RNL 语言标准 0.3 版（中文版）

0.3 版 · 修订 2 + 后续变更（2026-11-08，c82633b）

RNL（Rich Netlist）写在电路网表的注释里，描述电路怎样画成原理图、各部分是什么意思；电路本身只由网表定义。本文是这门语言 0.3 版的标准。

- 英文版 `rnl-0.3.md` 是正式文本。本文是它的中文译本；两者不一致时，以英文版为准。
- 标准词表 `rnl-words-0.3.json` 及其 JSON Schema `rnl-words.schema.json` 是本标准的一部分（5.3、附录 A）。
- RNL 的原则见 `rich-netlist.zh.md`（中文版，以英文版 `rich-netlist.md` 为准）；本标准称之为"原则"。
- 0.x 各版本之间不承诺兼容（16.1）。
- 本文按研究仓库 rnl-dev 里 0.3 版的草案修订 0.2 版，提交 d26d049 (2026-10-15)；改动见附录 D。

目录

1 概述 7
1.1 范围 7
1.2 本标准的结构 7
1.3 一瞥 7
1.4 术语与约定 8
1.5 核心原则 11
1.6 一致性 11
2 宿主嵌入 13
2.1 两层解析 13
2.2 载体 13
2.3 入口 14
2.4 记录的范围 14
2.5 停用记录与宿主文字 15
2.6 层 15
2.7 包文件 16
2.8 记录头 16
3 词法 18
3.1 字符 18
3.2 正式记号 18
3.3 自由文本 18
3.4 注释 19
3.5 空白与紧邻 19
3.6 名字 19
3.7 字面量 20
3.8 点 20
4 文法 21
4.1 文法 21
4.2 文法之外的约束 22
4.3 语序 22
4.4 错例 23
4.5 标准例句 24
5 对象与语句 26
5.1 对象 26
5.2 集合 26
5.3 语句与签名 29
5.4 消费方式 31
5.5 管道与嵌套 31
5.6 定义作为对象 32
5.7 句柄与名字 32
5.8 事实的键 32
5.9 没有名字的对象的编号 33
6 名字与作用域 35
6.1 层级 35
6.2 前缀 35
6.3 宿主引用 36
6.4 端子 36
6.5 子对象 36

6.6 命名 37
6.7 名字的范围 37
6.8 继承与遮蔽 37
6.9 定义体里的名字 38
6.10 空位 38
6.11 下标 38
6.12 端子名：DEVICE 39
7 失效与诊断 42
7.1 两个检查阶段 42
7.2 失效对象 42
7.3 严格与容错的输入 42
7.4 恢复 42
7.5 冲突不是失效 42
7.6 诊断 42
8 生效 44
8.1 整层读完再生效 44
8.2 阶段 44
8.3 外观队列 44
8.4 按键覆盖 45
8.5 冲突 45
9 定义 47
9.1 共同规则 47
9.2 符号：DEFINE-SYMBOL 47
9.3 样式：DEFINE-STYLE 50
9.4 普通定义：DEFINE 51
9.5 普通定义的用途 54
10 选择、规则与逐个生成 56
10.1 选择关键字 56
10.2 条件 `wired` 58
10.3 选择的求值 58
10.4 规则 59
10.5 逐个生成 59
11 放置与外观 61
11.1 位置：PLACE 61
11.2 朝向：ORIENT 61
11.3 符号与样式：SET-SYMBOL 与 SET-STYLE 62
11.4 一物多画 62
11.5 分组：GROUP 62
11.6 绘制顺序与图层：LAYER 63
11.7 坐标与单位 63
11.8 叠画：BUNDLE 64
12 虚拟元件 66
12.1 概念 66
12.2 创建虚拟元件 66
12.3 相对端子的位置 66
12.4 端口 67
12.5 标准虚拟元件 67
12.6 无连接标记 67
13 走线与文字 68

13.1 导线：WIRE 68
13.2 汇合点 70
13.3 文字：TEXT 70
13.4 标注：场景里的图元 72
13.5 （不用） 73
13.6 文字记号与属性引用 73
13.7 整个网络画成导线：ROUTE 75
14 布局意图 76
14.1 共同规则 76
14.2 ALIGN 76
14.3 ORDER 76
14.4 DISTRIBUTE 77
14.5 DISTANCE 77
14.6 信号流向 77
14.7 对称 78
14.8 优先级：PRIORITY 78
14.9 朝向关系：ORIENT-RELATION 79
15 电气语义 81
15.1 语义套用 81
15.2 电流方向：CURRENT 81
16 文件、版本、包与代称 83
16.1 版本 83
16.2 包与 USE-PACKAGE 84
16.3 标准包 86
16.4 扩展与未知内容 89
16.5 代称 89
17 修改模型 92
附录 A 关键字签名（规范性） 94
A.1 记录 94
A.2 头段记录与包 94
A.3 定义 95
A.4 符号内容与图元 95
A.5 放置与外观 96
A.6 选择 96
A.7 套用 97
A.8 导线 97
A.9 集合、分组与叠画 97
A.10 约束与优先级 97
A.11 电气语义 98
A.12 其他标准词 98
附录 B 标准例句（说明性） 99
B.1 放置与朝向 99
B.2 布局约束 99
B.3 外观 100
B.4 符号与图元 100
B.5 导线与文字 101
B.6 定义与套用 101
B.7 选择 102
B.8 集合与其他关键字 104

B.9 头段记录与包 104
附录 C 留给以后版本的内容（说明性） 105
附录 D 与 0.2 版相比的改动（说明性） 107

1 概述

1.1 范围

RNL（Rich Netlist）写在宿主网表的注释里。它描述网表定义的电路怎样理解、怎样呈现为原理图：用哪些符号，元件放在哪里，元件的朝向、对齐、连线、标注和样式怎样，几个元件合起来是什么意思（原则第 1 节）。

器件、类型、参数、连接和子电路接口只由宿主网表定义。RNL 引用它们，不能改变它们。RNL 也不校验绘图暗示的电气含义（原则 2.2）。

本标准定义：

- 怎样从宿主网表的注释里取出 RNL（第 2 章）；
- RNL 文本的词法和文法（第 3、4 章）；
- 文本描述什么对象，名字怎样解析，失效怎样处理，各条描述怎样合在一起生效（第 5–8 章）；
- 各类描述的词汇（第 9–17 章）；
- 标准词表，与本文一起发布，即 `rnl-words-0.3.json`（5.3、附录 A）。

本标准不规定布局算法和唯一的渲染结果（原则第 1 节），也不规定宿主网表怎样解析（原则 2.1）。标准包 `rnl-std` 和 `rnl-zh` 这类代称包是用 RNL 写的；本标准规定包怎样起作用（16.2、16.3、16.5），包的内容另行发布。

本文定义语言的 0.3 版。0.x 各版本之间不承诺兼容（16.1）。

1.2 本标准的结构

本标准由一个抽象模型和一种文本语法组成。抽象模型是对象、事实、消费方式和生效阶段（第 5–8 章）；文本语法是它的书写形式（第 2–4 章）。工具之间交换的是文本。

附录 A 列出各关键字的签名，由标准词表生成，是规范性的。附录 B 给出标准例句，附录 C 列出留给以后版本的内容，附录 D 列出与 0.2 版相比的改动，三者都是说明性的。注和例子是说明性的。其余都是规范性的。

1.3 一瞥

```
* Five-transistor OTA, input stage
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
.subckt ota5t inp inn out vdd vss vbias
M1 x    inp   tail vss nch W=2u L=0.5u
M2 out  inn   tail vss nch W=2u L=0.5u
M5 tail vbias vss  vss nch W=4u L=1u
* @RNL {
*   %dp1 = #M1(in_p) and #M2(in_n) treated @AS $DIFFERENTIAL_PAIR;
*   {#M1 #M2} @ALIGN("y"), i.e., same y;
*   #M5 is at @PLACE(20, 6) => it @ORIENT("R0");
*   %g0 = $GROUND is @ATTACH to &vss;
*   net &vss has a @WIRE connecting {#M5:3 %g0:p}
* }
.ends ota5t
.end
```

- `*` 开始一行 SPICE 注释。`@RNL {…}` 是一条记录，花括号里是用 `;` 分开的成员（第 2 章）。文件开头的头段记录声明语言版本（16.1）。
- `#M1`、`&vss`、`#M5:3` 引用宿主的元件、网络和端子。`%dp1`、`%g0` 是本层的 RNL 名字。`$DIFFERENTIAL_PAIR`、`$GROUND` 是标准包（16.3）里的普通定义（9.4）。`%g0` 是套用 `$GROUND` 得到的用例，标准包给它配了地符号，所以画成一个地（第 12 章）。
- `@AS`、`@ALIGN`、`@PLACE`、`@ORIENT`、`@ATTACH`、`@WIRE` 是关键字。参数写在紧贴关键字的括号里，只放字面量。AS 把写在它前面的对象填进写在它后面的那个定义的角色；ATTACH 把写在它前面的定义添在写在它后面的对象上（9.4、第 12 章）。
- `(in_p)` 标出角色。`%dp1 =` 给语句的结果起名。`=>` 把一段的结果交给下一段；后面只跟一个名字时，给前一段的结果起名（5.5）。

- `and`、`treated`、`, i.e., same y`、`is at`、`it`、`is … to`、`net`、`has a … connecting` 是自由文本，不改变含义（3.3）。标准例句在记号之间加这类词，让语句读起来像句子（4.5、附录 B）。去掉它们，这条记录就是：

```
%dp1 = #M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR;
{#M1 #M2} @ALIGN("y");
#M5 @PLACE(20, 6) => @ORIENT("R0");
%g0 = $GROUND @ATTACH &vss;
&vss @WIRE {#M5:3 %g0:p}
```

1.4 术语与约定

1.4.1 规范用语

- **应**（需要强调时写**必须**）、**不应**：对文件或实现的要求。
- **建议**、**建议不要**：推荐的做法；也可能有理由不这样做，建议先弄清这些理由。
- **可以**：允许的事。

语言本身的含义用陈述句写："语句失效""后写的生效"。

1.4.2 失效与诊断

- **失效**的条目（字段、语句、记录或文件；7.6）不起作用；失效的语句以失效对象为值（7.2）。每一处失效都是**错误**，不论原因是什么（7.6）。
- **警告**标出合法、但可能不是作者本意的条目；它照常生效，本标准另有规定的除外，如求解器在时间上限内没判定完的事实（8.5）。**提示**标出常见的覆盖、遮蔽或取舍，它们也可能是写错了。具体情况列在 7.6；本标准说某处给警告或提示的地方，建议实现把它报告出来。
- **解释失败**时，什么都不画，并报错（1.5 第 3 条、7.6）。

1.4.3 写法

- 正文里关键字写大写的名字，不带 `@`：PLACE、SET-SYMBOL。写语法和例子时带 `@`。
- 标准词、名字和字面量用代码字体写：`priority`、`"high"`、`instance`、`$GROUND`。
- 括号里的节号指本标准的节，如（9.4）。"原则 2.4"指 RNL 原则文档的 2.4 节，即 `rich-netlist.zh.md`；它以英文版 `rich-netlist.md` 为准。
- 例子用标准写法（4.3），不带自由文本，以自由文本为题的地方除外。标为错误的例子示范的是写错的写法。

1.4.4 术语

术语	英文	含义
代称	alias	标准词的另一种写法，或 `$` 名字以及一个定义的角色名、参数名、局部名的另一种写法；用 ALIAS 声明，或用 USE-ALIASES 加载（16.5）
代称包	alias package	只声明代称的包文件；用 USE-ALIASES 加载（2.7、16.5）
标注	annotation	写在场景里、不附着在对象上的图元或文字（13.4）
套用	apply, application	AS 和 ATTACH 做的事：把对象填进一个定义的角色。每次套用产生一个用例（9.4）
附着	attach	TEXT 附着在它收到的点或对象上：它可以读这个对象的属性，没给位置时放在对象那里或旁边（13.3）。不要与关键字 ATTACH 混淆，ATTACH 套用普通定义（9.4）
定义体、体	body	定义的花括号里的语句，或内嵌的包的花括号里的语句（9.1、16.2）
叠画	bundle	BUNDLE 产生的对象：几个宿主元件画成一个，代替它们画出来。它的端子代表各成员的同名端子（11.8）
总线	bus	网络槽位收网络集合的导线，只有一个成员也算；网表里没有总线，它只是一种画法（13.1）
载体	carrier	承载 RNL 的宿主整行注释：注释符（`*`，Spectre 语言方式里还有 `//`）写在第 1 列，后面什么都没有、只有空白，或者一个空格再接内容（2.2）

术语	英文	含义
提交	commit	记录对它的语句的处理：记录里的每条语句都成为它所在层的一部分，有没有名字都一样（5.4）
约束	constraint	给求解器的位置或朝向的条件：ALIGN、ORDER、DISTRIBUTE、DISTANCE、ORIENT-RELATION（第 14 章）
定义	definition	DEFINE、DEFINE-SYMBOL 或 DEFINE-STYLE 产生的东西，通常用 `$` 起名（第 9 章）。DEFINE 产生的是**普通定义**；符号（DEFINE-SYMBOL）和样式（DEFINE-STYLE）是**特殊的定义**。在角色、参数、套用或用例已经表明是哪一种的地方，本文把普通定义只叫"定义"；要与符号、样式区分时写"普通定义"
事实	fact	语句携带的断言，如"M1 在 (10, 20)"（5.1）
失效对象	failed object	失效语句的值（7.2）
补位	fill	标准包的规则为没有用别的方式画出的端子和网络添上的地、电源和端口符号（10.5、16.3）
自由文本	free text	语句的记号之间的文字；它不改变含义（3.3）
生成集	generated set	生成规则产生的集合；其中单个成员不能被引用（10.5）
分组	group	用 GROUP 标过的集合；提示求解器把成员当作一块，不保证任何几何关系（11.5）
句柄	handle	事实所说的那个东西：宿主的元件、网络、端子，或 RNL 创建的点、导线、集合等对象。同一个句柄的各条事实合在一起生效（5.1、5.7）
头段记录	header record	记录 `@RNL(type="header") {…}`，文件的第一条记录：版本、代称、端子名、根上的包（2.8、16.1）
遮蔽	hiding	下层的定义，或后面的包里的定义，代替上面来的同名定义被使用（6.8、16.2）
宿主、宿主网表	host, host netlist	承载 RNL 的网表。宿主适配器读它，把它的注释和对象提供给 RNL（2.1）
风格	house style	包或实现定下的整体画法：用哪些符号、样式和布局定义，没画导线的连接用标签还是飞线表示，等等。定下风格的包是风格包。风格不是 DEFINE-STYLE 意义上的样式
条目	item	失效所针对的：记录头的一个字段、一条语句、一条记录、一个文件，或整个解释（7.6）
元件	instance	宿主网表里的一个元件，包括子电路元件（X 行）。`XI<3:0>` 这样的叠合元件代表几个元件（6.11）
引用名	key	文字在它附着的对象上的名字：M1 上 `key="wl"` 的 TEXT 就是 `#M1::wl`（13.3）
关键字	keyword	`@` 加一个大写的词，如 `@PLACE`，或声明过的代称（16.5）；一段至多一个（4.1、5.3）
标签	label	附着在对象上的 TEXT；它可以引用对象的属性，如名字、值（13.3）
层	level	文字上的一个范围：主文件的文件层（就是电路的顶层）、被 include 的文件的文件层，或一个子电路定义；每个包也是一层。层按文字的包含关系排成一棵树（2.6、6.1）
局部对象	local object	普通定义的体里起了名的对象；每个用例各有一份，从外面写作 `%c1::mid`（9.4）
主对象	main object	填用例第一个角色的对象。它有坐标点时（端子的引脚、元件的原点、点），画出的用例就画在这个点上，没有自己的位置；否则用例有自己的位置（12.1）
成员、段、语句	member, segment, statement	成员是集合里用 `;` 分开的一部分。段是成员里用 `=>` 分开的一部分（`=>` 后面只有一个名字时不是段，4.2）。语句是带关键字的段，或不带关键字的段里的一个对象（4.2、5.3）
名字	name	`%x =`、`$X =`、`=> %x` 或 `=> $X` 给对象起名字；名字不产生对象，在一层里只绑定一次（6.6）

术语	英文	含义
包	package	一组定义和规则，用 USE-PACKAGE 放在某一层的上面；写成包文件（2.7），或内嵌在 USE-PACKAGE 的体里（16.2）。只引用别的包的包是**集成包**，如 `rnl-std`（16.3）
根上的包	packages at the root	排在所有层之前的包：不用写就使用的标准包，以及头段记录里 USE-PACKAGE 的包（16.3）
引脚	pin	符号上画端子、接导线的点（9.2.4）
优先级	priority	几何事实的档位：`"disabled"`、`"low"`、`"normal"` 或 `"high"`。冲突时先比优先级，再比写在哪里（8.5、14.8）
记录	record	注释里的一条 `@RNL(…) {…}`。括号里的字段组成记录头，作用于记录里的每条语句（2.3、2.8）
RNL 对象	RNL object	带关键字的语句或 `{…}` 产生的东西：一个句柄和一些事实（5.1）
角色	role	对象在语句里填的槽位，标在紧跟对象的括号里，如 `#M1(in_p)`。定义的角色在它的角色表里起名，在体里写作 `?name`；内置关键字的角色就是它的槽位名（5.3、9.4）
规则	rule	以选择的结果为对象的语句。写在某一层的规则，在这一层求值，在下面每一层再各求一次，每次只看那一层的对象（10.4）
场景	scene	一层的对象（来自宿主的和来自 RNL 的），连同已经生效的事实（第 8 章）
选择	selection	SELECT-… 关键字的结果：在它的范围里满足条件的某一种对象，用到时才求值（第 10 章）
集合	set	有序的一组对象，写成 `{…}`，或由选择得到（5.2）
对象槽位	slot	关键字收对象的地方之一，如 DISTANCE 的 `from` 和 `to`。每个槽位有名字，可以当作角色写（5.2.1、5.3）
求解器	solver	由 RNL 算出位置和朝向、并写进生成的记录的工具（14.8、第 17 章）
标准写法	standard form	每个关键字唯一的一种写法，工具输出的就是它：关键字写在哪里，哪些参数按位置写，标不标角色（4.3、5.3）
标准包	standard package	包 `rnl-std`，含定义、符号、样式和规则；每个文件都使用它，除非文件另作说明（16.3）
标准例句	standard sentence	用标准写法写、在记号之间加了自由文本、读起来像句子的语句（4.5、附录 B）
标准词	standard word	标准词表定义的词：关键字、槽位名、参数名和取值、记录头字段、种类、子对象名、属性名、样式属性、`true` 和 `false`（5.3）
样式	style	只指用 DEFINE-STYLE 定义的属性：颜色、线型、线宽、字号等（9.3）。布局上的偏好不是样式
下标	subscript	宿主名字或端子名后面紧跟的 `<…>` 或 `[…]`，如 `&D<3>`、`&D<7:0>`，代表一个或几个名字（6.11）；不是文字记号里的上下标（13.6）
子对象	sub-object	用 `::` 取到的对象的一部分，如 `#M1::position`、`%dp::in_p`（6.5）
符号	symbol	DEFINE-SYMBOL 产生的东西：图元、引脚和文字，为元件、叠画或用例画出（9.2）
生效	take effect	一层整个读完后，它的事实按阶段变成场景（第 8 章）
记号	token	关键字、引用、点、括号、花括号、`;`、`=>` 或 `=`；其余是自由文本或注释（3.2）
端子	terminal	元件的一个连接，与它的网表行一致：`#M1:d`，或按序号写 `#M1:1`。`#:inp` 是本层的端口 `inp`（6.4）
用例	usage	一次套用产生的对象。它记下用的是哪个定义、每个角色由哪些对象填；有符号时就画出来（9.4、9.5）
虚拟元件	virtual part	有符号的用例：画出来，但不属于网表，如地、电源、端口符号、汇合圆点（第 12 章）

术语	英文	含义
标准词表	word list	一个语言版本全部标准词和关键字签名的机器可读的表，本版是 `rnl-words-0.3.json`（5.3、附录 A）

1.5 核心原则

本标准的规则遵循以下几条原则。第 1–6 条细化原则 2.9；第 7 条说明词汇怎样分工。

1. **一切皆描述。** RNL 文本是一组描述。解释器先收齐全部描述，再给出结果。没有执行过程，名字可以在绑定它的语句之前使用。
2. **书写位置只作排序数据。** 描述写在哪里（所在的层和在文本里的先后）只在三处起作用：描述互相矛盾时保留后写的（几何事实先比优先级；8.3、8.4、8.5）；绘制顺序（11.6）；集合成员的次序（5.2）。此外结果与书写顺序无关。管道的每一段有自己的位置：后面的段算比前面的段后写（5.5）。
3. **矛盾。**
   - 读不懂的描述（语法错、名字找不到、种类不对等），以及自相矛盾的描述，失效（原则 2.4、2.9；第 7 章）。
   - 能读懂但互相矛盾的描述，在本标准给出确定办法的地方按办法取舍，如按键覆盖（8.4）、外观队列（8.3）和几何事实的取舍（8.5）；可以给提示。
   - 暂时还没有确定办法的，整个解释失败（7.6）。本版没有这种情形（7.5）。以后的版本可以定出办法，改为容错；现在能用的文件不受影响。
4. **每条描述看到的是生效前的状态。** 一条描述读到的是它所在阶段开始时的状态；它自己的效果和同一阶段其他描述的效果都不在内（8.2）。
5. **对象与名字。** 每条带关键字的语句和每个 `{…}` 都产生一个新对象，由句柄和事实组成。引用不产生对象。名字只是指向一个对象，在一层里只绑定一次。覆盖针对事实，不针对名字（5.1、6.6）。
6. **不为可读性限制写法。** 难读、但含义确定的写法照常允许，由作者自己克制。只有没有确定含义的写法才是错误。
7. **关键字与定义。** 实现必须自己懂、用别的描述写不出来的，是关键字。能用已有描述写进普通定义的体里的，是定义，它的名字只作提示（9.5）。

1.6 一致性

文件。 一份宿主网表的记录，按第 2–4 章读、按第 5–17 章解释，不出错误（7.6），包括它使用的包的内容里的错误，这份网表就符合 RNL 0.3。警告和提示不影响一致性。包文件单独检查时（16.2）同样判断。头段记录声明了别的语言版本的文件，不受本版本约束（16.1）。

实现指解释器（只用来检查的解释器也叫 lint）、查看器、求解器、编辑器以及其他读写 RNL 的工具。符合标准的实现：

1. 按第 2 章从宿主的注释里取出 RNL，按第 3、4 章读它；
2. 按第 5–17 章给每条描述应有的效果，使用标准词表里的签名（5.3）；
3. 报告每个错误，给出位置和原因（7.6）；错误不能调低，也不能关掉；
4. 建议报告本标准列出的警告和提示（7.6）；
5. 可以另加自己的检查，但只能是警告或提示：不应让本标准允许生效的条目失效，也不应让本标准规定失效的条目生效；
6. 可以提供更严的模式，停下或拒绝整份输入，但不作为默认；
7. 让用户能看到诊断；怎样呈现由实现决定（7.6）；
8. 声明自己支持哪些语言版本，对不支持的版本的文件按 16.1 处理；
9. 把不认识的关键字、参数和取值当作错误（16.4）；
10. 按 16.3 在根上使用标准包；
11. 写 RNL 时，新语句用标准写法写，不重排用户写的语句（4.3）；保存文件时写头段记录和版本（16.1）；不认识的语句、失效的语句、宿主文字、普通注释和不支持的版本的文件都原样保留（16.4）；只改第 17 章允许它改的记录。
12. 求解时，按 8.5 取舍冲突，把自己定下的每个决定都写回，交给别的工具的结果也只用求解器的记录能含有的形式（14.8）。

本标准不规定求解器在满足约束的各种位置中怎样选，也不规定查看器怎样画 RNL 没有定下的部分；留给它们决定的地方在正文里写明（如 9.2.6、13.1）。

一**致性测试**查错误（连同位置和每处失效的范围）和得到的场景。警告可以作为可选的期望来查；提示不查（7.6）。

2 宿主嵌入

2.1 两层解析

- 宿主适配器：
  - 识别宿主合法的普通注释，判断哪些行是载体（2.2）；
  - 确定每段注释属于哪一层，以及每条记录、每个子电路来自哪个文件、哪一条 include（6.1）；
  - 给出源位置和文本的顺序；
  - 提供宿主对象：元件、网络、端子、子电路端口和模型类型。
- RNL 核心只在这些注释里找记录。它不应对整个网表做不看上下文的 `@RNL` 搜索：宿主语句、宿主字符串里出现的同样字符不是 RNL。
- 解释器先解析完整的宿主网表，再处理 RNL（原则 2.9）。

2.2 载体

记录写在宿主的整行注释里，这样的注释行叫载体。宿主适配器逐行判断哪些行是载体，去掉注释符和它后面的一个空格，把其余内容交给 RNL 核心（2.1）。

- **只用整行注释。** 行尾注释（`$`、`;`、`//`、`!` 等）不读也不写，也不用块注释。
  - 行尾注释里的 `@RNL` 只是宿主文字，给警告。
  - 注：行尾那段算不算注释，随方言、模式和选项而变：HSPICE 的 `$` 可以紧跟在数后面（`1k$x`），ngspice 的 `ps` 方式里 `$` 是普通字符，Eldo 可以用 `.COMCHAR` 改；CDL 里 `$SUB=` 这类写法是数据。写进行尾还要改宿主语句那一行（原则 2.1）。SPICE 和 Spectre 都没有块注释（Spectre 只在 `reliability{}` 里有）。
- **各宿主的注释符。**

宿主	注释符
SPICE 系：SPICE3、ngspice、HSPICE、Xyce、LTspice、PSpice、WRspice、Eldo，以及 CDL 网表	`*`
Spectre 语言方式的段落：`.scs` 文件，或 `simulator lang=spectre` 之后的文字	`//` 或 `*`
Spectre 里 SPICE 语言方式的段落，包括没有 `.scs` 后缀、开头也没写 `simulator lang=spectre` 的被 include 的文件	`*`

  - `*` 在 Spectre 的两种语言方式里都是注释。`//` 只在 Spectre 语言方式里有文档，在 Eldo 和 CDL 的文档里没有。
  - 一条记录的几行可以混用 `//` 和 `*`。工具写入记录时应跟随本文件已有的记录；没有记录时，SPICE 方式写 `*`，Spectre 方式写 `//`。
  - 载体属于外层写法，外层写法不改（16.1）。以后支持新的宿主时只增加载体，不改已有的。
- **载体行。** 记录的每一行都应满足下面两条：
  - 注释符写在第 1 列，前面没有空白。
  - 注释符后面要么什么都没有，或者只有空白，这是空注释行；要么先是一个 ASCII 空格，再写内容。所以 `*@RNL`、`*#M1`、`//@RNL` 都不是载体行。

    注：HSPICE、PSpice 要求 `*` 在第 1 列；有缩进的行，各家处理不同。紧贴注释符的文字在别的工具里有含义：ngspice、WRspice 把 `*#` 行当命令执行，WRspice 对 `*@` 行也是这样；`*.` 开始一条 CDL 指令（如 `*.PININFO`），`*|` 开始 DSPF 的数据，`//pragma` 是 Spectre 的加密封套。
  - 记录还没写完时遇到不合这两条的注释行（如 `*#M1 …`、`** …`、有缩进的 `* …`），记录被截断：整条记录失效（2.4）。记录之外，这样的行里的 `@RNL` 不是入口，给警告。
  - 写 RNL 的行建议不要以 `\` 结尾。这样的行给警告，工具不应写出这样的行。RNL 照常读，没有东西失效；危及的是宿主网表。注：WRspice 会把下一行接进这行注释；Spectre 用 `-E` 运行（先过 C 预处理器）、Eldo 开了 `SLASHCONT` 时也可能这样；下一行若是宿主语句，就被吞进注释了。
- **不读 RNL 的位置。** 这些位置的 `@RNL` 给警告，工具不应往这些位置写：

  - 每个宿主文件的第 1 行，以及宿主当作标题的别的行（如 WRspice 里 `.newjob` 的下一行）。主文件的第 1 行是它的标题，被 include 时却不是标题；第 1 行一律不读，同一个文件作主文件和被 include 时读起来就一样。所以头段记录最早写在第 2 行（2.8）。
  - 主文件 `.end` 之后的文字。注：`.end` 之后，ngspice 接着读，Xyce 停下，HSPICE、PSpice 可能当作下一个电路。
  - 宿主不当作网表注释的区域：控制脚本（ngspice 会执行 `.control … .endc` 里的 `*#` 行）、别的语言的块和文件（WRspice 的 `.verilog`、Spectre 的 `reliability{}`、用 `ahdl_include` 引入的文件）、密文、预处理去掉的部分、没选中的 `.lib` 段。适配器说不准的区域按不读处理，并给警告。
- **照读但给警告的位置：**
  - 语句和它的续行（`+` 行）之间的注释行。它们对 RNL 的含义是唯一的，所以照读；但宿主处理不同（PSpice 报错，SPICE 3f4 会把后面的 `+` 行接到注释上）。这些行给警告，工具不应往这里写。
  - 写成明文的加密区（`.prot … .unprot`、`protect … unprotect`）。照读，给警告：加密以后记录就没了。
- **写入建议：**
  - RNL 不限制行长（2.4）。建议生成器在空白处换行，每行不超过 132 个字符：这是 PSpice 旧版的上限，在常见的宿主里最严（HSPICE 是 1024，Eldo 是 2000）。知道宿主上限的适配器按宿主的上限。一个记号在一行里放不下时，工具应报错，不应写出（字符串不能续到下一行，3.7）。
  - 编码见 3.1。

2.3 入口

- 记录以 `@RNL` 开始。入口只认大写。
- 各宿主都用同一个入口 `@RNL`；它就是原则 2.1 的子标签。
- `@RNL` 可以写在注释中间；同一行上它之前的文字属于宿主注释。
- **`@` 前的空格。** 入口 `@RNL` 和关键字的 `@`，紧前一个字符应是 ASCII 空格（U+0020）；TAB、换行、行首都不算。载体行上，注释符后面的那个空格算作 `@` 前的空格，如 `* @RNL`。字符串、加引号的名字、`/* */` 注释里的 `@` 不受限制。
  - 所以 `{`、`=>`、`=`、`;` 后面的关键字前也要留空格：`#M1 @SET-SYMBOL { @DEFINE-SYMBOL {…}}`、`%x = @DEFINE(…)`、`#M2 @PLACE(30, 20) => @ORIENT("MY")`。
  - 注：宿主注释里常有 `@`，如邮箱地址、`@param`；有了这条，这类文字不会被当作 RNL。WRspice 还会执行以 `*@` 开头的行（2.2）。
  - 记录外的 `x@RNL` 不是入口，只是宿主注释。记录里前面没有空格的 `@` 既不是关键字，也不是自由文本的字符：它是语法错误，所在的成员失效（7.4）。

2.4 记录的范围

- 记录写作 `@RNL(…) {…}`。括号里的记录头可以省；花括号不能省，单句也要写花括号。记录头的括号紧贴 `@RNL`，与关键字的参数括号紧贴关键字一样（3.5）：`@RNL (type="header") {…}` 是错误，整条记录失效。
- 记录从 `@RNL` 开始，到与它的第一个 `{` 相配的 `}` 结束。找相配的 `}` 时跳过字符串、`/* */` 注释和参数括号里的内容。
- **跨行的记录。** 一条记录可以写在连续的几行注释（载体，2.2）里。去掉每行的注释前缀，把各行内容接起来；换行是空白（3.5）。
  - 只有 `*`（或 `*` 和空白）的空注释行是空白，记录照常继续。
  - 还没写到收尾的 `}`，就遇到不是载体的行（宿主语句，`.subckt`、`.ends`、`.include` 这类行，完全空白的行，或不按载体写法写的注释行，2.2），或者文件结束，记录就不完整，整条失效。所以记录不会跨子电路和 include 的边界。记录里要留空行，就写一个只有 `*` 的行。
  - 记录还没结束时，在字符串和注释之外又出现入口 `@RNL`：前一条记录不完整，整条失效，从新的入口重新读。这样，漏写一个 `}` 只丢一条记录。
  - 不完整的记录是错误，诊断说明记录不完整，并指出在哪里截断。多打一个空行就让整条记录失效，这很容易发生，不应静默放过（7.6）。

  - 记录里的每一行注释都是 RNL。中间夹一行普通说明，如 `* the next line is for M2`，会当作自由文本读，含有不能作自由文本的字符时是错误。记录里的说明写在 `/* */` 里（3.4）。
  - RNL 不限制行长；宿主的限制归适配器。长语句可以在空白处拆到几行；记号和字符串不能拆开（3.5）。给生成器的建议见 2.2。
- 记录结束后，同一行上再出现记号是错误：这些记号失效，记录本身照常生效。新的 `@RNL` 除外，所以一行可以写几条记录。
- 记录的外层写法（注释载体、入口 `@RNL`、记录头、花括号配对）与语言版本无关，以后的版本都不改（16.1）。记录头的字段见 2.8。

2.5 停用记录与宿主文字

- 记录之外、不含 `@RNL` 的注释行是宿主注释，整行忽略。
- **停用记录。** 注释行里的 `/*`，如果前面只有空白或注释，或者紧跟在一条记录之后，就开始一段注释，到 `*/` 为止。其间的 `@RNL` 不是记录。
- 注释行中间文字里的 `/*`（如 `models/*.lib`）只是宿主文字。
- **宿主注释的结束。** 载体是整行注释（2.2）：宿主注释到行尾结束，行内没有东西能提前结束它，所以宿主注释的结束与 RNL 文字没有冲突。

2.6 层

- 层按文字划分，上下级见 6.1：
  - 主文件的文件层：写在主文件里、不在任何 `.subckt … .ends` 之内的部分。它就是电路的顶层。
  - 每个被 include 的文件的文件层。`.lib` 的段同样处理。
  - 每个 `.subckt … .ends`。
- 记录属于它所在的层（原则 2.5）。
- **多文件的顺序。** 被 include 的文件自成一层，排在 include 它的那一层之下（6.1）。所以 include 行写在哪里不影响顺序，这一点与 USE-PACKAGE 不同（16.2）；并列的几个被 include 的文件互不影响。同一层里按文本先后。
- **scope。** 记录可以用 `@RNL(scope="name")` 指定目标层。写的是子电路名，指那个子电路的定义。下面的文件里，最后一条记录写在主文件里，作用于子电路 `ota`：

```
* A record written outside its subcircuit
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
.subckt ota inp inn out tail vss
M1 x inp tail vss nch
M2 out inn tail vss nch
.ends ota
X1 vip vin vo vt 0 ota
* @RNL(scope="ota") { #M1 @PLACE(10, 20)}
.end
```

  - 每个子电路只有一张图（6.1），所以 scope 指定义，不指某一个元件。
  - scope 只能向下：目标应是本层在文字上包含的子电路，即本文件里定义的，或本层直接或逐级 include 进来的文件里定义的。子电路里不写 scope：子电路的 RNL 只管它自己。子电路里的记录写了 `scope` 是错误，当作没写（2.8）。目标找不到或不在本层之下时是错误；记录里的语句没有层可以解析，都失效（第 7 章）。
  - 记录里的引用都在目标层解析。它的语句提交到目标层，绑定的名字也发布到目标层，与目标层自己的记录共享（6.7）；目标层已经把这个名字绑定到别的对象时，再绑定是错误；绑定到同一个对象则可以（6.6）。scope 只影响它自己的记录，不改变宿主网表。
  - 顺序：scope 带进一个子电路的语句排在这个子电路自己的文字之后；几处都 scope 到同一个子电路时，外层的排在后面（8.3、8.5）。外面明确写的覆盖里面写的，与外层用例的优先级盖过它的体同理（14.8）。
  - 求解器的输出和用户在编辑器里做的修改不应使用 scope；它们写进子电路自己的文字（第 17 章）。scope 只用于作者自己选择写在外面的语句。

2.7 包文件

包可以写成单独的包文件，由 USE-PACKAGE 找到（16.2）。包文件是纯 RNL 文件，建议扩展名 `.rnl`，用 UTF-8 写。它不是网表，不用 `.include` 加载。

- 包文件的**读法**与网表里的载体相同（2.2、2.4），只是每一行都是载体，注释符和它后面那个空格都算空的：
  - 行首的 `@` 算前面有空格（2.3），所以 `@RNL` 和关键字可以从第 1 列写起；
  - 空行是空注释行，记录照常继续；
  - 记录之外的文字是说明，忽略，与宿主注释相同（2.5）；`/* */` 照常起作用；
  - 第 1 行照读：包文件不是网表，没有标题行（2.2）。
- 包文件里写 `@RNL(…) {…}` 记录，按 16.1 的顺序解析。第一条记录是头段记录，里面写 VERSION 和 PACKAGE。包里能写什么，包括引用别的包，见 16.2；代称包也是包文件（16.5）。

下面是一个包文件的全文，没有注释前缀。第三行在记录之外，是说明：

```
@RNL(type="header") { @VERSION("0.3"); @PACKAGE(name="acme-style", version="1.2")}

ACME house style: NMOS symbol and its rule.
@RNL {
  $NMOS_ACME = @DEFINE-SYMBOL {
    $NMOS;
    ? @TEXT("${name}", key="name") => @PLACE(-2.5, 1.5)
  };
  # @SELECT-INSTANCE(type="nmos") => @SET-SYMBOL $NMOS_ACME
}
```

2.8 记录头

`@RNL(…)` 括号里的字段作用于记录里的每条语句，都写名字。字段名和 `type` 的值按标准的拼写写，区分大小写，不能有代称（3.6、16.5）。

```
* @RNL(type="header") { @VERSION("0.3")}    /* the first record of the file: read as version 0.3
(16.1) */
* @RNL { %dp1 = #M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR}    /* no type: "authored" */
* @RNL(type="generated", priority="high", author="rnl-solver 0.3", hash="sig1:9f3a62c1") {
*   #M1 @PLACE(12, 20); #M2 @PLACE(28, 20)
* }
* @RNL(author="rnl-editor 0.1") { #M2 @PLACE(x=30, priority="high")}
```

字段	含义	影响解释
`type`	记录的种类：`"authored"`（默认）、`"generated"`、`"header"`	是
`scope`	目标子电路（2.6）	是
`priority`	记录里语句的默认优先级；给 AS、ATTACH 的只有 `"disabled"`（14.8）	是
`author`	作者、工具及其版本、包的版本等；任意字符串	否
`hash`	求解器输入的散列（第 17 章）	否
`numbering`	记录所用编号的校验值（5.9）；只用于 `type="generated"`	是
`modified`	最后修改的时间，ISO 8601 字符串	否

- **type**：取值固定，各有确定的行为。
  - `"authored"`，默认：手写的、编辑器写的或导入的。
  - `"generated"`：求解器的结果。求解器求解时，不把 `type="generated"` 的记录算进解释：它们不带来对象、名字或事实，也不约束结果；此外怎样用它们，由求解器自己决定。求完把它们整条换掉，不论作者是谁。编辑器拖动器件时重画的导线也算求解结果（14.8、第 17 章）。`type` 不隐含优先级：求解器明确写出 `priority="high"`。
  - `"header"`：文件的设置（16.1）。头段记录可以不写。写的话，它在文件层，不在 `.subckt` 里，不带 `scope`；它是文件的第一条记录，至多一条。里面写 VERSION（16.1）、ALIAS 的代称和 USE-ALIASES 的代称包（16.5）、DEVICE 的端子名（6.12）和 USE-PACKAGE 的根上的包（16.3）；包文件还在这里写 PACKAGE（2.7、16.2）。头段记录本身按标准拼写读，不认代称（16.5）。写在文件后面的头段记录，或第二条头段

记录，是错误，整条忽略；已经读过的记录不回头改。经 `.lib` 读入的一节，按一个文件算（2.6）：它的头段记录是节里的第一条记录。

- **元数据**：`author`、`hash`、`modified` 不影响解释；删掉它们，画出来的结果不变。工具拿它们判断怎样处理一条记录（第 17 章）。`type`、`author` 都不影响取舍次序（8.3）。
- **出错**：
  - 不认识的字段只给警告，忽略；它不影响这条记录和别的字段。这给以后的扩展留了位置。给这个警告时，应指出最接近的字段名。注：这样一来，影响解释的字段拼错了（如 `priorty="low"`）也只得到警告，记录按优先级 `"normal"` 生效；以后新增的这类字段，要靠版本（16.1）让旧工具知道自己过时了。
  - 认识的字段写错（值不对，`numbering` 写在 `type` 不是 `"generated"` 或带 `scope` 的记录里，或同一字段写了两次）是错误，按没写这个字段处理：`priority` 回到 `"normal"`，`type` 回到 `"authored"`，元数据丢掉。例外是目标找不到的 `scope`：记录里的语句没有层可以解析，都失效（2.6）。按位置写的字段是错误，忽略。
- 记录没有标识。工具按 `type`、`author` 和语句的键（5.8）找自己的记录；在另一条记录里引用一个对象靠它的名字，求解器的记录还可以用编号（5.9）。别的说明写在注释里。
- 求解结果不是要采纳的提案：它直接生效，过时的结果只给提示（第 17 章）。

3 词法

3.1 字符

- 正式记号都由 ASCII 字符组成。
- 非 ASCII 字符只出现在字符串（包括加引号的名字）、自由文本、注释和代称里，以及不加引号的角色名和参数名里。在自由文本和注释以外，不加引号的非 ASCII 词只出现在 `@` 后面和括号里：`@` 后面的应是声明过的代称；括号里的是代称，或角色名、参数名本身（3.6、16.5）。
- **编码**。宿主适配器按宿主的规则解码，RNL 处理的是字符。写 RNL 的工具不应改变宿主文件的编码和换行符，不应加字节顺序标记（BOM）；不应写出宿主编码表示不了的字符，改为给警告。新文件建议用 UTF-8。
- **宿主注释里的非 ASCII 字符**。 适配器应写明哪些宿主验证过注释里的非 ASCII 字符；在别的宿主上，非 ASCII 字符给提示。

  注：各宿主的支持不一。ngspice（v42 起）、WRspice（标题除外）、Xyce 和新版 LTspice 接受注释里的非 ASCII 字符；HSPICE、PSpice、Eldo 的文档没写；Spectre 的手册说网表是 ASCII 文件；QSPICE 用 Windows-1252。不加 BOM 还有一个原因：旧版 ngspice 会把它粘到被 include 文件的第一个记号上，WRspice 在第 1 行以它开头时中止读入。

3.2 正式记号

记号	写法
关键字	`@` 加大写的名字，如 `@PLACE`；或声明过的代称（16.5）
引用	前缀加名字，可带端子和子对象，如 `#M1`、`#M1:d`、`%dp1::in_p`（第 6 章）。宿主名字或端子名后面可以紧跟下标，如 `&D<3>`、`&D<7:0>`（6.11）。求解器的记录里还有 `%#` 加无符号整数，即编号，如 `%#3`（5.9）
点	`(x, y)`，不紧贴关键字或对象：前面是空白、注释、`{`、`;`、`=>` 或 `=`（3.8）
括号	参数和角色的 `(` `)`，紧贴关键字或对象（3.5）
花括号	集合的 `{` `}`
分隔符	`;`（分开成员）、`=>`（管道和命名）、`=`（成员开头的命名）

括号内（参数、角色和点的 `()`，数组的 `[]`）另有字符串、数、`,`、`=`、`_` 和名字（参数名、角色名、`true`、`false`），在普通定义的体里还有参数空位 `?name`（6.10）。下标的 `<>` 和 `[]` 里只有不带符号的整数、`,` 和 `:`（6.11）。

3.3 自由文本

- 括号外，除了下面列出的字符，任何字符都可以作自由文本，包括全部非 ASCII 字符（如中文标点 `：`、`、`、`；`、`（`、`）`）和 `-`、`:`、`'`、`!`、`+`、`/`、`*`、`|`。例外是：
  - `@`、`#`、`&`、`%`、`$`、`?`、`"`、`;`、`=`、`{`、`}`、`(`、`)`：在括号外，它们开始正式记号或属于正式记号（3.2）。`?` 开始空位（6.10）；在符号体里，单独的 `?` 指用这个符号画出的对象（13.3）；
  - `[`、`]`、`<`、`>`：数组和下标的括号（3.7、6.11）；
  - `/*`：开始注释（3.4）。
- 几条位置规则让含义唯一：
  - 紧贴引用、后面是名字或引号的 `:` 或 `::`，照常是端子或子对象（6.4、6.5）。后面不是名字时，`:` 是自由文本，如 `Input pair: %dp1 = …`、`#M1: the left one`。
  - 紧贴名字的 `<` 或 `[` 是下标（6.11）。
  - 关键字的名字里，`-` 后面跟字母时才属于名字（关键字见 3.6）。`@GROUP- the pair` 是 GROUP 加自由文本。
- 自由文本可以写在括号外任意两个记号之间，也可以写在成员开头。
- **自由文本是语句的一部分**。写在里面的引用不是说明，而是这条语句的对象：`Unlike #M2, #M1 @PLACE(x=0)` 把 M2 也放到了 x = 0。只作说明的文字写进注释（3.4），或者让它单独成为一个成员，用 `;` 与下一个成员隔开：只有自由文本的成员是空成员，忽略（4.1）。
- 标准不检查自由文本，自由文本不影响含义（原则 2.8）：`do not move #M2 @PLACE(x=30)` 仍会移动 M2。对否定词这类词，实现可以另给自己的警告（7.6）。

- 同理，误写的 `#M1.d` 里，`.d` 是自由文本，引用的是 M1 本身，不是它的漏极（给警告，见下）。
- 括号内没有自由文本。
- **警告。** 下面这些写错可能静默通过；它们给警告（7.6）：
  - 自由文本里的词与这一层的宿主元件名、看得到的 RNL 名字（不带前缀）或关键字名相同，如 `#M1 #M2 M3 @ALIGN("y")` 里的 `M3`、`WIRE {…}` 里的 `WIRE`：多半漏了 `#`、`%`、`$` 或 `@`。网络名常是普通词（如 in、out），不查。
  - 引用后面紧跟 `.` 加名字，如 `#M1.d`：端子用 `:` 写。
  - 引用后面紧跟 `-`、`!`、`'`、`+`、`/`、`*`、`\`、`^`、`~` 或 `|`，如 `&vdd!`、`#X1-a`：多半是名字里有这个字符，要加引号，如 `&"vdd!"`（3.6）；不加引号，引用就静默地指 `vdd` 或 `X1`。
  - 自由文本里的 `|`：管道写作 `=>`（4.2 第 3 条）。
  - 紧贴引用的 `:` 或 `::` 后面跟着非 ASCII 的词，如 `%c1::中点`：`::` 后面含非 ASCII 字符的名字要加引号，如 `%c1::"中点"`（3.6）；不加引号，`::` 和这个词是自由文本，引用指的是 `%c1`。
- 其余这类写错多数已经是错误：括号里没有自由文本，所以 `@ORIENT(MY)` 是错误；`#M1 @PLACE 1 2` 与签名不符；`#M1 (in_p)` 是错误，因为隔了空白的括号开始一个点，而 `(in_p)` 不是点（3.5）；`R1 -> R2` 里的 `>` 不能作自由文本。
- 给作者的写作参考由标准词表生成（5.3）。求解器和编辑器写的记录不应含自由文本。

3.4 注释

`/* … */` 可以写在任何允许空白的地方，包括括号内。注释不能嵌套，里面可以写任何字符。含有引用、或含有 `@`、`#`、`;` 这类字符的说明，写在注释里（3.3）。

3.5 空白与紧邻

- 空白是空格、TAB 和换行。记号不能拆开跨行，字符串也不能跨行。
- 以下位置不允许有空白和注释：前缀与名字之间，`%#` 与编号之间；`:`、`::` 的两侧；名字与它的下标之间（6.11）；关键字与它的参数括号之间，以及 `@RNL` 与它的记录头括号之间（2.4）；对象与它的角色括号之间。
- 隔了空白或注释的 `(` 开始一个点（3.8）。所以 `#M1 @PLACE (10, 20)` 是 PLACE 加一个点，PLACE 没有参数：是错误。`#M1 (in_p)` 是 M1 加一个不合写法的点：也是错误。诊断建议去掉空白。
- 名字（加不加引号都一样）后面紧跟的 `<` 或 `[` 总是开始一个下标（6.11）。点写成 `(x, y)`，与下标不冲突；名字后面要写点，就用空白隔开，如 `&D (1, 2)`。下标里可以写空白和注释，如 `&D<1, 2:3>`（6.11）。
- 紧跟关键字的 `.` 是自由文本，如 `{#M1 #M2} @GROUP.` 里的句号；`@BUNDLE.LOCK` 是 BUNDLE 加自由文本 `.LOCK`（16.4）。

3.6 名字

- **不加引号的名字**是一个或多个 ASCII 字母、数字、下划线，按最长匹配取。
- **宿主名字**（`#`、`&` 之后）和端子名按宿主方言的规则（原则 2.1）：哪些名字合法，区不区分大小写，能不能以数字开头，如 `&0`。SPICE 不区分大小写，Spectre 区分。含别的字符的名字加引号，如 `#"M1<0>"`、`&"vdd!"`、`#XU1:"in+"`；引号里的写法与字符串相同（3.7）。名字后面可以紧跟下标：`#M1<0>` 就是 `#"M1<0>"`，`&D<7:0>` 是一组网络（6.11）。引号里的 `<` 或 `[` 是名字的一部分，不解析。
- **RNL 名字**（`%`、`$`、`?` 之后的名字，角色名，普通定义的参数名）：
  - 区分大小写：`$DIFFERENTIAL_PAIR` 和 `$differential_pair` 是两个名字。
  - `%`、`$`、`?` 后面的名字可以以数字开头，如 `%1st`。在没有前缀的位置（`(role)` 和 `parameter=`），不加引号的名字由 ASCII 字母、数字、下划线和任何非 ASCII 字符组成，不以 ASCII 数字开头，免得像数：`#M1(正输入)`、`@AS(间距=3) $X`。这两处都在括号里，括号里没有自由文本（3.3），所以不会把别的词吞进名字。
  - 可以加引号，含义相同：`$"a"` 就是 `$a`；引号只是转义。含别的字符的名字和非 ASCII 的名字写在引号里，如 `%"差分对"`、`%"input pair"`、`?"正输入"`、`%dp::"正输入"`。在角色名和参数名的位置上，非 ASCII 的名字可以不加引号（上一条），但空格或别的 ASCII 标点仍要加引号，如 `#M1("input pair")`。在角色名和参数名的位置上，加引号的名字永远是名字本身，不当代称。这一点不适用于 `$` 名字和 `::` 后面的名字，它们的非 ASCII 代称本来就写在引号里（16.5）。
  - 按写出的字符逐个比较，不做 Unicode 归一化。空名字是错误。

  - 不收下标（6.11）：后面紧跟 `<` 或 `[` 是错误，如 `%p<1, 2>`。点写成 `(1, 2)`，与名字之间隔空白。
- **代称**（16.5）在头段记录里声明，或来自代称包。关键字和标准词的代称可以含非 ASCII 的字母；它们只在 `@` 后面和括号里不加引号写。`$` 名字的代称照 `$` 名字写，非 ASCII 的加引号，如 `$"接地补位集"`。`#`、`&`、`%`、`$`、`?` 和 `::` 后面不加引号的名字如上所述只含 ASCII 字符，所以 `#M1和#M2` 是 `#M1`、自由文本 `和` 和 `#M2`。
- **关键字**只认大写，词间用 `-` 连接，如 `@AS`、`@SET-SYMBOL`；大写也便于在自由文本里认出它们。`@at` 是不认识的关键字，除非头段记录把它声明为代称（16.5）。
- **标准词**区分大小写，按标准的拼写写：参数名、记录头的字段名、子对象名、选择条件的名字和枚举值（如 `"x"`、`"low"`、`"MY"`）。指宿主东西的值（选择条件里的名字、模型名、网络名）按宿主的规则。别的写法可以通过代称接受（16.5）。
- **`true` 和 `false`** 保留，只认小写。它们不能作角色名或参数名。

3.7 字面量

- **数**是十进制。可以带正负号和前导零，可以省略小数点前的零或小数点后的数字，可以带指数：`0`、`+1`、`01`、`.5`、`1.`、`-2.5E-4`。`01` 就是 1，不是八进制。没有单位和工程后缀（`2u`、`10k` 是错误），没有十六进制数，没有 `NaN` 和 `Infinity`。位数不限。实现装不下一个值时，这条语句在这个实现里失效，报的错误说明这是实现的限制；不悄悄改值。坐标按容差比较（11.7）。
- **字符串**写在双引号里，不能跨行。引号之间的字符照原样取；只有 `"` 本身写成 `""`，如 `"say ""hi"""`。没有反斜杠转义：`\` 是普通字符，没有 `\n` 或 `\uXXXX`；非 ASCII 字符直接写。字符串里写不了换行；文字里的换行用文字记号 `\\` 写（13.6）。TEXT 有自己的反斜杠命令（13.6）；那是 TEXT 怎样读它的文字，不是字符串的转义。记录头里的字符串也这样写；这属于外层写法，不改（16.1）。
- **数组**写作 `[value, value, …]`。元素是字面量，数组可以嵌套。
- **真假值**是 `true` 和 `false`。
- 没有别的字面量。枚举写成字符串，如 `axis="x"`；没有 `null`、映射和构造值。
- 没有表达式：运算符是错误。

3.8 点

- 点写作 `(x, y)`，x、y 是数或 `_`（在普通定义的体里，也可以是参数空位，6.10）。`_` 让这一轴自由，由约束或求解器决定。
- 写一个点就是创建一个点对象（5.1）。
- 点没有算术；相对位置写成约束（第 14 章）。
- 点的 `(` 前面有空白或注释，或者紧跟在 `{`、`;`、`=>`、`=` 之后。紧贴关键字或对象的括号放它的参数或角色（3.5）。名字后面紧跟的 `<` 无论如何都是下标（6.11）。

4 文法

4.1 文法

记录的文法用 EBNF 给出。每条产生式写作 `name = expression ;`；`|` 分开各个选项，`[ … ]` 是可选的，`{ … }` 重复零次或多次，`( … )` 分组，双引号里的文字照原样写，`(* … *)` 是注释。

```
record         = "@RNL" [ parameters ] set ;          (* the bracket touches "@RNL" *)
set            = "{" [ member ] { ";" [ member ] } "}" ;
member         = [ front-naming ] segment { naming } { "=>" segment { naming } } ;
front-naming   = ( "%" rnl-name | "$" rnl-name ) "=" ;
naming         = "=>" ( "%" rnl-name | "$" rnl-name ) ;
segment        = argument { argument } [ call { argument } ]
               | call { argument } ;
call           = keyword [ parameters ] ;          (* the bracket touches the keyword *)
keyword        = "@" keyword-name ;                (* upper case, or a declared alias *)
argument       = object [ "(" role-label ")" ] ;   (* the bracket touches the object *)
object         = reference | placeholder | point | set ;
reference      = "#" [ host-name [ subscript ] ] path
               | "&" host-name [ subscript ] path
               | "%" rnl-name path
               | "%#" integer path                (* a number, in a record of the solver, 5.9 *)
               | "$" rnl-name ;
placeholder    = "?" rnl-name path
               | "?" ;                             (* alone, in a symbol body *)
path           = { ":" terminal-name [ subscript ] | "::" rnl-name } ;
subscript      = "<" subscript-item { "," subscript-item } ">"
               | "[" subscript-item { "," subscript-item } "]" ;
subscript-item = integer [ ":" integer [ ":" integer ] ] ;   (* unsigned, 6.11 *)
host-name      = name | string ;
rnl-name       = name | string ;
terminal-name  = name | string ;
role-label     = bracket-name | string ;
point          = "(" coordinate "," coordinate ")" ;   (* after white space, a comment, "{", ";", "=>" or
"=", 3.8 *)
coordinate     = number | "_" | parameter-placeholder ;
parameters     = "(" [ value { "," value } [ "," named-value { "," named-value } ]
                     | named-value { "," named-value } ] ")" ;
named-value    = parameter-name "=" value ;
parameter-name = bracket-name | string ;
value          = number | string | "true" | "false"
               | "[" [ value { "," value } ] "]"
               | parameter-placeholder ;
parameter-placeholder = "?" rnl-name ;
```

- `name`、`bracket-name`、`keyword-name`、`string`、`number` 和 `integer` 的写法见第 3 章。`name` 是不加引号的名字（3.6）。`bracket-name` 是角色括号里或参数的 `=` 前面不加引号的名字：由 ASCII 字母、数字、`_` 和任何非 ASCII 字符组成，不以 ASCII 数字开头（3.6）。`keyword-name` 由字母、数字和 `_` 组成，词间用 `-` 连接；`-` 后面跟字母时才属于它（3.3）。标准关键字是大写的（3.6）；代称由任何文字的字母、数字和 `_` 组成，不含 `-`（16.5）。`string` 和 `number` 见 3.7；`integer` 是不带符号的十进制整数。
- `rnl-name`、`host-name`、`terminal-name`、`role-label` 和 `parameter-name` 也可以加引号，含义相同，只是在角色名和参数名的位置上，加引号的名字永远不是代称（3.6）。关键字、参数名、角色括号里的槽位名、子对象名、`true` 和 `false` 可以写成声明过的代称；不加引号的非 ASCII 代称只用在 `@` 后面和括号里，`::` 后面的代称加引号（16.5）。
- 下标直接跟在宿主名字或端子名后面，中间不能有空白；下标里可以写空白和注释（3.5、6.11）。哪些名字收下标，见 4.2 第 10 条。
- 参数括号紧贴关键字，角色括号紧贴对象；隔了空白或注释的 `(` 开始一个点（3.5）。
- `=>` 是管道还是命名，看它后面是什么（4.2 第 3 条）。

- 带名字的空位只写在普通定义的体里（6.10）。在对象的位置上，它是一个角色，可以带路径；在收字面量的位置上（参数的值、数组的元素、点的坐标），它是定义的一个参数，不带路径（6.10、9.4）。单独的 `?` 只写在符号体里（13.3）。
- 自由文本和注释可以写在括号外任意两个记号之间，也可以写在成员开头（3.3、3.4）；空白和注释也可以写在括号里。文法里不写出它们。
- 空成员（如 `;;`）忽略；最后一个成员之后的 `;` 可以省略。

4.2 文法之外的约束

1. 段不能为空。
2. 一段至多一个关键字。`@RNL` 只出现在记录开头。
3. 一个 `=>`，后面是一个 `%` 或 `$` 名字，再往后是 `=>`、`;`、`}` 或成员结束时，它是命名。别的 `=>` 都开始下一段；那一段恰好有一个关键字，写在那里的名字是那一段的对象。管道的第一段可以没有关键字，这时它恰好有一个对象。所以要先给结果起名、再接着管道，写 `=> %g0 => @PLACE(1, 2)`。在 `=> %g0 @PLACE(1, 2)` 里，`%g0` 是 PLACE 的对象；它通常没有绑定，这条语句因为找不到 `%g0` 而失效。诊断可以问是不是漏了 `=>`。
4. 没有关键字的段里，每个实参各是一条语句：`{#M1 #M2}` 里的 `#M1 #M2` 是两条语句。这样的段带命名时（`%p =` 或 `=> %p`），只能有一个对象：写 `%p = {#M1 #M2}`，不写 `%p = #M1 #M2`。
5. 角色只标在有关键字的段的实参上。管道后面的段不标角色。
6. 一段里的实参要么全部标角色，要么全部不标。同一个角色只标一次。
7. 收定义的关键字，定义的槽位由关键字定：ATTACH 的定义是写出的第一个对象；AS、SET-SYMBOL、SET-STYLE、SELECT-USAGE 和 SELECT-ANCESTOR-USAGE 的定义是最后一个对象。SELECT-USAGE 可以省略它的定义：最后一个对象是定义时，才填定义的槽位（10.1）。定义不标角色，也不计入第 6 条（5.6）。
8. 命名有两种写法，可以并用。`name =` 只能是成员的第一个记号（前面可以有自由文本），一个成员至多一个。`=> name` 跟在段后面，在成员末尾，或在下一段的 `=>` 之前，一个成员可以有几个。第 3 条把它与管道分开。
9. 按位置写的参数在前，写名字的在后：写了一个 `name=value` 以后，后面的参数都写名字。只有签名里列了位置次序的参数可以按位置写，按那个次序写。同一个参数不写两次。
10. 下标只跟在 `#` 或 `&` 后面的宿主名字之后，或 `#` 引用里的端子名、端口名之后。RNL 名字、用例的端子名（角色名）和子对象名不收下标（6.11）。
11. 一个引用里至多一处下标展开成一组名字（6.11）。

4.3 语序

- 关键字连同参数可以放在对象之间的任何位置；对象之间的先后有意义。`#M1 #M2 @DISTANCE(…)`、`#M1 @DISTANCE(…) #M2`、`@DISTANCE(…) #M1 #M2` 相同。
- 每个关键字有一种标准写法，工具应只写这一种。标准写法记在标准词表里（5.3）：关键字放在哪里，哪些参数按位置写、哪些写名字，标不标角色。
- 默认的标准写法是：对象在前，然后是关键字和参数，定义写在关键字后面。ATTACH 先写定义，再写关键字，再写对象：`$GROUND @ATTACH &vss`。签名里列了位置次序的参数按位置写，写在前面；其余的写名字，按标准词表的次序：`@PLACE(10, 20)`、`@ORIENT("R90")`、`@DISTANCE("bottom-to-top", min=4)`、`@TEXT("W/L", key="wl")`。PLACE 只在两轴都写时按位置写 x 和 y；只写一轴时写名字，如 `@PLACE(x=10)`。去掉也不改变含义的参数不写，如 WIRE 的 `mode="auto"`。内置关键字不标角色；普通定义有两个或更多角色时，AS 和 ATTACH 标出全部角色。
- 版面：括号里每个逗号后空一格；参数的 `=` 两边不空；命名的 `=` 和 `=>` 两边各空一格；关键字紧贴它的括号；`{` 和关键字之间空一格，如 `{ @…`（2.3）；数写成能读回同一个值的最短的十进制。
- **命名的标准写法。** 没有管道的成员在开头命名，`%x = statement`；有管道的成员在末尾命名，`statement => … => %x`，名字跟在它所命名的那一段后面。这样开头的名字总是给整个成员起名，读的人不用猜它指管道的哪一段。文法两种写法都收。
- "只写这一种"指工具新写的语句。工具改写已有的记录时，不应重排用户写的语句（16.4）；只在用户要求格式化时，才把它们改成标准写法。

- 标准例句见 4.5。

4.4 错例

下面每一条都是错误；注释说明原因。

错例

```
#M1 @PLACE (10, 20)    /* a bracket after white space is a point, and PLACE has no parameters (3.5) */
```

错例

```
#M1 (in_p) #M2 (in_n) @AS $DIFFERENTIAL_PAIR    /* role brackets touch the object; a separate bracket is a point (3.5) */
```

错例

```
#M2 @PLACE(62, 40) (MY)    /* the second bracket is a point, and its content is not a point */
```

错例

```
&vss @ATTACH $GROUND    /* the definition of ATTACH is written first: $GROUND @ATTACH &vss */
```

错例

```
R1 -> R2 {#R1 #R2} @ORDER("x")    /* ">" cannot be free text (3.3) */
```

错例

```
#M1 @PLACE(1, 2), #M2 @PLACE(3, 4)    /* a comma does not separate statements: two keywords in one segment */
```

错例

```
#M1 #M2 @SET-SYMBOL $NMOS    /* three objects, and SET-SYMBOL has two slots (5.2.1); write {#M1 #M2} @SET-SYMBOL $NMOS */
```

错例

```
#M1 => #M2    /* the segment after a pipe has no keyword */
```

错例

```
#M1 #M2 #M3 @DISTANCE("x", min=1)    /* three objects for two slots */
```

错例

```
#M1 @PLACE(x=10, 20)    /* a parameter given by position after one given by name (4.2 item 9) */
```

错例

```
{#M1 #M2} @ORDER(axis="x", direction="increasing")
/* ORDER has no axis: the axis and the direction are one parameter, direction, as in @ORDER("left-to-right") (14.3) */
```

错例

```
#M1 @SET-SYMBOL($NMOS)    /* brackets hold only literals */
```

错例

```
{#M1 #M2} @ALIGN(axis=y)    /* enumerations are written as strings */
```

错例

```
%ax = (_, _);
#M1(a) #M2 %ax @AS $MIRRORED    /* roles are marked on all objects or on none */
```

错例

```
$NMOS(symbol) #M1(target) @SET-SYMBOL    /* the definition is written last and is not marked with a role */
```

错例

```
#R1(source) #R2(chain) #R3(chain) #R4(load) @AS $CHAIN    /* the same role is marked twice */
```

错例

```
(0, 0) @PIN(terminal=1)    /* a pin cannot be written in the scene (13.4) */
```

错例

```
#M1 @CURRENT("in")    /* CURRENT takes only terminals (15.2) */
```

错例

```
&D<1:0>[1:0]    /* a name has at most one subscript (6.11) */
```

错例

```
#XI0<1:0>:D<1:0>    /* one reference has at most one expansion (6.11) */
```

错例

```
&D<_, 1>    /* a "<" directly after a name starts a subscript, and its content is not a subscript (3.5) */
```

错例

```
%p<1, 2>    /* RNL names take no subscript; a point is written (1, 2) (6.11) */
```

4.5 标准例句

标准例句是用标准写法写、在记号之间加上自由文本、读起来像一句话的语句。标准例句用于写作参考，也用于格式化工具给语句加词或去词。自由文本不影响含义（3.3）：例句只是建议，用户和语言模型可以照写，也可以不写，或写别的词。

- **读作事实**。语句说的是图上成立的事实，不是要做的操作。对象在前当主语，关键字当谓语或名词，参数是它的值。只加连接的词，如英文的 `is`、`are`、`has`、`is at`、`there is`，中文的 `是`、`处于`、`位于`、`有`、`的`、`按……排列`。少数语句也可以读作操作，如 `#M1 @SET-SYMBOL to $NMOS`。
- **词由语句算出**。工具生成的词只取决于关键字、参数、对象的个数和种类、语句写在哪一层，不取决于电路的意思。例如写 `is` 还是 `are` 按对象的个数；只给一轴时写 `the x of`（`x 坐标`）；选择的候选的说法按范围和要选的种类（附录 B）。这是生成例句的办法，不限制用户。
- **不写像有含义、其实语句没有这个含义的词**。不写 `must`、`required`、`only`（`必须`）这类词；方向词只在参数有这个方向时写。读的人会相信句子里的词，建议写作参考提醒他们注意这一点。
- **补一句**。读的人未必懂朝向码和轴。可以在关键字后面用 `, i.e., …`（`, 即……`）说出代码本身的意思，如 `#M2 @ORIENT("MX"), i.e., flipped upside down`、`#M2 @朝向("MX"), 即上下镜像`。
- **ORIENT-RELATION 和 DISTANCE** 有两个槽位，读作 `from A to B`（`A 到 B`）；中文里 ORIENT-RELATION 接 `的`，DISTANCE 接 `有`。方向只是一个轴（`"x"` 或 `"y"`）的 DISTANCE 读作 `between A and B`（`A 和 B 之间有`）。
- **选择**一律读作在候选里选：英文是 `among <candidates>, @SELECT-…`，中文是 `从<候选>中 @选出…`。范围是 `#` 时，候选读作 `the <kind> at each level in # and below`（本层 # 及以下每一层的<种类>），说出规则在每一层各求一次值（10.4）。种类算不出时，读作 `for X,`（对于 X,）。

- **AS 与 ATTACH。** 对象合起来构成这个东西时用 AS：`#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR`。这个东西添在对象上时用 ATTACH：`$GROUND is @ATTACH to &vss`（9.4）。
- **管道。**后一段指前一段的结果。英文用 `it`（前一段是 AS、ATTACH 或单个对象时）或 `they`（前一段是选择时）；中文一律用 `其`，与别的对象并列时写 `其与`。管道里的名字前面写 `named`（`记作`），如 `… => named %pout`（`… => 记作 %pout`）；它照样是命名（4.2 第 3 条）。
- **命名。** 单条语句的名字写在开头（4.3）。给 AS 的语句起名时，英文去掉 `are`：`%dp = #M1(in_p) and #M2(in_n) treated @AS $DIFFERENTIAL_PAIR`。
- **集合的名字。** 指集合的名字建议以 `_SET`（`$` 名字）或 `_set`（`%` 名字）结尾，中文以 `集` 结尾，如 `$GROUND_FILL_SET`、`%bias_set`、`%"接地补位集"`；标准包照此命名。
- **不写 `each`。**不写 `each`（`每一个`）这类词。一个集合是整个套用还是逐个成员套用，要看定义的角色表，标准例句不说是哪一种。
- **中文。**中文例句用代称包 `rnl-zh` 的代称（16.5）。中文逗号后面接 `@` 时仍要空一格（2.3）。

各关键字的标准例句列在附录 B。

5 对象与语句

5.1 对象

- 每条带关键字的语句产生一个新的 RNL 对象，由句柄和事实组成。`{…}` 也产生一个新对象，即集合（5.2）。
- 引用（`#M1`、`&vss`、`%x`、`$X`，以及端子、子对象）不产生对象，只取已有的句柄。单独的 `#M1` 也是语句，得到 M1，不带事实；单独的 `#` 得到本层，不带事实。
- 句柄是被描述的那个东西：宿主的元件、网络、端子，或本层；或由 RNL 创建的点、导线、集合、用例、约束等。句柄不会是另一个 RNL 对象。
- 一条语句的对象，要么用新的句柄（语句创建，如 WIRE、图元、`{…}`、选择、AS、ATTACH、约束），要么沿用输入的句柄、再加上新的事实（语句修改，如 PLACE、ORIENT、SET-SYMBOL、SET-STYLE、GROUP、CURRENT；ADD-TO、REMOVE-FROM 沿用目标集合的句柄）。是哪一种写在签名里（5.3）；WIRE 收到网络时创建，收到导线时修改（13.1）。
- 两条内容相同的语句是两个对象，如两句 `#M1 @PLACE(1, 2)`。它们的事实不矛盾，一起生效。
- 事实只在产生它的语句处提交一次（5.4）。本层生效时，同一个句柄的全部事实合在一起，矛盾按 1.5 第 3 条处理。
- 属性就是事实：位置、朝向、符号、样式都是对象上的事实，没有可以改写的属性字段。本层生效时，把同一个句柄的事实合起来算出最终的值：PLACE 的 x、y 按几何事实（5.8、8.5），符号、样式按外观队列（8.3）。
- 句柄的种类由它的来历决定：宿主对象由宿主给出，RNL 对象由创建它的语句决定。签名规定每个对象槽位收哪些种类（附录 A），收到种类不对的对象时，这条描述失效。能在读入时查的读入时查；要到后面才知道的（用例有没有符号、定义的角色）在本层生效时查（7.1）。

对象的种类：

种类	怎样产生	画不画
本层	宿主；`#`	不画
元件	宿主；`#M1`	用它的符号画
端子	宿主；`#M1:d`、`#:inp`。用例的端子是它的定义的角色（6.4）	作为符号的引脚画
网络	宿主；`&vss`	不画；导线、标签是另外的对象
点	`(x, y)`	不画；汇合圆点是套在点上的 `$JUNCTION`（13.2）
导线	收到网络的 WIRE；网络槽位收到集合的导线是总线。WIRE 收到导线时改写它（13.1）	画
标签	附着在对象上的 TEXT；写了引用名时，是那个对象的子对象 `object::key`（13.3）	画
叠画	BUNDLE（11.8）	代替各成员画一次
标注	场景里的图元（LINE、RECTANGLE、CIRCLE、ELLIPSE、ARC、BEZIER），或附着在点上的 TEXT（13.4）	画
用例	AS、ATTACH（9.4）	有符号才画，这时它是虚拟元件（第 12 章）
约束	ALIGN、ORDER、DISTRIBUTE、DISTANCE、ORIENT-RELATION（第 14 章）	不画
集合	`{…}`、选择、UNION、逐个生成的结果（5.2、第 10 章）	本身不画，成员各自画
定义	DEFINE-SYMBOL、DEFINE-STYLE、DEFINE（第 9 章）	不画，被用到时才起作用

这些种类里，元件、端子、网络、点、导线、标签、叠画、标注、用例有种类名（5.3），`text` 也是种类名，指任何 TEXT，不论标签还是标注；其余几种在签名里怎样写见附录 A。

另有子对象（6.5），句柄是父对象和名字；以及失效对象（第 7 章）。

5.2 集合

- `{…}` 产生一个集合对象，成员有序。集合可以命名（`%pair_set = {#M1 #M2}`），有名字的集合可以被别的语句增删成员（5.2.2）。

- 集合保留重复的成员和写出的次序，如 `{#M1 #M2 #M3 #M1}`。是不是同一个成员按句柄判断：分别写出的两个 `{…}` 是两个句柄，内容相同也不是同一个成员。
- **引用打散一层**：集合里写的引用如果指向一个集合，就在原处放进那个集合的成员，所以 `{%a_set %b_set}` 是 a_set 的成员接着 b_set 的成员。结果是集合的语句（如选择）同样打散；下标展开的一组引用（如 `&D<7:0>`，6.11）也一样。要去重，写 `%a_set %b_set @UNION`（5.2.2）。
- **直接写的 `{…}` 保留**：集合里直接写的 `{…}` 作为一个成员保留，所以 `{{%a_set} {%b_set}}` 是由两个集合组成的集合。
- **散写的对象也打散**：只有一个槽位的关键字收到几个不加花括号写的对象时，把它们合在一起，与写在 `{…}` 里完全相同（5.2.1 第 3 条），所以其中的集合打散。设 `%bias_set = {#R1 #R2 #R3}`，`%lbl` 是一段文字：
  - `%bias_set %lbl @ALIGN("x")` 就是 `{%bias_set %lbl} @ALIGN("x")`：R1、R2、R3 和文字逐个 x 相同；
  - `{%bias_set} %lbl @ALIGN("x")` 把集合保留为一个成员：文字与 R1–R3 的中心 x 相同；
  - 集合是唯一的对象时，用的就是它本身：`%bias_set @ALIGN("x")` 对齐 R1、R2、R3。

  在 ORDER、ALIGN、DISTRIBUTE 的成员里，只有写成 `{%x}` 的集合才作为一个成员、按它的中心算（14.1）。

  注：打散在约束里最容易读错。`{%stage1_set %stage2_set} @ORDER("x")` 把两组的成员逐个排序，常与组内的 ALIGN 矛盾。要在两组之间排序，写 `{{%stage1_set} {%stage2_set}} @ORDER("x")`，或用 DISTANCE 的两个槽位（14.5）。保留打散，是因为它是把集合拼起来的唯一写法。散写的对象完全按写在 `{…}` 里读，这样多加花括号不会让含义颠倒：要是散写的对象不打散，`%bias_set @ALIGN("x")` 会把集合算作一个成员，什么也不对齐；`{%bias_set} @ALIGN("x")` 会打散它，对齐 R1–R3；`{{%bias_set}} @ALIGN("x")` 又把它整个保留。
- 集合的内容在用到它的阶段求出，与选择一样：初始内容和各条增删语句里的选择，都按这个阶段开始之前的状态求值，增删按写的次序（它们的位置，8.5）处理。所以写在前面的使用也看到全部增删（1.5 第 1 条）；同一个集合在不同阶段可以不同：生成对象的规则看到的不含生成的对象，样式规则看到的含（8.2）。

  注：内容不在某一步一次算完，因为那样给选择起个名字就会改变结果：`%g_set = # @SELECT-USAGE $GROUND; %g_set @SET-STYLE $grey` 会选不到生成的地，而不起名的同一个选择选得到；名字只是引用对象的另一种写法（1.5 第 5 条）。

5.2.1 对象与对象槽位

一条语句写出的对象按以下规则对到签名的对象槽位。

1. **标了角色**：标了角色的对象整个填进它的角色，不拆开。一条语句里要么全标，要么全不标。
2. **先拆一层**：没标角色时，如果整条语句只有一个对象，而且它是集合，先把它拆开一层。定义、管道送来的结果都算对象。所以 `{#M1 #M2} @DISTANCE(…)` 等于 `#M1 #M2 @DISTANCE(…)`；`{#M1 #M2} @SET-SYMBOL $X` 是两个对象（集合和定义），不拆。
3. **按签名配位**：
   - 有多个槽位的关键字（定义也占一个槽位）：对象个数应与槽位个数相等，按顺序一一对应；不等时这条语句失效。AS、ATTACH 角色表末尾的集合角色可以不给对象（9.4）。
   - 只有一个槽位的关键字（如 PLACE、ORIENT、ALIGN、ORDER、DISTRIBUTE、GROUP、除 SELECT-USAGE、SELECT-ANCESTOR-USAGE 以外的各选择关键字（这两个还收定义，5.6），以及定义关键字，它们的槽位是定义体）：一个对象直接放入；多个对象合成一个集合放入，引用的集合在其中打散一层，直接写的每个 `{…}` 保留为成员。这些对象正是由唯一的集合按第 2 条拆开的时，放入的就是那个集合本身。
   - 唯一的对象是空集合时：
     - 要单个对象的槽位逐个做零次：什么也不发生，也不失效。名字照常绑定到这个空集合，用这个名字的语句什么都不做。
     - 要单个对象的槽位收到直接写出的空集合时给警告，如 `{} @PLACE(1, 2)`。由选择、规则得到的空集合不警告：规则在写它的层和下面每一层各求一次，某一层没有这类对象是常态。用户自己写的选择在它求值的每一层都没选到东西时，实现可以给一次警告，多半是条件写错了。
     - 要集合的槽位收空集合，如空的定义体 `@DEFINE(roles=["a"]) {}`，或没有成员的约束（自然成立）。例外是 BUNDLE 和 WIRE 的网络槽位：它们要用这个集合造一个新对象，空的造不出来，这条语句失效（11.8、13.1）。

    - 一个对象也没写的语句，如单独的 `@PLACE(1, 2)`，与签名不符，失效。

4. **对象槽位收到集合**：
    - 要单个对象的槽位对每个成员各做一次。每个成员是一条描述，结果是这些结果组成的集合。成员本身还是集合时，再拆开逐个做。唯一的例外是 PRIORITY：收到 GROUP 标过的集合时，作用于这个分组本身（11.5、14.8）。
    - 要集合的槽位整个收下。
    - 收对象或集合的槽位（如 DISTANCE 的两个槽位）把集合整个收下，集合在那里按范围算（14.5）。
    - 一条语句里至多一个要单个对象的槽位可以收到多成员的集合。超过一个时这条语句失效：按顺序配对和两两组合都说得通，含义不唯一。

写法	结果
`{#M1 #M2} @SET-SYMBOL $X`	两个对象，不拆。元件槽位收到集合：M1、M2 各得到这个符号
`#M1 #M2 @SET-SYMBOL $X`	三个对象对两个槽位：这条语句失效
`{#M1 #M2} @DISTANCE(…)`、`%pair_set @DISTANCE(…)`	唯一的对象是集合，拆开：从 M1 到 M2
`%a_set %b_set @DISTANCE(…)`	两个对象，各是一个集合，整个收下：两组之间（14.5）
`{#M1 #M2} @ALIGN("y")`、`#M1 #M2 @ALIGN("y")`	ALIGN 只有一个槽位，收到 {M1 M2}
`%bias_set %lbl @ALIGN("x")`，其中 `%bias_set = {#R1 #R2 #R3}`	散写的对象合成 `{%bias_set %lbl}`，其中 bias_set 打散：R1、R2、R3 和文字逐个 x 相同
`{%bias_set} %lbl @ALIGN("x")`	直接写的 `{…}` 保留：文字与 R1–R3 的中心 x 相同
`%pair_set @PLACE(10, 20)`	槽位收到的是 pair_set 本身，逐个成员放：两个成员都在 (10, 20)。要放集合的中心，写 `%pair_set::position @PLACE(10, 20)`
`%a_set %b_set @PRIORITY("disabled")`，两个集合都带 GROUP	松散的对象与写在 `{…}` 里一样打散：PRIORITY 收到的是成员，在它们上面失效（种类不对）。要关两个分组，每个分组写一条 PRIORITY（11.5、14.8）
`{#M1 #M2} @SET-SYMBOL $X => @ALIGN("y")`	管道的结果是唯一的对象，放入 ALIGN：对齐 M1、M2
`{#M1 #M2} @SET-SYMBOL $X => #M3 @DISTANCE(…)`	两个对象，不拆。槽位 `from` 把集合整个收下，按范围算：从 M1、M2 合起来到 M3（14.5）
`# @SELECT-NET(…) => $PULL_DOWN @ATTACH &vss`	三个对象对三个槽位（两个角色和定义）。第一个角色是单个角色，所以对每个选中的网络各套用一次定义，每次第二个角色都是 vss
`# @SELECT-INSTANCE(type="nmos") => @ORIENT("MY")`	逐个镜像；没有 NMOS 时做零次，什么也不发生
`# @SELECT-INSTANCE(name="Q*") => @SET-SYMBOL $NPN`	两个对象，不拆；没有 BJT 时做零次，什么也不发生
`{{#M1 #M2 #M3}} @ALIGN("y")`	拆开一层后剩一个集合对象：对齐 M1、M2、M3
`#R1 {#R2 #R3} #R4 @AS $CHAIN`	四个对象对四个槽位（三个角色和定义）；集合角色 `chain` 收到 {R2 R3}（9.4）
`&D<7:0> @WIRE {…}`	两个对象，不拆。网络槽位整个收下这组网络：一根总线（13.1）
`#XI0<3:0> @ALIGN("y")`	唯一的对象是下标展开的一组，拆开：对齐四个元件（6.11）

- 成员不够的约束自然成立，不失效，如只有一个成员的 ALIGN（第 14 章）。
- 要把一个集合当作整体：PLACE 用它的 `::position`（6.5）；作 ORDER、ALIGN、DISTRIBUTE 的成员时写成 `{%x}`，按它的中心算；作 DISTANCE 的对象时按它的范围算（14.1、14.5）。
- 定义的角色分单个角色和集合角色（9.4）。角色表里写成 `"{name}"` 的是集合角色，整个收下集合；其余是单个角色，收到集合时对每个成员各套用一次定义。

5.2.2 增删成员与去重

- `X @ADD-TO S` 在集合 S 的末尾追加 X（X 是集合时追加它的各成员）；`X @REMOVE-FROM S` 去掉 S 里这些成员的所有出现。目标写在最后，所以选择可以用管道送入：`# @SELECT-NET(name="vss") => @ADD-TO %gnd_set`。
- 目标应是集合，不自动包装成集合。`%tail = #M5` 之后，`#M3 @ADD-TO %tail` 失效，因为 M5 不是集合。

- 几条增删按位置的次序处理（1.5 第 2 条）。集合在某层求值时只含这一层的成员（10.3）。
- `%a_set %b_set @UNION` 产生一个去重的新集合，次序取第一次出现处；是不是同一个成员按句柄判断。
- 产生新集合而不改动原来的：拼接写 `{%a_set %b_set}`；并集写 `%a_set %b_set @UNION`；差集写 `%c_set = {%a_set}; %b_set @REMOVE-FROM %c_set`。
- 一个集合依赖于产生它的语句里写到的集合（花括号里、UNION 里，或作为选择的范围），以及用 ADD-TO、REMOVE-FROM 加进它或从它去掉的集合。依赖成环就是错误，不论有没有删除：按位置的次序（8.5）逐条看，依赖闭合这个环的那条语句失效，诊断列出整个环；其余照常生效。所以 `%a_set @ADD-TO %b_set; %b_set @ADD-TO %a_set` 在第二条失效；`%b_set = {%a_set}; %b_set @REMOVE-FROM %a_set` 这样删除没有确定结果的，在删除那条失效。

5.3 语句与签名

- 语句由至多一个关键字、若干对象和一组参数组成。
- 每个关键字有签名，声明：
  - 对象的个数，每个对象槽位的名字和它收的种类。AS、ATTACH 的个数是定义的角色表的长度，再加定义本身（AS 写在最后，ATTACH 写在最前），种类是角色声明的种类（9.4）；
  - 参数：名字、类型、是否必需、可选值，以及按位置写时的顺序；
  - 输出：新对象、输入的句柄、子对象、定义，或没有句柄；WIRE 收到网络时给出新导线，收到导线时给出这条导线（13.1）；
  - 每个输入的消费方式（5.4），以及对失效严格还是容错（7.3）；
  - 关键字能写在哪里（见下面的标准词表）。
- 对象个数固定，个数不定的用集合。没有可选槽位，例外是：AS、ATTACH 没填的集合角色是空集合（9.4）；SELECT-USAGE 的定义可以省略（10.1）；USE-PACKAGE 的体可以省略（16.2）。定义也占一个槽位。
- 对象按角色或按顺序对到槽位，全标或全不标；同一个角色只能标一次。管道送来的对象算第一个对象；AS、ATTACH 按角色配它，它不占定义的槽位（5.5）。对象怎样对到槽位，见 5.2.1。
- 不论什么关键字，语句里写出的对象都能标角色。角色名就是签名里的槽位名：`#M5:3(to) #M1:3(from) @DISTANCE("x", min=1)` 与按顺序写的 `#M1:3 #M5:3 @DISTANCE("x", min=1)` 相同。角色名不是这个关键字的槽位名时，这条语句失效，不会被忽略（原则 2.3、2.8）。只有一个槽位的关键字，对象也能标角色，但标了不增加什么。
- AS、ATTACH 的参数是定义的参数表里的参数（9.4），另有一个标准参数 `priority`（14.8）；定义的参数不能叫 `priority`。
- **标准词表**：每个语言版本发布一份机器可读的标准词表，是标准的一部分。各关键字收什么、有哪些参数只写在表里，正文写它们的含义。附录 A 由表生成，两者矛盾算勘误。解释器的检查、文档、编辑器的补全、一致性测试的骨架、给 LLM 的写作参考都从表生成。0.3 版的表发布为 `rnl-words-0.3.json`。
  - 表里收全部标准词：关键字及其签名、槽位名、参数名和取值、记录头的字段、种类名、子对象名、`${…}` 读的属性名、样式属性，以及 `true`、`false`。代称（16.5）是给表里的词声明的，版本检查和"最接近的名字"都查这张表。
  - 表是 JSON 文件，JSON Schema `rnl-words.schema.json` 说明表本身的结构。读写它的是工具和标准的维护者，不是描述电路的人。每个词带 `since`，即加它的版本；预留的词标出来（0.3 版没有预留的关键字）。文件声明的版本里还没有的词，按不认识的词处理，诊断说明要哪个版本（16.1）。
  - 表里为每个关键字写：
    - 每个槽位：名字、收哪些种类、要单个对象还是要集合（5.2.1）、消费方式（5.4，严格或容错由它决定，7.3）；定义的槽位；AS、ATTACH 的槽位按定义的角色表；
    - 每个参数：名字；类型（数、字符串、真假值，或数组及元素的类型；取值固定的列出取值）；必需与否；默认值；能按位置写的参数的次序（其余的只能写名字）；以及组合条件，如 PLACE 的 `x`、`y` 至少写一个，RECTANGLE 的中心写法和对角写法只能写一种；
    - 输出：新对象及其种类、沿用输入、沿用目标（ADD-TO）、定义、子对象（写了 `key=` 的 TEXT），或没有；WIRE 另记它收到哪种对象时改写这个对象（13.1）；

  - 能写在哪里：层里；只在符号体里（PIN）；只在头段记录里（如 VERSION、ALIAS、DEVICE）；只在包文件的头段记录里（PACKAGE）；
  - 事实怎样覆盖（5.8）：按键覆盖，及其键；按几何事实（8.5）；按外观队列（8.3）；或累加；
  - 标准写法和标准例句（4.3、4.5、附录 B）；
  - 一句说明和正文的节号。
- 不进表的：含义（关键字产生什么事实、在哪个阶段、怎样冲突）写在正文。包里普通定义的签名就是它的角色表和参数表（9.4）。

```
{
 "keyword": "PLACE",
 "since": "0.2",
 "group": "Placement and appearance",
 "summary": "Puts the origin of an object at a point; one axis is enough.",
 "section": "11.1",
 "where": [
  "level",
  "symbol"
 ],
 "slots": [
  {
   "name": "target",
   "takes": "one",
   "kinds": [
    "instance",
    "usage",
    "bundle",
    "point",
    "text",
    "position"
   ],
   "mode": "absorb"
  }
 ],
 "params": [
  {
   "name": "x",
   "type": "number",
   "since": "0.2"
  },
  {
   "name": "y",
   "type": "number",
   "since": "0.2"
  },
  {
   "name": "priority",
   "type": "enum",
   "values": [
    "disabled",
    "low",
    "normal",
    "high"
   ],
   "since": "0.2"
  }
 ],
 "positional": [
  "x",
  "y"
 ],
 "at_least_one": [
  "x",
  "y"
 ],
 "output": "same",
```

```
 "override": "geometry",
 "key": "object and axis",
 "form": "{target} @PLACE(x, y)",
 "sentence": "#M1 is at @PLACE(10, 20)"
}
```

- **内置关键字的槽位名**：每个槽位都有名字，就是能标的角色名。槽位名是标准词，与关键字一样随版本，改名就是不兼容的改动；同一关键字的各槽位不重名。定义的槽位也有名字，表里、诊断里用，但不能标（4.2 第 7 条）。种类名只有一套，签名和定义的角色声明（9.4）共用：`instance`、`terminal`、`net`、`point`、`wire`、`usage`、`bundle`、`text`、`annotation`、`label`（附录 A）。

5.4 消费方式

方式	关键字	输入的事实	失效（默认）	名字
吸收	大多数关键字	带到输出	严格	嵌套集合里的名字只在该集合内有效
封装	DEFINE、DEFINE-SYMBOL、DEFINE-STYLE	成为定义的内容，不外传	容错	局部；每次套用时改名，每个用例有自己的一份（9.4）
提交	记录，`@RNL`	按顺序并入本层	容错	发布到本层，本层各记录共享；可以先用后绑定

- 一个成员的产物是它各段、各嵌套集合产生的全部事实，不只是最后的句柄。它们随所在的顶层成员一起提交。
- 记录里的每条语句都提交，有没有名字都一样。提交后起什么作用，看对象是什么：场景里的描述画出来或参与求解；定义可以被使用，本身不画；引用和只由引用组成的集合不起作用。普通 `{…}` 里的语句随所在的成员提交，所以 `{ @LINE {…}}` 的线会画出来（13.4）。
- 定义体里的语句不提交，只成为定义的内容。普通定义在套用时才展开（9.4）。
- 带体的 USE-PACKAGE 也按封装方式收体：体里的语句是包的内容，不提交到本层。包里的名字怎样可见，见 16.2。

5.5 管道与嵌套

- `a => @K(…)` 把 `a` 的结果作为 K 的第一个对象；AS、ATTACH 把它填进第一个角色，不填进定义的槽位。结果是集合也只算一个对象（5.2.1）。`=>` 后面的段只能按位置收这个对象，不能给它标角色。
- 管道不是逐步执行的，而是展开成用句柄连起来的几条描述：每一段描述的是前一段结果的句柄；逐个成员做的一段展开成多条平行的描述（1.5 第 1 条）。
- **各段的先后**。每一段有自己的位置；凡是书写位置起作用的地方（1.5 第 2 条），后面的段都算比前面的段后写：按键覆盖（8.4）、外观队列（8.3）、取舍次序（8.5）、绘制顺序（11.6），以及导线的改写（13.1）。所以在 `#M1 @SET-SYMBOL $A => @SET-SYMBOL $B` 里，M1 用 `$B` 画。
- **失效只波及依赖它的描述**。一条描述失效，只有两种原因：它自己读不懂或自相矛盾；或者它要用的句柄本该由一条失效的描述创建。
  - `$GROUND @ATTACH &vss => @PLACE("20", 38) => @ORIENT("R90") => %g1` 里只有 PLACE 失效（PLACE 要数的地方写了字符串，5.3）；地照样创建，ORIENT 照样生效。
  - `#XI0<3:0> @BUNDLE(bad=1) => @PLACE(10, 20) => %s` 里叠画没建成（BUNDLE 没有参数 `bad`），PLACE 没有要描述的对象，也跟着失效。
- 名字绑定什么：写在开头的 `%x =` 绑定整条管道最后的句柄；`=> %x` 绑定它左边那一段的结果，写在末尾就是最后的句柄，写在中间就是那一段的结果，如 `#M1 @AS $A => %a => @AS $B => %b`。PLACE、ORIENT、SET-SYMBOL、SET-STYLE、GROUP 沿用输入的句柄，所以 `$GROUND @ATTACH &vss => @PLACE(24, 38) => %g0` 里的名字 `%g0` 是这次套用的用例，也就是这个地。AS、ATTACH 产生新句柄：`#M1 @AS $A => @AS $B` 的第二段把 `$B` 套用到第一段产生的用例上（嵌套），不是 M1。要给 M1 套用两个定义，写成两条语句。名字左边那一段失效时：它本来沿用输入的句柄，名字仍指这个句柄（上例的 `%g1` 是那个地）；它本来要创建新对象，名字指失效对象（上例的 `%s`）。

- 对象可以写成一个集合，集合里是语句（嵌套）。嵌套集合里绑定的名字只在该集合内有效。嵌套的语句各是一条描述：`{%a = #M1 @PLACE(1, 2); #M9} @ALIGN("y")` 在宿主没有 M9 时 ALIGN 失效，M1 的坐标照常生效。
- 标准写法是平铺的：给对象起名，再在别的语句里引用它。单条语句在开头写 `%x =` 起名，管道用 `=> %x`（4.3）。

5.6 定义作为对象

用到定义的关键字把定义作为它的一个对象，定义在哪个槽位由关键字定：ATTACH 的定义是写出的第一个对象；SET-SYMBOL、SET-STYLE、AS、SELECT-USAGE、SELECT-ANCESTOR-USAGE 的定义是最后一个对象，SELECT-USAGE 的可以省略（10.1）。

```
%ax = (_, _);
%w13 = &x @WIRE {#M1:1 #M3:1};
#M1 @SET-SYMBOL $NMOS;
%w13 @SET-STYLE $signal;
#M1(a) #M2(b) %ax(axis) @AS $MIRRORED;
$GROUND @ATTACH &vss;
# @SELECT-USAGE $GROUND => @SET-SYMBOL $GROUND_SYMBOL
```

- 定义不标角色，也不计入"全标或全不标"。定义也是对象、占一个槽位，所以 `{#M1 #M2} @SET-SYMBOL $X` 是两个对象（5.2.1）。
- 定义的槽位由关键字固定，所以解析语句时不用知道一个 `$` 名字指的是定义还是集合：`$GROUND_STUB @ATTACH $GROUND_FILL_SET` 里前一个是定义，后一个是集合。
- 定义不能经管道送入：送来的对象填第一个槽位，AS、ATTACH 按角色配它，不占定义的槽位（5.5）。所以 `$GROUND @ATTACH &vss` 不能写成 `$GROUND => @ATTACH &vss`。
- 定义可以是嵌套的匿名定义：`#M1 @SET-SYMBOL { @DEFINE-SYMBOL {…}}`。
- 普通定义可能写在后面或别的层里，所以 AS、ATTACH 的对象个数、角色名和参数在本层生效时核对（7.1）。

5.7 句柄与名字

- 宿主对象的句柄由宿主给出：元件名、网络名。端子是元件加端子序号，所以 `#M1:1` 与 `#M1:d` 是同一个端子。
- RNL 创建的东西，句柄由创建它的语句产生。别的语句只能通过名字引用它：名字是找到它的唯一办法；例外是求解器的记录，它用编号引用没有名字的对象（5.9）。除此以外，没有名字的对象只能在它自己的语句里使用（管道、嵌套）。
- 一个名字不会指两个东西：同一层里只绑定一次（6.6）。
- 子对象的句柄是父对象和它的名字，不必另起名字。
- 普通定义展开出的局部对象，句柄是用例和局部名，从外面写作 `%c1::mid`（9.4）。没有名字的用例，它的局部对象只在诊断里按位置显示；只有求解器的记录能经用例的编号引用它们，如 `%#5::mid`（5.9）。
- 生成集里的单个对象没有名字，不能单独引用：不按种类和关联的对象认对象（没有自然键），集合也不提供按下标引用。

5.8 事实的键

- 有键的事实按键覆盖：同一个键，后写的生效；后写的失效时，保留较早的有效值。
- 几何事实例外：PLACE 的 x、y（每个坐标一个键）和 ORIENT（每个对象一个键）。同一个键写了两个不同的值就是冲突，按 8.5 取舍：优先级相同时保留后写的，与按键覆盖结果相同；优先级不同时保留高的。PLACE 只改它写出的坐标，从没写过的坐标是自由的。
- 用已经绑定的名字再创建对象，不是按键覆盖：名字同一层只绑定一次（6.6）。要改 RNL 创建的对象，写修改它的语句，如 `%g0 @PLACE(40, 38)`。
- 没有键的事实累加：不冲突的并存，冲突的按 8.5 取舍。
- SET-SYMBOL、SET-STYLE 不按键覆盖，按外观队列生效（8.3）。
- VERSION 只在头段记录里写一次（16.1），所以谈不上覆盖。
- CURRENT 每个端子一个键（15.2），LAYER 每个对象一个键（11.6）。

- 用例没有键：一个定义套用几次就是几个用例，互不覆盖。两句相同的套用产生两个用例，所以 `$GROUND @ATTACH &vss` 写两次就画两个地。对一个用例写的修改语句（如 PLACE），其事实是有键的，照上面的规则处理。
- 对象的优先级是一条有键的事实；优先级之间按什么次序覆盖，见 14.8。
- 写了引用名（`key=`）的 TEXT，每个附着对象、每个引用名是一段文字，后写的按项覆盖它（13.3）。没写引用名的 TEXT 不按键覆盖：每写一次就多一段文字（13.3）。
- 导线没有键：写两次就是两条导线（13.1）。WIRE 收到导线时改写它；改写有键，键是这条导线，后写的改写换掉先写的（13.1）。
- 约束没有键。要改一条约束，就把它关掉（`"disabled"`，14.8）再写新的，或者直接改文本。

5.9 没有名字的对象的编号

```
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
.subckt amp in out vdd vss
M1 out in vss vss nch
R1 vdd out 10k
* @RNL {
*   #M1 #R1 @ALIGN("x");
*   &out @WIRE {#M1:1 (_, _) #R1:2};          /* %#1: the wire; %#2: the free point */
*   {#M1 #R1} @TEXT("${name}")                /* %#3: the label of M1; %#4: that of R1 */
* }
* @RNL(type="generated", priority="high", author="rnl-solver 0.1", hash="sig1:…", numbering="…") {
*   #M1 @PLACE(10, 0); #R1 @PLACE(10, 8);
*   %#2 @PLACE(12, 4);
*   %#1 @WIRE(mode="straight") {#M1:1 (12, 2) %#2 (12, 6) #R1:2};
*   %#3 @PLACE(13, 1); %#4 @PLACE(13, 9)
* }
.ends amp
```

- 求解器的记录用编号引用经名字找不到的对象，以便写回它的每个决定（14.8）。编号写作 `%#` 加一个无符号整数，如 `%#3`，用法同名字：`%#5:p` 是用例 `%#5` 的端子，`%#5::mid` 是它的一个局部对象。编号只用来引用它的对象，不像名字那样绑定，所以 `%#3 = …` 和 `=> %#3` 是语法错误（4.1）。
- **哪些对象**：没有自己的名字、也不能经 `::` 从有名字或有编号的对象找到的点、导线、标签、叠画、标注和用例（6.5、9.4）。只在嵌套集合里绑定的名字（6.7）不算，因为求解器的记录用不了它。对求解器的一条记录，这些对象依次是：
  1. 本层作者记录（`type="authored"`，2.8）的成员产生的，以及这些成员产生的用例的定义体里产生的。它们按求解器看到的那样数，即不带 `type="generated"` 的记录来解释这一层（2.8）；带上这些记录解释时，编号不变；
  2. 这条记录自己的成员产生的，以及它们产生的用例的定义体里产生的。它们按记录写出的文本数，当作其中的编号都能解析；下面的校验值对不对得上，不改变它们。

  约束和集合不编号。规则逐个生成的对象（10.5）也不编号：求解器把它们写成显式语句，落在第 2 项里。求解器的每条记录各有自己的编号：第 1 项对每条记录都一样，第 2 项只数这条记录自己的成员。
- **次序**：作者记录按 2.6 的次序取：本层自己的文本，再是从外面用 `scope` 写进本层的记录。一个成员里，按产生对象的记号的先后编号：点是它的 `(`，其余对象是关键字。一个记号为集合的每个成员各产生一个对象时（5.2.1），每个对象各有一个编号，按集合成员的次序。用例定义体里的对象紧接在用例之后编号，按定义体的次序，嵌套的用例照此一层层往下。编号从 1 起。
- **写在哪**：编号只能写在 `type="generated"`、带字段 `numbering`、不带 `scope` 的记录里（2.6、2.8）。写在别处是错误：作者记录用名字引用对象。在它所在的记录里，编号可以用在产生其对象的成员之前，名字也是这样（6.7）。
- **校验值**：`numbering` 是编号对象来历的校验值。按编号的次序，每个编号一行：
  - 产生这个对象的那个成员（记录或定义体里的成员，连同它的各段，4.1）的记号，去掉自由文本和注释，记号之间隔一个空格。这里的记号指词法单位：关键字；引用，连同它的端子、子对象

和下标，中间不留空白；照原样写的字符串或数；名字；`_`；以及 `{ } ( ) [ ] , = ; =>` 中的一个。所以 `&out @WIRE {#M1:1 (_, _) #R1:2}` 写成 `&out @WIRE { #M1:1 ( _ , _ ) #R1:2 }`；

- 对象在用例的定义体里时，先写这个用例的那一行（用例本身有没有编号，都照样写），再接一个制表符和定义体里那个成员的记号，照定义里的写法；
- 那个记号为集合的每个成员各产生一个对象时，后面接一个制表符和这个成员的句柄，写成引用：宿主元件或网络写名字；端子写元件和端子号，如 `#M1:1`；对象的位置或某个键上的文字，写对象后接 `::position`，或 `::` 和键；RNL 对象写本层绑定给它的 `%` 名字，有几个时取按 2.6 的次序最先绑定的那个，不管集合里写的是哪个名字；没有这样的名字的对象写编号，或作为用例的局部对象，写用例的引用（照这几条写）、`::` 和局部名字：有几个局部名字时，取定义体里最先绑定的那个，同一个成员里也一样（`%p = (0, 0) => %q` 里取 `p`），不管集合里写的是哪个名字；几个用例都把它当作局部对象时，经最先套用的那个用例写。名字只由 ASCII 字母、数字和 `_` 组成时照写，否则写在双引号里，`"` 写两个（3.6）。例如在 `%p = (0, 0); %q = %p; %s = {%q}` 之后，`%s @TEXT("x")` 产生的标注那一行是 `%s @TEXT ( "x" )`、一个制表符和 `%p`，校验值是 `675df5171032726d`。用 `$D = @DEFINE(roles=["x"]) { %p = (0, 0) => %q }; %u = #M1 @AS $D; %s = {%u::q}` 时，`%s @TEXT("x")` 那一行末尾是一个制表符和 `%u::p`，校验值是 `a8c9571193175903`。

校验值是这些行（每行以换行符结束）的 UTF-8 文本的 SHA-256 的前 16 位十六进制，小写。

- 一条记录的校验值与本层对不上时，记录里用到编号的语句都不生效；有这样的语句时，整条记录给一个提示：求解结果对没有名字的对象已经过时。不用编号的记录可以不写 `numbering`。记录里的其他语句照常生效（第 17 章）。编号超过最后一个编号对象是错误。
- 例子里，校验值对六个编号各有一行：WIRE 那个成员两行（先是导线，再是点），TEXT 那个成员两行，后面分别接 `#M1` 和 `#R1`；改写导线的 WIRE 里写成坐标的两个点是这条记录自己的对象，编号 5 和 6，再占两行。不求解的查看器照求解器选定的样子画出自由点、导线的走法和两个标签。
- 改写成 `{#R1 #M1}`，标签 3、4 就对调了：校验值对不上，用到编号的语句在这一层重新求解之前都不生效，M1、R1 的位置照旧。改动 ALIGN，或移动 M1，编号都不变。
- 注：校验值只看编号对象是怎样产生的。移动器件、改不产生无名对象的语句，编号都不变；增、删、改、调换产生这类对象的成员，编号就过时，直到这一层重新求解。求解器的记录一般不含改集合或定义的语句；不论有没有，第 1 项都不带这些记录来数。

6 名字与作用域

6.1 层级

- **层级按文字的包含关系。** 一个层的 RNL 作用于本层，以及它在文字上包含的各层；谁实例化谁不起作用。
  - 文件包含它定义的子电路和它 include 的文件。
  - 写在子电路里的 `.include`，和嵌套在子电路里的 `.subckt` 定义（SPICE3、ngspice 的局部定义），挂在那个子电路下面。
- 从上往下，到一个子电路的层级链是：根上的包（标准包，以及头段记录里 USE-PACKAGE 的包，16.3）→ 主文件的文件层 → ……（逐级 include 的文件）→ 定义这个子电路的文件 → 子电路自己。例如：

```
packages at the root (the standard package)
└ file level of tb.sp (the top level)
   ├ load                    a subcircuit defined in tb.sp
   ├ file level of ota.sp    a file included by tb.sp
   │  ├ ota                  ota instantiates cell
   │  └ cell
   └ file level of bias.sp
      └ bias                 also instantiates cell, but neither the RNL of bias.sp nor that of bias
reaches cell
```

- **子电路的 RNL 只管它自己。** 取舍时它的语句排在各文件层的内容之后，因而优先于它们（8.5）；但不传给它实例化的子电路，嵌套定义在它里面的除外。amp 实例化了 cell，amp 的定义和规则到不了 cell。
- **被 include 的文件影响不到 include 它的层。** 它的文件层的定义、规则、USE-PACKAGE 只作用于它的子电路和它再 include 的文件。从网表看，它写在子电路之外的器件属于 include 它的那一层，但它的 RNL 改不到这些器件。所以一个文件不会被它 include 的文件改动。
- **被 include 的文件的文件层没有自己的电路。** 只有主文件的文件层是电路。被 include 的文件在文件层写的宿主引用（`#M1`、`&vss`）失效（6.3），在那里起作用的只有定义、规则、USE-PACKAGE 和 `scope`。单独打开的文件就是主文件。
- **被 include 的文件继承 include 它的层。** 主文件的风格（它的规则和 USE-PACKAGE 的包）一直传到库文件的子电路里。所以同一个库文件在不同的主文件下，或单独打开时，可能画得不同，而它的求解结果只存一份（第 17 章）。要让库里用到的符号不受上层外观规则的影响，在库文件或子电路里用 USE-PACKAGE 再用一次标准包（16.2）。只想钉住标准包的版本、仍让上层的风格盖过它，就把 USE-PACKAGE 写在库文件的头段记录里（16.3）。主文件里只该作用于顶层的规则，写 `recursive=false`（10.1）。
- **一个子电路一张图。** 查看器不展开子电路：子电路的元件总是用它的符号画，子电路内部单独画、单独求解。每个子电路只有一条上层链，所以一份文档里每个子电路恰好一张图。
- **包。** `@USE-PACKAGE` 写在哪层，包的 `$` 定义和规则就在那层及以下生效（16.2）。标准包隐式用在根上，头段记录里 USE-PACKAGE 的包也放在根上（16.3）。include 把文件放在本层下面，文件继承本层；USE-PACKAGE 把包放在本层上面，本层继承包。
- **规则按定义对待。** 写在某层的规则在该层和下面每一层各求一次，每次只看那一层（10.4）。选择加 `recursive=false` 就只在写规则的那一层选，规则实际上只作用于这一层。这正是原则 2.5 对定义的要求：定义能把自己的生效范围限定在当前层，或让下层继承，未限定时默认继承。原则 2.5 说只属于所在层的描述，指逐个对象写的语句，它们的宿主引用本来就只在本层解析。

6.2 前缀

前缀	指什么	在哪里解析
`#`	宿主的元件；单独的 `#` 是本层	本层的宿主
`&`	宿主的网络	本层的宿主
`%`	在本层起的名字：场景对象的，已有对象的（如 `%tail = #M5`），或只在本层用的定义的	本层
`$`	定义，或集合（6.6）	绑定它的层及其下层

前缀	指什么	在哪里解析
`?`	定义的角色或参数；符号体里单独的 `?` 是用这个符号画出的对象（13.3）	普通定义或符号的定义体

`$X` 与 `%X` 是两个不同的名字。

6.3 宿主引用

- `#M1`、`&vss` 只在本层的宿主里解析，不跨层（原则 2.5）。不同子电路里的同名元件互不相关。
- 找不到的宿主引用失效（原则 2.4）。被 include 的文件的文件层没有自己的电路，所以那里的宿主引用都失效（6.1）。
- 名字后面可以紧跟下标，表示一个或一组宿主名字，如 `&D<3>`、`#XI0<3:0>`（6.11）。
- 某一层有哪些网络，由宿主适配器按宿主的语义决定：
  - 局部网络：这一层的网表行用到的结点，含端口。
  - 全局网络：`0` 在各种 SPICE 方言里都是全局地。仿真器自己认的地的别名（如 ngspice 的 `gnd`，HSPICE 的 `GND`、`GND!`、`!GND`、`GROUND`）由这种方言的适配器并到 `0`；并入后 `&gnd` 与 `&0` 是同一个句柄。`.GLOBAL vdd`（Spectre 写 `global vdd`）声明的名字，在任何子电路里都是同一个结点。名字以 `!` 结尾（如 `vdd!`）是 Virtuoso 原理图的习惯，起作用的是声明。
  - 全局网络在用到它的每一层各有一个句柄，即这一层里的那一份；网表行没有用到它的层没有这一份。所以 `&0` 指写它的那一层的那一份：每张图各画各的，事实属于那一层，下层的描述改不到上层。
  - 要指每一层里的那个全局网络，在上层按名字选，加条件 `global=true`（10.1）。

6.4 端子

- `object:terminal` 是端子：`#M1:1`、`#M1:d`、`#XU1:inp`、`%g0:p`。`#:inp` 是本层的端口 inp。
- 数字是宿主的端子序号，从 1 起。
- 名字是宿主的端子名：头段记录里 `@DEVICE` 给的名字（6.12），或子电路的端口名。不能不加引号写的名字（3.6）要加引号，如 `#XU1:"in+"`。端子名、端口名后面可以紧跟下标，如 `#X1:D<7:0>`、`#:D<7:0>`（6.11）。
- 大小写：子电路的端口名和 DEVICE 给的端子名都按宿主的规则：SPICE 不分大小写，`#M1:D` 也行；Spectre 区分。用例的端子名是它的定义的角色表里的名字，按 RNL 名字的规则（3.6）。
- 引脚的 `label=` 只用于显示，不能用来引用端子：端子属于网表，引用端子不取决于用的是哪个符号。
- 用例的端子就是它的定义的角色，按角色表的顺序编号；集合角色不是端子（9.4）。只有带符号的用例，它的端子才有引脚、能接导线（第 12 章）。
- 找不到的端子失效。端子在符号里有没有引脚，在定了符号之后检查（7.1）。
- 除了子电路的端口名，端子名由 DEVICE 给出；SPICE 系的标准名字见 6.12。
- 网表里没有定义的子电路（如 `XM1 … sky130_fd_pr__nfet_01v8` 里的 PDK 器件），只有用户在头段记录里用 DEVICE 给出端子名（6.12），或用了写有 DEVICE 语句的包（16.2）时，才有端子名；否则只能写序号。

6.5 子对象

- `object::name` 是子对象。它总是对象，不是数值。它的句柄是父对象和名字。
- 子对象有：
  - 文字的引用名：写了 `key=` 的 TEXT 在它附着的对象上产生的子对象，如 `#M1::wl`。符号里写了引用名的文字，是用这个符号画出的每个对象的子对象（13.3）。`::name`、`::value` 不是标准子对象，只是常用的引用名（13.3）。
  - `::position`：对象的原点。`#M1::position @PLACE(…)` 等于 `#M1 @PLACE(…)`。对画出来的用例，它是画这个用例的那个点：主对象有坐标点时就是那个点，否则是用例自己的原点（12.1）。对 TEXT，它是文字自己的位置（13.3）；对中心写法的 RECTANGLE、CIRCLE、ELLIPSE，它是中心（13.4）。
  - 端子的 `::net`：它所在的网络，如 `#M5:1::net` 就是 `&tail`。叠画的端子见 11.8：成员接在不同网络上时，是这些网络的集合。
  - 用例的角色名：填进这个角色的对象，如 `%dp1::in_p` 是 M1，`%g0::p` 是网络 vss。集合角色取到的是那个集合。
  - 用例体里起了名的局部对象，如 `%c1::mid`（9.4）。

  - 集合的 `::position`：成员所占范围的中点，按每个成员的那一点（原点、引脚或点）算；嵌套的集合一直取到最里层的对象，与符号无关。`%pair_set::position @PLACE(10, 20)` 把 pair_set 的中心放在 (10, 20)。集合的范围没有子对象：要让两组分开，写 DISTANCE（14.5）；要把一组作为约束的一个成员，写成 `{%x}`（14.1）。
- 标准子对象名保留：普通定义的角色名、参数名、局部名，以及文字的引用名，都不能与它们相同（按原样比较，3.6）。局部名不能与角色名相同，文字的引用名也不能与它所附着的用例的角色名、局部名相同。局部名可以与参数名相同：参数是值，`::` 不取值。定义的参数也不能叫 `priority`（14.8）。
- `::` **后面的名字按这个次序找**：用例自己的角色名和局部名；它们的 `definition=` 代称；标准子对象名和它们的代称；文字的引用名（代称见 16.5）。
  - 引用名排在最后，所以不能与排在它前面的名字拼写相同：标准子对象名和本文件声明的它们的代称（如 `rnl-zh` 的 `"位置"`）、用例的角色名和局部名，以及这些名字的 `definition=` 代称。这样的引用名是错误，与上面的重名一样（13.3）。
- `::` 只取对象自己的部分；在场景里按条件找对象，一律用选择关键字。所以"某个定义的全部用例"写成选择（10.1），不写成定义的子对象。要用例成员的范围，就给成员的集合起名。
- 端子用 `:`、子对象用 `::`，免得以后的版本新增的子对象名与网表的端口名冲突。
- **子对象名**：标准子对象名有两个，`position` 和 `net`，都保留。其余子对象名都是动态的：文字的引用名、用例的角色名、定义体里的局部名。以后的版本新增的标准子对象名按版本区分（16.1）：旧版本的文件里，同名的引用名照旧有效。

6.6 命名

- `%name = statement`、`$name = statement` 给语句的结果起名；管道里 `=> %name`、`=> $name` 给它们左边那一段的结果起名（5.5）。名字只是引用对象的另一种写法，不产生对象。
- `$name` 绑定定义或集合，含选择的结果。`$` 集合在某层求值时只含这一层的成员（10.3）。其他场景对象属于某一层，不能有 `$` 名字。
- `%name` 可以绑定本层的任何对象：本条语句创建的对象、已有的对象（`%tail = #M5`）、集合（`%pair_set = {#M1 #M2}`），或只在本层用的定义。
- **同一层里一个名字只绑定一次**。再绑定同一个对象可以，如 `%x = #M1` 写两次；绑定别的对象是错误，名字保留第一个有效的绑定。只有全部绑定都失效时，名字才绑定到失效对象（7.2）。创建对象的语句、定义、集合、选择的结果都一样。
- 覆盖针对事实，不针对名字：要改 RNL 创建的对象，写修改它的语句，如 `%g0 @PLACE(40, 38)`（5.8）。

6.7 名字的范围

- `@RNL` 提交的名字发布到本层：本层的所有记录共享，名字可以在绑定它的语句之前使用（原则 2.9）。
- 嵌套集合里绑定的名字只在该集合内有效。
- 定义体里绑定的名字是体内局部的（6.9）。

6.8 继承与遮蔽

- `$` 定义沿层级（6.1）向下继承，绝不向上：进本文件的子电路，也进被 include 的文件。一个层看不到它 include 的文件里的 `$` 定义；要给上层提供定义，做成包，用 USE-PACKAGE 使用（16.2）。子电路符号是唯一向上的例外（9.2.6）。
- 下层定义了与继承来的同名的 `$X`，就在这一层及其下层遮蔽继承来的定义，并给提示；上层看到的仍是自己的。
- 一层里 USE-PACKAGE 的包，它的 `$` 定义算作这一层上面的一层（16.2）。它们遮蔽上面各层或根上的包的定义时，与任何遮蔽一样给提示。头段记录里使用的包定义了标准包里的名字，或排在它前面、放在根上的包里的名字时，也给提示（16.3）。
- `%` 名字只在本层有效，不继承：不进本文件的子电路，也不跨 include。包里的 `%` 定义是包内私有的辅助定义。

6.9 定义体里的名字

- 定义体里的名字在写定义的地方解析，套用处不重新解析。
- 所以遮蔽一个定义只改变这个名字指什么：别的定义内部已经用到的定义不变。例如在下层遮蔽 `$DIFFERENTIAL_PAIR`，不会改变上层写的定义体内用到的差分对。
- 体内的 `%name` 先在体内的局部名里找，再在本层的定义里找。本层的场景对象（点、导线等）不能从体内引用，这样的引用是错误。
- **体内的写法**：角色和参数写 `?name`。局部名用 `%name =`（管道里用 `=> %name`）绑定，用 `%name` 引用。从外面经用例引用局部名（9.4）。
- 定义体看不到正在定义的名字：`$NMOS = @DEFINE-SYMBOL {$NMOS; …}` 里，体内的 `$NMOS` 是外面（上层或包里）这个名字的定义。下层就是这样遮蔽并扩展上层定义的；外面没有这样的定义时是错误。
- DEFINE-SYMBOL、DEFINE 体里的宿主引用是错误：`#M1`、`&vss`、`#:inp`、`#M1:d` 和单独的 `#` 都不能写在那里。体里的名字在写定义的地方解析，普通定义却会在别的层、别的文件里套用，在那里宿主引用仍会指到定义所在的那一层，含义不唯一。普通定义要用宿主对象，经角色传进来。
- **普通定义的体什么时候检查**：
  - 按定义查一次：不依赖实参的都在这时查，包括关键字、签名（对象个数、参数名和类型）、`$` 和 `%` 名字（在写定义的地方解析）、宿主引用、局部名的撞名（9.4），以及声明了种类的角色用得对不对：角色 t 声明为 `"instance"` 时，体里写 `?t::net` 是错误。签名在读入时查（7.1）；名字要整层读完才能解析，所以要用到名字的检查在本层生效时做。两者每个定义都只做一次，与套用几次无关。
  - 按套用查：依赖实参的，如没声明种类的角色收到的对象、符号里有没有那个引脚，每次套用时查，报在套用处，并指出普通定义里的原句。
  - 错误的范围：体里查出错的语句（含撞名的局部名）在定义处报一次，每次展开都不含它，体的其余部分照常（体是容错的，9.1）。角色表、参数表写错（重名、与标准子对象名撞名、不认识的种类名），整个普通定义失效，用到它的套用也都随之失效。
  - 从没被套用的普通定义也查。这些检查与套用无关，所以能查出包里没用到的定义的错误。

6.10 空位

- `?name` 只出现在普通定义的体里，名字应在角色表或参数表里；否则读入时就是错误。
- 写在对象的位置是角色，如 `?t`、`?t::net`；写在字面量的位置是参数，如 `min=?gap`、`[?w, 2]`、`(?x, 0)`（9.4）。角色写在字面量的位置，或参数写在对象的位置，读入时就是错误。角色名和参数名在同一张表里，不能重名。
- 角色的空位可以带路径，如 `?t::net`；参数的空位不带路径。

6.11 下标

写法	表示
`&D<3>`	与 `&"D<3>"` 相同
`&D<7:0>`	一组：`D<7>`、`D<6>`、…、`D<0>`
`&D<1,2:3>`	`D<1>`、`D<2>`、`D<3>`
`&D<0:6:2, 9>`	`D<0>`、`D<2>`、`D<4>`、`D<6>`、`D<9>`
`&D[3:0]`	`D[3]`、`D[2]`、`D[1]`、`D[0]`
`#XI0<3:0>`	四个元件
`#X1:D<7:0>`	X1 的八个端子
`#:D<7:0>`	本层的八个端口
`&"D"<1:0>`	`D<1>`、`D<0>`
`&"D<7:0>"`	名字就叫 `D<7:0>` 的结点：引号里的不解析

- 网表里没有总线：Virtuoso 等工具出网表时把总线拆成单个的位，如 Spectre 网表里的 `D\<3\>`；子电路的总线端口、叠合元件也拆开。下标是写一个或一组这类名字的简写。它只是拼名字，不靠宿主认总线，所以含义唯一。

- **写法**：宿主名字（加不加引号都行）后面紧跟 `<…>` 或 `[…]`，里面是用 `,` 分开的若干项。每项是 `i`、`a:b` 或 `a:b:s`，几种可以混写。i、a、b、s 都是不带符号的十进制整数，s 至少为 1。名字与括号之间没有空白；括号里可以有空白和注释（3.5）。
- **展开**：各项按写的次序展开、拼接，重复的照留。`a:b` 从 a 到 b，`a:b:s` 每次走 s，不越过 b：`<7:0>` 是 7、6、…、0，`<0:7:2>` 是 0、2、4、6。步长写在最后，与 Virtuoso、Xschem 相同，与 MATLAB 的 `a:s:b` 不同。生成的名字是括号前的名字、左括号、不带前导零的十进制数和右括号，括号照写的那一种：`&D<07>` 是 `D<7>`，`&D[3]` 是 `D[3]`。
- **一个还是一组**：只有一项且是单个数时，是一个引用：`&D<3>` 就是 `&"D<3>"`。否则是一组，即对一个集合的引用，如 `&D<3:3>`、`&D<3,5>`。一组放进集合时打散一层（5.2）；作语句唯一的对象时先拆一层（5.2.1），所以 `#XI0<3:0> @ALIGN("y")` 对齐四个元件；放在要单个对象的槽位时逐个成员做；放在 WIRE 的网络槽位时整个收下，是一根总线（13.1）。
- **用在哪里**：`#`、`&` 之后的宿主名字，以及 `#` 引用里的端子名、端口名。RNL 名字（`%`、`$`、`?` 之后的）、用例的端子名（即定义的角色名）和子对象名不收下标，后面紧跟 `<`、`[` 是错误。
- **名字后面不能直接写点**：名字后紧跟的 `<`、`[` 一律开始一个下标，后面不合写法就是语法错误，不会读成点。要在名字后面写点，就隔开，如 `&D (1, 2)`。
- **一处展开**：一个名字后面至多一个下标，所以 `A<1>[2]` 这样的名字加引号写，`&"A<1>[2]"`。一个引用里至多一处下标展开成一组：`#XI0<3:0>:a`、`#X1:D<7:0>` 可以，`#XI0<1:0>:D<1:0>` 是错误；两处都是单个数的可以，如 `#XI0<3>:D<1>`。展开成一组的引用后面接 `::` 时，对每个成员各取子对象，如 `#XI0<3:0>::name`。
- **引号里的不解析**：`&"D<7:0>"` 指名字就叫 `D<7:0>` 的结点；`&"D"<7:0>` 才展开。网表里的名字含有不表示总线位的 `<`、`[` 时，加引号照原样引用。
- **比的是适配器读出的名字**（3.6）：Spectre 网表里的 `D\<7\>`（反斜杠是 Spectre 的转义）由适配器读成 `D<7>`，所以 `&D<7>` 能引用它。
- 每个成员各自解析。找不到的成员是失效对象，按收它的槽位的规定严格或容错处理（7.3）。
- 参数里的字符串不展开，如 PIN 的 `terminal=`（9.2.4）、选择的条件 `name=`。
- 不收：Virtuoso 的重复 `<*2>A`、`A<0:2*2>` 和并列 `A,B`（集合已经能写）；KiCad、Xschem 的 `D[0..7]`，它的成员是不带括号的 D0 … D7。
- 叠合元件（如 Virtuoso 的 `I0<3:0>`：几个宿主元件画成一个）写成叠画：`#XI0<3:0> @BUNDLE`（11.8）。

6.12 端子名：DEVICE

```
* sky130 transistors, defined in the PDK and not in this netlist
* @RNL(type="header") {
*   @VERSION("0.3");
*   @DEVICE(name="X*", model="sky130_fd_pr__nfet_*", terminals=["d", "g", "s", "b"]);
*   @DEVICE(name="X*", model="sky130_fd_pr__pfet_*", terminals=["d", "g", "s", "b"])
* }
XM1 out in 0 0 sky130_fd_pr__nfet_01v8 w=1 l=0.15
XM2 out in vdd vdd sky130_fd_pr__pfet_01v8 w=2 l=0.15
* @RNL { #XM1:d @CURRENT("in") /* XM1 is a sky130 NMOS; its terminal 1 is named d */}
.end
```

- **端子名由 RNL 起。** `@DEVICE` 给一类器件的端子起名：`terminals` 按网表行上结点的顺序列出名字。它只给宿主的端子序号起名，不改端子的顺序，也不改子电路的接口；它是原则 2.2 允许的显示别名。
- **只写在头段记录里**（2.8）：DEVICE 写在别处是错误。包文件的头段记录也算（16.2）。内嵌的包里不能写 DEVICE，要写在文件的头段记录里。
- **匹配**：`name` 比元件名，可以用通配符 `*`（任意多个字符）。`model` 可以不写；它比元件行上写的模型名或子电路名（Spectre 里是元件的 master，原生器件如 `resistor`），也可以用 `*`。两项都按宿主的大小写规则比，两项都对上，DEVICE 才与元件匹配。选择条件里比名字的条件也收 `*`，规则相同（10.1）。
- **几条都匹配时后写的生效**：适配器的默认在最前，然后是这一层用到的包里的 DEVICE 语句（按包的排序位置，16.2），再是外层文件的头段记录里的，再是被 include 的文件的；同一条头段记录里按写的先后。所以文件自己写的盖过它的包，包盖过适配器；具体的语句建议写在通用的后面。所有包的 DEVICE 语句都排

在各文件的之前，不论它们的 USE-PACKAGE 写在哪里：头段记录总在文件最前面，要是按 USE-PACKAGE 的位置排包，包反倒会盖过文件自己写的。写了 `model` 的 DEVICE 不算更具体，不另设特异性规则。

- **范围**：文件的头段记录里的 DEVICE，作用于这个文件和它 include 的文件；被 include 的文件只能改它自己和它再 include 的文件（6.1）。包文件的头段记录里的 DEVICE，作用于用到这个包的各层（16.2）。例如一个 sky130 的包可以写 DEVICE 语句、符号和选符号的规则，用户写一条 `@USE-PACKAGE(name="sky130-rnl")` 就既有端子名又有符号。
- **适配器的默认**：每种方言的标准元件定义，算作写在最外层。不用写进文件，工具也可以把它们写出来。同一种方言的工具应给出相同的定义。它们是元件那一行所在文件的版本的默认（16.1），与那个文件头段记录里的 DEVICE 一样。SPICE 系：

```
@DEVICE(name="M*", terminals=["d", "g", "s", "b"]);
@DEVICE(name="Q*", terminals=["c", "b", "e", "s"]);
@DEVICE(name="J*", terminals=["d", "g", "s"]);
@DEVICE(name="Z*", terminals=["d", "g", "s"]);
@DEVICE(name="D*", terminals=["a", "k"]);
@DEVICE(name="R*", terminals=["p", "n"]);              /* and one each like it for C, L, V, I, B,
F, H, W */
@DEVICE(name="E*", terminals=["p", "n", "cp", "cn"]);  /* and one each like it for G, S */
@DEVICE(name="T*", terminals=["p1", "n1", "p2", "n2"]); /* and one like it for O */
@DEVICE(name="U*", terminals=["a", "b", "ref"])
```

Spectre 的元件名不以元件字母开头，所以默认的定义按 `model`，即元件的 master 来认。Spectre 的原生器件：

```
@DEVICE(name="*", model="resistor", terminals=["p", "n"]);      /* and one each like it for
capacitor, inductor, vsource, isource, ccvs, cccs, bsource, port */
@DEVICE(name="*", model="diode", terminals=["a", "k"]);
@DEVICE(name="*", model="vcvs", terminals=["p", "n", "cp", "cn"]); /* and one each like it for
vccs, relay */
@DEVICE(name="*", model="transformer", terminals=["p_plus", "p_minus", "s_plus", "s_minus"])
```

- `switch` 每个结点一个端子，按 Spectre 给位置的编号命名：`t0` 是公共端，`t1` 到 `tn` 是各个掷点。
- 模型卡的元件，master 是一条 `model` 语句的名字，端子名按那个模型的 master 所属的器件族取。它的类型（选择条件 `type`，10.1）用 SPICE 的写法：

器件族	模型的 master	端子	类型
MOS	`bsim1`、`bsim2`、`bsim3`、`bsim3v3`、`bsim4`、`bsim6`、`bsimbulk`、`bsimsoi`、`bsimcmg`、`bsimimg`、`b3soipd`、`mos0`、`mos1`、`mos2`、`mos3`、`mos9`、`mos11`、`mos705`、`mos902`、`mos903`、`psp102`、`psp103`、`psp104`、`pspnqs102`、`pspnqs103`、`ekv`、`hisim`、`hisim2`、`hisimhv`、`hvmos`、`bta_soi`、`ucsd_hvmos`	`d g s b`	`nmos`；模型的 `type` 是 `p` 或 `pmos` 时为 `pmos`
双极晶体管	`bjt`、`bjt500`、`bjt503`、`bjt504`、`bjt504t`、`bjt3500`、`bjt301`、`vbic`、`hicum`、`hicum0`、`hicum2`、`mextram`	`c b e s`	`npn`；模型的 `type` 是 `pnp` 时为 `pnp`
二极管	`diode`、`juncap`、`juncap200`	`a k`	`d`
JFET	`jfet`	`d g s`	`njf`；模型的 `type` 是 `p` 或 `pjf` 时为 `pjf`
电阻	`resistor`、`rdiff`、`r3`	`p n`	`r`
电容	`capacitor`	`p n`	`c`
电感	`inductor`	`p n`	`l`
继电器	`relay`	`p n cp cn`	`sw`

- **网表里有定义的子电路**（含 include、lib 进来的库文件里的）：端口名照定义，这是宿主的事实（原则 2.2）。DEVICE 不改它们；与这样的元件匹配而名字不同的 DEVICE 给警告。DEVICE 只给没有端子名的器件起名：原生器件，以及网表里没有定义的子电路（PDK 器件），后者由用户或 LLM 写 DEVICE。
- **名字的个数**：元件的结点比 `terminals` 的名字多时，多出的端子只有序号；比它少时，引用多出的名字失效。

- **大小写**：DEVICE 给的端子名按宿主的规则比（6.4）。
- 序号总能用，子电路的端口名和 `#:port` 也一样。

7 失效与诊断

7.1 两个检查阶段

- **读入**：词法；文法；内置关键字的签名，即对象个数、种类和参数（含数组的形状）；以及普通定义的体里每个 `?name` 是否在它的角色表或参数表里、写的位置合不合适（6.10）。普通定义的体里不依赖实参的签名错误也在这时报（6.9）。
- **生效**：
  - 整层读完后，解析名字和宿主引用，并检查名字所指对象的种类；
  - 普通定义的体里的名字按定义各解析一次，不随套用重复（6.9）；
  - 核对 AS、ATTACH 的对象个数、角色名、角色声明的种类和参数（9.4）；
  - 每次套用检查展开结果里依赖实参的部分；
  - 定了符号以后，检查导线和约束接到的端子在符号里有没有引脚，以及 PLACE、ORIENT、约束作用到的用例和导线接到的用例有没有符号；作用于画在主对象坐标点上的用例的 PLACE 和约束失效（12.1）。

7.2 失效对象

- 失效的语句产生失效对象，带有诊断：原因、位置和上游原因。
- 失效对象留在成员中的原位，其他成员也留在原位。所以宿主网表里没有 M9 时，`{#M1 #M2 #M9} @ALIGN("y")` 整句失效，不会变成只对齐 M1、M2。
- 名字绑定到失效对象时（包括读入时就失效的定义），引用这个名字的语句收到的也是失效对象。依赖失效描述的语句随之失效，不论先后（原则 2.9）；它们也是错误，诊断写明上游原因（7.6）。
- 名字第二次绑定别的对象是错误，名字保留第一个有效的绑定；只有全部绑定都失效时，名字才绑定到失效对象（6.6）。

7.3 严格与容错的输入

- 签名为每个输入声明严格还是容错：
  - 严格：输入里有失效对象时，整条语句失效；
  - 容错：去掉失效的成员并记录诊断，其余照常生效。
- 默认：吸收的输入严格；封装的输入（定义体）和提交的输入（`@RNL`）容错（5.4）。
- AS、ATTACH 的实参严格；展开出的语句容错：失效的逐条单独丢弃。
- 管道和嵌套里，失效只波及依赖它的描述（5.5）：严格的输入里有失效对象时，这条语句失效，但嵌套在里面、与失效无关的语句照常生效；语句对每个成员各做一次时，每个成员是一条描述，各自失效。

7.4 恢复

- 语法错误使它所在的成员整个失效，范围到下一个 `;` 或所在集合的末尾。
- 括号不配对时，整条记录失效。
- 记录还没写完就被截断时，整条记录失效，诊断说明记录不完整（2.4）。
- 从错误中恢复，绝不超出它所在的记录：别的记录照常读（原则 2.9）。

7.5 冲突不是失效

- 冲突在整层读完后按确定的办法处理：按键覆盖（8.4）、外观队列（8.3）和几何事实的取舍（8.5）。
- 还没有确定办法的冲突，使整个解释失败（1.5 第 3 条；7.6）。本版没有这种情形：集合成环，失效的是闭合环的那条语句（5.2.2）。

7.6 诊断

- **检查与解释是同一个过程。** 解释器读入输入，给出生效后的场景和一份诊断；只用来检查时也叫 lint。本标准只规定这一份诊断。编辑器为即时反馈做的局部检查（如语法高亮）不是这里说的诊断：本节不约束它们，它们也不能代替诊断。

- **条目**。失效针对一个条目：记录头的一个字段、一条语句、一条记录、一个文件，或整个解释。是哪一种，在定义各种失效的地方写明（见下面的失效范围）。
- **三级**：
  - **错误**：条目失效，不论原因在它自己、在别的条目，还是在它依赖的上游。错误包括原则 2.4 的基本错误（语法错误、不认识的词、签名不符、引用找不到、种类不对、自相矛盾、版本不支持）、本标准另定的失效（如没有符号的用例收到 ORIENT、端子在符号里没有引脚；12.1、9.2.4），以及随之失效的（7.2）。
  - **警告**：条目合法、照常生效（本标准另有规定的除外，如没判定完而丢弃的事实，8.5），但可能不是作者想要的，如给要单个对象的槽位直接写出的空集合，像 `{} @PLACE(1, 2)`（5.2.1）；直接写作记录的成员、没有嵌套在语句里的无名定义（9.1）；生成规则带不到下层的角色（10.5）；记录头里不认识的字段（2.8）；包的版本被另一个替代（16.2）；导线接到的端子不在导线标的网络上（13.1）；检查没在求解器的时间上限内做完而丢弃的几何事实（8.5）。
  - **提示**：常见的覆盖、遮蔽、取舍，也可能是误用，如下层或下层使用的包遮蔽上层或根上的包的定义（6.8）；后 USE-PACKAGE 的包遮蔽先 USE-PACKAGE 的（16.2）；取舍时丢弃的几何事实（8.5）；没写版本而用了包的最新版本（16.2）；没声明版本的主文件按工具的默认版本读（16.1）；编号已经过时的求解器记录（5.9）。
- **失效范围**在定义各种失效的地方给出：字段（记录头认识的字段写错，按没写处理，2.8）；语句（大多数情况）；记录（括号不配对、记录被截断、`scope` 的目标找不到；7.4、2.4、2.6）；文件（版本不支持，16.1）；整个解释（还没有确定办法的冲突，1.5 第 3 条：整个解释停止，什么也不画；以后的版本可能把这种情况改为容错；本版没有这种情形）。
- **要求**：
  - 错误必须给出。警告和提示是本标准列出的那些，建议实现给出。
  - 实现可以另加自己的检查，但只能是警告或提示；它们不应让条目失效，否则不同的实现会画出不同的结果。
  - 实现应让用户能看到诊断；怎样呈现由实现决定（原则 2.9）。
  - 实现可以更严，如停下或拒绝整份输入（原则 2.9），但不应以此为默认。它可以让用户调高、调低警告和提示的级别；错误不能调低，因为错误就是失效。
  - 一致性测试查错误（位置和失效范围）和生效后的场景；警告可以作可选的期望；提示不测。
  - 注：警告是猜作者的意图，会随经验改变，写成要求就把这些猜法冻进了标准。原则 2.4 也只要求基本错误。
- **内容**：诊断给出
  - 位置：文件、行列或范围；哪条记录、哪条语句、哪个字段或引用（原则 2.8）；
  - 原因：期望什么、收到什么，如"in_p 要元件，收到的是网络"（9.4）；名字找不到时，给最接近的名字；
  - 上游原因：随之失效的语句各自写明上游原因，呈现时可以把它们收在根源下面，免得一个笔误报出全图几十条错。同一条规则在许多对象上失效时，可以合成一条诊断，列出这些对象，如包给每个 MOS 写 `W/L=${w}/${l}`，而有的元件没有 `w`；
  - 来历：普通定义展开出的语句，同时指出套用处和定义里的原句；规则产生的，指出规则和求值的那一层；包里的，指出包名、版本和文件（16.2）。
- 诊断的文字和语言不规定。本版没有标准的诊断码；实现可以给诊断编号，编号不另成容错类别。
- **包**：在用了包的文档里，包内容的错误照常报告，归到包；包内容的警告和提示只在检查包本身时给。注：用户改不了包；要是照常给警告，包作者有意写的宽泛规则会让用户被警告刷屏。
- 文件不能压制诊断，也不能要求更严的模式。
- 不认识的关键字、参数和取值是错误，与语法错误一样（原则 2.3）：这条语句失效（16.4）。本版识别但不解释的内容只有记录头里不认识的字段，它只给警告（2.8）。

8 生效

8.1 整层读完再生效

- 解释器解析完整的宿主网表，读入全部记录，再让每层按 8.2 的阶段生效。
- 同一层内，语句的先后只在 1.5 第 2 条说的几处起作用，不影响名字和引用的解析（原则 2.9）。
- 两层之间谁覆盖谁，按层级决定，不看文本的先后（原则 2.9；8.3）。

8.2 阶段

1. **读入**：词法、文法、签名。
2. **名字与定义**：解析 `$` 名字、`%` 名字和宿主引用；求出 DEFINE-SYMBOL、DEFINE-STYLE 的定义体。
3. **创建对象**：创建显式写出的场景对象（点、导线、标签、叠画、图元、约束、用例）和集合，展开普通定义的体。选择不在这里展开。
4. **符号队列**：SET-SYMBOL 生效，给第 3 步的对象定符号。这个队列里规则的选择，这时才在各层求值。
5. **逐个生成**：以选择的结果为对象的 AS、ATTACH，以及附着在选择结果上的 TEXT（10.5、13.3），按包、上层、本层的顺序，在每层各求一次。
6. **给生成的对象定符号**：同一个符号队列再对新对象走一遍。生成看到的是生成之前的符号；展开时若给已有对象换了符号，这次生成不回头重算。
7. **几何**：PLACE、ORIENT、约束按取舍次序逐条纳入，与已纳入的冲突就丢弃（8.5、14.8），然后求解。这一步要用到引脚的位置和符号的外框（9.2.3），所以放在定符号之后；作用于没有符号的用例的 PLACE、ORIENT、约束在这里失效。画在主对象坐标点上的用例（12.1），位置在这里由主对象算出，作用于它的 PLACE、约束失效。没有任何位置和约束的对象由求解器摆放，并把位置写回（14.8）。
8. **样式队列**：SET-STYLE 生效，按 8.3 的两级取值。符号图元没写的样式属性和 `fill=` 的颜色，这时才从所属对象的样式落到具体值。
9. **渲染**。

- 这些阶段就是 1.5 第 4 条说的内部顺序：一条描述读到的是它所在阶段开始之前的状态，它自己和同阶段其他描述的效果都不在内。
- 集合在用到它的阶段求值（5.2）：同一个集合，在第 5 步用到时不含生成的对象，在第 8 步用到时含。
- 选择条件只读前面阶段已定的信息：
  - 宿主信息、对象种类、用例、端子的电流方向、显式导线在第 3 步就定了；
  - 符号队列里的选择不能按对象用了哪个符号选；
  - 生成规则能看到符号。
- 端子在符号里有没有引脚，在第 6 步之后检查（7.1）。
- 求解和渲染的方法不由本标准规定，本标准只规定它们的输入（原则第 1 节）。

8.3 外观队列

- 有两个队列：符号队列收 SET-SYMBOL，样式队列收 SET-STYLE。
- 某一层的每个队列依次由下面几组组成：
  1. 根上的包的规则：标准包在最前，再是头段记录里 USE-PACKAGE 的包（16.3）；某个包在本层或上层（文件层或子电路）又用 USE-PACKAGE 使用时，它改排在那个 USE-PACKAGE 的位置（16.2）；
  2. 子电路给自己的 `#` 指派的符号，即子电路符号（只在符号队列；9.2.6）；
  3. 上层的规则，由外到内；
  4. 本层写的全部 SET-SYMBOL 或 SET-STYLE，包括规则和逐个对象写的语句；
  5. 外面用 `scope` 写进本层的 SET-SYMBOL 或 SET-STYLE，外层的在后（2.6）。
- 每组内按文本先后。别的包的内容排在它的 USE-PACKAGE 所在的位置（16.2）：在上层用到的随上层排，在本层用到的排在本层文本里 USE-PACKAGE 的位置。逐个生成产生的 SET-SYMBOL、SET-STYLE 归入生成它的规则所在的组，位置取规则的位置。
- **分两级**：

  - 直接的：直接指明这个对象的 SET-SYMBOL、SET-STYLE，即输入是它的句柄、名字或指向它的 `::` 路径，或是在 `{…}` 里把它列为成员的集合（这种集合起了名也算）。对象定义时写的样式参数（9.2.5、9.3）也算，相当于在创建它的语句处对它写了一次 SET-STYLE。
  - 传来的、选出来的：成员经选择或逐个生成收到的；所属对象传给它符号里图元的（9.2.5）；对象传给附着在它上面的文字的（11.3）；用例传给它的符号和它体里创建的可画对象的，经嵌套的用例一层层往下，但不传给填进它角色的成员（12.1 第 4 条）。传来的事实带着原语句的位置。
  - 每个属性（符号队列里就是符号）先看直接的事实，有就取位置最后的；没有再看传来的、选出来的，取位置最后的；都没有，用渲染器的默认（9.3）。位置按上面的队列次序。
  - 所以编辑器单独给某个对象改的符号、样式，不会被后写的规则盖掉；规则、包、主题照样管没有单独改过的对象。
- 记录头的 `type`、`author` 不影响先后（2.8）。
- 符号整体替换；样式按属性合并。应用一个样式，等于把它定义的每个属性各写一次，所以只定义了 `linewidth` 的样式只改线宽。
- 规则作用于写它的层及其下层，不作用于上层；包里的规则作用于用到这个包的所有层（10.4）。规则在每一层各求一次，只看那一层的对象；选择加 `recursive=false` 的规则只在写它的那一层生效（10.4）。

8.4 按键覆盖

有键的事实（5.8）按键覆盖，后写的生效（原则 2.9）。覆盖是合法的描述方式，不是错误；是否提示由实现决定。PLACE、ORIENT 是几何事实，按 8.5 取舍（5.8）。

8.5 冲突

几何事实合在一起可能没有解，即使两两之间都不矛盾，如 `{#M1 #M2} @ALIGN("y")`、`#M1 @PLACE(y=0)` 和 `#M2 @PLACE(y=10)`。下面的规则决定丢弃哪些事实。

- **几何事实**是：四种原子约束（ALIGN、ORDER、DISTRIBUTE、DISTANCE）、朝向关系 ORIENT-RELATION（14.9）、PLACE 的每个坐标、TEXT 的 `dx` 和 `dy`（13.3；它们与 PLACE 在同一轴上互相取代，不在这里取舍），以及 ORIENT。符号（SET-SYMBOL）不在其中，它在几何之前就定了（8.2）。实现自己的准则（器件不重叠、网格等）也不在其中：它们是求解器自己的考量，排在全部语言描述之后。描述逼得器件重叠，就重叠。
- **取舍的单位**：
  - 一条原子约束语句是一个单位，冲突时整条丢弃（ORIENT-RELATION 也是）；不会只留下它的一部分成员。这与失效一致：原子约束的成员是严格输入（7.3）。要单个对象的槽位收到集合、语句对每个成员各做一次时，每条这样的描述各是一个单位（5.2.1）。
  - 一条 PLACE 是两个单位：x 一个，y 一个，可以各自丢弃。TEXT 的 `dx`、`dy` 也各是一个单位；它们先与同一段文字同一轴上的 PLACE 按位置互相取代（13.3），留下的再参加取舍。
  - ORIENT 是一个单位：这个对象的朝向等于某个值。没有 ORIENT 就没有约束：由求解器选，并把朝向写回（14.8）；没有求解结果的渲染器按 `"R0"` 画。朝向是取 8 个值的变量，端子的位置由它算出；检查冲突时可以直接试各种朝向，不必改写成端子的相对位置。
  - 普通定义不是约束：用例展开出的每个事实各是一个单位，各自排序、各自丢弃。
- **自相矛盾的事实**：每条事实先单独检查。单独就无解的事实，如从一个对象到它自己、`min=4` 的 DISTANCE（9.5），是自相矛盾：它失效（7.6），不参加取舍。只有单独有解的事实才排序、纳入。
- **取舍次序**：几何事实排成一个次序。先按优先级 `priority` 排，`"high"` 高于 `"normal"` 高于 `"low"`（14.8）；优先级相同，按位置，后写的在前。位置：
  - 按层和文本先后：根上的包（16.3）、上层的内容、本层的语句、外面用 `scope` 写进本层的语句（外层的在后，2.6）；别的包的内容在离本层最近的那个 USE-PACKAGE 所在的位置（16.2）；
  - 普通定义展开出的事实取 AS 或 ATTACH 语句的位置，逐个生成的事实取生成规则的位置（10.5）；
  - 一条语句产生几个事实时：展开出的按定义体里的先后，对每个成员各做一次的按成员次序，PLACE 先 x 后 y；每种情况都是后面的在前。管道的各段是各自的描述，后面的段算后写（5.5）：

`#R1 @PLACE(0, 0) => @PLACE(5, 5)` 的事实次序是 y = 5、x = 5、y = 0、x = 0；`#M1 @PLACE(0, 0) => @ORIENT("R90")` 里 ORIENT 排在 PLACE 前。

- **取舍**：按取舍次序逐条纳入；每纳入一条，就检查已纳入的还有没有解，没有解，就丢弃这一条。结果唯一，就是按这个次序字典序最大的一致子集。有解，是指有实数的坐标，并给每个没定的朝向在 8 个值里取一个，使这些事实在容差内全部成立（11.7）；整数格点只是求解器的目标。
- **时间上限**：检查有没有解可能要很久，因为有些事实有几种成立的方式，要逐一去试：只给轴的 ORDER 或 DISTANCE、集合之间 DISTANCE 的上界、三个以上成员的集合的中心，以及没定的朝向。所以实现可以限定检查所花的时间。这个上限是求解器的选项，用户可以设定（14.8）；本标准不规定它的值。一条事实的检查没在上限内做完，就丢弃这条事实，并给警告，说明它没有判定（7.6）；这时结果可能与上面定义的不同，每次运行也可能不同。不设上限，或者每次检查都没到上限时，结果就是上面定义的那个。
- 纳入的事实对求解器都是硬的；丢弃的不再起作用。优先级为 `"disabled"` 的事实在排序之前就去掉（14.8）。
- 注：本节规定的是保留哪些事实，不是怎样求出它们。实现可以一次检查一整段事实，只在这段无解时再去找要丢弃的那一条；也可以只检查与新事实有共同对象的那些事实；只要保留的事实相同。
- 求解的实现应按本节的规定取舍冲突，结果应满足全部纳入的事实。不求解的实现只画显式几何，不要求它检查语句之间的冲突（原则 2.4）。
- 丢弃是合法的取舍，给提示，实现可以把它升为警告。给出时，提示列出每一组已纳入、与被丢弃的事实一起无解、而且少了其中哪一条都不再无解的事实，每条事实按写它的语句指出（7.6）。这些组里的事实所在的语句，也各给一条提示，列出因它而丢弃的事实，让作者看到哪些语句因它失去了作用。找全这些组没在时间上限内做完时，提示说明它列得不全。丢弃 `"low"` 的事实是预期内的，这些提示可以不给（7.6）：实现可以不列出与它冲突的那些组，也不给组里的语句提示。
- 冲突不是失效（7.5）。语句之间高层语义的冲突，以及与网表是否一致，不要求检查（原则 2.4）。

9 定义

9.1 共同规则

- 有三种定义：DEFINE-SYMBOL 产生符号，DEFINE-STYLE 产生样式，DEFINE 产生普通定义。符号和样式是特殊的定义（1.4.4）。它们都收一个集合作定义体；体内的语句逐条容错（7.3）。
- 定义是对象，用 `$name =` 或 `%name =` 命名（6.6），作关键字的对象被使用，所在的槽位由关键字定（5.6）。
- 提交一个定义只是发布它的名字。定义本身不画任何东西，只在被使用时起作用：符号在被 SET-SYMBOL 指派时，样式在被 SET-STYLE 指派时，普通定义在被 AS 或 ATTACH 套用时。
- 直接写作记录的成员、不嵌套在语句里、又没有名字的定义，不会被用到，给警告。嵌套在语句里的匿名定义照常使用。
- **并入**：DEFINE-SYMBOL 或 DEFINE-STYLE 的定义体里引用同类的定义，就是在该处并入它的内容。有键的成员后写的覆盖先写的；没有键的成员累加。没有 `base=` 参数。
- 普通定义之间不并入，用体内的 AS、ATTACH 组合。派生和扩充也这样做；没有继承关系（9.4）。

9.2 符号：DEFINE-SYMBOL

9.2.1 求值

- 符号是值：定义体在定义处求值，体内的语句按符号的局部坐标产生引脚和图元。它们成为定义的内容，不提交给场景。
- 成员的顺序就是绘制顺序。
- 局部坐标与场景同单位、同轴向（x 向右、y 向上），(0, 0) 是符号的原点（6.5）；本版没有缩放（11.7）。体内的点两轴都应写数，不能用 `_`。

9.2.2 体内的语句

- 体内只能写：图元、引脚、命名的局部点（如 `%tip = (0, 0.7)`）、并入别的符号、只用来画的用例（用 ATTACH 或 AS，9.4）以及写在这种用例上的 SET-SYMBOL 和 ORIENT（9.2.7），还有附着在所属对象 `?`（即用这个符号画出的对象）上的 TEXT 以及给这种文字定位的 PLACE（13.3）。
- 其他语句失效，被丢弃（体是容错的）；宿主引用也失效。
- 引脚只能写在 DEFINE-SYMBOL 体里。图元也可以写在场景里，那时用场景坐标，是标注（13.4）。

9.2.3 参数

- `box=[x1, y1, x2, y2]`：符号的外框，用局部坐标，轴对齐，随元件的朝向变换。语言本身不用它；查看器通常拿它当选择框，求解器当符号互不重叠的边界，怎样用由实现决定。不写时取符号里除文字以外全部图元的外接框，因为文字的大小随字体和渲染器变。

9.2.4 引脚

```
$RES_BOX = @DEFINE-SYMBOL {
  %p = (0, 3) @PIN(terminal=1, label="P", direction="up");
  %n = (0, -3) @PIN(terminal=2, label="N", direction="down");
  @LINE {%p (0, 1.5)};
  @LINE {(0, -1.5) %n};
  (0, 0) @RECTANGLE(width=1.2, height=3)
}
```

- 引脚是绑定到端子的点。命名后可以当点用，如上例的引线从 `%p` 画起、画到 `%n` 为止。
- 建议引脚放在整数点上（11.7）：元件放在整数点上时，它的引脚和接上的导线也都落在整数点上。放在别处不是错误，实现可以给提示。符号里的图元不受此限。
- `terminal=`：用这个符号画出的对象的一个端子，写序号、DEVICE 给的端子名（6.12）或子电路的端口名；用例则写它的定义的角色名或角色的序号（6.4）。一个符号里一个端子只有一个引脚：引脚以端子为键。

- **总线引脚**：`terminal=` 可以收数组，一个引脚对几个端子，如子电路的总线端口：`(-4, 0) @PIN(terminal=["D<3>", "D<2>", "D<1>", "D<0>"], label="D<3:0>", direction="left")`。数组至少有一个元素，每个元素是序号或名字。参数里的字符串不展开下标（6.11），所以名字要逐个写。
  - 这几个端子都画在这个引脚上：导线或约束接到其中任何一个，就是接在这个引脚上；导线接到其中任何一个时，这几个端子都算接了线（10.2）。
  - 同一个端子写在几个引脚里时，引脚按端子覆盖：后写的引脚得到它，前面的引脚保留自己其余的端子（9.2.7）。
  - 叠在同一点的几个引脚代替不了总线引脚：坐标相同不算接上（13.1），导线只接到其中一个引脚，其余的仍是 `wired=false`，标准包的补位规则会给它们补标签、补地。
- `label=`：引脚名，只用于显示，不能用来引用端子（6.4）。显不显示、显示在哪里由查看器决定。可以用文字记号，如 `\overline{CS}`（13.6）。
- **不画的端子就没有引脚**。没有引脚的端子没有绘图意义：导线或约束接到它上面时失效（"符号没有这个端子的引脚"），补位也不选它（10.1）。没有把有引脚的端子藏起来的写法；不想画衬底，就用一个没有衬底引脚的三端符号。
- `direction=`：导线离开引脚的方向，取 `"right"`、`"up"`、`"left"` 或 `"down"`。它写在符号的局部坐标里，随朝向一起转（11.2）。它是给求解器和查看器的提示，如自由走线从哪个方向离开引脚、电流在端子上的走向（15.2），或选朝向时的参考；它不是约束，不参加冲突。可以不写，不写时引脚没有方向，实现可以按图形推断一个。
- 没有引线长度：引线用 LINE 画在符号里。
- 没有电气类型：引脚和端子不设 input、output 之类的参数，网表里也没有这类信息。需要时，用户用语义定义的用例表达；导出到 Virtuoso 之类工具的导出器，按自己的配置把普通定义对应到电气类型。
- 引脚只是一个点：画不画引线由符号决定，引脚的点怎样标出来（小方框、圆点或不标）由查看器决定。所以没有引线的引脚不用专门的写法：写 PIN、不写 LINE 就是。例如原图把导线接到编辑器隐藏的衬底引脚上，就导入成保留衬底 PIN、不画衬底引线的符号。一般的符号，包括标准包里的，都画引线。

9.2.5 图元

位置是点对象，标量是参数。表中写法里的 *point* 代表任何点对象。

写法	含义
`@LINE {point point …}`	两个点是线段，更多是折线；最后一个点与第一个点位置相同时图形闭合，接头按转角画
`@RECTANGLE {point point}`	矩形；两个点是对角，不分先后
`(0, 0) @RECTANGLE(width=4, height=2)`	矩形；点是中心，`width`、`height` 是宽和高
`@CIRCLE {point point}`	圆；第一个点是圆心，第二个点在圆上
`(0, 0) @CIRCLE(radius=1)`	圆；点是圆心
`@ELLIPSE {point point}`	椭圆；两个点是外接框的对角
`(0, 0) @ELLIPSE(width=4, height=2)`	椭圆；点是中心，`width`、`height` 是外接框的宽和高
`@ARC {point point point}`	圆弧，从第一个点经过第二个点到第三个点
`@BEZIER {point point point point …}`	三次贝塞尔曲线，3n+1 个点：起点，然后每段两个控制点和一个终点；相邻两段共用端点
`(1, 0) @TEXT("…")`	附着在点上的文字：符号的固定文字，不能按元件改变
`? @TEXT("${name}", key="name") => @PLACE(1, 0)`	附着在所属对象（用这个符号画出的对象）上、读它属性的文字；引用名是 `name`，所以对 M1 就是 `#M1::name`，场景里引用名相同的文字按项覆盖它（13.3）

- **两种写法**：RECTANGLE、CIRCLE、ELLIPSE 按有没有写尺寸参数区分。写了尺寸（RECTANGLE、ELLIPSE 的 `width` 和 `height`，CIRCLE 的 `radius`），对象槽位收一个点，就是中心；收到集合时每个点各画一个（5.2.1）。没写尺寸，对象槽位收两个点的集合，点数不对时这条图元失效。尺寸只写了一部分是参数写错，这条图元失效。两点的写法里点是对象：可以命名，也可以是引脚的位置，图形跟着点走。
- 矩形和椭圆都是轴对齐的；斜的矩形用闭合的 LINE 画。

- **填充**：除 TEXT 外的图元都收 `fill=`。不闭合的 LINE、ARC、BEZIER，按从最后一个点到第一个点连一条直线围成的区域填，这条连线不描边；填充的 ARC 是弦和弧围成的弓形。只填充、不描边的图形（MOS 的栅极、箭头、汇合圆点）写 `fill="linecolor", linestyle="none"`（9.3）。
- **线端箭头**：LINE、ARC、BEZIER 收 `arrow=`：`"none"`（默认）、`"start"`、`"end"` 或 `"both"`，如 `@LINE(arrow="end") {%from %to}`。箭尖落在起点或终点上，指向线在这一端的走向：LINE 取首段或末段，ARC 取圆弧在端点的切线，BEZIER 取从相邻控制点到端点的连线（两点重合时取下一个不重合的点）。箭头的长度由 `arrowsize` 给（9.3），形状由实现决定。端点是点对象；被约束挪动时箭头跟着走，可以是任意角度，而画在点上的箭头用例只有 8 种朝向。WIRE 不收 `arrow=`。
- **样式参数**：图元可以直接把样式属性写成参数（9.3）：LINE、ARC、BEZIER 收 `linecolor`、`linestyle`、`linewidth`、`cap`、`join`、`fillcolor` 和 `arrowsize`；RECTANGLE、CIRCLE、ELLIPSE 收前六个；TEXT 收 `textcolor`、`size`、`font`、`bold` 和 `italic`。写了的是这个图元直接的样式，高于所属对象传来的样式（8.3），所以符号里写死的颜色不随主题变；没写的取所属对象的样式，所以换主题不用改符号。填充用 `fill=` 写：`"none"`（默认）、`"linecolor"`（用 `linecolor` 的颜色）或 `"fillcolor"`（用 `fillcolor` 的颜色）。
- **不设**：SVG 路径字符串、POLYGON（用闭合的 LINE）、单独的箭头图元（线端的箭头见上；别处用填充的闭合 LINE，或画在点上的箭头用例，9.2.7）、圆角矩形、椭圆弧（用 BEZIER 近似）和图片。
- **文字**：TEXT 另有 `size=`（字号，11.7）、`halign=`（水平：`"start"`（默认）、`"middle"`、`"end"`）、`valign=`（竖直：`"baseline"`（默认）、`"top"`、`"middle"`、`"bottom"`）和 `rotate=`（0、90、180 或 270，默认 0）。符号体里附着在 `?` 上的 TEXT 引用所属对象的属性，附着在点上的是固定文字；场景里的 TEXT 按附着的东西分成标注和标签（13.3、13.4）。文字记号见 13.6。
- **基线**：`valign="baseline"` 时，锚点在第一行的基线上，后面几行往下排，与 SVG、PostScript 和多数编辑器相同；`"top"`、`"middle"`、`"bottom"` 按整个文字框对齐。基线是默认。
- **文字的方向**：锚点就是文字自己的位置（13.3）。文字框按 `halign`、`valign` 放在锚点旁边，绕锚点按 `rotate` 转；符号里的文字再随元件的朝向变换，旋转、镜像都算。框里的字总是正读：横排从左往右，竖排从下往上，字形从不镜像。所以 `rotate=180` 按 0 显示、270 按 90 显示，文字框在锚点的另一边；元件镜像时锚点照常镜像，文字框沿镜像的方向移到锚点的另一边，但字不镜像。例：NMOS 的名字锚点在 (1.5, 1.5)，`halign="start"`，往右展开；`"MY"` 时锚点到 (−1.5, 1.5)，文字往左展开，离开本体；`"R90"` 时锚点到 (−1.5, 1.5)，文字竖排，从下往上读。场景里的文字不随元件转。

9.2.6 子电路符号

写在子电路 `.subckt ota5t … .ends` 之内：

```
%ota5t = @DEFINE-SYMBOL(box=[-4, -4, 4, 4]) {
  (-4, 1) @PIN(terminal="inp");
  (-4, -1) @PIN(terminal="inn");
  (4, 0) @PIN(terminal="out");
  @LINE {(-4, -4) (-4, 4) (4, 0) (-4, -4)};
  ? @TEXT("${name}", key="name") => @PLACE(-1, 0)
};
# @SET-SYMBOL %ota5t
```

- 子电路给自己的 `#` 指派的符号，就是该子电路的元件不论在哪里的默认外观，即它的子电路符号。引脚按名字对应子电路的端口。
- 它是原则 2.5"定义只向下生效"这条规则的例外：它作用于实例化该子电路的各层。
- 子电路的总线端口在网表里已拆成单个端口（6.11）；子电路符号用总线引脚把它们画成一个（9.2.4）。
- 在符号队列里，它排在根上的包的规则之后，在各层的规则（包括在某层用到的包）之前，所以上层可以用规则或给单个元件写的 SET-SYMBOL 换掉它（8.3）。
- 没有元件实例化文件层，所以写在文件层的 `# @SET-SYMBOL` 不起作用。
- 子电路没有子电路符号时，它的元件怎样画由查看器决定，本标准不规定。一个子电路里有几条 `# @SET-SYMBOL` 时，它们在符号队列的同一组里，按文本先后，后写的生效（8.3）；实现可以给警告。

9.2.7 变体与片段

```
$NMOS_BULK = @DEFINE-SYMBOL(box=[-3, -3, 2, 3]) {
  $NMOS;
  %b = (1, 0) @PIN(terminal=4, label="B", direction="right");
  @LINE {(0, 0) %b}
}
```

- 在体内并入另一个符号再添改，就是变体：引脚按端子覆盖（总线引脚按其中各个端子算，9.2.4），图元累加。
- 只在本层用的片段用 `%` 命名，如 `%MOS_CHANNEL = @DEFINE-SYMBOL {…}`，别的符号体里写 `%MOS_CHANNEL;` 并入它。
- 图元没有键：并入只加不减，并入的图元去不掉。要少画一部分，就另定义一个符号：把共用的部分拆成片段，各个符号并入自己要的部分，如 `$NMOS3 = @DEFINE-SYMBOL {%MOS_BODY}` 和 `$NMOS_4TERM = @DEFINE-SYMBOL {%MOS_BODY; %BULK}`。
- **体里的用例**：需要放在不同位置或不同朝向的图形，如箭头，写成画在点上的用例：

```
$ARROWHEAD = @DEFINE(roles=[["p", "point"]]) {};
$ARROWHEAD_SYM = @DEFINE-SYMBOL {
  @LINE(fill="linecolor", linestyle="none") {(0.4, 0) (-0.4, 0.3) (-0.4, -0.3) (0.4, 0)}
};
$ISRC_DOWN = @DEFINE-SYMBOL {
  (0, 3) @PIN(terminal=1, direction="up");
  (0, -3) @PIN(terminal=2, direction="down");
  @LINE {(0, 3) (0, 1.5)};
  @LINE {(0, -1.5) (0, -3)};
  (0, 0) @CIRCLE(radius=1.5);
  @LINE {(0, 0.9) (0, -0.5)};
  $ARROWHEAD @ATTACH (0, -0.6) => @SET-SYMBOL $ARROWHEAD_SYM => @ORIENT("R270")
}
```

  - 第一个角色填体里的一个点，用局部坐标。用例按它的符号画在这个点上，朝向绕这个点转，再随所在的符号一起变换；嵌套用例的变换逐级向外合成，所以箭头跟着器件转。
  - 只用来画：外面的规则看不到符号体里面，所以用例的符号要在体里用 SET-SYMBOL 指派；没指派符号的用例不画，给警告。普通定义的体在符号里不展开，约束和导线在符号里没有意义；这里的 ORIENT 是符号内部的数据，不参加冲突，没有优先级。
  - 场景里同样的写法就是画在点上的虚拟元件（12.1），规则可以给它指派符号；风格包的规则伸不到符号里面。
- 引脚名和公式：引脚名显不显示由查看器决定（9.2.4）；公式就是引用属性的 TEXT（13.6），没有特别的规则。文字记号见 13.6。

9.2.8 符号与对象相配

符号不分"给器件的"和"给用例的"。一个符号能不能用于某个对象，看它的引脚对不对得上对象的端子。

9.3 样式：DEFINE-STYLE

```
$ink = @DEFINE-STYLE(linecolor="#202020", textcolor="#202020", fillcolor="#ffffff", size=1.2) {};
$accent = @DEFINE-STYLE(linecolor="#1f4fd8", linestyle="bold") {$ink};
$shade = @DEFINE-STYLE(fillcolor="#1f4fd833") {}
```

- 括号里的参数是这个样式自己的属性，最后生效。集合里只能引用别的样式，在该处并入它们的属性，后写的覆盖先写的。没有基样式的样式写 `{}`。
- 样式只在被 SET-STYLE 指派时生效（11.3）。同样的属性也能作参数直接写在图元、TEXT 和 WIRE 上（9.2.5、13.1），按 8.3 的两级取值。
- 样式只管描线上色：位置、朝向、形状（符号或图元的点）和文字的内容都不属于样式。
- **属性**：

类	属性	取值
线	`linecolor`	颜色；也是 `fill="linecolor"` 的颜色

类	属性	取值
	`linestyle`	`"solid"`（默认）、`"dashed"`、`"dotted"`、`"dash-dot"`、`"double"`（两条平行的实线）、`"bold"`、`"none"`（不画线）
	`linewidth`	线宽，坐标单位（11.7）；`0` 是在任何缩放下都一样细的细线，也是默认
	`cap`	线端：`"butt"`、`"round"`、`"square"`
	`join`	转角：`"miter"`、`"round"`、`"bevel"`
填充	`fillcolor`	`fill="fillcolor"` 的颜色
文字	`textcolor`	文字的颜色；不给时取 `linecolor`
	`size`	字号，坐标单位（11.7）
	`font`	字体名，只是提示；查看器没有这个字体就用自己的
	`bold`、`italic`	整段文字的粗体、斜体，`true` 或 `false`；记号 `\textbf`、`\textit` 照样能给局部加上（13.6）
其他	`arrowsize`	线端箭头的长度，坐标单位（9.2.5）

- 线只有颜色、线型、线宽、线端、转角五个属性。`"bold"` 画两倍的线宽，线宽为 0 时画两倍的细线；虚线的划长和间隔、双线的间距由实现决定，随线宽成比例。以后的语言版本可以给 `linestyle` 增加取值（16.1）。
- `"none"` 不画线，线端的箭头也不画（箭头属于线，9.2.5）；`fill="linecolor"` 照样取 `linecolor` 的颜色，所以只填充、不描边的图形写 `fill="linecolor", linestyle="none"`，它的颜色照样随主题变。导线和图元都收它；导线写了它就是一根看不见的导线，但仍是导线（13.1）。
- **颜色**照 CSS 的十六进制写法：`"#RRGGBB"` 或 `"#RRGGBBAA"`，或简写 `"#RGB"`、`"#RGBA"`。透明度在后：`00` 全透明，`FF` 不透明；不写就是不透明。线和填充各有各的颜色，所以透明度也分开设，如上例 `$shade` 只让填充部分透明。本版不收颜色名和 `rgb()`。颜色是字符串。
- 本版不设：图案填充、渐变、阴影；可见性 `visible`（显示和隐藏是查看器的事，16.2）；主题、profile 这类对象（主题就是一个带样式和规则的包，用 USE-PACKAGE 引入，16.2）。
- 某个属性没有任何样式或参数给值时，用渲染器自己的默认值，这不是错误（原则 2.7）。标准包给出可移植的默认外观（16.3），用户写的规则和语句覆盖它。
- 不认识的属性与语法错误同等处理（原则 2.3）。

9.4 普通定义：DEFINE

```
$TAIL_MID = @DEFINE(roles=["a", "b", "t"]) {
  %mid = (_, _);
  {?a %mid ?b} @DISTRIBUTE("x");
  {%mid ?t} @ALIGN("x")
};
$CHAIN_ROW = @DEFINE(roles=["source", "{chain}", "load"]) {
  {?source ?chain ?load} @ORDER("x")
};
$GND_BELOW = @DEFINE(roles=[["t", "terminal"]], parameters=[["gap", 2]]) {
  %g = $GROUND @ATTACH ?t::net;
  ?t::net @WIRE {?t %g:p};
  {?t %g:p} @ALIGN("x");
  %g:p ?t @DISTANCE("bottom-to-top", exact=?gap)
};
%c1 = #M1:3(a) #M2:3(b) #M5:1(t) @AS $TAIL_MID;
#R1(source) {#R2 #R3}(chain) #R4(load) @AS $CHAIN_ROW;
# @SELECT-TERMINAL(net="vss", wired=false) => $GND_BELOW @ATTACH(gap=3)
```

- 普通定义由角色表、参数表和定义体组成。体内用 `?name` 引用角色和参数（6.10）。定义做什么由它的内容决定，见 9.5。
- **套用**：AS 和 ATTACH 把对象填进定义的角色，产生一个用例。两个关键字做同一件事，只是定义写的位置不同：
  - AS 的定义写在最后，读作"这些对象，作为某某"，用于语义和摆法，如 `#M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR`；

  - ATTACH 的定义写在最前，在写出的对象之前，标准写法把关键字放在定义后面，如 `$GROUND @ATTACH &vss`；读作"某某，添在这些对象上"，用于虚拟元件和补位（第 12 章、10.5）。
  - 挑了另一个关键字不算错，只是读着别扭；定义写错位置才是错误（4.2 第 7 条）。管道送来的对象直接按角色配位，两个关键字写出来一样；这时照定义的用途挑。
  - 怎样挑：对象合起来构成这个东西时用 AS，如电路结构、摆法、由这几个对象组成的局部画法；这个东西添在对象上时用 ATTACH，如地、端口符号、无连接标记、汇合圆点、每个元件一个的标签、补位。标准例句分别是 `#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR` 和 `$GROUND is @ATTACH to &vss`（4.5）。
- **角色表**：参数 `roles`，是一个数组，每项是角色名，或 `[name, kind]` 对。DEFINE 的两个参数都只能写名字。
  - 写成 `"{name}"` 的项是集合角色，整个收下一个集合；其余是单个角色（5.2.1 第 4 条）。
  - 集合角色收到单个对象，当作只有它的集合；收到空集合也合法，体里成员不够的约束自然成立（第 14 章）。
  - 单个角色收到集合，就对每个成员各套用一次，空集合就是零次；一次套用里至多一个单个角色可以收到多成员的集合（否则语句失效，5.2.1）。
  - 集合角色的 `?name` 写在 `{…}` 里时，集合打散一层，所以上例的 ORDER 排的是 R1、R2、R3 和 R4。
  - 集合角色收到含集合成员的集合时原样收下，由体决定怎样用。
  - 集合角色不对应端子（6.4）。
  - 一次套用里每个角色只能标一次；要给集合角色多个对象，就写成集合，如上例的 `{#R2 #R3}(chain)`。
  - **种类**：写成 `[name, kind]` 的项声明这个角色收哪种对象；只写名字的项收任何种类。种类名就是词表里的（5.3）：`"instance"`、`"terminal"`、`"net"`、`"point"`、`"wire"`、`"usage"`、`"bundle"`、`"text"`、`"annotation"` 和 `"label"`。在角色声明里，`"instance"` 也收叠画，`"terminal"` 也收叠画的端子（11.8）；写成数组表示几种之一，如 `["p", ["net", "terminal"]]`。集合角色的种类管它的每个成员，如 `["{members}", "instance"]`。没有"某个定义的用例"这一种。
  - 角色收到的对象种类不对时，整个套用失效，诊断写明哪个角色、要什么、收到了什么；集合角色里有一个成员种类不对也一样。选择逐个产生套用时，只有收错对象的那一次套用失效。
  - 注：体为空的定义，没有什么检查角色填的是什么，所以只有声明了种类，才查得出填错的角色；体不为空时，展开出的每条语句也各自检查（9.5）。声明了种类，体里对角色的用法也能在定义处检查（6.9）。
  - **可选的角色与个数**：单个角色应填上；集合角色可以不填，不填就是空集合。按顺序给出对象而对象少于角色时，剩下的角色应都是集合角色；给对象标角色时，没写的集合角色是空集合。没有最少、最多个数：成员不够的约束自然成立（第 14 章）。形状不同的变体写成另一个定义（见下面的派生与扩充）。同一对象填两个角色、期望的连接这类合法性规则，见 9.5。
- **参数表**：参数 `parameters`，可以不写。每项是参数名，或 `[name, default]`。
  - 参数在 AS、ATTACH 的括号里传入，如 `$GND_BELOW @ATTACH(gap=3)`，按参数表的顺序按位置写，或写名字；两种都用时，按位置的在前（4.2 第 9 条）。标准写法写名字（4.3）。名字不在参数表里，这次套用失效。选择逐个产生套用时，每次套用都用同一组参数。
  - 体里能写字面量的地方都能改写参数的空位：参数的值、数组的元素、点的坐标。所以在定义体里，"参数里只放字面量"（4.1）这条规则成了"字面量或参数的空位"；展开后它们仍是字面量。AS、ATTACH 自己的参数只能是字面量，不能传对象；对象经角色传入。
  - 没传、也没有默认值的参数，在用到它的地方当作没写：作具名的参数时就是没写这个参数，如 `@TEXT(?label, size=?size)` 里的 `size=?size`，这时文字用 TEXT 自己的默认字号或它的样式的字号（11.3）；作点的坐标时就是 `_`；写在按位置的参数里或数组的元素里没有确定含义，那条语句失效。所以不需要"必填"标记。
  - 参数用于数值、文字这类量；结构不同的变体建议写成独立的定义，语言不强制这一点。参数只传进体里；不能按参数的值选择用例。
- 角色名和参数名在同一张表里，应各不相同；它们都不能是标准子对象名（6.5）。参数不能叫 `priority`，它是 AS 和 ATTACH 的标准参数（14.8）。
- 体内绑定的 `%` 名字是局部名：每次套用都为它们生成新的对象，从外面经用例引用（见下）。几次套用共用的对象，如对称轴，经角色传入。

- 套用可以给对象标角色（`#M1(a) #M2(b) … @AS $T`、`$T @ATTACH #M1(a) #M2(b) …`），也可以按角色表的顺序给出对象（`#M1 #M2 … @AS $T`）。单个角色没填，或给的对象多于角色时，整条语句失效；没填的集合角色是空集合。
- 一次套用产生一个用例，它记下定义和每个角色填了什么，可以命名（如 `%c1`）。删掉这条语句，展开出的描述也一起消失。用例没有键：每次套用都是一个单独的对象（5.8）。
- **展开**：
  - 空位换成实参和参数值；
  - 局部名换成新句柄"用例 + 局部名"，写作 `%c1::mid`；匿名的用例在诊断里按语句的位置显示；
  - 体外的名字不在套用处重新解析（6.9）；
  - 展开的结果在 AS 或 ATTACH 语句的位置提交，冲突取舍时也取这个位置（8.5）。
- 嵌套的用例逐层往下取（`%c1::inner::mid`）。用例也可以填进别的定义的角色：有了自己写的定义 `$OTA = @DEFINE(roles=[["pair", "usage"], ["load", "usage"]]) {}`，`#M1(in_p) #M2(in_n) @AS $DIFFERENTIAL_PAIR => %cm1 @AS $OTA` 是一个放大器，它的角色 pair 填的是差分对的用例，角色 load 填的是电流镜 `%cm1`（5.5）。
- **循环套用**：展开中的一次套用，若直接或经别的定义套用了某个外围用例正在展开的定义，就闭合了一个循环。这次套用失效，诊断列出整条链；其余照常生效。下例把 `$A` 套用到 M1 时展开 `$B`，其中对 `$A` 的套用闭合了循环 `$A → $B → $A`，失效；`$B` 里的 ORIENT 照样把 M1 翻转：

错例

```
$A = @DEFINE(roles=["x"]) {?x @AS $B};
$B = @DEFINE(roles=["x"]) {
  ?x @AS $A;                /* closes the cycle $A → $B → $A: this application fails */
  ?x @ORIENT("MY")
};
#M1 @AS $A
```

- **从外面引用局部对象**：

```
$DP_BLOCK = @DEFINE(roles=["in_p", "in_n"]) {
  %pair = {?in_p ?in_n} @GROUP
};
%dp1 = #M1(in_p) #M2(in_n) @AS $DP_BLOCK;
%c1::mid @PLACE(x=20);              /* moves the midpoint of c1 */
{%c1::mid #M9:1} @ALIGN("x");       /* aligns the drain of M9 with the midpoint of c1 */
%dp1::pair @PRIORITY("disabled")    /* turns off only the group expanded for dp1 */
```

  - 体里每个起了名的局部对象，都能经用例从外面引用，写作 `usage::local_name`；没起名的够不着，只有求解器的记录能用编号引用它们（5.9）。起不起名由定义的作者选择，与"要引用就命名"（5.7）一样；不另设导出表。
  - 路径可以连写：`%st1::dp::mid` 是 st1 体里的局部用例 dp 的 mid。经角色走也一样：`%st1::pair::mid` 是填进角色 pair 的那个用例的 mid。
  - 它们是普通的引用：可以加事实（PLACE、SET-STYLE、TEXT、PRIORITY 等），可以用在别的语句里，可以填进别的定义的角色；与展开出的事实冲突时按 8.5 取舍。不能从外面给局部名重新绑定：`%c1::mid = …` 是错误。
  - 名字：经 `::` 从一个用例取到的名字应唯一。所以局部名不能与角色名或标准子对象名相同，这在定义处检查；用例上文字的引用名也不能与这些名字中的任何一个相同（6.5、13.3）。局部名可以与参数名相同：参数是值，`::` 不取值。
  - 优先级：从外面直接写在局部对象上的 PRIORITY 语句，盖过用例的优先级（14.8）。
  - 够不着的：匿名用例的局部对象（求解器的记录可以用编号引用，5.9），以及选择逐个生成的用例的局部对象。以用例为范围的选择仍只选它的角色成员，不选局部对象（10.1）：局部对象只能按名字引用，包里的规则看不到定义内部。
  - 所以局部名是定义的接口的一部分：定义改了局部名，外面的引用就找不到它，失效（第 7 章），求解结果随之过时，并给提示建议重新求解（第 17 章）。标准包写明它的定义中哪些局部名可以依赖。

- **体内的写法**：角色和参数写 `?name`，局部名写 `%name`（6.9、6.10）。
- **派生与扩充**：只用组合，没有继承关系（9.1）。派生定义的体里套用基础定义，再加自己的语句：

```
$FILTER = @DEFINE(roles=["in", "out"]) {};
$LOWPASS = @DEFINE(roles=["in", "out", "r", "c"]) {
  %f = ?in(in) ?out(out) @AS $FILTER;
  {?r ?c} @ORDER("x")
}
```

  - 增加内容靠体里的语句，数值上的变体靠参数；要在下层遮蔽并扩充一个定义，体里引用外层的同名定义（6.9）。
  - 按基础定义选择不需要特别的规则：派生定义展开出的 FILTER 用例本来就在场景里，所以 `# @SELECT-USAGE $FILTER` 选得到它，按 `$FILTER` 写的规则对低通滤波器同样起作用，只作用在里面那个 FILTER 用例上，不作用于派生定义自己加的语句。
  - 没有专门去掉基础定义一部分的机制：派生定义的体可以写 `%f::x @PRIORITY("disabled")` 关掉其中一部分，但外层用例上的优先级会盖过它（14.8）；要稳定地少一部分，就把基础定义拆小，或另写一个。

9.5 普通定义的用途

定义做什么由它的内容决定，不由关键字决定：

```
$PAIR = @DEFINE(roles=["in_p", "in_n"]) {};                  /* meaning: roles only */
$SIDE_BY_SIDE = @DEFINE(roles=["a", "b"]) {                 /* layout: constraints in the body */
  {?a ?b} @ALIGN("y");
  {?a ?b} @ORDER("left-to-right")
};
$PROBE = @DEFINE(roles=["p"]) {};                            /* virtual part: a rule gives its
usages a symbol (chapter 12) */
$PROBE_SYM = @DEFINE-SYMBOL {(0, 0) @PIN(terminal=1); (0, 0) @CIRCLE(radius=0.5)};
# @SELECT-USAGE $PROBE => @SET-SYMBOL $PROBE_SYM
```

标准包里的 `$CURRENT_MIRROR`、`$MIRRORED` 和 `$GROUND` 分别属于这三种；`$DIFFERENTIAL_PAIR` 有它的角色，体里还有一个默认的布局（见下）。

- 它们都是同一种东西：一个用例（记下套用了哪个定义、每个角色填了什么），加上定义体展开出的描述。两部分的使用者不同：
  - 体展开出的约束由求解器处理，必须满足，冲突时按 8.5 取舍；
  - 用例给人和 LLM 读，也给 RNL 自己的选择和规则用（10.1）。

  所以布局定义的用例同样带有语义；体为空的定义，如 `$PAIR`，只用角色记录含义，它的含义由标准包或它的作者写明。记录含义的定义，体里也可以带画法（见下）。
- 语义用例不是注释：它的引用应找得到，角色应对得上，基本合法性受检查（原则 2.4），选择和规则也能用它；不规定的只是怎样画。
- "是什么"和"怎么画"可以写在一个定义里，也可以写成两个。定义可以在体里带一个默认的画法，优先级用 `"low"`：作者自己写的事实排在它前面，用例的优先级能关掉或抬高它，如 `%dp1 @PRIORITY("disabled")`，或者对这个定义的每个用例写 `# @SELECT-USAGE $DIFFERENTIAL_PAIR => @PRIORITY("disabled")`，用例仍记着这些对象是什么（14.8）。从语义到画法这一步也可以写成包里的规则，换包就换了画法：`# @SELECT-USAGE $DIFFERENTIAL_PAIR => @AS $DP_LAYOUT`；删掉布局的用例不影响语义的用例。标准包把默认画法写在定义体里，其中的约束优先级都是 `"low"`，如差分对的尾管放在两管中间、`$BRANCH` 的成员同一列、`$CHAIN` 从左到右（16.3）。
- **画法都写成 RNL**：某种语义画成某个样子，只能靠 RNL 写出来，写在它的定义体里，或写成布局定义再用规则套用，如上一条，可以放在包里；不要依赖渲染器对某种语义的默认约定。定义带不带画法由它的包决定，本标准不另作规定。RNL 没有描述的部分怎样摆由实现决定（原则第 1 节），但那不是对某种语义的画法约定，不能依赖。渲染器决定没有描述的对象的默认符号和样式（原则 2.7），是另一回事。
- **词汇归标准包**：首批语义定义和标准虚拟元件都是标准包的内容，标准包不属于语言标准（16.3）。标准包里的定义怎样写明含义和角色，也由标准包决定。用户的定义的含义就是它的体，名字只给人读。

- **基本合法性**（原则 2.4）：定义的语法不检查两个角色是否填了同一个对象。一次套用的基本合法性是：引用找得到，角色收到的种类对，单个角色都已填上（9.4），以及展开出的每条语句自身合法。

  错例

  ```
  $DP_LAYOUT = @DEFINE(roles=[["in_p", "instance"], ["in_n", "instance"]]) {
    {?in_p ?in_n} @ALIGN("y");
    ?in_p ?in_n @DISTANCE("x", min=4)
  };
  #M1(in_p) #M1(in_n) @AS $DP_LAYOUT      /* the expanded DISTANCE asks M1 to be at least 4 from
  itself */
  ```

  - 上例展开出的 DISTANCE 自相矛盾：它失效并给诊断，只丢弃这一条（5.5、7.3）；ALIGN 照常成立。体为空的定义，如上面的 `$PAIR`，查不出这种填法：`#M1(in_p) #M1(in_n) @AS $PAIR` 不报错。
  - 两端接在同一网络上的虚拟元件有确定的含义，所以"同一对象填两个角色"一般不能算错误。
  - 更多的检查，如要求两个角色不同，本版不提供。
- **关键字与普通定义**（1.5 第 7 条）：实现必须自己懂、用别的描述写不出来的，是关键字，如原子约束、GROUP、WIRE、TEXT、CURRENT；能写进普通定义的体的，是普通定义，如差分对、电流镜、地和电源、信号流向（14.6）。DEFINE-SYMBOL、DEFINE-STYLE、SET-SYMBOL、SET-STYLE 也是关键字：符号体用局部坐标，在定义处求值；样式的属性是字面量；SET-SYMBOL、SET-STYLE 给已有的对象设一个属性，在全部对象都齐了以后在队列里决定，而 AS、ATTACH 每次产生一个新对象。
- 定义按它的名字解析到的定义认，不按名字字符串：下层自己定义了 `$DIFFERENTIAL_PAIR`，那就是另一个定义，包里按标准的 `$DIFFERENTIAL_PAIR` 写的规则不认它。

10 选择、规则与逐个生成

本章讲选择关键字、用到它们的规则，以及逐个生成。

10.1 选择关键字

选出的每种对象各有一个选择关键字，每个的条件固定。选择的结果是集合（5.2）。

每个选择关键字收一个范围对象。`#` 表示求值时所在的那一层（10.3）。给一个对象或集合，就在这些对象里选：比如给网络的集合时，SELECT-TERMINAL 选接在这些网络上的端子（怎样从范围走到要选的种类，见表后）。范围不能省略，所以简单的选择也写 `#`：`# @SELECT-INSTANCE(name="R*") => @SET-SYMBOL $RESISTOR`。

关键字	选出	条件
SELECT-INSTANCE	宿主元件。范围里给端子时，选它们所属的元件；给网络时，选在上面有引脚的元件	`name`（元件名，如 `name="R*"`）；`model`（元件行上写在模型位置的名字：`.model` 名或子电路名）；`type`（`.model` 里的类型，如 `"nmos"`）
SELECT-TERMINAL	宿主元件的端子；只选所属元件的符号里有引脚的	所属元件的 `name`、`model`、`type`；`terminal`（端子名或序号）；`net`（所在网络的名字）；`wired`（10.2）；`current`（电流方向，15.2：`"in"`、`"out"` 或 `"unspecified"`）
SELECT-NET	网络。范围里给元件、端子或导线时，选它们所在的网络	`name`；`port`（是否本层的端口）；`wired`（10.2）；`global`（是否全局网络，6.3）
SELECT-WIRE	导线。范围里给网络时，选画在这些网络上的导线	`net`（导线所在网络的名字；总线有一个网络对上就算）；`bus`（`true`：只选总线，13.1；`false`：只选不是总线的导线，即网络槽位收到的是一个网络、不是集合）
SELECT-LABEL	标签，即附着在对象上的 TEXT（13.3）。范围里给对象时，选附着在它们上面的标签	`key`（文字的引用名；收 `*`，区分大小写；没有引用名的标签对不上任何 `key=`）
SELECT-ANNOTATION	标注，即场景里的图元和附着在点上的 TEXT（13.4）	除 `role` 和 `recursive` 外没有别的条件
SELECT-USAGE	`scope @SELECT-USAGE $D`：范围里定义 `$D` 的用例，往里找。给 `#` 时是本层的用例；给用例时是这些用例本身和它们里面的用例。不写 `$D` 时是任何定义的用例	定义是最后一个对象（5.6），可以省略。除 `recursive` 外没有别的条件
SELECT-ANCESTOR-USAGE	`objects @SELECT-ANCESTOR-USAGE $D`：把这些对象当成员的 `$D` 的用例，往外找；间接的成员也算	定义是最后一个对象（5.6），不能省略。除 `recursive` 外没有别的条件

除此之外，每个选择关键字都收条件 `recursive`，前六个还收 `role`，它只在范围里有用例时才有意义（两者都见下）。

- **每种一个关键字。** 每个选择关键字按它选出的种类命名，如某个网络的导线写 `&clk @SELECT-WIRE => @SET-STYLE $signal`，M1 的标签写 `#M1 @SELECT-LABEL`。没有一种选择能选出所有种类的对象：要覆盖每一种，就每种写一条规则，标准包给每个对象配基础样式就是这样做的。
- **范围怎样走到要选的种类。**范围里的每个对象朝要选的种类走一步；走不到的对象不选出东西，这不是错误：
  - SELECT-INSTANCE：元件就是它自己；端子走到所属的元件；网络走到在上面有引脚的元件。
  - SELECT-TERMINAL：元件走到它的端子；网络走到接在上面的端子；端子就是它自己。
  - SELECT-NET：网络就是它自己；元件走到它各个端子所在的网络；端子走到它所在的网络；导线走到它所在的网络。
  - SELECT-WIRE：网络走到画在上面的导线；导线就是它自己。SELECT-LABEL：对象走到附着在它上面的标签；标签就是它自己。SELECT-ANNOTATION 只取范围里的标注。

  - 经过端子的一步只算符号里有引脚的端子，与 SELECT-TERMINAL、补位、ROUTE 相同（9.2.4）。所以 `&vss @SELECT-INSTANCE` 就是 `&vss @SELECT-TERMINAL => @SELECT-INSTANCE`：用三端符号画的 NMOS 只有衬底接在 vss 上时选不到；对这样的管子 M1，`#M1 @SELECT-NET` 也不含 vss。本层的端口，如 `#:inp`，照样走到它的网络。符号队列给出符号之前，全部端子都算（8.2）。
  - 以用例为范围时，只取要选种类的成员；成员不再当作范围（见下）。
  - 例：受时钟控制的管子是 `&clk @SELECT-INSTANCE`；只要栅极接 clk 的，是 `&clk @SELECT-TERMINAL(terminal="g") => @SELECT-INSTANCE`；M1 接的网络是 `#M1 @SELECT-NET`。
  - 注：只算有引脚的端子，选出来的就与画出来的一致，后面的规则加上的地、导线和样式都与图对得上。
- **只选本层。** 每个选择关键字都有条件 `recursive`，默认 `true`。写 `recursive=false` 时只选写这条语句那一层的对象；在下层求值时为空，所以规则只在它自己那一层生效（10.4）：`# @SELECT-TERMINAL(net="vss", wired=false, recursive=false) => $GROUND_STUB @ATTACH` 只给本层补地。单个宿主对象直接引用即可，如 `&vss`：宿主引用本来就只在本层解析。
- **SELECT-USAGE 和 SELECT-ANCESTOR-USAGE。** 两个都按名字解析到的定义认定义（9.5），选出的总是用例，都没有 `role=`。
  - SELECT-USAGE 往里找，与别的选择关键字一样。范围给 `#` 时，选求值那一层的用例；给用例或用例的集合时，选这些用例本身和它们里面的用例，一直到最里层。范围里的元件、网络和端子不是用例，选不出东西；这时给警告，提示要找它们所在的用例用 SELECT-ANCESTOR-USAGE。
  - SELECT-ANCESTOR-USAGE 往外找：选出把范围里的对象当成员的用例，直接的、间接的都算（见下例）。范围里的用例本身不算在内。范围写 `#` 是错误：一层不是任何东西的成员。
- **以用例为范围。** 以用例为范围时，别的选择关键字选出这些用例里本种类的成员，`role=` 只取那个角色的成员。是用例的成员继续往里走，一直到最里层：

```
$OTA = @DEFINE(roles=[["pair", "usage"], ["load", "usage"]]) {};
%dp1 = #M1(in_p) #M2(in_n) #M5(tail) @AS $DIFFERENTIAL_PAIR;
%cm1 = #M3(ref) #M4(out) @AS $CURRENT_MIRROR;
%ota1 = %dp1(pair) %cm1(load) @AS $OTA;
%all_set = %ota1 @SELECT-INSTANCE;                  /* {#M1 #M2 #M5 #M3 #M4} */
%pair_set = %ota1 @SELECT-INSTANCE(role="pair");    /* {#M1 #M2 #M5} */
%up_set = #M1 @SELECT-ANCESTOR-USAGE $OTA           /* {%ota1} */
```

  - `role=` 的值也认这个用例的定义的 `definition=` 代称（16.5）。只要直接成员，就把角色当子对象用，如 `%dp1::tail`。遇到已经走过的用例就停。成员指填进角色的对象，不含体里的局部对象（9.4）。
  - **只取本种类的成员。** 成员不再当作范围往下走，别的种类的成员不算。SELECT-TERMINAL 只取端子成员：`$GROUND @ATTACH &0; $GROUND @ATTACH #M5:3` 之后，`# @SELECT-USAGE $GROUND => @SELECT-TERMINAL` 只是，不含网络 0 上的其他端子。差分对 %dp1 的成员是元件，所以 `%dp1 @SELECT-TERMINAL` 是空集合。要成员元件的端子，写 `%dp1 @SELECT-INSTANCE => @SELECT-TERMINAL(terminal=2)`（三个管子的栅极，尾管也在内）；要成员网络上的端子，写 `… => @SELECT-NET => @SELECT-TERMINAL`。每一段只做一件事；要往下走，就多写一段。范围里直接给的对象照前面的规则：网络给出它上面的端子，元件给出它的端子。
- 在场景里按条件找对象，一律用选择关键字；`::` 只取对象自身的部分（6.5）。
- **叠画。** 叠画的成员选不到，选到的是叠画（11.8）。SELECT-INSTANCE 按叠画第一个成员的条件选到叠画本身，每个叠画只算一次。SELECT-TERMINAL 选到叠画的端子，`net=` 对上其中任何一个成员的网络就算对上。
- **条件**写名字，形如 `name=value`；写出的条件都满足时才选中。要"或"，就写几条规则。比较名字的条件（`name`、`model`、`net`）收通配符 `*`，它匹配任意多个字符，与 DEVICE 一样按宿主的大小写规则比较（6.12）。按元件字母选，写 `name="R*"`。Spectre 的元件名不以元件字母开头，按 `model` 选，如 `model="resistor"`（6.12）。
- **`model`** 比较元件行上写在模型位置的名字：M、Q、D 等元件是 `.model` 名；X 元件是子电路名，包括网表里没有定义的 PDK 器件，如 `# @SELECT-INSTANCE(model="sky130_fd_pr__nfet_01v8")`；Spectre 里是 master。它与 DEVICE 的 `model=` 相同（6.12）。`type` 比较 `.model` 里的类型；X 元件没有类型。
- 比较照 3.6：条件名和标准的枚举值区分大小写；指宿主东西的值（名字、模型名、网络名）按宿主的规则。

- 网络可以按名字选：`# @SELECT-NET(name="0")` 选这个网络，`# @SELECT-TERMINAL(net="0")` 选接在它上面的端子。对象也可以按语义选：`# @SELECT-USAGE $DIFFERENTIAL_PAIR => @SELECT-INSTANCE(role="in_p")` 选出每个差分对的正输入管。
- 没有求交集的关键字，如求一个集合里属于某个用例的成员；用两次 REMOVE-FROM 写（5.2.2）：

```
%s_set = {#M1 #M3 #M5};
%s_set @SELECT-ANCESTOR-USAGE $DIFFERENTIAL_PAIR => @SELECT-INSTANCE => %dp_m_set;
%out_set = {%s_set};
%dp_m_set @REMOVE-FROM %out_set;    /* the members of %s_set in no differential pair */
%in_set = {%s_set};
%out_set @REMOVE-FROM %in_set       /* the members of %s_set in a differential pair */
```

- **地和电源网络**由标准包提供的集合给出：`$GROUND_NET_SET`（默认是网络 0）和 `$SUPPLY_NET_SET`（默认为空）。没有网络类别。
  - 包里不能写宿主引用（16.2），所以这种集合的默认内容只能写成选择，如 `# @SELECT-NET(name="0")`。
  - 用户用 ADD-TO 和 REMOVE-FROM 增删网络：

```
&vss @ADD-TO $GROUND_NET_SET;                                  /* the vss of this level
*/
# @SELECT-NET(name="vss") => @ADD-TO $GROUND_NET_SET;          /* the nets named vss of
every level */
# @SELECT-NET(name="vdd!", global=true) => @ADD-TO $SUPPLY_NET_SET  /* only the global vdd! */
```

    第二条语句也会加上子电路里叫 vss 的局部网络。只在某处不算某个网络，就复制一份集合，再从副本里删掉它：`%g_set = {$GROUND_NET_SET}; &vss2 @REMOVE-FROM %g_set`。
  - 补位规则用这样的集合作范围，如 `$GROUND_NET_SET @SELECT-TERMINAL(wired=false) => $GROUND_STUB @ATTACH`。
  - 在自己的层里写 `$GROUND_NET_SET = …` 会遮蔽包里的集合，包里的规则看不到新的这个（6.8）。
  - 不按网络名猜哪些是地和电源。导入器把编辑器文件里网络的电源域（如 `powerDomain`）写成 ADD-TO。
  - 求解器可以把标准集合当提示读（如电源在上、地在下），与它可以读标准包里的定义一样；这不作要求。
  - 注：地和电源网络没有关键字（1.5 第 7 条）：这些集合的用处，如选端子补地、换符号，都能写成包里的规则。
- **全局网络**。SELECT-NET 的条件 `global=true` 只选全局网络（6.3）。不同层里同名的局部网络是不同的结点，按名字选会都选上；只要全局的那一个，就加这个条件。

10.2 条件 `wired`

- 端子不是任何显式导线的点（端点或中间的点），也不与这样的端子共用总线引脚（9.2.4）时，它是 `wired=false`。总线导线也是显式导线（13.1）。导线接到叠画的端子，就是接了每个成员的这个端子（11.8）。
- 接在网络上的宿主元件的端子都没被显式导线接触时，网络是 `wired=false`；虚拟元件的端子不算。
- 只算显式写出的导线，不算生成的。有没有虚拟元件没有影响。
- 导线按写出的样子算，从 8.2 第 3 步起就算数，后来因为端子没有引脚而失效的也算（7.1、9.2.4）。导线被改写过的，算最后那次改写（13.1）。

10.3 选择的求值

- 选择不在写它的地方展开，每次求值只看一层。含选择的语句（规则，10.4）在写它的那一层求一次，在下面每一层各求一次；每次求值时 `#` 就是那一层，选择只选这一层的对象。所以顶层写的 `# @SELECT-INSTANCE(model="nch") => @ALIGN("y")` 在每张图里各对齐一次，不会把不同图里的元件放在一起对齐。
- 选择在用到它的阶段求值：外观规则在符号队列和样式队列里，生成规则在生成阶段（8.2）。集合也一样（5.2）。

- 选择选出的种类在读入时就知道。把它交给收不了这种对象的槽位，当即是错误，放在集合里也一样，如 `# @SELECT-TERMINAL(net="vss") => @SET-SYMBOL $NMOS`。
- SELECT-USAGE 和 SELECT-ANCESTOR-USAGE 选出的总是用例；从用例走到成员时，选出的种类由关键字决定。所以每个选择选出的种类都在读入时就知道。
- 选择的结果是集合，可以用 `%` 或 `$` 命名。集合在某层求值时只取这一层的成员：顶层写的 `&vss @ADD-TO $GROUND_NET_SET` 只加顶层的 vss，集合在子电路里求值时不含它（6.6）。
- 选择的范围写成本层的宿主对象时，如顶层的 `&Q<0:3:2, 1> @SELECT-TERMINAL(name="R*") => @TEXT("${net}", key="netname")`，规则在下层求值时这些对象不存在：选择在那里为空，什么也不做，不给警告，正如集合只取这一层的成员。10.5 的警告只针对生成规则的其余角色。

10.4 规则

- 收选择结果的语句是规则。
- 任何关键字都能收选择的结果，按 5.2.1 的规定逐个做或整个收：`# @SELECT-INSTANCE(type="nmos") => @ORIENT("MY")`、`# @SELECT-INSTANCE(model="nch") => @ALIGN("y")`。
- 以选择的结果为对象的 SET-SYMBOL 和 SET-STYLE 是外观规则，进外观队列（8.3）。AS 和 ATTACH 是生成规则（10.5）。其他关键字的规则，等于对选中的每个对象各写一条语句，或对它们全体写一条，在该关键字的阶段生效。
- 规则作用于写它的层及其下层，每一层各求一次，只看那一层（10.3）；包里的规则作用于用到这个包的所有层。选择加 `recursive=false` 时，规则只在写它的那一层生效。规则按定义对待（6.1）。

10.5 逐个生成

"对所有没接导线的 vss 端子，各放一个地并用导线接上"：

```
$GROUND_STUB = @DEFINE(roles=[["t", "terminal"]], parameters=[["gap", 2]]) {
  %g = $GROUND @ATTACH ?t::net;
  ?t::net @WIRE {?t %g:p};
  {?t %g:p} @ALIGN("x");
  %g:p ?t @DISTANCE("bottom-to-top", exact=?gap)
};
# @SELECT-TERMINAL(net="vss", wired=false) => $GROUND_STUB @ATTACH => %stubs_set
```

标准包提供一个这样的 `$GROUND_STUB`，只是它只给最小距离（`min=?gap`），求解器可以把导线画得更长，如把长短不同的元件的地排齐；它还让地朝下（`@ORIENT("R0")`），约束的优先级都是 `"low"`；这里的定义遮蔽了它（6.8）。

- AS 或 ATTACH 的单个角色收到选择的结果时，对选中的每个对象各套用一次定义，其余角色在每次套用中都相同（5.2.1、9.4）。集合角色把选择的结果整个收下，定义只套用一次。
- 虚拟元件也这样逐个创建：`# @SELECT-NET(port=true) => $IO_PORT @ATTACH` 给每个端口网络一个端口符号；`$RLOAD` 是一个画成电阻的两角色定义，`# @SELECT-NET(name="out*") => $RLOAD @ATTACH &vss` 在每个选中的网络与 vss 之间接一个；因为 `&vss` 属于本层，这条规则只在本层生效，并给下面说的警告。
- 地也可以直接画在引脚上，不用导线：`# @SELECT-TERMINAL(net="vss", wired=false) => $GROUND @ATTACH`。这时角色填的是端子，用例画在它的引脚上（12.1）。这样的地没有导线，所以端子仍是 `wired=false`：在同一引脚上再显式写一个地时，规则仍会生成一个，两个重叠。所以标准包的补位用带导线的定义补地和电源（`$GROUND_STUB`、`$SUPPLY_STUB`）：写回以后，导线让端子不再是 `wired=false`，补位就跳过它。对用户直接写在引脚上的地，补位用 `# @SELECT-USAGE $GROUND => @SELECT-TERMINAL => @REMOVE-FROM $GROUND_FILL_SET` 这样的规则扣掉它的端子（10.1），所以不会再补第二个。照常放一个地、用长度为 0 的导线接上（13.1）也行。
- 配对在每次套用的内部：定义的局部名每次套用都是新的，所以一次套用里的 `%g` 就是这次套用建的那个地。
- 结果是一个生成集。它可以命名，可以整个交给 SET-SYMBOL 或 SET-STYLE（如 `%stubs_set @SET-STYLE $grey`），但不能引用其中单个成员（5.7）。

- **手写一个。** 要单独描述其中某一个，就手动创建它并接上线。它的端子就不再是 `wired=false`，补位跳过它。没有另外取消生成的办法。
- **快照。** 所有生成规则看同一个快照，即 8.2 第 3、4 步的结果（按键覆盖之后，含全部对象的符号），彼此看不到对方的结果。所以同一条规则写两次，就生成两份。
- 生成的事实在冲突取舍时取生成规则的位置（8.5）。
- 生成的对象没有位置时由求解器摆放；要说明怎样摆，就用带约束的定义。
- **写回。** 单个生成的对象不能被引用（5.7）。求解器写回它们时，写成显式的语句，如 `$IO_PORT @ATTACH &out => @PLACE(40, 20) => %p_out`，加上接到它的导线。只有规则把已经写出的去掉时，显式的对象才取代生成的对象，标准包的补位就是这样，它写两个集合的差集（见下面的按网络补位）；别的写法的规则会在显式对象旁边再生成一个，没有位置。所以要由求解器摆放其对象的规则，应写成这样的差集。只要朝向、不用接线的对象（如画在引脚上的地）也可以经选择写回，如 `#M1:3 @SELECT-ANCESTOR-USAGE $GROUND => @ORIENT("R180")`。这样写出的显式用例，定义体里没有局部名字的对象用编号引用（5.9）。写出的用例照原样带上规则在套用上明写的优先级，或者它从规则所在记录的记录头继承的 `"disabled"`（14.8）；此外什么都不写，它的体保持原来的优先级，记录头别的优先级不写。
- 生成不递归：生成规则展开出的内容里有规则时，这些规则失效。生成阶段只有一个，这样的展开产生的规则没有可以生效的阶段。
- 生成规则的其余角色填的是本层的对象（场景对象或宿主对象）时，规则只在写它的那一层生效，并给警告：这些对象属于那一层，下层没有。填定义的角色没有这个限制。
- 生成产生的 SET-SYMBOL 或 SET-STYLE 归入产生它的规则所在的组，位置取规则的位置（8.3）。
- 绑定到选择或生成集的名字不能重新绑定（6.6）。
- **按网络补位。** 没有内置的"已有就不补"规则。标准包的补位写两个集合的差集；标准包写的是：

```
$PORT_FILL_SET = # @SELECT-NET(port=true);                    /* the nets that get a
port tag */
$SUPPLY_NET_SET @REMOVE-FROM $PORT_FILL_SET;                  /* none on a supply net
*/
$GROUND_NET_SET @REMOVE-FROM $PORT_FILL_SET;                  /* nor on a ground net:
the stubs stand there */
# @SELECT-USAGE $IO_PORT => @SELECT-NET => @REMOVE-FROM $PORT_FILL_SET;   /* less those that
have one */
$IO_PORT @ATTACH $PORT_FILL_SET
```

  - 这几条语句如果写在文件里而不是包里，就会遮蔽标准包的集合（6.8），并多出第二套端口补位。
  - 手写的端口符号取代生成的：`$IO_PORT @ATTACH &inp => @PLACE(0, 20) => %p` 之后，inp 不再得到生成的端口符号。用户还可以从 `$PORT_FILL_SET` 去掉网络，如 `# @SELECT-NET(name="vdd") => @REMOVE-FROM $PORT_FILL_SET`，或把自己的端口定义的用例算进来：`# @SELECT-USAGE $MY_PORT => @SELECT-NET => @REMOVE-FROM $PORT_FILL_SET`。
  - 生成的端口符号不算：生成规则在第 5 步用 `$PORT_FILL_SET`，差集里的选择按第 5 步之前的状态求值（5.2）。
  - 注："已经有一个"不内置，因为对多角色的定义它说不清，用户自己的端口定义不会算在内，而布局定义的重复只是约束的重复，由冲突取舍和优先级处理（8.5、14.8）。
- 按端子补位时，取代生成对象的仍是手写对象的导线：接了导线的端子不再是 `wired=false`。仍然没有取消生成的办法：从集合里去掉网络只是改了集合。

11 放置与外观

11.1 位置：PLACE

```
#M1 @PLACE(10, 20);
#M1 @PLACE(x=10)
```

- PLACE 给对象的位置赋值，x、y 至少应写一个。PLACE 只设定写出的坐标。每个坐标起初是自由的，没写过的一直自由，由约束和求解器决定。
- PLACE 只管坐标，不管旋转、镜像、缩放和端子的方向；朝向由 ORIENT 设置（11.2）。
- 一条 PLACE 是两个几何事实，x、y 各一个，冲突时可以各自丢弃（8.5）。参数 `priority=` 给它的优先级（14.8），如 `#M1 @PLACE(12, 20, priority="high")`：按位置写的 x、y 在前，写名字的参数在后（4.2 第 9 条）。
- 对象：元件、自己有位置的虚拟元件（12.1）、叠画（11.8）、点、TEXT（标注和标签；PLACE 定的是文字自己的位置，13.3）、原点 `::position`（含中心写法的 RECTANGLE、CIRCLE、ELLIPSE 的中心），以及集合的 `::position`。PLACE 不收别的图元；要移动它们，就移动它们的点（13.4）。没有符号的用例、画在主对象坐标点上的用例收到 PLACE 时失效（12.1）。符号体里的 PLACE 只给附着在 `?` 上的 TEXT 定位置，坐标是局部的（9.2.2）。
- 输出沿用输入的句柄（5.1）。
- 对象是集合时逐个放（5.2.1）：`%pair_set @PLACE(10, 20)` 把两个成员都放在 (10, 20)。要放集合的中心，写 `%pair_set::position @PLACE(10, 20)`（6.5）。
- 坐标的轴向、单位、小数和容差见 11.7。

11.2 朝向：ORIENT

```
%pair_set = {#M3 #M4};
#M2 @ORIENT("MY");          /* flipped left-right */
%pair_set @ORIENT("R90")    /* each member turns about its own origin and stays where it is */
```

- 对象：元件、有符号的用例、叠画。没有符号的用例收到 ORIENT 时失效（12.1）；点、标签、图元、导线、网络不收。
- 对象是集合时逐个做（5.2.1）：每个成员绕自己的原点转，位置不变，每个成员是一条描述。不是把整个集合当一块转。不提供整个集合一起转：`%pair_set::position` 是点，ORIENT 不收。
- 画在主对象坐标点上的用例绕那个点转；它有自己的朝向，不跟着主对象转（12.1）。
- 没写 ORIENT 就是没有约束，由求解器选；不求解的渲染器按 `"R0"` 画（8.5）。几个对象的朝向之间的关系（同向、互为镜像）写 ORIENT-RELATION（14.9）。
- ORIENT 是一个几何事实，参加冲突的取舍（8.5）；参数 `priority=` 给它的优先级（14.8）。
- **朝向码**：一个码表示一种朝向。只认下表的八个写法，区分大小写（3.6）。名字与 Virtuoso 相同，y 向上（11.7）时含义也相同。等价的别的写法（如 `"MXR180"`，就是 MY）、任意角度都不认：它们是写错的参数值，这条 ORIENT 失效。工具写回时只写这八个。没有"在现在的朝向上再转"的相对写法（第 17 章）。

码	(x, y) 变成	含义
`"R0"`	(x, y)	不转
`"R90"`	(−y, x)	逆时针转 90°
`"R180"`	(−x, −y)	转 180°
`"R270"`	(y, −x)	逆时针转 270°，即顺时针转 90°
`"MY"`	(−x, y)	关于 y 轴镜像，左右翻
`"MX"`	(x, −y)	关于 x 轴镜像，上下翻
`"MYR90"`	(−y, −x)	先 MY 再 R90
`"MXR90"`	(y, x)	先 MX 再 R90

- **合成次序**：MXR90、MYR90 先镜像再转；名字按动作的先后读，与 Cadence、OpenROAD 的 OpenDB 相同。

- **怎样作用**：朝向绕符号的原点作用于局部坐标，再把原点放到对象的位置上：符号上一点画出的位置 = 对象的位置 + 朝向作用于这一点的局部坐标。符号里的图元、引脚位置都随之刚性变换，引脚的出线方向也跟着转（9.2.4）。文字按 9.2.5 的规定显示；字形不镜像。
- **只改变画在哪里**：朝向只改变引脚画在哪里，不改变引脚属于哪个端子。子电路符号上下翻以后，inp 的引脚画到了下面，接 inp 的导线跟着过去。不会为了让 inp 留在上面而把两个引脚对调：那等于改了哪根导线接哪个端子，是网表的事（原则 2.2）。要 inp 画在别处，换一个符号（11.3）。

11.3 符号与样式：SET-SYMBOL 与 SET-STYLE

```
%w13 = &x @WIRE {#M1:1 #M3:1};
#M1 @SET-SYMBOL $NMOS;
# @SELECT-INSTANCE(model="nch") => @SET-SYMBOL $NMOS;
%w13 @SET-STYLE $signal;
# @SELECT-INSTANCE => @SET-STYLE $base
```

- SET-SYMBOL 给对象指派符号。对象可以是元件、用例、叠画，或它们的集合（含选择和逐个生成的结果），对每个成员各指派一次（5.2.1）。在子电路里，`# @SET-SYMBOL` 给的是子电路符号（9.2.6）。
- 用例被指派了符号就画出来：它就是虚拟元件（第 12 章）。
- SET-STYLE 给对象指派样式。对象可以是元件、用例、叠画、导线、图元（含文字），或它们的集合（含选择和逐个生成的结果）。不收网络、层、约束、定义、点。端子随所属对象的符号画，跟所属对象的样式。用例收到的样式传给它的符号和它体里创建的可画对象（8.3），所以没有符号的语义用例收到样式不出错；体里没画东西时，看不出效果。
  - 注：否则标准包的规则 `# @SELECT-USAGE => @SET-STYLE $base` 会对每个语义用例给警告。
- 输出沿用输入的句柄；对象是集合时，输出就是这个集合。
- 两者都经外观队列生效，不按键覆盖。分两级：直接指明对象的 SET-SYMBOL 或 SET-STYLE，高于经选择到达对象的和传给对象的（8.3）。
- 有默认规则时（标准包，16.3），可以不写。
- **参数与样式**：写在对象上的样式参数，如 TEXT 的 `size=`，是这个对象直接的样式，高于选择来的、传来的样式。之后写的、直接指明这个对象的 SET-STYLE 按位置盖过它们（8.3）；编辑器就靠这个单独改一个对象。定义的参数没传时当作没写，样式随之生效（9.4）。
- **标签与汇合点**：对象的样式传给附着在它上面的文字。在"选择来的、传来的"这一级里，它与选这段文字的规则（如 SELECT-LABEL）按位置比；文字自己的参数和直接指明这段文字的 SET-STYLE 在直接的一级。所以 `#M1 @SET-STYLE $signal` 写在标签规则之后时，M1 的名字也取这个颜色，拖动名字也不会换颜色。符号里附着在 `?` 上的文字也是标签，SELECT-LABEL 选得到；附着在点上的固定文字是符号的图元，用所属对象的样式。附着在网络或点上的文字不从附着的东西得到样式，只用自己的。汇合点是 `$JUNCTION` 的用例（13.2）；圆点是它的符号，颜色跟用例的样式。

11.4 一物多画

没有办法把一个宿主对象画几次：宿主句柄如 `#M1` 总是指这个对象唯一的那一次画出。以后的版本若加一物多画，多出的画法会是另外的对象，有自己的名字，宿主句柄的含义不变（附录 C）。

注：几个元件画成一个，是叠画（11.8）。

11.5 分组：GROUP

```
%stage1_set = {#M1 #M2 #M5} @GROUP;
%stage2_set = {#M3 #M4} @GROUP;
{%stage1_set %stage2_set} @GROUP;
%stage1_set %stage2_set @DISTANCE("x", min=1);
{%stage1_set::position %stage2_set::position} @ALIGN("y")
```

- GROUP 给一个集合加一条事实：求解器把它当作一块。它沿用集合的句柄，所以 `%stage1_set` 仍然是那个集合。
- 集合本身就是求解器登记的东西。GROUP 不是第五种原子约束，也不是另一种对象。

- 要把几个对象当整体，PLACE 收集合的 `::position`（6.5）；写成 `{%x}` 的集合是 ORDER、ALIGN、DISTRIBUTE 的一个成员，按它的中心算；交给 DISTANCE 的集合按它的范围算（14.1、14.5）。任何集合都可以这样用，不需要 GROUP。
- 嵌套由包含关系得出：`{%stage1_set %stage2_set}` 的成员是五个元件（引用打散一层，5.2），所以两级都包含在里面。
- **含义**：GROUP 只是给求解器的提示，"求解时当作一块"，不保证任何几何关系。位置上的要求，如不穿插（例子里的 DISTANCE）或居中，写成原子约束；GROUP 对搜索和打分的影响由实现决定。不规定默认不穿插：那等于在用户背后加约束。两组可以部分重叠，即一个成员在两个互不包含的组里；由求解器处理。
- 分组后的集合仍是集合，引用写在 `{…}` 里会打散一层。要整组作约束的一个成员，写成单独的 `{…}`，或写它的 `::position`，如 `{#R1 {%rc_set} #M1} @ORDER("left-to-right")`（14.1、14.3、14.6）。
- GROUP 是关键字，不是标准包里的定义：它的含义只有求解器懂，写不进定义的体（1.5 第 7 条）。
- 查看器可以把分组画出来，如画一个框；这是查看器的事，本标准不规定。
- GROUP 不是原子约束：它不参加冲突的取舍（8.5），也没有优先级，只有开和关。`%stage1_set @PRIORITY("disabled")` 让求解器收不到这个组，也就是取消分组。外层用例为 disabled 时，它展开出的分组也关掉；从外面经 `::` 直接写在某个分组上的 PRIORITY 可以把它重新打开（14.8）。位置上的要求（如两组不穿插）写成原子约束，各有优先级。定义展开里的分组，只有是命名的局部对象、从外面引用时，才能单独关掉（9.4）。

11.6 绘制顺序与图层：LAYER

```
@RECTANGLE(fill="fillcolor") {(0, 0) (40, 30)} => @LAYER("bottom") => %f1;    /* a frame, drawn
below everything */
(2, 28) @TEXT("input stage") => @LAYER("top") => %n1
```

- 有三个图层：`"bottom"`、`"middle"`、`"top"`。先画 bottom，再画 middle，最后画 top。同一图层里按位置先后画：后面的对象画在前面的上面，盖住它。
- `@LAYER(…)` 设对象所在的图层；不写就在 `"middle"`。每个对象一个键，后写的 LAYER 覆盖（5.8）。LAYER 不改变对象在图层里的位置。
- **位置**按书写的次序（1.5 第 2 条），与 8.3、8.5 用的次序相同：同一层里按文本先后；包的内容在它的 USE-PACKAGE 的位置，根上的包在最前（16.3）；用 `scope=` 写进来的语句排在目标子电路自己的文字之后。各类对象取下面的位置：
  - 导线、图元、文字（标注和标签）：创建它的语句；
  - 用例：实际生效的那条 SET-SYMBOL；符号由规则指派的，取规则的位置（8.3）；
  - 规则、定义展开、逐个生成产生的对象：产生它的那条语句，同一次展开里按体内的先后；
  - 元件：在所在图层的最底下，元件之间按网表顺序；放进别的图层的元件在那一层的最底下。叠画当作元件，取它第一个成员的网表顺序。
- 改变对象的语句，如 PLACE、ORIENT、SET-STYLE、LAYER，不移动它的位置。
- 对象：元件、有符号的用例、叠画、导线、图元、标签；集合逐个做（5.2.1）。没有符号的用例收到 LAYER 不出错，只是没有效果（与 SET-STYLE 相同，11.3）；点、网络、约束、定义收到时失效。
- 符号里的成员没有图层：按成员顺序（9.2.1）跟着所在的对象一起画。
- 不设数字的 `z=`。覆盖（按事实的键）和遮挡（按图层和位置）是两回事。
- "图层"在这里指绘制的先后，与宿主的层（2.6）无关。

11.7 坐标与单位

- **轴向**：x 向右，y 向上；y 越大越靠上。所以 ORDER、DISTANCE 的 `"bottom-to-top"` 是从下往上（14.3、14.5），R90 是逆时针（11.2）。SVG 等格式和一些编辑器是 y 向下，画出、导入、导出时由实现把 y 取反。
- **单位**：单位没有物理尺寸。一个单位画多大、打印多大，由查看器决定。
- **小数**：坐标和长度都可以写小数（3.7），不必是整数，位数不设上限。坐标照写的值画，不吸附到整数上。
- **网格**：建议引脚放在整数点上（9.2.4）；符号里的图元不受此限。元件放在整数点上、朝向是八种之一时，整数的引脚位置变换以后仍是整数，导线也就落在整数点上：整数点就是通常说的网格。语言没有自己的

网格。求解器给自由坐标取不取整数是它自己的规则，排在全部语言描述之后（8.5）；约束逼出小数时就用小数。走线方式算出的拐点照算出的值，不取整（13.1）。

- **容差**：检查约束是否满足、判断冲突时（8.5），差在容差以内就算满足。容差是求解器的选项，用户可以设定；本标准不规定它的值，但它大于 0，这样写不成有限小数的值也能写回（见下）。不求解的查看器检查时用自己的设置。所以恰好卡在容差边上的情况，两个工具的结论可能不同。求解器写回坐标时，写成读回来还是同一个值的十进制小数，不另外截短；写不成有限小数的值，如 1/3，写足够多的位数，使记录读回时，它满足的事实仍在容差内成立（第 17 章）。
- **符号的坐标**：DEFINE-SYMBOL 体里的局部坐标与场景同单位、同轴向，原点就是符号的原点（9.2.1、6.5）。没有缩放：符号不带缩放参数，元件也不能缩放。导入器把别的工具的符号换算成 RNL 的单位。
- **其他长度**：语言里的长度都按坐标单位算，随图一起缩放：DISTANCE 的 `min`、`max`、`exact`，CIRCLE 的 `radius=`，RECTANGLE、ELLIPSE 的 `width=`、`height=`，字号 `size`（同 SVG 的 font-size），线宽 `linewidth`，箭头长度 `arrowsize`（9.3），以及定义里当长度用的参数（如 `gap=3`）。线宽 0 是例外：在任何缩放下都一样细的细线，也是默认（9.3）。
- **原点与纸面**：场景的原点没有特别的含义。每个子电路一张图（6.1），各有自己的坐标；查看器按内容的范围取景。RNL 不记纸面大小，也不记编辑器的吸附网格。

11.8 叠画：BUNDLE

Virtuoso 的叠合元件，如 `I0<3:0>`，是一个符号，网表里却是四行：

```
* A unit cell repeated four times
* @RNL(type="header") { @VERSION("0.3")}
.subckt unit in out vdd vss
R1 in out 1k
.ends unit
XI0<3> D<3> out vdd vss unit
XI0<2> D<2> out vdd vss unit
XI0<1> D<1> out vdd vss unit
XI0<0> D<0> out vdd vss unit
* @RNL {
*   #XI0<3:0> @BUNDLE => @PLACE(30, 10) => %x;    /* four instances drawn as one symbol */
*   &D<3:0> @WIRE {(20, 10) %x:in};               /* a bus to the in pin of the bundle */
*   &out @WIRE {%x:out (40, 10)};                 /* all four members are on out: an ordinary pin
*/
*   %x @TEXT("I0<3:0>", key="name")               /* the name, written out */
* }
.end
```

- **新的对象**：BUNDLE 收一个宿主元件的集合，产生一个叠画，代替这些元件画一次。成员按写的次序，第一个成员决定引脚和符号。
- **引脚和符号**：端子名、端子个数照第一个成员。外观队列把叠画当作它的第一个成员：选择按第一个成员的条件选到叠画（10.1），子电路符号照用。直接写在叠画上的 SET-SYMBOL、样式、PLACE、ORIENT、LAYER、约束、文字都是叠画的。
- **端子**：叠画的端子，如 `%x:in`，代表各成员的同名端子：它是一个总线引脚。
  - `::net`：各成员接同一个网络时就是这个网络，如 `%x:out::net` 是 `&out`；否则是按成员次序的网络集合，如 `%x:in::net` 是 D<3>、D<2>、D<1>、D<0>。所以定义里的 `?t::net @WIRE {…}` 收到集合时画成总线，收到单个网络时是普通导线（13.1）。
  - 导线接到叠画的端子，就是接了各成员的这个端子（10.2）。
- **属性**（13.6）：各成员相同时就是这个值，否则用 `,` 把各值连起来。`${w}` 都是 1u 时显示 1u；`${name}` 是 `XI0<3>,XI0<2>,XI0<1>,XI0<0>`，要显示 `I0<3:0>`，就写同引用名的文字，如上例。端子的 `${net}` 同理，如 `D<3>,D<2>,D<1>,D<0>`。
- **选择**：成员选不到，选到的是叠画（10.1）。所以"给每个没接线的引脚补一个网名"这类规则在叠画上只补一个标签，显示整条总线。定义里收元件的角色也收叠画，收端子的角色也收叠画的端子（9.4）。
- **成员不单独画**：关于画图的描述（位置、朝向、符号、样式、图层、约束、导线、文字）直接写在成员或成员的端子上时失效，与没有符号的用例相同（12.1 第 3 条）。不涉及画图的描述照常，如 CURRENT 和语义用例的角色。

- **都是同一种**：建议各成员的元件字母、模型名或子电路名、参数（按原文比较）、端子个数都相同。不同时给警告；照样叠画，引脚和符号照第一个成员。
- **一个元件只在一个叠画里**：一个元件写进几个 BUNDLE 时，归后写的那个（1.5 第 2 条）；先写的保留其余成员，可以给提示。先写的叠画成员都被拿走时是空的，不画，给警告。
- BUNDLE 收到空集合时失效（5.2.1）。成员不是宿主元件时失效（种类不对）。找不到的成员使整条语句失效（严格，7.3）。

12 虚拟元件

12.1 概念

- 虚拟元件是只存在于 RNL 里的元件，如地、电源、端口符号、无连接标记。它在电气上不存在：只用于画出，不产生也不改变电气事实。
- 虚拟元件就是有符号的用例，没有单独的关键字或对象种类。`$GROUND @ATTACH &vss` 产生一个用例；标准包的一条规则给 `$GROUND` 的用例指派地符号，它就画出来。
- 规则：
  1. 用例的端子就是它的定义的角色：`%g0:p` 是端子，即符号上 p 引脚的位置；`%g0::p` 是填进角色 p 的成员，即网络 vss。这与元件一样：`#M5:3` 是端子，`#M5:3::net` 是它所在的网络。
  2. 用例被指派了符号才画、才能放置；没有符号的用例不画。标准包只给要画出来的定义配符号，所以差分对这类语义用例不画。
  3. 没有符号的用例对画图没有意义：用到它或它的端子的 PLACE、ORIENT、约束和导线失效，与没有引脚的端子相同（9.2.4）；这在 8.2 的第 6 步之后检查。收到样式不出错：样式传给它体里创建的可画对象（8.3），体里没画东西时看不出效果（11.3）。
  4. 画出来的用例只画它自己的符号。它不包含、也不带动它的成员，不是视觉容器。
  5. **画在哪里**：填第一个角色的对象是用例的主对象。主对象有坐标点时（端子是它的引脚的位置，元件是它的原点，点就是它自己），符号的原点画在这个点上，用例自己没有位置：用到它的 PLACE 和约束失效。没写 ORIENT 时由求解器选它的朝向并写回（14.8），没有求解结果的渲染器按 `"R0"` 画（8.5）；它不跟着引脚的出线方向。可以写 ORIENT：用例绕这个点转，有自己的朝向，不跟着主对象（11.2）。主对象没有坐标点时（如网络），用例自己有位置，用 PLACE 给出或由求解器定，它的 `::position` 是它自己的原点。用例没有自己的名字；文字经角色引用主对象，如 `${p}`（13.6）。
  6. 定义体一律展开，所以虚拟元件的定义也可以带体。
- 按原则 2.2 的说法，虚拟元件是有符号、角色填宿主对象的用例；标准虚拟元件是标准包里的定义，带一条给它默认符号的规则；用户自定义的虚拟元件是用户自己的定义，加上一条符号规则。
- 外观由规则按定义指派，如 `# @SELECT-USAGE $GROUND => @SET-SYMBOL $GROUND_SYMBOL`。换画法不改变端子和接在上面的导线。
- 一个用例是不是虚拟元件，只看它有没有符号。
- 注：该画出来却没配符号的定义，只有它的某个用例被放置、被约束或接线时才会报出来。

12.2 创建虚拟元件

```
$GROUND @ATTACH &vss => @PLACE(24, 38) => %g0;
&vss @WIRE {#M5:3 %g0:p}
```

- 在 `$D @ATTACH &net` 里，定义后面的对象按定义的角色表配位。它用 ATTACH 写，读作"把这个添在网络上"（9.4）。多端子的虚拟元件就像网表里的一行元件：定义 `$DIFF_PORT` 有角色 a、b 时，写 `$DIFF_PORT @ATTACH &inp(a) &inn(b)`，或按顺序写 `$DIFF_PORT @ATTACH &inp &inn`。
- 角色一般填网络：虚拟元件的端子和元件的端子一样接在网络上；它和某个端子的关系（画在哪、接谁）由导线和约束表达（12.3）。第一个角色填端子时，虚拟元件直接画在这个端子的引脚上（12.1 第 5 条），如 `$GROUND @ATTACH #M5:3`。角色收哪几种对象可以在定义里声明（9.4），如 `roles=[["p", ["net", "terminal"]]]`。
- 这种关联只是描述，不建立电气连接，也不校验（原则 2.2）。
- 同一个网络可以有任意多个虚拟元件，如五个地、两个探针。用例没有键：每套用一次就多一个用例（5.8）。
- 个别虚拟元件可以另换符号，如用自己的符号 `$EARTH` 写 `%g1 @SET-SYMBOL $EARTH`。
- ATTACH 除 `priority`（14.8）外没有自己的参数。端口的方向归标准包的内容（12.4）；显示的名字是主对象的名字，要显示别的文字就写标签（13.3）。定义的参数照常经 ATTACH 传入（9.4）。
- 同一层里 `%g0` 只绑定一次（6.6）；要改它的位置，写 `%g0 @PLACE(…)`。

12.3 相对端子的位置

参数里不能放宿主引用，所以相对端子的位置写成约束：

```
{%g0:p #M5:3} @ALIGN("x");
%g0:p #M5:3 @DISTANCE("bottom-to-top", exact=2)    /* the pin of the ground 2 below the source of
M5 */
```

要把虚拟元件直接画在引脚上，不隔距离也不画导线，就让它的第一个角色填这个端子（12.1 第 5 条）。

12.4 端口

```
$IO_PORT @ATTACH &inp => @PLACE(0, 20) => %p_inp
```

- 子电路的端口名就是它内部的网络名，所以端口符号关联的是网络 `&inp`。
- 端口本身（`#:inp`）不画，只靠虚拟元件显示。标准包用集合补上端口符号，已经显式写了端口符号的网络不补（10.5）；它们的位置由求解器定。
- 端口的方向只是显示上的事：网表里没有方向，画哪种符号由用户决定。有哪些端口定义和符号，归标准包的内容。
- **端口符号的朝向**：端口符号有方向，如标准包的 `$IO_PORT_SYMBOL` 引脚在一端，标签往另一边伸出。没写 ORIENT 时，由求解器按导线的方向选朝向，以拐点少为目标。这是求解器的目标，不属于语言（14.8），所以手写时一般不写端口符号的 ORIENT。

12.5 标准虚拟元件

标准虚拟元件归标准包，标准包不属于语言标准（16.3）。例如，标准包提供地 `$GROUND`、电源 `$SUPPLY`、端口符号 `$IO_PORT` 和无连接标记 `$NO_CONNECT`（12.6），以及它们的符号、指派这些符号的规则和补位（10.5）。

同一个网络 vss，三种写法画出来不同：

1. `&vss @ADD-TO $GROUND_NET_SET`：vss 上每个没接导线的端子下面各补一个地，带短导线（标准包的补位，10.5）。教科书多这样画：每个接地的端子下面各一个地符号。
2. `$GROUND @ATTACH &vss => %g`：整个网络只有一个地，由求解器摆。各端子用导线接到它的引脚，如 `&vss @WIRE {#M5:3 %g:p}`，或整个网络写 ROUTE。
3. `$GROUND @ATTACH #M5:3`：地直接画在 M5 源极的引脚上，不画导线（12.1 第 5 条）。

- 电源同样写，集合是 `$SUPPLY_NET_SET`，定义是 `$SUPPLY`。网络 0 默认就在 `$GROUND_NET_SET` 里（10.1）。
- 第三种写法的端子，标准包的补位会把它扣掉，不会再补一个地（10.5）。
- 端口：标准包给子电路的每个端口网络补一个端口符号（10.5）。要放置它、约束它或改它显示的名字，就自己写出来：`$IO_PORT @ATTACH &out => %pout`，导线接 `%pout:p`，名字写 `%pout @TEXT("V_{OUT}", key="name")`（13.3）。不要端口符号的网络从 `$PORT_FILL_SET` 里去掉。

12.6 无连接标记

```
$NO_CONNECT @ATTACH #M1:4    /* terminal 4 of M1 is left unconnected on purpose */
```

- 无连接标记是标准包里的定义 `$NO_CONNECT`，它的角色收端子；不另设关键字。它的主对象是端子，所以叉画在引脚上（12.1 第 5 条）。标准包给它一个叉的符号，原点在叉的中心。
- 用例记下这个端子是故意不接的：`# @SELECT-USAGE $NO_CONNECT => @SELECT-TERMINAL` 选出这些端子。
- 检查（网络上还有别的宿主端子、有导线碰到这个端子）不作要求（原则 2.4）；认得标准 `$NO_CONNECT` 的实现可以给警告。
- 无连接标记不算导线（10.2）：只有这个标记的端子仍是 `wired=false`。按端子补位要跳过它们，就减去上面的选择，写法同 10.5 的差集。
- 注：无连接标记不做关键字，因为它对实现没有画图以外的用处：求解器不读它，画法能写进定义和符号（1.5 第 7 条）。
- 网名标签不是虚拟元件，见 13.3。

13 走线与文字

13.1 导线：WIRE

```
&x @WIRE(mode="straight") {#M1:1 (10, 37) #M3:1} => @SET-STYLE $signal => %w13;
&out @WIRE(mode="manhattan") {#M2:1 #M4:1};
&tail @WIRE {#M1:3 #M5:1}                    /* no mode: "auto", routed freely */
```

- WIRE 收一个网络（或网络的集合，即总线，见下）和一组有序的点，至少两个点。点可以是端子（包括虚拟元件的端子）或点对象。点可以重复；首尾两点相同时，导线是一圈。
- **走线方式**：导线经过点列里的每一个点；相邻两点之间怎样走，由参数 `mode=` 决定。设相邻两点是 A(x1, y1)、B(x2, y2)，dx = x2 − x1，dy = y2 − y1：

`mode`	从 A 到 B
`"auto"`（默认）	由求解器决定，可以按实际情况优化，并把走法写回（见下，14.8）；求解之前，查看器按自己的设置画
`"straight"`	一段直线
`"horizontal-vertical"`	A ⟶ (x2, y1) ⟶ B
`"vertical-horizontal"`	A ⟶ (x1, y2) ⟶ B
`"manhattan"`	|dx| ≥ |dy| 时，先横走 dx 的一半，竖走 dy 的全程，再横走另一半：A ⟶ (x1 + dx/2, y1) ⟶ (x1 + dx/2, y2) ⟶ B；否则横竖对调
`"manhattan45"`	|dx| ≥ |dy| 时，先横走 (|dx| − |dy|)/2，再走一段 45° 斜线（横竖各 |dy|），最后横走剩下的；否则横竖对调

  - 除 `"auto"` 外，每种走法都是严格的规则，查看器照定义画。dx 或 dy 为 0 时，每种走法都是一段直线。\|dx\| = \|dy\| 时，`"manhattan"` 按 \|dx\| ≥ \|dy\| 的情况走，`"manhattan45"` 就是一段斜线。走法算出的拐点保持确切的值，不取整，可能落在半个单位上（11.7）。
  - 走法加出的拐点不是对象：不能引用，也不能在上面放汇合点。要在拐点上接支线或画圆点，就把拐点写成点。自由点 `(_, _)` 先由求解器定坐标，再按走法连。
  - 不在表里的 `mode` 值是无效的参数值：这根导线失效。
- **样式参数**：WIRE 也收线的样式属性作参数：`linecolor`、`linestyle`、`linewidth`、`cap`、`join`（9.3）。它们是这根导线直接的样式（8.3），如电源轨写 `@WIRE(linestyle="bold")`，衬底连接写 `@WIRE(linestyle="dashed")`。
- **改写导线**：WIRE 收到的不是网络而是一根导线时，不产生新导线，而是改写这根导线。这条语句照新写一条 WIRE 那样读，它的点列和走法换掉这根导线原来的；导线所属的网络不变。样式参数是这根导线的直接样式（8.3）。
  - 改写有键，键是这根导线：后写的换掉先写的点列和走法（8.4）；同一条管道的两段，后面那段算后写（5.5）。每次改写的样式参数都仍是这根导线的直接样式，按 8.3 逐项取舍。改写不是几何事实（8.5）。
  - 改写失效时，如列出的端子没有引脚（9.2.4），是错误，整条改写不起作用，它的样式参数也一样；导线保留先前的点列和走法：较早的改写给的，或建立时的（5.8）。每次失效的改写都报告，即使后面有改写换掉了它。
  - 与新导线一样，改写从 8.2 第 3 步起就算数：`wired=`（10.2）读的是最后那次改写的点，即使它到第 6 步之后才失效（7.1）。
  - 网络槽位收一个网络、网络的集合，或一根导线；集合里有导线，是种类不对。
  - 求解器给每根 `"auto"` 导线定的走法，都这样写回（14.8）。建议求解器按次序保留这根导线在作者记录里的各个点（作者记录改写过它的，按最后一次改写），只在它们之间加写成坐标的拐点，并写 `mode="straight"`：导线的连接由作者写下，求解器只定它的几何。
  - 编辑器里用户改了一根导线的形状，编辑器在自己的记录里也这样写；求解器就按改写后的导线，不再改它。
  - 要在语句本身里固定走法，就写出拐点，并改用 `"auto"` 以外的走法；保留确切形状的导出器写 `mode="straight"` 并列出全部拐点。

- **同一网络上的几个端子**（原则 2.6）用共用点的几根导线连接。只有共用同一个点对象（命名的点）或同一个端子，导线才接在一起；坐标相同不算（原则 2.2）。没有一句写一棵树的语句，也没有图对象。
- **T 形接头**：直通的导线把接头点列为中间的一个点，不必断成两根；支线从这个点出发（见 13.2 的例子）。没有"导线某段上的一点"。
- **引脚直接相接**：两个引脚画在同一点、中间没有导线时，也写一根导线把两个端子连上，如地符号直接画在电容的引脚上：`&0 @WIRE(mode="straight") {#C1:2 %g3:p}`。它的两端重合，长度为 0，画不出东西，但写明了连接：两个端子算接了线（10.2），标准包不会再补一个地。坐标相同本身不算接上（原则 2.2）。导出器转换靠位置相同连接引脚的原理图时，写这样的导线；导线从引脚或汇合点上直穿过去的，把那个点列为中间点（T 形接头）；汇合点只接一根导线、又正好落在引脚上的，把导线直接接到那个端子。
- **网络只是注明**，必须写。端点实际所在的网络不必与它一致，不校验（原则 2.2）；网络也不从端点推导，端点可以在不同的网络上。
  - 点列里的端子不在导线的网络上（总线导线：不在它的网络集合里）时，给警告：多半是端子写错了，如把 `#R1:2` 写成 `#R1:1`。端子的网络查得到（虚拟元件的端子在填它角色的对象的网络上）；点没有网络，不查。导线合法，照画（原则 2.2、7.6）。
- **总线**：网络槽位收一个网络，或整个收下一个网络的集合。收到集合时，这根导线是总线导线，一个成员也算。网表里没有总线（6.11）；总线只是把几个网络画成一根线的画法。

```
* A 4-bit bus between two blocks, with two taps
* @RNL(type="header") { @VERSION("0.3")}
.subckt blk D<3> D<2> D<1> D<0>
RL D<3> D<0> 1k
* @RNL {
*   $BLK_SYM = @DEFINE-SYMBOL {
*     (-3, 0) @PIN(terminal=["D<3>", "D<2>", "D<1>", "D<0>"], label="D<3:0>", direction="left");
*     @RECTANGLE {(-3, -2) (3, 2)}
*   };
*   # @SET-SYMBOL $BLK_SYM                                        /* one bus pin for four
ports (9.2.4) */
* }
.ends blk
XA D<3> D<2> D<1> D<0> blk
XB D<3> D<2> D<1> D<0> blk
RT3 D<3> 0 1k
RT2 D<2> 0 1k
* @RNL {
*   %t3 = (20, 10); %t2 = (24, 10);
*   %bus = &D<3:0> @WIRE(linestyle="bold") {#XA:D<3> %t3 %t2 #XB:D<3>};   /* a bus between the
bus pins of XA and XB */
*   %bus @TEXT("D<3:0>");                                          /* the name of the bus,
written out */
*   &D<3> @WIRE {%t3 #RT3:1};                                      /* a tap: a single wire
that shares %t3 with the bus */
*   &D<2> @WIRE(mode="straight") {#RT2:1 (23, 9) %t2}              /* its last stretch
enters the bus at 45° */
* }
.end
```

  - 集合的次序就是位的次序，重复的照留；直接写在集合里的集合按次序取到最里层的成员。有不是网络的成员时，这根导线失效（种类不对）。空集合也失效：导线至少带一个网络（5.2.1）。
  - 网络槽位是严格的（7.3）：有一位找不到，整根导线失效，与单个网络找不到时一样。
  - 点、`mode=`、样式参数、汇合点、T 形接头都与别的导线相同。没有 BUS 关键字：实现画总线与画导线相同，加粗是样式的事（1.5 第 7 条）。导入器可以直接写 `linestyle="bold"`；标准包用 SELECT-WIRE 的条件 `bus=true` 默认给总线导线加粗（10.1、16.3）。
  - **分接**不加机制：单根导线与总线共用一个点就是分接，总线把分接点列为中间点（T 形接头）。一小段 45° 斜线进总线，只是单根导线末段的走法。分接符号是套在分接点上的定义，归标准包（16.3）。位名是单根导线上的标签 `${net}`（13.3）。单根导线的网络在不在总线里不校验（原则 2.2）；实现可以给提示。
  - 总线导线的属性 `net` 是各网络名按次序用 `,` 连起来，不加空格，如 `D<3>,D<2>,D<1>,D<0>`；要显示 `D<3:0>`，就照写这段文字。实现不压缩这个列表：压缩取决于宿主怎样写名字，结果不唯一。

  - 所以 `{&a &b} @WIRE {…}` 是一根总线导线，不是重合的两根导线。
  - 几个宿主元件画成一个（叠合元件），写成叠画（11.8）。
- 导线接到的端子，在它的符号里必须有引脚；否则这根导线失效（9.2.4）。
- 线条接触、交叉或重叠都只是几何，不建立连接（原则 2.2）。不共用点的导线交叉时不画圆点；交叉处画小弧之类，归样式和实现。
- 收到网络的每条 WIRE 是一个新对象，没有键：写两次就是两根（5.8）。要从别的语句引用一根导线，就给它起名；求解器的记录用编号引用没有名字的导线（5.9）。端点是端子的，跟着器件动；写成坐标的点不动；命名的点可以加约束。
- **没画出来的连接**：RNL 里没有导线的连接怎样呈现（求解器布线并写回成 WIRE、飞线、标签），由求解器和查看器的风格以及所用的风格包决定，语言不规定。风格包可以用规则生成标签或导线，这些是 RNL。要某个网络画成导线、走法交给求解器，写 ROUTE（13.7）。

13.2 汇合点

```
%j1 = (10, 30);
&x @WIRE {#M1:1 %j1 #M3:1};          /* the through wire passes %j1 */
&x @WIRE {%j1 (30, 30) #M4:2};       /* the branch starts at %j1 */
$JUNCTION @ATTACH %j1                /* draws the dot */
```

- 分支就是几根导线共用一个点（13.1）。
- 汇合圆点显式写出：只在套了 `$JUNCTION` 的地方画圆点；渲染器不自己加。
- 圆点是用例，不是关键字。`$JUNCTION` 是标准包里的定义，只有一个角色，收点或端子；圆点画在这个主对象的坐标点上（12.1 第 5 条），与 `$NO_CONNECT`（12.6）、电流箭头（15.2）相同。引脚上接了两根导线时，可以套在端子上。标准包给这些用例指派实心圆点作符号：圆点的大小来自符号，颜色来自用例样式的 `linecolor`；圆点的大小没有单独的设置。
  - 同一个点套两次 `$JUNCTION`，就是重叠的两个圆点，看上去是一个。
  - 选出全部汇合点写 `# @SELECT-USAGE $JUNCTION`。
  - 绘制次序按指派符号的规则的位置（11.6）。标准包的规则在最前，所以圆点先画、导线后画；圆点与导线同色、比导线宽，看不出差别，不必加 LAYER。
- 圆点不按共用点上接了几根导线推导：真实的原理图里有只接两根导线也画了圆点的分支点，推导出的圆点还要另加一种去掉它的写法。导出器照原图写出圆点。共用点上接了三根以上导线却没套 `$JUNCTION` 时，实现可以给提示；本标准不要求。
- **求解器补圆点**：求解器的输出也是 RNL，可以有 `$JUNCTION` 的用例。求解器在 ROUTE 的分叉点（13.7）写上圆点；用户写的导线在共用点上接了三根以上却没套 `$JUNCTION` 时，求解器也可以在输出里补上。共用点一定是命名的点或端子（坐标相同不算），所以圆点写得出来。补不补圆点是求解器的设置，不是语言规则。
- 圆点只是画法：不建立连接，不校验（原则 2.2）。套了 `$JUNCTION` 的点上没有导线，也照画圆点。

13.3 文字：TEXT

```
(50, 18) @TEXT("5T OTA testbench");                          /* an annotation: attached to a point
*/
(1, 1) @TEXT("see here") => @PLACE(2, 2);                    /* attached to (1, 1); the text itself
is at (2, 2) */
#M1 @TEXT("W/L=${w}/${l}", key="wl") => @PLACE(12, 18);       /* a label: attached to M1, with the
key wl, so it is #M1::wl */
#R1 @TEXT("R_{D}", key="name");                              /* a key the symbol has: new content,
the position still from the symbol */
#M1::name @PLACE(13, 18);                                    /* the name of M1, dragged in an
editor */
%w13 @TEXT("${net}");                                        /* the net name on a wire; no key;
placed by the implementation */
&out @TEXT("${name}");                                       /* shows the name of the net out;
placed by the implementation */
# @SELECT-TERMINAL(wired=false) => @TEXT("${net}", key="netname")    /* net names on terminals
without wires */
```

- **附着**：TEXT 收一个点，或一个画得出来的对象，文字附着在它上面。附着的东西决定三件事：`${…}` 读谁的属性（13.6）；没给位置时文字放在哪；写了引用名时文字是哪个对象的子对象。
  - 点（`(x, y)`、命名的点、`::position`）：文字是标注（13.4）。它只能引用点的属性 `x` 和 `y`。
  - 对象：文字是标签。对象可以是元件、画出来的用例、叠画、有引脚的端子、导线或图元。也可以是网络：网络本身不画，附着在网络上的文字由实现放在它的某根导线或某个端子旁边。文件就是这样指定哪些网络显示名字。
  - 没有符号的用例、没有引脚的端子对画图没有意义：收到它们时这条 TEXT 失效（12.1、9.2.4）。
  - 收到集合时，TEXT 给每个成员附着一段文字（5.2.1），如上例最后一句。
  - 符号体里，附着在点上的文字是符号的固定文字；附着在单独的 `?` 上的文字，附着在用这个符号画出的对象上（见下）。
- **内容**：`text` 必写，写了引用名也一样；不写时这条 TEXT 失效。它可以按位置写在最前，如 `@TEXT("${name}", key="name")`。没有默认内容：要显示名字，写 `"${name}"`。文字记号和属性引用见 13.6。
- **位置**：文字有自己的位置，不是它附着的点或对象。
  - 文字的位置起初是自由的，如同 `(_, _)`。没有 PLACE、`dx`、`dy` 或约束定它时，由求解器决定并写回（14.8）；没有求解结果时由查看器决定：附着在点上的文字通常就放在这个点，附着在对象上的放在对象旁边；为了不重叠，两者都可以挪开。
  - PLACE 定的是文字自己的位置。所以 `(1, 1) @TEXT(…) => @PLACE(2, 2)` 不矛盾：前一部分把文字附着在 (1, 1) 上，后一部分把文字放在 (2, 2)。要把标注确切地放在某处，就写 PLACE。
  - `dx`、`dy`：偏移，只有 TEXT 收。附着的东西有坐标点时（点、元件的原点、画用例的那个点、端子的引脚），文字放在这个点加 (dx, dy) 处。导线和网络没有坐标点：附着在它们上的文字写了偏移，这条 TEXT 失效。
    - 文字仍只有一个位置，就是加了偏移以后的那个；约束和求解器都按它算，没有"偏移前"的位置。附着对象被挪动或由求解器摆放时，文字跟着走；别的约束拉动文字时，附着对象也跟着动。偏移等于 x、y 各一条精确的、沿递增方向（`"left-to-right"`、`"bottom-to-top"`；14.5）的 DISTANCE，只是写在文字上。
    - 偏移按场景的轴，不随附着对象的朝向转，因为场景里的文字不跟对象转。符号体里也可以写偏移，那时它是局部坐标，随元件变换，与 `=> @PLACE(…)` 写的局部位置一样。
    - **偏移与 PLACE 互相取代**：文字在每一轴上的位置，要么是绝对位置（PLACE 的 x 或 y），要么是偏移（`dx` 或 `dy`），不会同时成立。同一段文字的同一轴上，按位置（8.5 的位置）后写的那种生效，先写的那种整个不再起作用：写了偏移以后，这一轴的绝对位置是自由的，由偏移算出；写了 PLACE 以后，这一轴的偏移不再起作用。这是取代，不是冲突：与优先级无关，也不给提示。例如先写 `dx`、`dy`，后写 `#M1::name @PLACE(x=13)`，x 就是绝对的 13，`dx` 不再起作用，y 仍按 `dy`；编辑器拖动一段按偏移放的文字时就这样写。符号里给的位置排在最前（见下），场景里的 PLACE 或偏移都取代它。
    - 生效的那种照常是几何事实：偏移的 x、y 各是一个单位，它们的位置是写它们的 TEXT 语句的位置，优先级是记录头的默认（14.8）；它们与别的事实按 8.5 取舍。同一轴上的几条 PLACE 之间按 8.5 取舍。写了引用名的文字，`dx`、`dy` 分别覆盖（8.4）。
  - 文字可以按它自己的位置作约束的成员（14.1）。文字的 `::position` 就是它自己的位置：`%n1::position @PLACE(…)` 等于 `%n1 @PLACE(…)`。
  - 场景里的文字不跟附着对象的朝向转；要转就写 `rotate=`。当场景通过 PLACE 或偏移给定文字位置的两个轴时，两轴都不再取符号的局部位置：文字不继承附着对象的朝向，`halign`、`valign` 按场景的方向解释。
  - 求解器把它选定的位置写回（14.8）：经文字的引用名（见下）或名字，否则经它的编号（5.9）。生成规则直接产生的、没有引用名的文字（13.3、10.5）无法引用：求解器把它的位置留给查看器。规则产生的用例，定义体里的文字在求解器把用例写成显式语句后，用编号引用（10.5、5.9）。
- **引用名 `key=`**：
  - 不写引用名：文字是附着对象上一段没有引用名的文字；再写一次就是另一段，与导线一样（13.1）。可以用 `%x =` 给它起名，在本层引用、修改。规则生成的、没写引用名的文字没有句柄，求解器不能写回它们的位置。
  - 写了引用名：文字是附着对象的子对象 `object::key`，如 `#M1::wl`。同一个对象同一个引用名就是同一段文字，后写的语句按项覆盖它：字号、对齐、旋转等各自覆盖，没写的参数保持不变（8.4）。内容

每条都写（`text` 必写），后写的内容总是换掉先写的；只改对齐或旋转，也要把内容照写一遍，如 `#M1 @TEXT("${name}", key="name", halign="end")`。位置照 PLACE 的规则（8.5）。别的记录、编辑器、求解器靠这个句柄找到这段文字。

  - 附着在点上的文字不收 `key=`，因为点没有子对象：写了引用名，这条 TEXT 失效。
  - `::` 后面的名字按固定的次序查，引用名排在最后（6.5）：先是用例的角色名和局部名，然后是它们的 `definition=` 代称，然后是标准子对象名和它们的代称，最后是文字的引用名。所以引用名不应与排在它前面的任何名字同拼写：标准子对象名（`position`、`net`）或本文件声明的它们的代称（如 `rnl-zh` 的 `位置`）；所附着用例的角色名或局部名；或这些名字的 `definition=` 代称。这样的引用名是错误，与 6.5 的撞名相同。写在符号里的文字，代称取写这个符号的文件的。`name` 和 `value` 不是标准子对象名，只是常用的引用名。

- **符号里的文字**：
  - 符号体里，单独的 `?` 指用这个符号画出的对象：画 M1 时是 M1，画用例 `%g0` 时是 `%g0`。它只能作 TEXT 的对象，用在别处失效，如在符号里写 `? @PLACE(…)`，含义不通。
  - `? @TEXT("${name}", key="name") => @PLACE(1.5, 1.5)`：文字附着在用这个符号画出的对象上，`${…}` 读它的属性。这里的 PLACE 是局部坐标，随元件变换（见 9.2.5 的文字方向）。没写 PLACE 时，由实现把文字放在对象旁边。
  - 写了引用名的文字是用这个符号画出的对象的子对象，如 `#M1::name`。符号里的文字排在最前，场景里同一个引用名的文字按项覆盖它：`#R1 @TEXT("R_{D}", key="name")` 只改内容，位置仍取符号里的；`#M1::name @PLACE(13, 18)` 只改位置，内容仍取符号里的。场景里给的位置是场景坐标，不再随元件转。
  - 符号给的引用名在指派符号以后才有（8.2 第 4、6 步）。M1 的符号没有引用名为 `name` 的文字、场景里也没写时，`#M1::name` 失效。要不依赖符号，就在场景里写这个引用名的 TEXT。
  - 符号里附着在点上的文字是固定文字，如运放符号里的 `(-2, 2) @TEXT("+")`，不能按元件改。
- **规则**：`# @SELECT-TERMINAL(wired=false) => @TEXT(…)` 这类规则按符号选端子，所以与逐个生成一起在第 5 步求值（8.2）。实现自己补飞线或标签时，跳过已有标签的端子。
- 显示哪些网名、显示在哪里，没画导线的连接用标签还是飞线显示，默认都由实现决定：它们属于实现的风格（原则 2.7）。用标签代替导线会改变布局，所以求解器和渲染器用同一份设置。RNL 只在需要时写这些：表达意图（不写位置），或保存已经定下的位置。
- 没有沿导线的位置：导线本身的坐标就不是固定的。
- 参数：`text`（必写，可以按位置写）、`key`、`dx`、`dy`、`halign`、`valign`、`rotate`，以及文字的样式属性 `textcolor`、`size`、`font`、`bold`、`italic`（9.3）；附着在点上的文字没有 `key`。`halign`、`valign`、`rotate` 见 9.2.5。样式参数是这段文字直接的样式（8.3）；文字的样式怎样取见 11.3。

13.4 标注：场景里的图元

```
(50, 18) @TEXT("5T OTA testbench") => @SET-STYLE $note => %n1;
%f1 = @RECTANGLE {(0, 0) (40, 30)};
%n1 @PLACE(60, 18)                          /* moves the text; may also be written in another
record */
```

- LINE、RECTANGLE、CIRCLE、ELLIPSE、ARC、BEZIER 和 TEXT 可以直接写在场景里，也就能写在普通定义的体里。写在符号体里是局部坐标，写在场景里是场景坐标；由写在哪里决定，关键字不变。
- 原则 2.2 的 note 对象（自由文字、图形、边框等纯绘图对象，没有端子，没有电气语义）就是本标准的标注：写在场景里的图元和附着在点上的 TEXT。
- PIN 只能写在 DEFINE-SYMBOL 的体里：引脚绑定的是用这个符号画出的对象的端子，场景里没有这样的对象。
- 附着在点上的 TEXT 是标注（13.3）：它只能引用点的属性 `x` 和 `y`；不收 `key=`，因为点没有子对象（写了引用名，这条 TEXT 失效）。要显示对象的名字或值，把 TEXT 附着在对象上，它就是标签。
- 图元的位置由它的点给出。点可以是任何点对象：命名的点、加了约束的自由点，或跟着器件走的 `#M1::position`。PLACE 不收图元本身：要从别的记录移动中心写法的 RECTANGLE、CIRCLE、ELLIPSE，就移动它的 `::position`；LINE、ARC、BEZIER 和两点写法的图元有几个点，就给点起名后移动点。TEXT 例外：它有自己的位置，附着的点只给它默认的去处，PLACE 直接定文字的位置（13.3）。

- 外观：图元可以直接写样式参数（9.2.5），也收 SET-STYLE，按 8.3 的两级取值；都没给的属性用渲染器的默认值。
- 边框写成 RECTANGLE，或首尾相同的 LINE。没有框住一组对象的图元：画一个 RECTANGLE，用约束把它的角定在这组对象外面，如 `%tl %stage1_set @DISTANCE("left-to-right", exact=1)`（集合按范围算，14.5）；框住的是各成员的点加上边距。查看器可以把 GROUP 画成框，这归查看器（11.5）。没有 `::box`。
- 绘制次序按图层和位置（11.6）。
- **类别**：是不是标注，由对象的种类得出：场景里的图元就是标注，所以渲染器可以整类显示或隐藏。SELECT-ANNOTATION 选出场景里的全部图元，包括普通定义的体展开出来的（如风格包画的框）；符号里的图元、标签、导线、用例（含汇合点）不是标注。更细的类别（说明文字、标题框、某个包画的装饰）用命名的集合，一个图元可以同时在几个集合里。显示和隐藏是查看器的事；语言没有 `visible`。
- 器件上的文字是附着在器件上的 TEXT，即标签（13.3）。附着在 `#M1::position` 上的 TEXT 是标注；默认它也跟着器件走，但只能引用这个点的坐标，也没有引用名。

13.5 （不用）

本节号不用。

13.6 文字记号与属性引用

```
#R1 @TEXT("R_{D}", key="name");                              /* subscript */
(50, 18) @TEXT("\textbf{5T OTA} testbench\\VDD = 1.8 V");    /* bold, two lines */
#M1 @TEXT("W/L=${w}/${l}", key="wl");                        /* host parameters */
$NMOS_M = @DEFINE-SYMBOL {
  $NMOS;
  ? @TEXT("${model}", key="model") => @PLACE(1.5, -1.5)      /* the model name, in a symbol */
}
```

- **文字记号**是 LaTeX 文本模式的一个子集，不是完整的 TeX：

写法	含义
`_{…}`、`^{…}`	下标、上标
`\textbf{…}`、`\textit{…}`	粗体、斜体
`\overline{…}`	上划线，如低有效的 `\overline{RST}`
`\frac{…}{…}`	分式：分子在上、分母在下，中间一条横线，如 `\frac{${w}}{${l}}`；不能上下叠排的查看器显示成"分子/分母"
`\alpha` … `\omega`、`\Gamma` … `\Omega`	希腊字母，名字与 LaTeX 相同
`\pm`、`\times`、`\cdot`、`\infty`、`\approx`、`\neq`、`\leq`、`\geq`、`\ll`、`\gg`、`\propto`、`\parallel`、`\angle`、`\circ`、`\to`、`\uparrow`、`\downarrow`	常用符号
`\\`	换行
`\_`、`\{`、`\}`、`\$`、`\%`、`\&`、`\#`、`\textbackslash`	字符 `_`、`{`、`}`、`$`、`%`、`&`、`#`、`\` 本身

- 希腊字母：小写 `\alpha` 到 `\omega`（LaTeX 没有 `\omicron`，直接写 o），以及变体 `\varepsilon`、`\vartheta`、`\varpi`、`\varrho`、`\varsigma`、`\varphi`；大写只有与拉丁字母不同的 `\Gamma`、`\Delta`、`\Theta`、`\Lambda`、`\Xi`、`\Pi`、`\Sigma`、`\Upsilon`、`\Phi`、`\Psi`、`\Omega`。表里没有的符号直接写成 Unicode 字符（字符串可以含非 ASCII 字符，3.7）。
- 文字记号可以嵌套，如 `\textit{V}_{\textit{DD}}`、`I_{D_{1}}`。`V_{in}^{+}` 的上下标怎样排（叠放或先后），由查看器决定。
- **上下标要带花括号**：单独的 `_`、`^` 是普通字符。这与 LaTeX 不同，LaTeX 里 `g_m` 的 m 是下标：名字和说明里常有 `vin_1`、`bias_ctrl`，按那条规则会排坏。代价是 `g_m` 照原样显示。
- 其他字符照原样显示，包括单独的 `{`、`}`、`$`、`~`、`%`、`&`、`#`，它们在 LaTeX 里另有含义。

- **反斜杠**：`\\` 是换行，与 LaTeX 相同；反斜杠本身写 `\textbackslash`。反斜杠后面不是认得的命令时，照原样显示并给警告，如 `C:\temp`（`\temp` 是不认识的命令，见下条）；不要警告就写 `C:\textbackslash{}temp`。只有反斜杠后面紧跟 `\`、`{`、`}`、`_`、`^`、`$`、`%`、`&`、`#` 或认得的命令名时，才需要写 `\textbackslash`。RNL 字符串没有反斜杠转义（3.7）：`"\textbf{V}"` 照这样写，不写成 `\\textbf`。
- 不认识的命令（`\` 后面跟着不在上表里的字母，如 `\sqrt`、`\alpah`）和不配对的花括号：照原样显示并给警告；这段文字仍然有效。
- **命令名之后**：命令名是 `\` 后面连续的字母。认得的命令名后面的空格吞掉，不显示；紧跟的空 `{}` 只作分隔，也不显示。所以 `10 \mu A`、`10 \mu{}A` 都显示 10 µA；要留空格，写 `\mu{} A`；`\textbackslash{}temp` 显示 \temp。`\_`、`\{` 这类转义字符后面的空格照留。

  注：以后的版本支持更多 TeX，只会放宽。到时数学模式写 `\(…\)`，不写 `$…$`，因为 `$` 用于属性引用。
- **分式**：`\frac` 后面要紧跟两个花括号里的组；少一个时按不认识的文字记号处理，照原样显示并给警告。分子、分母里可以写文字记号和 `${…}`。
- 上下标、分式升降多少、缩小多少，由查看器决定。字体，以及整段文字的粗体和斜体，来自样式或参数 `font`、`bold`、`italic`（9.3）。
- 文字记号在 TEXT 的 `text` 和 PIN 的 `label` 里解释（如 `\overline{CS}`）。别的字符串参数（`key`、枚举值、选择条件）不解释文字记号。
- 注：导入导出时，KiCad 的 `_{…}`、`^{…}` 写法相同，它的 `~{…}` 就是 `\overline{…}`；Altium 在每个字符后加 `\` 的上划线改写成 `\overline{…}`。
- **属性引用** `${name}` 显示附着对象的一个属性。要写 `${` 本身，写 `\${`；从 `\${` 到与它配对的 `}` 都照原样显示，这个 `}` 不算不配对的花括号。能引用的属性：
  - 元件：`name`；`value`，宿主行里写的值，原文照搬，没有就是空的（不退而取模型名）；`model` 和 `type`，与选择条件相同（10.1），所以 X 元件的 `${model}` 显示子电路名；以及宿主参数，如 `w`。与标准属性同名的宿主参数写 `${#…}`（见下）。
  - 端子和导线：`net`，所在的网络，显示成它的名字。总线导线的 `net` 是各网络名按次序用 `,` 连起来，不加空格（13.1）。叠画的端子：各成员接同一个网络时是这个网络的名字，否则用 `,` 把各名字连起来（11.8）。
  - 网络：`name`。
  - 用例：各角色和它的定义的参数。`${p}` 就是填角色 p 的对象，按下面的规则显示。用例没有自己的名字：地、电源、端口的符号写 `${p}`，p 是网络时显示网名；用例画在引脚上（12.1 第 5 条）时 p 是端子，也显示它所在网络的名字。
  - 叠画：各成员相同的属性取这个值，否则把各值用 `,` 连起来（11.8）。
  - 点：`x`、`y`，最后的坐标，即求解以后的。
  - 图元：没有属性。
- **属性路径**：`${a::b::c}` 里，第一个名字取附着对象的属性，后面每个 `::name` 取前一个值的属性，如 `${p::x}`（角色 p 是点时它的 x）、`${pair::in_p::w}`。
  - 属性的值可以是对象（用例的角色；端子、导线的 `net`），也可以是文字（`name`、`value`、宿主参数等）。文字没有属性，后面不能再接 `::`。
  - 路径上的值也可以是集合，如集合角色。这时路径的其余部分从它的每个成员取，成员是集合的按它自己的成员取，按集合的次序；这些值照集合的显示方式显示（见下）；有一个成员没有这个属性，文字就失效。所以 `$CASCADE` 的用例第一级是两个差分对时，`${first::in_p::w}` 显示 `w`，两个 `in_p` 各一个，如 `1u,2u`。
  - `${…}` 里的 `::` 只取属性，与语句里取子对象的 `::` 不同（6.5）：`${m::name}` 是元件 m 的名字，而语句里的 `#M1::name` 是 M1 上引用名为 `name` 的文字（13.3）。
  - `${m::#type}` 只找宿主参数，与 `${#type}` 相同（见下）。
- **显示**：路径最后的值是文字时，照原样显示。是对象时：元件和网络显示名字；端子和导线显示所在网络的名字，总线导线把各网络名用 `,` 连起来；叠画按上面说的；集合（如集合角色）显示它的成员，各按这里的规则，用 `,` 连起来。点、用例、图元显示不出来：这段文字失效。所以角色 p 是点时，要写 `${p::x}`，不写 `${p}`。

- 引用取原文，不求值：`w=1u` 显示 1u。各方言给出什么，由它的宿主适配器定；如 Spectre 里电阻的值是参数 `r`。
- 上面列的属性总是有的，没有值时为空。引用不存在的属性名时，这段文字失效。
- 引用来的内容照原样显示，不当作文字记号解释：宿主的名字和值不是按这套文字记号写的。
- 引用只能从附着对象的属性往下取，不能跳到任意对象。没有 KiCad 的 `${R3:VALUE}` 这种引用别的对象的写法。
- **代称**：`${…}` 里的属性名也认本文件声明的代称，按标准属性、代称、宿主参数的次序查；`${#name}` 只找宿主参数（16.5）。用例的角色名、参数名是 RNL 名字，只认它的定义的 `definition=` 代称，每一步按这一步的用例的定义查，如 `${输入对::正输入::w}`（16.5）。
- **与标准属性同名的宿主参数**：标准属性优先，同名的宿主参数写 `${#type}`（RNL 里 `#` 指宿主）。任何宿主参数都可以这样写；没有同名的标准属性和代称时，`${w}` 是 `${#w}` 的简写。标准属性名按原样比较，宿主参数名按宿主的规则比较：SPICE 宿主里 `${Type}` 不是标准属性，所以它是宿主参数 type。
- **名字的排法**：语言没有办法给引用内容的个别字符改格式。要把 M1 显示成 M_1，就写文字本身，`#M1 @TEXT("M_{1}", key="name")`，改名后它不跟着变。"器件名末尾的数字一律排成下标"这类统一的约定，属于查看器的风格。

13.7 整个网络画成导线：ROUTE

```
&x @ROUTE;                                  /* x is drawn as wires; the solver decides the
branches and paths */
%jo = (_, _);
&out @WIRE {#M2:1 %jo};
&out @ROUTE;                                /* this wire is drawn as written; the solver adds the
rest */
$ROUTED_NET_SET = # @SELECT-NET;
$GROUND_NET_SET @REMOVE-FROM $ROUTED_NET_SET;
$ROUTED_NET_SET @ROUTE                      /* as a house-style package would write it: every net
except the ground nets */
```

- ROUTE 让文件可以要求把整个网络画成导线，而不必说在哪里分叉、圆点画在哪里。
- **签名**：一个槽位，收网络；收到集合时逐个成员处理（5.2.1），不当作总线。参数：`mode`（给求解器的走法提示，取值同 WIRE，默认 `"auto"`）和导线的样式参数（13.1）。
- **含义**：这个网络在本层画成导线。网络上全部有引脚的端子（器件的，以及套在网络或它的端子上的虚拟元件的）连成一棵树；在哪里分叉、怎样走由求解器决定。已经写出的导线是这棵树的一部分，求解器补齐其余的。
- 注：WIRE 的点列为空不表示这个意思。点列可能是算出来的，选择碰巧选空时就会悄悄把全部连上；所以 ROUTE 是单独的关键字；本版也没有可选槽位（5.3）。
- **求解器写回**：具体的 WIRE（带拐点和 `mode="straight"`）、分叉用的命名点和汇合点（13.2），带上 ROUTE 的样式参数，写进求解器自己的记录。ROUTE 本身是用户的意图，留着；下次求解时重新计算（第 17 章）。
- **不求解的查看器**画飞线或什么也不画，由查看器决定（原则 2.7）。
- ROUTE 不是几何事实，不参加冲突（8.5）。没有引脚的端子不在树里。网络上要连的点不到两个时，ROUTE 什么也不做。
- 风格包用集合和规则选择哪些网络画成导线，如上例；没有规定网络怎样显示的关键字。

14 布局意图

14.1 共同规则

- 约束产生一个约束对象。命名的约束可以引用，便于定位、编辑，如 `%row1 = {#M1 #M2} @ALIGN("y")`；约束也可以匿名。
- **词汇**：位置的原子约束只有 ALIGN、ORDER、DISTRIBUTE、DISTANCE 四种，加上 PLACE、ORIENT、GROUP。别的关系写成普通定义：对称（14.7）、居中（三个成员的 DISTRIBUTE）、行列（ALIGN 加 ORDER）、信号流向（14.6）、镜像对、按比例的间距（几条精确的 DISTANCE）。没有 LEFT、ABOVE（它们就是带方向的 ORDER 或 DISTANCE），也没有 RATIO、NEAR、INSIDE。相对朝向（同向、互为镜像）写 ORIENT-RELATION（14.9）；一对器件的栅极相向还是朝外，仍用一个 ORIENT 或端子上的 ORDER 定（14.3）。符号不重叠、间距常数、网格归求解器（11.7），不属于约束的词汇。
- **没有相对整张图的位置**："最左""最右""最上"没有关键字，也没有整张图或符号外框的范围。写成相对一组对象的 ORDER 或 DISTANCE，求解器据此找出布局。如端口在最右边：`{#M6 #M7 #CL} %pout @DISTANCE("left-to-right", min=2)`，或者在 M7 本来就在最右时写 `{#M7 %pout} @ORDER("left-to-right")`。
- **成员**是能定位的对象：元件、自己有位置的虚拟元件（12.1）、叠画、点、TEXT（取文字自己的位置，13.3）、端子、按中心写的 RECTANGLE、CIRCLE、ELLIPSE 的中心、集合的 `::position`，以及集合（见下）。导线、网络、约束、定义、没有符号的用例、画在主对象坐标点上的用例不能作成员：收到时这条约束失效（12.1）。导线上的位置应先写成点（13.1）。
- **取哪一点**：元件取它的原点，即它的符号的原点，PLACE 放的也是它；端子取引脚的位置，是器件位置加上符号里引脚位置的刚性变换，不能单独移动；点取它自己；虚拟元件取它自己的位置（12.1）；TEXT 取文字自己的位置；按中心写的 RECTANGLE、CIRCLE、ELLIPSE 取中心。不取符号外框的中心：三端、四端 MOS 符号的外框不同，排成一行会错位。要按引脚对齐就写端子。视觉上的对称要求原点落在符号的对称轴上，这归符号库。
- **集合作成员**：ORDER、ALIGN、DISTRIBUTE 的成员是集合时，算作一个成员，取它的中心，即它的 `::position`。DISTANCE 的两个槽位收对象或集合，集合按范围算（14.5）。中心和范围都按成员的那一点算，嵌套的集合取到最里层的对象，与符号无关。空的集合不算成员；集合里有不能定位的对象时，这条约束失效。引用仍打散一层（5.2）：`{%a_set %b_set} @ALIGN("y")` 是把两个集合的成员拼起来逐个对齐。要两个集合各算一个成员，写 `{{%a_set} {%b_set}} @ALIGN("y")`，或者用 `%a_set::position` 和 `%b_set::position`。

  注：本意是 `{{%a_set} {%b_set}}` 却写成 `{%a_set %b_set}`，是常见的错误（5.2）。
- 投影只有 `"x"` 和 `"y"`。
- 约束不改变网表。冲突时一条原子约束整条保留或整条丢弃（8.5）。
- 四种原子约束和 ORIENT-RELATION 都收参数 `priority=`（14.8）。

14.2 ALIGN

`{…} @ALIGN("y")`：成员在 `axis` 这一投影上相等，`"y"` 是 y 相同（排成一行），`"x"` 是 x 相同（排成一列）。`axis` 必写，可以按位置写。成员不到两个时这条约束自然成立，不失效：选择和集合有几个成员，写语句的时候不知道。

14.3 ORDER

```
%rc_set = {#R2 #C2};
{#M1 #M1:2 #M2:2 #M2} @ORDER("x");              /* the two gates face each other */
{#R1 {%rc_set} #M1} @ORDER("left-to-right")     /* left to right; R2 and C2 count as one member,
by their centre */
```

- `{…} @ORDER(direction)`：成员沿方向所在的投影严格单调。成员不到两个时自然成立。
- **方向**：参数 `direction`，必写，按位置写在最前。取值：

  - `"left-to-right"`、`"right-to-left"`：沿 x 递增、递减；
  - `"bottom-to-top"`、`"top-to-bottom"`：沿 y 递增、递减（y 向上，11.7）；
  - `"x"`、`"y"`：只定在哪一轴上排，正序、倒序都可以。

  DISTANCE 的取值与此相同。
- 成员是集合时按它的中心算（14.1）。只比中心，所以两个集合的成员仍可能交错；要两个集合分开，用 DISTANCE（14.5）。
- ORDER 是严格的：相邻成员至少差一个坐标单位（11.7）。要坐标相等，用 ALIGN。

14.4 DISTRIBUTE

`{…} @DISTRIBUTE("x")`：相邻成员在 `axis` 投影上的差相等，步长可以是正、负或零。比的是成员那一点之间的步长（集合取中心），不是外框之间的间隙。`axis` 必写，可以按位置写。成员不到三个时这条约束自然成立。

14.5 DISTANCE

```
#M1:2 #M1 @DISTANCE("left-to-right", min=1);      /* the gate is on the left */
#R1 #R2 @DISTANCE("left-to-right", exact=8)       /* R2 exactly 8 units to the right of R1 */
```

```
%stage1_set = {#M1 #M2 #M5};
%stage2_set = {#M3 #M4};
%stage1_set %stage2_set @DISTANCE("x", min=1);               /* the two sets do not interleave in
x; either may be on the left */
%stage1_set %stage2_set @DISTANCE("left-to-right", min=1)    /* all of stage 2 is to the right of
all of stage 1 */
```

- `A B @DISTANCE(direction, min=…, max=…)`：A 填槽位 `from`，B 填槽位 `to`；各是一个对象或一个集合。
- **方向**：参数 `direction`，必写，按位置写在最前，取值同 ORDER（14.3）。只约束这一轴，不是欧氏距离。
  - 沿递增方向（`"left-to-right"`、`"bottom-to-top"`）约束 B − A；
  - 沿递减方向（`"right-to-left"`、`"top-to-bottom"`）约束 A − B；
  - `"x"` 和 `"y"` 约束 |B − A|，边界不能为负。
- **集合按范围算**：A、B 是集合时，沿递增方向约束 B 的最小值减 A 的最大值，B 整个在 A 之后时为正；沿递减方向反过来，约束 A 的最小值减 B 的最大值；`"x"` 和 `"y"` 约束两者之间的间隔，哪边在前都行，两者交错时为负。单个对象是只有一点的范围，结果与上一条相同。范围按成员的那一点算（14.1）。一边是空集合时这条约束自然成立。所以要两个集合不交错，写 DISTANCE 就行，不需要别的写法。
- `min`、`max`、`exact` 至少应写一个；`min` 不应大于 `max`。
- `exact=` 是精确的距离，就是 `min`、`max` 相等的简写：沿递增方向时 B − A = exact，沿递减方向时 A − B = exact，`"x"` 和 `"y"` 时 |B − A| = exact，哪边在前都行（不能为负）。它不能与 `min` 或 `max` 同写：同写是无效的参数，这条语句失效。工具写精确距离时写 `exact`（4.3）。

14.6 信号流向

信号流向（原则 2.4）写成普通定义的用例，没有关键字。标准包提供不提供这样的定义、叫什么名字，归标准包（16.3）；例子用 `$FLOW`，它按参数给出的方向，给一条写成集合的路径排序。

```
%rc_set = {#R2 #C2} @GROUP;
{#R1 {%rc_set} #M1} @AS $FLOW          /* the signal goes from R1 through the block of R2 and C2
to M1, left to right */
```

- 信号流向是软的概念：不是所有电路都有严格的方向。所以它写成低优先级的约束（14.8）。
- 路径上由几个对象组成的一段，单独写成一个 `{…}`，如 `{%rc_set}`，按它的中心算（14.1）。GROUP（11.5）只让求解器把它当作一块，可以不写。
- 方向在套用定义时作为参数给出（9.4）：`{…} @AS(direction="top-to-bottom") $FLOW` 就是从上往下（y 向上，11.7）。

14.7 对称

对称写成普通定义的用例。对称轴只是一个自由点，不需要关键字。标准包提供 `$MIRRORED`，角色是 `a`、`b`、`axis`：a、b 同高，关于轴点对称，a 在左，b 的朝向是 a 左右翻过来的。同标准包的每个定义一样，这些约束的优先级都是 `"low"`（9.5）。

```
%ax = (_, _);
%sym12 = #M1(a) #M2(b) %ax(axis) @AS $MIRRORED
```

- `$MIRRORED` 不定实际的朝向：栅极朝内还是朝外，仍要给一边写 ORIENT（14.9）。要两边同向的对称，另写一个普通定义，不带其中的 ORIENT-RELATION。

14.8 优先级：PRIORITY

```
$DP_LAYOUT = @DEFINE(roles=["in_p", "in_n"]) {
  {?in_p ?in_n} @ALIGN("y", priority="low")                    /* a default layout, the first to
give way */
};
%dp2 = #M1(in_p) #M2(in_n) @AS(priority="high") $DP_LAYOUT;     /* this pair keeps the default
layout strictly */
#M1 @PLACE(12, 20, priority="high")                             /* fixes the position of M1 */
```

```
# @SELECT-USAGE $DP_LAYOUT => @PRIORITY("disabled")    /* a house-style package turns the default
layout off, to apply its own */
```

- **四档**：`"disabled"`、`"low"`、`"normal"`、`"high"`。没写优先级的事实都是 `"normal"`，不论写在用户的记录里、包里还是定义体里。
- **分级，不加权**：冲突时先比优先级，高的保留；优先级相同，后写的保留（8.5）。约束不做加权求和。

  注：加权时，十条低优先级的约束能压过一条高优先级的；而且各求解器量"违反了多少"的办法也不同。
- **纳入的都是硬的**：`"low"` 的意思是"冲突时先丢"，不是"尽量接近"。纳入的事实，不论优先级是哪一档，求解器都必须满足；实现自己的美观目标只在纳入的事实之内优化，排在语言的全部描述之后。
- **`"disabled"`**：这条事实在排序之前去掉，不交给求解器。这不是失效：对象照样存在，依赖它的描述不受影响，不给警告，后写的优先级可以把它重新打开。
- **写法**：
  - PLACE、ORIENT、四种原子约束、ORIENT-RELATION、AS、ATTACH 都收参数 `priority=`，优先级跟着这条事实。`priority` 是 AS、ATTACH 唯一的标准参数；定义的参数不能叫这个名字。
  - PRIORITY 语句用来事后改优先级。它只收有自己句柄的对象：约束、用例，以及带 GROUP 的集合（只能开关，11.5）。收到带 GROUP 的集合时作用于分组本身；别的集合逐个成员处理（5.2.1）。收到元件或别的对象时失效（种类不对）：`#M1 @PLACE(…) => @PRIORITY(…)` 是错误，因为 PLACE 沿用 M1 的句柄；优先级要写成 PLACE 的参数。
  - 记录头 `@RNL(priority=…)` 给记录里收这个参数、又没写它的语句一个默认值，也给 TEXT 的 `dx`、`dy`（13.3）。给 AS、ATTACH 的只有 `"disabled"`，所以关掉的记录连同其中的用例和它们的体一起关掉，和记录里别的语句一样。记录头是 `"high"`、`"normal"` 或 `"low"` 时，用例只在它自己的语句或 PRIORITY 语句给了优先级时才有优先级，否则它的体保持各自的优先级，和规则生成它时一样，所以求解器写出的用例（10.5）和生成的那个意思相同。用例明写的优先级照旧盖过它的体（见下）。
- **覆盖的次序**，从低到高：记录头的默认值；语句自己的参数；PRIORITY 语句（几条之间后写的生效；包里的按 USE-PACKAGE 的位置）；外层用例的优先级；从用例外面经 `::` 直接写在它的某个局部对象上的 PRIORITY 语句（几条之间后写的生效）。这些都没有时，优先级是 `"normal"`。
- **用例的优先级向下传**：用例的优先级（它自己的参数，或写在它上面的 PRIORITY 语句）作用于它展开出的全部事实，包括嵌套的用例展开出的，并盖过它们自己的优先级；嵌套时，最外层带优先级的用例说了算。要只调里面的一部分，就从外面经 `::` 直接写在那个局部对象上（9.4）：它比用例的优先级更具体，所

以盖过用例的优先级，越具体的越优先。这种语句若写在外层定义的体里，仍受外层用例的优先级管。记录里的每条语句都能直接改，所以记录头只给默认值。

  - 完整地说：用例的优先级管住它的定义体里写的全部 PRIORITY 语句，这些语句不再起作用。从外面经 `::` 直接写在局部用例上的 PRIORITY 也这样管住：那个用例算作带优先级的最外层，这条语句对它展开出的全部事实说了算。没被管住的语句之间，写在最里层对象上的胜出；写在同一对象上的，从产生这个对象的定义体外面写的，胜过写在这个体里的；再比后写的胜出。
  - 例如 `%o1` 为 disabled，顶层写 `%o1::in @PRIORITY("high")`，`%o1` 的定义体里写 `%in::row @PRIORITY("low")`：`row` 的事实是 `"high"`。体里那句受 `%o1` 的优先级管，而顶层那句为 `in` 盖过了这个优先级。`%o1` 若没有优先级，结果是 `"low"`：体里那句写在更里层的对象上。要两种情形都是 `"low"`，从外面写 `%o1::in::row @PRIORITY("low")`。

- **求解器的输出**：求解器把结果写成 PLACE、ORIENT 等，用 `"high"`，放在这一层自己的文字的最后一个记录里，如 `@RNL(type="generated", priority="high") {…}`。子电路写在它的 `.subckt … .ends` 之内，哪怕它在被 include 的文件里；顶层写在主文件的文件层（第 17 章）。求解器求解时，不把 `type="generated"` 的记录算进解释（2.8），所以旧结果不会约束下一次求解；求完后删掉它们，在最后写上新的结果。
  - 两次求解之间，新写的 normal 语句若与求解结果冲突，会被丢弃，下一次求解后才生效。查看器可以按记录头的 hash 建议重新求解；这类丢弃不必逐条报告。
  - 编辑器里的拖动写成 `"high"` 的 PLACE，同样写在这一层自己的文字里，放在求解器的记录之后。这就把位置固定住了：以后每次求解都把它当硬约束。要放开，删掉那条语句。
- **求解器写回什么**：求解器的记录应包含它定下的、文本里没定的每个位置、朝向和走法：有名字或引用名的对象用名字或引用名，别的用编号（5.9），导线的走法用 WIRE 改写（13.1），规则生成的对象写成显式语句（10.5）。没写进记录的决定不算结果：那个对象仍是自由的，如同这一层没求过解。所以不求解的查看器画出的，正是求解器画的那张图。求解器引用不到的，它不去定，如生成规则直接产生的、没有引用名的文字的位置；这样的文字由查看器放（13.3）。
- **求解器的结果就是 RNL**：求解器以别的形式把结果交给别的工具时，比如求解器与查看器之间的消息，这种形式应只含有求解器的记录能含有的东西，使任何结果都能存成这样一条记录。
- **求解器的优化目标**只在满足全部保留下来的事实的摆法里挑；保留下来的事实不论优先级都是硬的。优化目标不属于语言：目标的种类和次序因求解器而异，本标准定不出它们的共同词汇。求解选项是 hash 所覆盖的输入的一部分（第 17 章），存进文件本有利于复现结果；本版没有扩展机制（16.4），所以求解器把自己的设置放在它自己的文件里。

14.9 朝向关系：ORIENT-RELATION

```
#M1 #M2 @ORIENT-RELATION("MY");                  /* M2 is drawn as M1 flipped left-right */
{#M1 #M3} {#M2 #M4} @ORIENT-RELATION("MY");      /* M1 and M3 alike, M2 and M4 alike, the two
pairs mirror images */
#M5 {#M7 #M8} @ORIENT-RELATION("R0")             /* M5, M7 and M8 have the same orientation */
```

- `A B @ORIENT-RELATION(relation)`：两个槽位 `from`、`to`，各是一个对象或一个集合。关系是 ORIENT 的八个朝向码之一（11.2），按位置写。B 的朝向等于先按 A 的朝向变换、再做这个关系的变换。变换在图的坐标系里做，所以 `"MY"` 总是关于竖直线左右翻，与 A 本身转没转无关。
- 一边是集合时，这一边的成员朝向都相同，两边之间满足关系。要一个集合全部同向，就把其中一个成员拿出来，如 `#M5 {#M7 #M8} @ORIENT-RELATION("R0")`，或者把集合写两遍：`{#M5 #M7 #M8} {#M5 #M7 #M8} @ORIENT-RELATION("R0")`。
- 没有"只给一个集合、集合里朝向都相同"的写法：关键字说的是两者之间的关系，只给一个集合时看不出关系作用在什么上。只写一个对象与签名不符，是错误。只写一个集合时，照 5.2.1 先拆一层，所以 `{#M1 #M2} @ORIENT-RELATION("MY")` 就是 `#M1 #M2 @ORIENT-RELATION("MY")`；拆出来不是两个对象时是错误。
- 成员是有朝向的对象，与 ORIENT 相同（11.2）：元件、叠画、有符号的用例。端子、点、网络、导线、没有符号的用例是错误：这条约束失效。一边是空集合时这条约束自然成立。
- 它是约束：产生一个约束对象，可以命名，收 `priority=`，可以用 PRIORITY 改（14.8）；冲突时整条作为一个单位保留或丢弃（8.5）。写了 ORIENT 的成员朝向已经定了；别的成员由求解器在满足关系的朝向里选。

- 它不定实际的朝向：两个 MOS 互为镜像时，栅极朝内、朝外都满足 `"MY"`。要定下来，再写一个 ORIENT，或者端子上的 ORDER（14.3）。

15 电气语义

15.1 语义套用

```
%dp1 = #M1(in_p) #M2(in_n) #M5(tail) @AS $DIFFERENTIAL_PAIR;
%cm1 = {#M3 #M9}(ref) {#M4 #M6}(out) @AS $CURRENT_MIRROR;
%in = {#M8 #M5} @AS $MATCHED
```

- 对象的含义用 AS 套用语义定义来标记（9.5）：套用产生一个用例，对象填进定义的角色表里的各个角色。标记含义与写布局关系是同一种语句。
- 用例可以作为别的用例的成员（嵌套），见 9.4、10.1 的例子。
- 集合角色在角色表里写成 `"{name}"`（如例子里 `$MATCHED` 的 `"{members}"`，`$DIFFERENTIAL_PAIR` 的 `"{tail}"`，`$CURRENT_MIRROR` 的 `"{ref}"` 和 `"{out}"`），收一个集合，表达个数不定的成员（9.4）。
- 角色可以声明收哪种对象，如 `roles=[["in_p", "instance"], ["in_n", "instance"]]`；填了种类不对的对象，整个套用失效（9.4）。体为空的定义不检查角色填的是什么，不声明就查不出填错。
- 用例的角色名是它的子对象，如 `%dp1::in_p`。
- 套用定义时展开它的体，体里可以有约束，如优先级为 `"low"` 的默认画法，用例的优先级能关掉它（9.5、14.8）。
- 不校验：不检查含义是否符合网表的连接、类型或参数（原则 2.4）。如 `#M3(ref) #M4(out) @AS $CURRENT_MIRROR` 不检查 M3 的栅极和漏极是否相连。
- 选出某种含义的用例用 SELECT-USAGE，选出含有某个对象的用例用 SELECT-ANCESTOR-USAGE；用例的成员用以用例为范围的选择取到（10.1）。
- 语义定义的词汇归标准包（9.5），标准包提供 `$DIFFERENTIAL_PAIR` 等定义。
- 单条描述的基本合法性（原则 2.4）是：它的引用找得到，它的角色收到种类正确的对象，它的单个角色都填了，它展开出的每条语句自身合法（9.5）。不检查两个角色是否填了同一个对象。

15.2 电流方向：CURRENT

```
#M5:1 @CURRENT("in");                /* drain: the current flows in */
#M5:3 @CURRENT("out")                /* source: the current flows out */
```

- CURRENT 设定端子上静态电流的方向：`"in"`（流入）、`"out"`（流出）或 `"unspecified"`（不定，用来覆盖包里规则给的值）。参数是 `direction`。
- 它是关于端子的事实。它与 ORIENT 一样是修改语句，沿用输入的句柄。它有键，每个端子一个，后写的生效。从没写过它的端子就是不定。
- 它只给在端子上。多端子器件内部的电流拆不清楚：三端器件说"a 流向 b、b 流向 c"，b 处是流入还是流出说不清；而端子上静态电流的方向总有确定的含义。
- 它只收宿主的端子，含本层的端口：`#:inp @CURRENT("in")` 的意思是电流从这里流进子电路。虚拟元件的端子电气上不存在：收到时 CURRENT 失效。没有引脚的端子也可以（与导线不同）。
- 不校验：既不对照网表检查，也不检查各端子的方向是否自洽（原则 2.4）。
- 用途：电流流出时，端子上电流的走向就是导线离开引脚的方向（9.2.4，随符号的朝向转）；流入时反过来。求解器可以用它在摆法里挑，比如尽量让电流朝下（14.8）；朝上还是朝下是风格。没有写 CURRENT、写的是 `"unspecified"`，或者引脚没写 `direction=`（9.2.4）的端子，不给出方向；实现从图形推断出来的方向，在这里不算。规则可以按端子的电流方向选端子，用 SELECT-TERMINAL 的条件 `current`（10.1）。
- **箭头**：CURRENT 不画东西。按 CURRENT 画不画箭头、怎样画，是查看器的偏好（属于它的风格），与显示哪些网名同类（13.3）。查看器自己画时，跳过已有标准电流箭头用例的端子，就像它认得标准的 `$NO_CONNECT`（12.6）。要保留的箭头（导入的原图里的，或作者定下的）写成显式的对象：

```
$CURRENT_ARROW @ATTACH #M5:1 => @ORIENT("R270") => @TEXT("I_{SS}", dx=1);   /* on the drain pin,
with I_SS beside it */
```

```
$CURRENT_ARROW @ATTACH (12, 20) => @ORIENT("R270")                    /* in the middle of
a wire, at absolute coordinates */
```

- 画在端子上的箭头是主对象为端子的用例（12.1）。它不跟着引脚转，所以朝向要写出来；旁边的文字附着在这个用例上（13.3）。
- 导线中间的箭头不锚定在导线上：它画在一个点上，用绝对坐标。
- 流入和流出怎样区分，箭头的定义和符号怎样画，归标准包。符号可以只是一条末端带箭头的线（9.2.5）。
- 画出来的箭头与 CURRENT 分开：两者若是一回事，导出一个箭头就会改变布局的目标。

16 文件、版本、包与代称

本章讲文件的语言版本、包和包文件、标准包、未知内容，以及代称。

16.1 版本

```
* @RNL(type="header") { @VERSION("0.3")}
```

- **一个文件一种语言版本**。不同版本的 RNL 写在不同的文件里，用 include 连起来（6.1）。
- **写在头段记录里**。VERSION 只能写在头段记录（`type="header"`）里，这条记录是文件的第一条记录（2.8）。版本只能声明一次，所以一个文件不会有两个版本值。写在别处的 VERSION 是错误，并被忽略。
- **读的次序**：
  1. 宿主适配器给出注释；
  2. 按记录的外层写法切出全部记录，读出它们的记录头，这一步与版本无关；
  3. 第一条记录是头段记录时，读出它的 VERSION；
  4. 其余记录按这个版本解析。
- **与版本无关的部分**。注释载体、`@RNL` 入口、记录头的写法、花括号配对，以及头段记录里 VERSION 的写法，以后的版本都不改；否则解析器找不到版本。支持新的宿主时只增加载体，不改已有的（2.2）。头段记录里别的内容按声明的版本读。
- **没写头段记录**。头段记录可以不写，头段记录里的 VERSION 也可以不写。这时被 include 的文件取 include 它的文件的版本；主文件取工具默认的版本，并给提示。工具保存文件时应写上头段记录和版本，所以手写的文件可以省略，工具写过的文件都有。包文件应有头段记录（16.2）；没写 VERSION 时，与主文件一样取工具默认的版本。内嵌的包与它所在的文件是同一个版本（16.2）。
- **不支持的版本**。工具不支持某个文件的版本时，这个文件的 RNL 都不解释，报错，保存时原样保留；别的文件照常解释。
- 实现应声明它支持的版本（原则 2.3）。语言版本、包的版本（USE-PACKAGE 的 `version=`）、宿主的版本是三回事。包的版本怎样写、怎样对上，见 16.2。包文件按它自己声明的语言版本读，不论用它的文件是哪个版本。
- **版本号**。版本写成 `"major.minor"`：两个不带符号的十进制整数，如 `"1.0"`、`"1.2"`。没有补丁号、日期或 profile。`"1.0"` 之前的版本是草案，写成 `"0.minor"`；0.x 之间不承诺兼容，各算一个大版本。写错的版本（如 `"1"`、`"v1.2"`）按不支持的版本处理，不当作没写，免得文件按别的版本误读。
- **文件按它声明的版本读**。工具应知道每个标准词是哪个版本加的：在声明了较早版本的文件里，后来才加的词不算标准词。所以以后新增的标准子对象名不影响旧文件里拼写相同的引用名（6.5），新的属性名不影响同名的宿主参数（13.6），AS、ATTACH 新的标准参数不影响同名的定义参数（9.4）。
- **同一大版本里，新的小版本只增加、不改义**。
  - 可以：加关键字、参数、取值、标准词、子对象名、属性名；给原来失效的写法一个含义。附录 C 的各条，以及本标准说是纯放宽的地方，都属于这一类；把整个解释失败的情形（1.5 第 3 条）改为容错也是。
  - 不升大版本就不可以：改变合法写法的含义；删掉任何东西；改默认值；把合法的写法变成错误。
  - 旧文件在新工具里含义不变。新文件在旧工具里，旧版本已有的部分照常读，新增的部分是未知内容（16.4），并给警告，说文件的版本比工具支持的新。工具也可以选择不画这个文件，但不应假装读懂。
- **大版本不同**。工具不支持的大版本的文件，不解释，原样保留（见上）。隐式用在根上的标准包也不起作用（16.3）：这个文件的 RNL 都不解释，查看器按自己的默认外观画网表，并报告版本不受支持。工具可以支持几个大版本。把文件升到新的大版本是显式的迁移，由工具改写文件和它的头段记录；工具不应悄悄按新大版本的规则读旧文件。本标准的每个新大版本都附迁移说明。
- **支持的版本**。工具按大版本说明支持到哪个小版本，如 `"1.0"`–`"1.3"`。支持 `"1.3"` 就是也能读 `"1.0"` 到 `"1.2"`。
- **升版本**。工具往文件里写只有较新版本才有的写法时，把头段记录里的版本升上去。升之前，检查中间新加的标准词是否与文件里已有的名字（文字的引用名、角色名、定义参数、宿主参数）拼写相同，每个这样的名字给一个警告，因为一升版本，它们的含义就可能变；版本照样升。

- **0.2 版**。本版取代 0.2 版。工具可以继续按 0.2 版的规则读 0.2 版的文件，但只写 0.3 版：往 0.2 版的文件里写时，把它升到 0.3 版，并在头段记录里写 `@USE-PACKAGE(name="rnl-std", version="0.2")`，除非这个文件或 include 它的文件已经钉住了标准包，使文件的名字和画法不变（16.3）。

16.2 包与 USE-PACKAGE

```
@USE-PACKAGE(name="acme-style", version="1.2");                    /* looked up in the
package directories of the tool */
@USE-PACKAGE(name="acme-style", version="1.2", path="lib/acme-style.rnl");    /* reads this
package file */
@USE-PACKAGE(name="acme-style", version="1.2") {…}                 /* an inline package
*/
```

包在哪里生效。

- 包的 `$` 定义和规则在写它的 USE-PACKAGE 的那一层及以下各层生效。本版没有只用于本层的 USE-PACKAGE。写在子电路里，包只管这个子电路；写在文件层，管本文件的子电路和它 include 的文件（6.1）；写在头段记录里，范围与写在文件层相同，但排在根上（16.3）。
- 包里的规则作用于用到这个包的每一层（10.4）。
- **排序位置**。包的内容排在它的 USE-PACKAGE 所在的位置，包内再按包的文本先后，与定义展开出的事实取 AS 或 ATTACH 语句的位置同理。冲突的取舍（8.5）和外观队列（8.3）都按这个位置。USE-PACKAGE 写在一层的语句之前，包就是一组默认值；写在它们之后，包里的规则能盖过前面的语句。头段记录里的 USE-PACKAGE 排在根上（16.3）。

包里的名字。只有排序位置跟着 USE-PACKAGE；包里的名字照旧是包自己的。

- 包里的名字先在包自己的定义里找，再在它引用的包里找（见下面的"包引用包"），再在 USE-PACKAGE 那一层生效的标准包里找。不在那一层或上层的用户定义里找，也不在别的包里找；与定义体一样（6.9），USE-PACKAGE 那一层的同名定义不会截走它。所以内嵌的包与同样内容的包文件含义相同，搬出去不用改；包文件可以单独检查，这时用工具默认的标准包。标准包和它直接、间接引用的包只看自己和自己引用的包，不看标准包，免得查找又绕回它们自己。
- 包的 `$` 定义是公开的，按 USE-PACKAGE 那一层的上面一层对待：那一层写了同名的定义就遮蔽包里的，并给提示（6.8）；这不算一层里的第二次绑定。包的 `$` 定义遮蔽上面各层或根上的包的定义时，也给提示（6.8）。
- 包的 `%` 定义是包内私有的。

包里有什么。

- 包里有定义、`$` 集合和规则。包文件的头段记录里写 VERSION 和 PACKAGE，还可以写 USE-PACKAGE（引用别的包，见下）、DEVICE（6.12），以及只管本文件的 ALIAS 和 USE-ALIASES（16.5）。代称包是只有头段记录的包文件，由 USE-ALIASES 加载（16.5）。
- 包没有自己的电路，与被 include 的文件的文件层一样（6.1）：
  - 写具体宿主对象的语句（`#M1`、`&vss`、`#:inp`）失效；单独的 `#` 照旧是规则求值的那一层；
  - 不是规则、只画一个场景对象的语句，如 `(0, 0) @RECTANGLE …`，没有层可画，是错误；
  - 记录头的 `scope=` 和 `type="generated"` 是错误（2.8）；
  - 在包的头段记录以外的记录里写 USE-PACKAGE 是错误：引用别的包只写在头段记录里。
- 内嵌的包的体照这些规则，只是没有头段记录：包名和版本是它的 USE-PACKAGE 的参数，DEVICE 写在文件自己的头段记录里。内嵌的包不能引用别的包；要把一个集成包冻结进文件，就把它展开后的内容按先后写进体里（16.3）。

包文件。包文件怎样读见 2.7，那里有例子。它的头段记录用 `@PACKAGE(name=…, version=…)` 声明这个文件是哪个包、哪个版本；没写 PACKAGE 的文件不是包，用 USE-PACKAGE 加载它是错误。一个包一个文件：分在几个文件里就要在它们之间定先后，本版没有这种规定。包文件按它自己声明的语言版本读（16.1）。

查找。

- 带体的 USE-PACKAGE 是内嵌的包，不查找。

- 写了 `path=` 就读那个文件。路径相对于写这条 USE-PACKAGE 的文件所在的目录，用 `/` 分隔；它指本地文件，不是网址。文件不存在，或它的 PACKAGE 对不上，就按名字找，并给警告。
- 否则到工具设置的包目录里，按 PACKAGE 的名字和版本找。标准包由工具自带，或装在包目录里。包目录是工具的设置，不写进文件；同一个版本有几份时，按这项设置里目录的先后。
- 诊断说出用的是哪个文件、哪个版本。
- 找不到包时，这条 USE-PACKAGE 失效（错误），用到包里名字的语句随之失效（7.3）；诊断给出的原因是缺这个包。
- `name=` 必写。它按整个字符串比较，区分大小写；包名建议只用小写字母、数字和 `-`。
- 有了 `path=`，网表和它的包文件可以放在一起，整体搬走；本版没有给它们用的清单格式。

包的版本。

- 包的版本写法与语言版本相同，`"major.minor"`；每个 0.x 各算一个大版本（16.1）。
- 有写的那个版本就用它。没有时，用同一大版本里比它新的版本中最小的一个，并警告包的版本不同、画面可能不同；同一大版本里只有更旧的版本，或者一个也没有，就是没找到。不用更旧的版本，因为里面可能缺名字。`path=` 指的文件也按这条核对。
- 不写 `version=` 时，用找到的最新版本，并给提示。内嵌的包的 `version=` 只标识版本，不查找。
- 按名字和版本找到的包，整份文档里只读一次：同一个版本在用到的各处是同一组定义。每个内嵌的包各是一份。不同版本里的同名定义是两个定义（9.5）；见 16.3 的例子。
- 建议包的作者照语言的规矩（16.1）：同一大版本里不删、不改已有的名字、角色、参数和引脚；画法可以改。这是约定，工具查不了。
- 本版没有包的内容摘要（如 USE-PACKAGE 的 `digest=` 参数）。要画面完全不变，就钉住版本，或把包内嵌进文件。记录头的 `hash` 怎样算归工具（第 17 章）。

包引用包。包可以引用别的包，这样大的包可以拆成几块，再按几种大小组合起来；标准包就是这样构成的（16.3）。文件里只有头段记录、只引用别的包的包，叫集成包；本标准平时就叫它包。

- **写法**。在包文件的头段记录里，为引用的每个包写一条 `@USE-PACKAGE(name=…, version=…)`。`path=` 相对于这个包文件所在的目录；查找和版本同上。引用不带体；带体的是错误，因为内嵌的内容没有版本，参加不了取最新版本的比较（见下）。
- **排序**。用一个包时，先按它的头段记录里引用的先后，把引用的包展开，放在它前面；引用的包又引用别的包的，同样先展开。然后放这个包自己的内容。在一次展开里，每个包只放一次，放在它第一次出现的位置。所以集成包的内容就是它引用的各包的内容按先后接起来。
- **名字**。引用的包的 `$` 名字在引用它的包里看得到，也转给这个包的使用者；与这个包自己的 `$` 名字一样，按 USE-PACKAGE 那一层的上面一层对待。几个引用的包都定义了 `$X` 时，后引用的遮蔽先引用的，包自己的定义又遮蔽它们全部；每一处遮蔽都给提示，与一层里用了几个包时相同。本版不区分只给包自己用的引用和转给使用者的引用。`%` 名字照旧是各包私有的。
- **只生效一份**。包按名字认，照下面的"哪个 USE-PACKAGE 生效"，经别的包带进来的也算，位置是它展开时所在的位置。
  - 文件直接用的包：照下面的"哪个 USE-PACKAGE 生效"，写在文件里、离这一层最近的 USE-PACKAGE 生效，版本和内嵌的内容都取它的；别的包引用这个包的地方也是这样。它的大版本与引用它的包要求的不同时，给警告：引用它的包可能用到它没有的名字。
  - 只经别的包带进来的包：在同一条层链上的全部引用里，取同一大版本中最新的版本，版本不同时给提示；同一大版本里新版本只增加，所以也能满足较旧的引用。不同大版本的引用是错误：后展开的那条引用失效，用到它的名字的语句随之失效（7.3）。
- **换掉、去掉其中一个**由"只生效一份"推出，不另加规则。写同名的空内嵌包就去掉它，如 `@USE-PACKAGE(name="rnl-std-fill") {}`，它关掉自动补位；写同名的另一个版本或别的内嵌内容就换掉它。被去掉的包若被别的包用到，那些包里用到它的名字的语句失效。
- **成环**。引用成环是错误，闭合这个环的那条引用失效。哪条引用闭合环，取决于展开从哪里开始；环上的几个包都被直接用到时，每个各报一次这个环。

- **诊断报在哪里**。版本不同的提示和大版本不同的警告，报在文件里写的那条 USE-PACKAGE 上（包里的提示和警告不给用户看，7.6）。引用失效的错误报在那条引用上，并说明是文件里哪条 USE-PACKAGE 把它带进来的。
- 代称包不能被引用（16.5）。包文件头段记录里的 USE-ALIASES 照旧只管这个文件。

几个包里的同一个 `$` 名字。

- 一层里用了几个包，都定义了 `$X` 时，后用的遮蔽先用的，并给提示。
- 查名字的次序：本层 → 本层用的包（后写的先查）→ 上一层 → 上一层用的包 → ..... → 根上的包（排在后面的先查）。
- 本版没有包限定的名字、给包另起的名字或层级标签（如 `$acme.DIFFERENTIAL_PAIR`、`as=`）。注：以后的版本要加它们，需要名字后面加 `.` 以外的写法，因为名字后面紧跟一个点在句末很常见（`align #M1 and #M2.`）。

哪个 USE-PACKAGE 生效。从根往下到一层的层链上，一个包只生效一份。写在离这一层最近的那一层的 USE-PACKAGE 生效，版本和内嵌的内容也取它的；同一层里后写的算近，头段记录里的算写在文件层的最前面。经别的包带进来的包，算写在带它进来的那条 USE-PACKAGE 处，版本按上面"包引用包"选。包排在那条 USE-PACKAGE 的位置；那条 USE-PACKAGE 在头段记录里时，排在根上。

注：否则包里的生成规则（如补端口）会在下层再跑一遍，把它们的对象生成两份。

重新使用包。子电路或库文件可以在它的文件层或子电路里重新使用标准包。这时标准包的外观规则排在上层（如主文件）流下来的规则之后，重新盖过它们；包的 `$` 名字按那条 USE-PACKAGE 的上面一层对待，遮蔽上层的同名定义。这一层用到的符号就不受上层外观规则的影响（6.1）。

- 它只影响符号和名字。上层流下来的几何规则和生成规则照样作用；求解结果因此过时时，给提示，建议重新求解（第 17 章）。
- 只想钉住版本、仍让上层的风格盖过包，就把 USE-PACKAGE 写在头段记录里（16.3）。

内嵌的包。USE-PACKAGE 可以带一个体，体就是包的内容。这时文件自带这个包，不查找：

```
@USE-PACKAGE(name="acme-style", version="1.2") {
  $NMOS_ACME = @DEFINE-SYMBOL {$NMOS; (-1, 0) @CIRCLE(radius=2.5)};
  # @SELECT-INSTANCE(type="nmos") => @SET-SYMBOL $NMOS_ACME
}
```

- 内嵌的包与不带体的 USE-PACKAGE 含义相同：包名和包的版本、排在 USE-PACKAGE 的位置、名字照"包里的名字"查找、`$` 名字按上一层对待、`%` 名字私有、按包名认的"只生效一份"，都照上面。体按封装方式收（5.4）。内嵌的包与所在的文件是同一个语言版本（16.1）。
- 内嵌的包与包文件的区别只在：内容写在网表里，没有头段记录。所以它的语言版本跟所在的文件，`version=` 只标识版本，不能声明 DEVICE 或 ALIAS，也不能引用别的包。每个内嵌的包各是一份。
- 内嵌的包通常只用于两件事：去掉一个包，写同名的空内嵌包（16.3）；把整套包冻结进文件，体里是它们展开后的全部内容，自身是完整的。
- 要扩展标准包，或换掉其中一块，就写一个包文件，在它的头段记录里引用它要用的包（见上面的"包引用包"），再在网表里用 `path=` 指向它。用带内容的内嵌包顶替标准包的一个子包时，它算标准包的一部分，只看自己：用不到标准包别的子包里的名字。

其他规则。

- 处理包时不应执行代码，也不应隐式访问网络（原则 2.7）。
- 查看器怎样显示画面，能不能按用户的偏好（如暗色背景、打印）改外观，不归本标准。没有任何样式或参数给值的属性，取渲染器的默认值（9.3，原则 2.7）。

16.3 标准包

- 标准包不属于本标准。它与别的包一样用 RNL 写成，单独发布；它的语义定义、虚拟元件和别的内容由包自己定义。
- 标准包里的名字没有强制前缀，如 `$DIFFERENTIAL_PAIR`。用户可以写同名的定义，按遮蔽取代标准包的（6.8）。
- **集成包**。标准包 `rnl-std` 只有头段记录。它引用最小的集成包 `rnl-std-core`，再引用两个子包：

  - `rnl-std-core` 引用 `rnl-std-style`（样式和默认的样式规则）、`rnl-std-basic` 和 `rnl-std-semiconductors`（符号和定义，包括虚拟元件，如地、电源、端口符号），以及两种方言的规则包，它们给元件配符号：`rnl-std-spice` 按元件字母和模型类型，`rnl-std-spectre` 按 master 和模型类型（见下文的方言）。它只把网表画出来，不加东西。
  - `rnl-std-fill` 有补位用的集合，和补地、补电源、补端口符号的规则；它引用 `rnl-std-basic`。
  - `rnl-std-structures` 有电路结构的定义（如差分对、电流镜）和布局的定义（如 `$MIRRORED`、`$FLOW`）；它们体里的约束，优先级都是 `"low"`（9.5、14.8）。
  - 各子包里的名字互不重复，所以用户不必知道一个名字来自哪个子包。
  - 更全的集成包 `rnl-std-full` 引用 `rnl-std` 和几个领域的包：`rnl-std-functional`（放大器、比较器、转换器等）、`rnl-std-rf`、`rnl-std-logic`、`rnl-std-signal-flow`。可选的 `rnl-std-no-labels` 关掉标准符号的默认标签，给自己写全部标签的文件用；它不在任何集成包里。以后还可以加别的集成包和包。整个标准系列一起发布，用同一个版本号；集成包钉住它的子包的版本。
  - 这是标准系列 0.3 的结构。
- **隐式加载**。标准包 `rnl-std` 用在根上，文件里没写也一样，只有版本不受支持的文件除外（16.1）。它的版本是工具按文件的语言版本给的默认：工具有的、用这个语言版本写的标准系列的最新版本。所以 0.3 版的文件用标准系列 0.3，没写头段、工具按 0.3 读的主文件也一样（16.1）。可以给提示。所以用的版本会随工具升级而变；这个文件、或 include 它的文件的头段记录里的 USE-PACKAGE 可以钉住它，钉住的版本优先（见下）。
  - 注：所以没有任何 RNL 的网表也按标准符号画，写的人不必先写 USE-PACKAGE 就能用标准的定义。标准包是一个包，不是查看器里的代码，所以查看器不必自带元件的符号。
- **另一种方言的规则包**。两种方言里元件的名字意思不同：在 SPICE 里，`I0` 是电流源；在 Spectre 里，元件的名字不说明什么，Virtuoso 把子电路的实例叫作 `I0`。所以在 0.3 版的文件里，另一种方言的规则包隐式去掉：SPICE 方言的文件不用 `rnl-std-spectre`，Spectre 方言的文件不用 `rnl-std-spice`。这相当于文件的头段记录写了同名的空内嵌包，但只管这个文件自己的各层。
  - 文件的方言，是读它第一条宿主语句时的语言模式（2.2）：这条语句是文件里第一行不是注释、不是空行、不是主文件的标题、也不是 `simulator lang=` 的那一行。`.scs` 文件从 Spectre 模式开始，别的文件从 SPICE 模式开始，第一条语句之前的 `simulator lang=` 行改变读这条语句时的模式（2.2）。所以在 Spectre 网表里，`.scs` 文件是 Spectre 方言，除非它的第一条语句在 `simulator lang=spice` 之后；别的文件是 SPICE 方言，除非它的第一条语句在 `simulator lang=spectre` 之后。SPICE 网表的文件都是 SPICE 方言。第一条语句之后的 `simulator lang=` 行不改变这个文件的方言。
  - 往文件里写头段记录不改变它的方言，因为注释是跳过的。工具写头段记录，用写它那个地方的语言模式的注释符（2.2）。文件的第一条语句在一行 `simulator lang=` 之后时，头段记录可以写在这行之后，与文件其余部分的模式一致；也可以写在这行之前，如第 2 行，用文件开头的模式。不论写在哪里，它都必须是文件的第一条记录（2.8）。
  - 每个文件按自己的方言去掉，被别的文件 include 时也一样：Spectre 文件 include 的 SPICE 文件用 `rnl-std-spice`，不用 `rnl-std-spectre`。
  - 写出来的优先：一层的层链上（16.2），不论在本文件还是在 include 它的文件里，写了这个包的 USE-PACKAGE，包本身或同名的内嵌包都算，这一层就不隐式去掉它，那条 USE-PACKAGE 照 16.2 生效。所以写在子电路里的，只为这个子电路和它下面的层留下这个包，文件的其余各层照样去掉。Spectre 文件还想用 SPICE 的规则，就在头段记录里写 `@USE-PACKAGE(name="rnl-std-spice", version="0.3")`。
  - 不用标准包，或用的标准系列里没有这两个包时，什么也不去掉。把标准包冻结成内嵌内容的内嵌包里，也没有叫这两个名字的包（16.2），所以从中什么也不去掉；工具把标准包冻结进 0.3 版的文件时，写的是这个文件实际生效的内容，里面只有它自己方言的规则。
- **根上的包**。主文件或被 include 的文件的头段记录里可以写 USE-PACKAGE：

```
* @RNL(type="header") { @VERSION("0.3"); @USE-PACKAGE(name="rnl-std", version="0.2")}
```

  - 它的范围与文件层的 USE-PACKAGE 相同：写它的文件和这个文件 include 的文件。写在被 include 的文件里的，管这个文件和它再 include 的文件，不管 include 它的文件（6.1）。

  - 它排在根上：在所有层的规则之前，也在子电路符号之前（8.3 第 1 组）。它的 `$` 名字也在根上，所以一层里（如主文件里）的同名定义照样遮蔽它。文件层的 USE-PACKAGE 则排在它自己的位置：那是重新使用这个包（16.2），所以它盖过流下来的规则，它的 `$` 名字按上面一层对待。写在头段记录里，包的意思是"这个文件依赖这个版本，上层的风格照样可以盖过它"；写在文件层，意思是"这个文件用这个包定自己的画法"。
  - 哪个 USE-PACKAGE 生效照 16.2：写在离这一层最近的那一层的生效，头段记录里的算写在文件层的最前面。
  - 根上几个包的先后：标准包在最前；其余按最先写它的文件（由外到内）和那个文件头段记录里的先后。里层文件的头段记录再写同一个包，只改变那个文件的各层用的版本或内嵌内容；位置不变，所以钉版本不改变先后。
  - 头段记录里的包定义了标准包里的、或排在它前面的根上的包里的 `$` 名字时，遮蔽那个定义，并给提示（6.8）。
  - 钉不钉版本由作者选择。建议主文件钉住版本。被 include 的文件不钉就跟随 include 它的文件；建议它只在依赖某个版本时才钉。工具不应自己给被 include 的文件加版本；把 0.2 版的文件升到 0.3 版时写的钉住系列 0.2 除外（16.1）。
  - 建议不要用主文件文件层的 USE-PACKAGE 钉版本：那里的包排在子电路符号之后，而且文件在别处被 include 时，这条 USE-PACKAGE 就是重新使用包（16.2），改变了文件的含义。
- **例**。`tb.sp` 的头段记录写 `@USE-PACKAGE(name="rnl-std", version="1.2")`，`tb.sp` include 的 `ota.sp` 的头段记录写版本 `"1.0"`。
  - `tb.sp` 的顶层和 `tb.sp` 里定义的子电路用 `"1.2"`，`ota.sp` 里的子电路用 `"1.0"`。`tb.sp` 的风格规则照样流到 `ota.sp`，盖过 `"1.0"` 的默认。
  - 默认的选法是标准包里的规则做的，如给 `$GROUND` 的用例配地符号的规则。每个版本在用它的各层里选自己的 `$GROUND`，所以 `ota.sp` 照常画。
  - 两个版本里的同名定义是两个定义（16.2）。所以 `tb.sp` 里按定义选的规则，如 `# @SELECT-USAGE $GROUND => @SET-SYMBOL $MY_GND`，只选得到 `"1.2"` 的 `$GROUND` 的用例；`ota.sp` 不钉自己的版本，才会跟着这条规则。按宿主属性选的规则，如 `# @SELECT-INSTANCE(type="nmos") => …`，不受影响。
  - `ota.sp` 要让自己的符号不受 `tb.sp` 的规则影响，就在文件层使用这个包（16.2）。
- **标准包的多少**。隐式加载的只认包名 `rnl-std`。
  - 要更多：在头段记录里写更全的集成包或单个子包。同名的子包只生效一份，不会生效两份。
  - 要更少：写同名的空内嵌包去掉不要的子包：`@USE-PACKAGE(name="rnl-std-fill") {}` 关掉自动补位。只要最小的集成包，就写空的 `rnl-std` 去掉默认的，再用 `rnl-std-core`：

```
* @RNL(type="header") {
*   @VERSION("0.3");
*   @USE-PACKAGE(name="rnl-std") {};
*   @USE-PACKAGE(name="rnl-std-core", version="0.3")
* }
```

- **扩展标准包，或换掉其中一块**。写一个包文件，在它的头段记录里引用要用的标准子包，再加自己的内容（16.2 的"内嵌的包"）。只添加到标准包上的定义和规则，也可以直接写在网表里；标准包已有的规则，只有同名的包才能去掉或换掉。
- **冻结或不要整个标准包**。在头段记录里写同名的内嵌包。这由按包名认的"只生效一份"推出，不另加规则：
  - `@USE-PACKAGE(name="rnl-std", version="1.2") {…}` 把标准包冻结进文件，文件就自包含了；体是各子包展开后的内容，按展开的先后接起来（16.2）；
  - `@USE-PACKAGE(name="rnl-std") {}` 不用标准包；外观就交给渲染器的默认（原则 2.7）。
- 加粗的总线、总线的分接符号、标总线位宽的标记等约定，也是标准包的内容。没有标准包时，由渲染器决定默认的符号和绘图风格（原则 2.7）。

16.4 扩展与未知内容

- **未知内容**（原则 2.3）。不认识的关键字、参数和取值是错误：这条语句失效，依赖它的语句随之失效（5.5、7.3），其余照常解释（7.6）。记录头的未知字段例外，只给警告（2.8）。画面可能因此不完整，诊断应说出这一点。
- **保存时保留**。原则 2.3 建议原样保留这样的内容；本标准把它定为要求：
  - 不是工具自己的记录不改（第 17 章），宿主正文和普通注释也原样不动；
  - 工具改写自己的记录时，里面它不认识的、失效的语句连同它们的自由文本原样保留，次序不变；
  - 版本不受支持的文件整个原样保留（16.1）；
  - 工具改写的记录不必逐字节不变。
- **本版没有扩展机制**。新的语义、虚拟元件和画法写成包里的定义、符号和样式（1.5 第 7 条），不用新的关键字；要实现必须懂的新机制，等本标准的新版本。工具私有的数据，如编辑器的锁定或视图状态，不写进 RNL 记录；放在工具自己的文件里，或记录之外的普通注释里。
  - 注：本版先让各工具读同一种 RNL；这么早就有厂商方言，会把语言分裂，而新的语义已经能写进包里。
  - 注：以后的版本可以把扩展做成带厂商前缀和 `.` 的关键字，如 `@ACME.ROUTE`，在头段记录里声明：工具不认识的必需扩展会让整个文件不解释，可选扩展则跳过并给提示。这样的关键字现在是不认识的关键字，是错误，所以以后加是纯放宽（16.1）；厂商前缀不能用标准关键字的名字。
- 不为扩展预留写法：标准关键字后面紧跟的 `.` 是自由文本，所以 `@BUNDLE.LOCK` 是 BUNDLE 加自由文本 `.LOCK`，句末的 `{#M1 #M2} @GROUP.` 也合法（3.5）。
- **识别但不解释**。本版这类内容只有记录头的未知字段（2.8）。以后的版本若加扩展，可选扩展是第二种。

16.5 代称

```
* @RNL(type="header") {
*   @VERSION("0.3");
*   @USE-ALIASES(name="rnl-zh");
*   @ALIAS("@at", "@PLACE")
* }
* @RNL {#M1 @位于(10, 20); {#M1 #M2} @对齐("y"); #M3 @at(x=4); #M1(正输入) #M2(负输入) @视作 $"差分对"}
```

- 代称给标准词另起写法，用来兼容别的大小写用法，或按中文等别的自然语言的习惯写 RNL。
- **代称从哪里来**：
  - 直接写在头段记录里的 ALIAS 语句，用于自己起几个代称；
  - 代称包，在头段记录里用 `@USE-ALIASES(name=…, version=…, path=…)` 加载。一种语言的代称做成一个包，各文件不用重写，用词也处处一致。标准的中文代称包是 `rnl-zh`；它随标准包发布，由工具自带（16.3）。
  - 两种可以并用，代称包也可以加载几个；合起来照下面的对应关系检查。
  - 代称只管声明它的文件，不随 include 继承；被 include 的文件或包文件要用代称，各自在自己的头段记录里声明。同一个标准词在不同文件里可以有不同的代称。写在符号里的文字，它的引用名对照写这个符号的文件的代称检查（6.5、13.3）。
  - 头段记录本身永远按标准写法读，不认代称：读到 USE-ALIASES 和 ALIAS 以前还不知道有哪些代称，而且头段记录多由工具写。所以只能写在头段记录里的关键字（VERSION、ALIAS、USE-ALIASES、DEVICE、PACKAGE）起代称没有用。
- **代称包**：
  - 代称包是一个包文件（16.2），它的头段记录里只有 VERSION、PACKAGE 和一串 ALIAS 语句，此外没有别的记录。它的查找和版本的选择照 16.2。
  - 用 USE-ALIASES 加载不是代称包的文件，或用 USE-PACKAGE 加载代称包，是错误。找不到代称包时，它的 USE-ALIASES 是错误，用到它的代称的语句随之失效（7.3）；诊断给出的原因是缺这个代称包。
  - 注：代称包不用 USE-PACKAGE 加载，因为 USE-PACKAGE 可以写在任何一层，管的是定义和规则，按位置排序；而代称管的是怎样读文件的文字，只能按文件算。两者分开，风格包就改不了文字的读法。
  - 代称包的内容随它所指的东西变：标准包里的名字改了，`rnl-zh` 跟着改，版本也升上去。
- **写法**。`@ALIAS("alias", "original")`：代称在前。两个参数是 `alias` 和 `original`，平时按位置写；第三个参数 `definition` 只能写名字（见下）。

- **原名**：
  - 本标准规定的关键字或标准词，按标准的写法。关键字连 `@` 写，如 `"@PLACE"`；别的词不带，如 `"axis"`、`"low"`。代称也这样写：关键字的代称带 `@`，别的代称不带。带不带 `@` 把关键字和拼写相同的标准词分开，代称在正文里的样子也与声明的完全一样。
  - `$` 名字。代称和原名都带 `$`，如 `@ALIAS("$接地补位集", "$GROUND_FILL_SET")`；正文里写 `$"接地补位集"`（`$` 后面的非 ASCII 名字要加引号，3.6），读作 `$GROUND_FILL_SET`。`%` 名字没有代称：写的人可以直接起别的语言的名字，包里的 `%` 名字又是包内私有的。
  - 定义里的名字，用 `definition=` 声明，如 `@ALIAS("正输入", "in_p", definition="$DIFFERENTIAL_PAIR")` 或 `@ALIAS("间距", "gap", definition="$GROUND_STUB")`。这种代称只在那个定义里查：套用它时的角色名（`#M1(正输入) #M2(负输入) @AS $DIFFERENTIAL_PAIR`）、AS 和 ATTACH 传给它的参数名（`$GROUND_STUB @ATTACH(间距=3) …`）、经 `::` 从它的用例取的局部名（`%c1::"中点"`）。`definition` 的值本身也可以写成代称。标准词的代称不用于这些位置，只有 AS、ATTACH 自己的参数 `priority` 除外。
    - 代称按名字认：它们跟着这个 `$` 名字走，在下层用同名定义遮蔽了它的地方也适用，就像 `$` 名字的代称换的是拼写一样。
    - 声明代称时不查原名在不在那个定义里；套用定义时找不到的名字，照常是错误（如"没有角色 in_p"）。
    - `::` 后面的名字按 6.5 的次序找：用例自己的名字、它们的 `definition=` 代称、标准子对象名和它们的代称、文字的引用名。
    - 选择条件 `role=` 的值，和 `${…}` 里沿角色往下取的路径，也认这些代称，每一步在那一步的用例的定义里查，如 `%c1 @SELECT-INSTANCE(role="第一级")` 或 `${第一级::正输入::w}`，其中 `%c1` 是 `$CASCADE` 的用例，它的第一级是一个差分对。范围里有几种定义的用例时，各按各的定义查；原名照样能用。
- **能起代称的**：词表里除下一条所列以外的全部词，即关键字、内置关键字的槽位名、参数名和取值、种类名、子对象名、属性名、样式属性、`true` 和 `false`（5.3）；以及上面说的 `$` 名字和定义里的名字。
- **不能起代称的**：外层写法，即 `@RNL`、记录头的字段名和 `type` 的值、`@VERSION`；以及 `@ALIAS` 和 `@USE-ALIASES` 本身。
- **另起名字**。名字本来就是对象的另一个名字：`$"差分对" = $DIFFERENTIAL_PAIR` 之后，`$"差分对"` 就是同一个定义。这种名字是正文的一部分，去不掉，也管不到角色名、参数名；要一整套别的语言的名字，用代称。
- **对应关系**：
  - 一个原名可以有几个代称；同一个范围里，一个代称只代表一个原名。每个定义的 `definition=` 代称各是一个范围。
  - 代称不应与文件声明的版本里的任何标准词拼写相同，也不应再声明给别的原名。`definition=` 代称不应与那个定义的角色名、参数名、局部名拼写相同，也不应与 AS、ATTACH 自己的参数 `priority` 的代称拼写相同。
  - 代称不能当原名用，也不能与它的原名相同。文件升版本时，新加的标准词与某个代称拼写相同，给警告，版本照样升（16.1）；升了以后，这个代称与新版本的标准词同拼写，照上面的规则是错误。
  - 代称包按它自己声明的语言版本读（16.1）；原名在用这个包的文件的版本里不是标准词时，是错误。
- **代称只增加写法**。标准写法始终有效，同一文件里可以混写。
- **代称只换整个记号**：
  - 代称换的是整个记号，不是文本宏，所以位置和诊断都对得上；
  - 不带 `@` 的标准词按拼写起代称，所以拼写相同的词（如参数名 `x` 和取值 `"x"`）共用代称；
  - `$` 名字的代称换的是拼写：写 `$"接地补位集" = {…}` 的人，就是在那一层定义 `$GROUND_FILL_SET`，它遮蔽标准包的那个，并给提示（6.8）。
- **词法**：
  - 关键字的代称是 `@` 后面紧跟的字母（任何文字）、数字和下划线，不含 `-`；它们组成一个记号，`@` 前面应有空格（2.3），中文标点后面也一样：`对 &b，@选出元件` 是错误，应写成 `对 &b， @选出元件`。后面接中文自由文本时，用一个空格或标点（`，`、`。`）隔开：`@对齐两个器件` 是一个不认识的关键字，是错误，它的诊断应问 `@对齐` 后面是不是漏了空格。记号不按已声明的代称切分，所以读出记号不用先看声明。
  - 不加引号的非 ASCII 词只出现在两处：`@` 后面，和括号里。在 `@` 后面，这个词应是声明过的代称，否则是错误（"非 ASCII 的名字要加引号"）。在括号里（参数名、内置关键字的槽位名、`true` 和 `false`、角色名），声明过的代称就是代称；在角色名或定义的参数名的位置上，不是代称的词就是名字本身（3.6）；在

别的位置上，不是代称的词是错误。别处照 3.6：宿主名字、RNL 名字、端子名和 `::` 后面的名字不加引号时只有 ASCII 字符，用在 `::` 后面的代称也要加引号，如 `#M1::"位置"`。

- 括号里不加引号的代称，与同一位置上可用的某个定义的参数名或角色名拼写相同时，是错误，因为含义不唯一。括号里加了引号的词永远是名字，不是代称。`$` 名字的代称和 `::` 后面的代称本来就写在引号里（3.6）。
- 枚举取值的代称写在字符串里，只在取值是枚举的参数里起作用。`${…}` 里的属性名也认代称（13.6）。别的字符串，如 TEXT 的 `text`，不换。

- **诊断与翻译**：
  - 诊断照文件里写的样子显示词，并附上它们的标准写法。
  - 翻译器两个方向都是机械的。翻成代称写法时，它在头段记录里写 USE-ALIASES；要文件自包含时，改为把用到的代称写成 ALIAS 语句。一个原名有几个代称时，取第一个声明的。翻回标准写法时去掉 USE-ALIASES 和 ALIAS，不丢信息。
  - 自由文本、注释和字符串不翻译，枚举取值和 `${…}` 里的属性名除外。
  - LLM、求解器和编辑器写标准写法。它们改用代称写的文件时，加的语句也用标准写法；混写是合法的。

17 修改模型

- RNL 只描述当前状态，不记录得到它的编辑过程：
  - 没有撤销、删除操作，删除就是编辑文本；
  - 没有相对位移，编辑器写入新的绝对值。
- **谁改哪些记录**。求解器换掉 `type="generated"` 的记录。别的工具只应改 `author` 是它自己的记录，如编辑器整理它自己写过的修改，或重新导入时换掉上次导入写的。工具不应碰没有 `author` 的 authored 记录（2.8）。
- 后追加的描述按先后覆盖较早的（8.4）；几何事实先比优先级（8.5）。`type` 和 `author` 不影响先后（8.3）。
- **输出跟着子电路走**。求解器的输出和用户在编辑器里做的修改，都写进那一层自己的文字：子电路写在它的 `.subckt … .ends` 之内，哪怕它在被 include 的文件里；顶层写在主文件的文件层。求解器用 `priority="high"` 把输出写在这一层自己的文字的最后；它求解时，不把 `type="generated"` 的记录算进解释（2.8），求完后换掉它们（14.8）。它应把自己定下的每个决定都写回（14.8）。`scope=` 只用于作者自己选择写在外面的语句（2.6）。文件只读时保存不保存，由工具决定。
- **结果过时只给提示**。上层或包改了以后，库里子电路的求解结果可能过时，如主文件换了符号。旧结果照常生效；查看器发现记录头的 `hash` 与当前输入不符时，给提示，建议重新求解（2.8）。
  - hash 按这一层继承后的实际输入算：流下来生效的规则、选定的符号、包的版本和求解器的选项，不算 `type="generated"` 的记录，也不算元数据字段。它按层算，不按文件算，也不只算子电路自己的文字，否则主文件的修改就查不出来。
  - hash 不改变解释。它的算法归工具；值建议带算法的标记，如 `"sig1:…"`，工具不认识这个算法就不比较。
  - 注：要是规定 hash 对不上就不用旧结果，它的算法就成了语言的一部分，各工具必须算得一样，不求解的查看器也画不出这一层。
  - 编号是例外：产生无名对象的语句变了，编号的校验值就对不上，用到编号的语句在这一层重新求解之前都不生效（5.9）。校验值只看这一层和它用到的定义的文本，各工具都算得一样，旧结果的其余部分照画。
- **生成的对象**。求解器给规则生成的对象定了位置、接了线以后，就把它们写成显式的语句，与导线一起写进自己的记录。显式的对象优先，补位规则不再生成同一个对象（10.5）。它们的定义体里没有局部名字的对象，用编号引用（5.9）。
- **改求解器做出来的对象**。求解器给它写出的对象起的名字，只绑定在它自己的记录里，而求解时这条记录不算进解释（2.8）。所以别的记录里用这个名字的语句，求解时找不到对象，求解器也就顾及不到它。要单独描述这样的对象，编辑器按 10.5 的说法在自己的记录里手写它，再写上修改，如 `%g_r4 = $GROUND_STUB @ATTACH #R4:2; %g_r4::g @PLACE(30, -4, priority="high")`：stub 的导线让这个端子有了导线，补位跳过它，下次求解不再写出那个对象；在此之前，求解器记录里的那个也照画。要改的若是生成的用例本身，如端口符号，从宿主对象出发的选择在求解时和画图时都找得到它，如 `&out @SELECT-ANCESTOR-USAGE $IO_PORT => @PLACE(40, 20, priority="high")`；用例的局部对象选不到（9.4）。
- 记录头的字段作用于记录里的每条语句（2.8）。
- **例：编辑器里的修改**。下面是一个完整的片段，带宿主的注释前缀。头段记录之后，设计者写了第一条记录，求解器写了第二条；用户在编辑器里把 VIN 往左拖，编辑器追加了第三条：

```
* Common-source stage
* @RNL(type="header") { @VERSION("0.3")}
.model nch nmos level=1
M1 out in 0 0 nch W=4u L=1u
RD vdd out 5k
VDD vdd 0 1.8
VIN in 0 DC 0.6 AC 1
* @RNL {
*   {#RD #M1} @ALIGN("x");
*   #M1 #RD @DISTANCE("bottom-to-top", min=2);
*   &vdd @ADD-TO $SUPPLY_NET_SET
* }
* @RNL(type="generated", priority="high", author="rnl-solver 0.3", hash="sig1:5be1c09a") {
*   #M1 @PLACE(10, 0); #RD @PLACE(10, 8); #VIN @PLACE(0, -2); #VDD @PLACE(20, 4)
```

```
* }
* @RNL(author="rnl-editor 0.1") { #VIN @PLACE(x=-4, priority="high")}
.end
```

  - 拖动写成一条新的记录，追加在求解器的记录之后，优先级是 `"high"`。两者都是 high，所以后写的保留，求解器写的 x 被丢弃，并给提示（8.5）；以后每次求解这一层，新的 x 都是固定的位置（14.8）。
  - 写进设计者的记录不行。那里的语句默认是 `"normal"`，盖不过求解器的 `"high"`；写成 `"high"` 也不行，因为求解器的记录在后面，同一优先级里后写的事实保留。这种失误不是错误，只给一条关于取舍的提示。这个位置要到下次求解这一层时才生效，那时设计者的语句是约束之一；在那之前图上不变。
  - 按位置写的参数在写名字的参数之前（4.2 第 9 条）：`@PLACE(-4, -2, priority="high")` 与 `@PLACE(x=-4, y=-2, priority="high")` 相同，`@PLACE(x=-4, -2)` 是错误。
  - 要改设计者写的内容（换符号、改名字、删约束），直接改设计者的记录。求解器的记录不碰，下次求解这一层时整个换掉。
- 要暂时停用一条语句，就用 `/* … */` 把它包起来。
- **原图往返**。RNL 不保证任意导出成 RNL 再导入后图不变（原则第 1 节）；导出器可以选择写坐标级的精确 RNL（原则第 3 节）。本版不定义什么是精确 RNL，也不定义往返的判据。

附录 A 关键字签名（规范性）

本附录由标准词表 `rnl-words-0.3.json` 生成（5.3）；两者不一致时是勘误。它列出每个关键字收什么、给什么；含义见最后一栏所指的节。

- **对象槽位**按标准写法的次序列出，带槽位名（可以写成角色，5.3）和收什么：*one*：单个对象；收到集合时，对每个成员各做一次（5.2.1），只有 PRIORITY 收到带 GROUP 的集合时作用于分组本身（14.8）；*set*：集合；为这个槽位散写的几个对象合成一个集合（5.2.1）；*any*：一个对象，或作为整体的一个集合；*shape*：一个点，或一组点（图元的两种写法，9.2.5）；*body*：花括号里的语句，封装（5.4）；*definition*：定义：`$` 名字、绑定了定义的 `%` 名字，或匿名定义（5.6、6.6）。吸收的输入是严格的，封装的输入是容错的（5.4、7.3）。
- **种类**是种类名（`instance`、`terminal`、`net`、`point`、`wire`、`usage`、`bundle`、`text`、`annotation`、`label`）或：*any*，任何对象；*level*，单独的 `#`：本层；*owner*，符号体里单独的 `?`：用这个符号画出的对象；*position*，对象的子对象 `::position`（它的原点）；*locatable*，有位置的对象：元件、叠画、有引脚的端子、点、文字、用中心写的图元的中心、有自己位置的画出的用例，或由它们组成的集合（14.1）；*orientable*，有朝向的对象：元件、叠画、有符号的用例，或由它们组成的集合（14.9）；*constraint*，约束关键字产生的对象；*set*，集合，写成名字。
- **参数**标了位置 *1*、位置 *2* 的可以按位置写，按这个次序；其余只能写名字（4.2）。`priority` 的取值总是 `"disabled"`、`"low"`、`"normal"`、`"high"`；不写时用记录头的优先级，只是 AS、ATTACH 从记录头只取 `"disabled"`（14.8）。
- **结果**：语句给出的对象（5.5），以及它的事实怎样覆盖（5.8）。没写覆盖方式的，事实累加。
- 没有注明的，关键字可以写在层的任何记录里；注明也可写在符号体里的，还可以写在 DEFINE-SYMBOL 的体里。

A.1 记录

`@RNL(…) {…}` 把它的语句提交到本层（5.4）。记录头的字段（2.8），都写名字：

字段	取值	默认值	影响解释
`type`	`"authored"`、`"generated"`、`"header"`	`"authored"`	是
`scope`	字符串	—	是
`priority`	`"disabled"`、`"low"`、`"normal"`、`"high"`	`"normal"`	是
`author`	字符串	—	否
`hash`	字符串	—	否
`numbering`	字符串	—	是
`modified`	字符串	—	否

A.2 头段记录与包

关键字	对象槽位	参数	结果	节
`@VERSION`（只在头段记录里）	—	`version`（位置 1；必需）	—	16.1
`@PACKAGE`（只在包文件的头段记录里）	—	`name`（必需），`version`（必需）	—	2.7
`@USE-PACKAGE`（也可写在头段记录里）	`body`：body，可省	`name`（必需），`version`，`path`	—	16.2
`@ALIAS`（只在头段记录里）	—	`alias`（位置 1；必需），`original`（位置 2；必需），`definition`	—	16.5
`@USE-ALIASES`（只在头段记录里）	—	`name`（必需），`version`，`path`	—	16.5
`@DEVICE`（只在头段记录里）	—	`name`（必需），`model`，`terminals`（必需；数组）	—	6.12

A.3 定义

关键字	对象槽位	参数	结果	节
`@DEFINE-SYMBOL`（也可写在符号体里）	`body`：body	`box`（数组）	定义（符号）	9.2
`@DEFINE-STYLE`	`extends`：body（样式）	样式属性（9.3）	定义（样式）	9.3
`@DEFINE`	`body`：body	`roles`（必需；数组），`parameters`（数组）	定义（普通定义）	9.4

A.4 符号内容与图元

关键字	对象槽位	参数	结果	节
`@PIN`（只在符号体里）	`position`：one（point）	`terminal`（必需；端子，或非空的端子数组），`label`，`direction`（`"right"`、`"up"`、`"left"`、`"down"`）	新的引脚	9.2.4
`@LINE`（也可写在符号体里）	`points`：set（point、terminal）	`arrow`（`"none"`、`"start"`、`"end"`、`"both"`；默认`"none"`），`fill`（`"none"`、`"linecolor"`、`"fillcolor"`；默认`"none"`），line 组的样式属性	符号里的一个图元；写在场景里时是新的标注（13.4）	9.2.5
`@RECTANGLE`（也可写在符号体里）	`points`：shape（point、terminal）	`width`（数），`height`（数），`fill`（`"none"`、`"linecolor"`、`"fillcolor"`；默认`"none"`），area 组的样式属性；`width`、`height` 同时写或都不写	符号里的一个图元；写在场景里时是新的标注（13.4）	9.2.5
`@CIRCLE`（也可写在符号体里）	`points`：shape（point、terminal）	`radius`（数），`fill`（`"none"`、`"linecolor"`、`"fillcolor"`；默认`"none"`），area 组的样式属性	符号里的一个图元；写在场景里时是新的标注（13.4）	9.2.5
`@ELLIPSE`（也可写在符号体里）	`points`：shape（point、terminal）	`width`（数），`height`（数），`fill`（`"none"`、`"linecolor"`、`"fillcolor"`；默认`"none"`），area 组的样式属性；`width`、`height` 同时写或都不写	符号里的一个图元；写在场景里时是新的标注（13.4）	9.2.5
`@ARC`（也可写在符号体里）	`points`：set（point、terminal）	`arrow`（`"none"`、`"start"`、`"end"`、`"both"`；默认`"none"`），`fill`（`"none"`、`"linecolor"`、`"fillcolor"`；默认`"none"`），line 组的样式属性	符号里的一个图元；写在场景里时是新的标注（13.4）	9.2.5
`@BEZIER`（也可写在符号体里）	`points`：set（point、terminal）	`arrow`（`"none"`、`"start"`、`"end"`、`"both"`；默认`"none"`），`fill`（`"none"`、`"linecolor"`、`"fillcolor"`；默认`"none"`），line 组的样式属性	符号里的一个图元；写在场景里时是新的标注（13.4）	9.2.5
`@TEXT`（也可写在符号体里）	`attach`：one（point、instance、usage、bundle、	`text`（位置 1；必需），`key`，`dx`（数），`dy`	新的文字；写了 `key` 时是附着对象的子对象；按键	13.3

关键字	对象槽位	参数	结果	节
	terminal、wire、annotation、net、`?`）	（数），`halign`（`"start"`、`"middle"`、`"end"`；默认 `"start"`），`valign`（`"baseline"`、`"top"`、`"middle"`、`"bottom"`；默认 `"baseline"`），`rotate`（`0`、`90`、`180`、`270`；默认 `0`），text 组的样式属性	覆盖（键：附着对象和引用名，写了 key 时）	

A.5 放置与外观

关键字	对象槽位	参数	结果	节
`@PLACE`（也可写在符号体里）	`target`：one（instance、usage、bundle、point、text、`::position`）	`x`（位置 1；数），`y`（位置 2；数），`priority`；`x`、`y` 至少写一个	输入的对象；几何事实（键：对象和轴）	11.1
`@ORIENT`（也可写在符号体里）	`target`：one（instance、usage、bundle）	`code`（位置 1；必需；`"R0"`、`"R90"`、`"R180"`、`"R270"`、`"MX"`、`"MY"`、`"MXR90"`、`"MYR90"`），`priority`	输入的对象；几何事实（键：对象）	11.2
`@LAYER`	`target`：one（instance、usage、bundle、wire、annotation、text）	`layer`（位置 1；必需；`"bottom"`、`"middle"`、`"top"`）	输入的对象；按键覆盖（键：对象）	11.6
`@SET-SYMBOL`（也可写在符号体里）	`target`：one（instance、usage、bundle、`#`）；`symbol`：definition（符号）	—	输入的对象；外观队列	11.3
`@SET-STYLE`	`target`：one（instance、usage、bundle、wire、annotation、text）；`style`：definition（样式）	—	输入的对象；外观队列	11.3

A.6 选择

关键字	对象槽位	参数	结果	节
`@SELECT-INSTANCE`	`scope`：any（`#`、任何对象）	`name`，`model`，`type`，`role`，`recursive`（布尔；默认 `true`）	新的集合，成员是元件	10.1
`@SELECT-TERMINAL`	`scope`：any（`#`、任何对象）	`name`，`model`，`type`，`terminal`（端子），`net`，`wired`（布尔），`current`（`"in"`、`"out"`、`"unspecified"`），`role`，`recursive`（布尔；默认 `true`）	新的集合，成员是端子	10.1
`@SELECT-NET`	`scope`：any（`#`、任何对象）	`name`，`port`（布尔），`wired`（布尔），`global`（布尔），`role`，`recursive`（布尔；默认 `true`）	新的集合，成员是网络	10.1
`@SELECT-WIRE`	`scope`：any（`#`、任何对象）	`net`，`bus`（布尔），`role`，`recursive`（布尔；默认 `true`）	新的集合，成员是导线	10.1
`@SELECT-LABEL`	`scope`：any（`#`、任何对象）	`key`，`role`，`recursive`（布尔；默认 `true`）	新的集合，成员是标签	10.1
`@SELECT-ANNOTATION`	`scope`：any（`#`、任何对象）	`role`，`recursive`（布尔；默认 `true`）	新的集合，成员是标注	10.1

关键字	对象槽位	参数	结果	节
`@SELECT-USAGE`	`scope`：any（`#`、任何对象）；`definition`：definition（普通定义），可省	`recursive`（布尔；默认 `true`）	新的集合，成员是用例	10.1
`@SELECT-ANCESTOR-USAGE`	`scope`：any（任何对象）；`definition`：definition（普通定义）	`recursive`（布尔；默认 `true`）	新的集合，成员是用例	10.1

A.7 套用

关键字	对象槽位	参数	结果	节
`@AS`（也可写在符号体里）	定义的各角色，然后是定义（`definition`，写在最后）	定义的参数（按参数表的次序按位置写，或写名字；9.4），`priority`	新的用例	9.4
`@ATTACH`（也可写在符号体里）	定义，然后是它的各角色（`definition`，写在最前）	定义的参数（按参数表的次序按位置写，或写名字；9.4），`priority`	新的用例	9.4

A.8 导线

关键字	对象槽位	参数	结果	节
`@WIRE`	`net`：any（net）；`points`：set（terminal、point）	`mode`（`"auto"`、`"straight"`、`"horizontal-vertical"`、`"vertical-horizontal"`、`"manhattan"`、`"manhattan45"`；默认 `"auto"`），wire 组的样式属性	新的导线；收到导线时改写它，按键覆盖（键：这个对象）	13.1
`@ROUTE`	`net`：one（net）	`mode`（`"auto"`、`"straight"`、`"horizontal-vertical"`、`"vertical-horizontal"`、`"manhattan"`、`"manhattan45"`；默认 `"auto"`），wire 组的样式属性	新的布线	13.7

A.9 集合、分组与叠画

关键字	对象槽位	参数	结果	节
`@BUNDLE`	`members`：set（instance）	—	新的叠画	11.8
`@GROUP`	`members`：set（任何对象）	—	输入的对象	11.5
`@ADD-TO`	`items`：any（任何对象）；`target`：any（set）	—	目标集合	5.2.2
`@REMOVE-FROM`	`items`：any（任何对象）；`target`：any（set）	—	目标集合	5.2.2
`@UNION`	`sets`：set（任何对象）	—	新的集合	5.2.2

A.10 约束与优先级

关键字	对象槽位	参数	结果	节
`@ALIGN`	`members`：set（locatable）	`axis`（位置 1；必需；`"x"`、`"y"`），`priority`	新的约束；几何事实	14.2
`@ORDER`	`members`：set（locatable）	`direction`（位置 1；必需；`"left-to-right"`、`"right-to-left"`、`"bottom-to-top"`、`"top-to-bottom"`、`"x"`、`"y"`），`priority`	新的约束；几何事实	14.3
`@DISTRIBUTE`	`members`：set（locatable）	`axis`（位置 1；必需；`"x"`、`"y"`），`priority`	新的约束；几何事实	14.4

关键字	对象槽位	参数	结果	节
`@DISTANCE`	`from`：any（locatable）；`to`：any（locatable）	`direction`（位置 1；必需；`"left-to-right"`、`"right-to-left"`、`"bottom-to-top"`、`"top-to-bottom"`、`"x"`、`"y"`），`min`（数），`max`（数），`exact`（数），`priority`；`min`、`max`、`exact` 至少写一个；`exact` 不与 `min` 同写；`exact` 不与 `max` 同写	新的约束；几何事实	14.5
`@ORIENT-RELATION`	`from`：any（orientable）；`to`：any（orientable）	`relation`（位置 1；必需；`"R0"`、`"R90"`、`"R180"`、`"R270"`、`"MX"`、`"MY"`、`"MXR90"`、`"MYR90"`），`priority`	新的约束；几何事实	14.9
`@PRIORITY`	`target`：one（constraint、usage、set）	`priority`（位置 1；必需）	输入的对象；按键覆盖（键：对象）	14.8

A.11 电气语义

关键字	对象槽位	参数	结果	节
`@CURRENT`	`terminal`：one（terminal）	`direction`（位置 1；必需；`"in"`、`"out"`、`"unspecified"`）	输入的对象；按键覆盖（键：端子）	15.2

A.12 其他标准词

- 种类：`instance`、`terminal`、`net`、`point`、`wire`、`usage`、`bundle`、`text`、`annotation`、`label`（5.3、9.4）。
- `::` 后面的子对象：`position`、`net`（6.5）。
- `${…}` 读的属性：`name`、`value`、`model`、`type`、`net`、`x`、`y`（13.6）。
- 真假值：`true`、`false`（3.7）。

样式属性（9.3）：

属性	类型和取值	默认值
`linecolor`	颜色	—
`linestyle`	`"solid"`、`"dashed"`、`"dotted"`、`"dash-dot"`、`"double"`、`"bold"`、`"none"`	`"solid"`
`linewidth`	数	`0`
`cap`	`"butt"`、`"round"`、`"square"`	—
`join`	`"miter"`、`"round"`、`"bevel"`	—
`fillcolor`	颜色	—
`textcolor`	颜色	—
`size`	数	—
`font`	字符串	—
`bold`	布尔	—
`italic`	布尔	—
`arrowsize`	数	—

关键字可以当参数收的样式属性（9.2.5）：*line*：`linecolor`、`linestyle`、`linewidth`、`cap`、`join`、`fillcolor`、`arrowsize`；*area*：`linecolor`、`linestyle`、`linewidth`、`cap`、`join`、`fillcolor`；*wire*：`linecolor`、`linestyle`、`linewidth`、`cap`、`join`；*text*：`textcolor`、`size`、`font`、`bold`、`italic`。

附录 B 标准例句（说明性）

标准例句是在标准写法的记号之间加上自由文本、读起来像句子的语句。自由文本不改变含义（3.3）；措辞的规则见 4.5。表里句首的 `…` 代表管道里它前面的段，`{…}` 和句末的 `…` 代表省略的内容。

中文版每条列出英文例句和中文例句。中文例句用代称包 `rnl-zh` 的代称（4.5、16.5）。

B.1 放置与朝向

关键字	情况	英文	中文
PLACE	两轴	`#M1 is at @PLACE(10, 20)`	`#M1 @位于(10, 20)`
	一轴	`the x of #M1 is at @PLACE(x=10)`	`#M1 的 x 坐标 @位于(x=10)`
	编辑器写的	`the x of #X2 is at @PLACE(x=3, priority="high")`	`#X2 的 x 坐标 @位于(x=3, 优先级="高")`
	管道	`… => it is at @PLACE(40, 0)`	`… => 其 @位于(40, 0)`
ORIENT	转	`#M2 @ORIENT("R90")`	`#M2 @朝向("R90")`
	镜像	`#M2 @ORIENT("MX"), i.e., flipped upside down`	`#M2 @朝向("MX")，即上下镜像`
		`#M2 @ORIENT("MY"), i.e., flipped left-right`	`#M2 @朝向("MY")，即左右镜像`
		`#M3 @ORIENT("MXR90"), i.e., flipped upside down, then rotated 90°`	`#M3 @朝向("MXR90")，即上下镜像再转 90°`
		`#M3 @ORIENT("MYR90"), i.e., flipped left-right, then rotated 90°`	`#M3 @朝向("MYR90")，即左右镜像再转 90°`
	管道	`… => they @ORIENT("R90")`	`… => 其 @朝向("R90")`
ORIENT-RELATION	两个对象	`from #M1 to #M2, they have @ORIENT-RELATION("MY"), i.e., flipped left-right`	`#M1 到 #M2 的 @朝向关系("MY")，即左右镜像`
	集合	`from {#M1 #M3} to {#M2 #M4}, they have @ORIENT-RELATION("MY"), i.e., flipped left-right`	`{#M1 #M3} 到 {#M2 #M4} 的 @朝向关系("MY")，即左右镜像`
	朝向相同	`from #M5 to {#M7 #M8}, they have @ORIENT-RELATION("R0"), i.e., the same orientation`	`#M5 到 {#M7 #M8} 的 @朝向关系("R0")，即朝向相同`
LAYER		`#X is on @LAYER("bottom")`	`#X 处于 @图层("底")`
	管道	`… => they are on @LAYER("top")`	`… => 其处于 @图层("顶")`
GROUP		`{#M1 #M2 #M5} as a @GROUP`	`{#M1 #M2 #M5} 视作一个 @分组`
	命名	`%s1_set = {#M1 #M2 #M5} as a @GROUP`	`%s1_set = {#M1 #M2 #M5} 视作一个 @分组`

B.2 布局约束

关键字	情况	英文	中文
ALIGN	y 相同	`{#M1 #M2} @ALIGN("y"), i.e., same y`	`{#M1 #M2} @对齐("y")，即 y 坐标相同`
	x 相同	`{#M1 #C1} @ALIGN("x"), i.e., same x`	`{#M1 #C1} @对齐("x")，即 x 坐标相同`
	低优先级	`{#M7 #M5} @ALIGN("y", priority="low"), i.e., same y`	`{#M7 #M5} @对齐("y", 优先级="低")，即 y 坐标相同`
	管道	`… => they @ALIGN("y")`	`… => 其 @对齐("y")`
ORDER	有方向	`{#M1 #M2 #M3} are in @ORDER("left-to-right")`	`{#M1 #M2 #M3} 按 @顺序("从左到右") 排列`
	不分方向	`the x of {#M1 #M1:2 #M2:2 #M2} are in @ORDER("x"), either ascending or descending`	`{#M1 #M1:2 #M2:2 #M2} 的 x 坐标按 @顺序("x") 排列，正倒序皆可`
	管道	`… => they are in @ORDER("bottom-to-top")`	`… => 其按 @顺序("从下到上") 排列`
DISTRIBUTE		`the x of {#R1 #R2 #R3} @DISTRIBUTE("x"), i.e., in equal steps`	`{#R1 #R2 #R3} 的 x 坐标 @等距("x")`
	管道	`… => the x of them @DISTRIBUTE("x"), i.e., in equal steps`	`… => 其 x 坐标 @等距("x")`
DISTANCE	有方向	`from #M1 to #M2, they have @DISTANCE("bottom-to-top", min=4)`	`#M1 到 #M2 有 @距离("从下到上", 最小值=4)`

关键字	情况	英文	中文
	集合	`from {#M6 #M7 #CL} to %pout, they have @DISTANCE("left-to-right", min=2)`	`{#M6 #M7 #CL} 到 %pout 有 @距离("从左到右", 最小值=2)`
	不分方向	`between %a_set and %b_set, they have @DISTANCE("x", min=1)`	`%a_set 和 %b_set 之间有 @距离("x", 最小值=1)`
PRIORITY		`%c1 has @PRIORITY("disabled")`	`%c1 的 @优先级("停用")`
	管道	`… => they have @PRIORITY("low")`	`… => 其 @优先级("低")`

- `the x of`（或 `the y of`），中文"的 x 坐标"（或"的 y 坐标"），只写在方向或轴是 `"x"`（或 `"y"`）的 ORDER、DISTRIBUTE 前面，以及只给一轴的 PLACE 前面（B.1）；ALIGN 改用注解 `, i.e., same x`，中文"，即 x 坐标相同"。
- 这里的 x 坐标是各成员的位置：元件是 PLACE 定的点（符号的原点），不是外框的中心；集合按中心算（14.1）。

B.3 外观

关键字	情况	英文	中文
SET-SYMBOL	一个对象	`#M1 @SET-SYMBOL to $NMOS`	`#M1 @使用符号 $NMOS`
	集合	`{#M1 #M2} @SET-SYMBOL to $NMOS`	`{#M1 #M2} @使用符号 $NMOS`
	子电路的 block symbol	`this subcircuit # @SET-SYMBOL to $NAND2_SYM`	`本子电路 # @使用符号 $NAND2_SYM`
	管道	`… => they @SET-SYMBOL to $PMOS`	`… => 其 @使用符号 $PMOS`
SET-STYLE		`{#CC #RZ} @SET-STYLE to $orange`	`{#CC #RZ} @使用样式 $orange`
	管道	`… => they @SET-STYLE to $grey`	`… => 其 @使用样式 $"灰色"`
DEFINE-STYLE		`$orange = @DEFINE-STYLE(linecolor="#e07000", textcolor="#e07000") {}`	`$orange = @定义样式(线色="#e07000", 字色="#e07000") {}`
	并入别的样式	`$wide = @DEFINE-STYLE(linewidth=2) based on {$orange}`	`$wide = @定义样式(线宽=2), 以 {$orange} 为基础`

B.4 符号与图元

关键字	情况	英文	中文
DEFINE-SYMBOL		`$TRI = @DEFINE-SYMBOL {…}`	`$TRI = @定义符号 {…}`
PIN		`at (-4, 0) there is a @PIN(terminal="in", direction="left")`	`在 (-4, 0) 有一个 @引脚(端子="in", 方向="左")`
LINE		`a @LINE running through {(0, 0) (4, 0)}`	`一条 @折线 经过 {(0, 0) (4, 0)}`
ARC		`an @ARC running through {(0, 2) (2, 0) (0, -2)}`	`一条 @圆弧 经过 {(0, 2) (2, 0) (0, -2)}`
BEZIER		`a @BEZIER with control points {(0, 0) (1, 1) (2, 1) (3, 0)}`	`一条 @贝塞尔 曲线, 控制点为 {(0, 0) (1, 1) (2, 1) (3, 0)}`
CIRCLE		`centered at (0, 0), a @CIRCLE(radius=1)`	`以 (0, 0) 为中心的 @圆(半径=1)`
RECTANGLE	中心加尺寸	`centered at (0, 0), a @RECTANGLE(width=4, height=2)`	`以 (0, 0) 为中心的 @矩形(宽度=4, 高度=2)`
	两个对角	`a @RECTANGLE with corners {(0, 0) (4, 2)}`	`一个 @矩形 以 {(0, 0) (4, 2)} 为对角`
ELLIPSE	中心加尺寸	`centered at (0, 0), an @ELLIPSE(width=4, height=2)`	`以 (0, 0) 为中心的 @椭圆(宽度=4, 高度=2)`
	外框的对角	`an @ELLIPSE with box corners {(0, 0) (4, 2)}`	`一个 @椭圆 以 {(0, 0) (4, 2)} 为外框对角`
TEXT	符号里	`on ? there is a @TEXT("${name}", key="name") => it is at @PLACE(-2, 2.5)`	`? 上有 @文本("${名字}", 引用名="name") => 其 @位于(-2, 2.5)`
ATTACH	符号里	`$ARROW is @ATTACH to (0, -1.5) => …`	`$"箭头" @附着 在 (0, -1.5) 上 => …`

- 写在场景里的图元（标注，13.4），句子相同。

B.5 导线与文字

关键字	情况	英文	中文
WIRE		`net &n1 has a @WIRE connecting {#R1:2 %j #R2:1}`	`网络 &n1 的 @导线 连接了 {#R1:2 %j #R2:1}`
	总线	`net &D<3:0> has a @WIRE connecting {%p1 %p2}, which is a bus`	`网络 &D<3:0> 的 @导线 连接了 {%p1 %p2}，此为总线`
	走法	`net &in has a @WIRE(mode="vertical-horizontal") connecting {#VIN:1 #R1:1}`	`网络 &in 的 @导线(模式="先竖后横") 连接了 {#VIN:1 #R1:1}`
ROUTE		`net &n1 is fully @ROUTE`	`网络 &n1 全部 @布线`
	集合	`nets {&a &b} are fully @ROUTE`	`网络 {&a &b} 全部 @布线`
	管道	`… => it is fully @ROUTE`	`… => 其全部 @布线`
TEXT	对象上	`on #M1 there is a @TEXT("W/L = ${w}/${l}", key="wl", dy=-3)`	`#M1 上有 @文本("W/L = ${w}/${l}", 引用名="wl", dy=-3)`
	点上	`at (_, _) there is a @TEXT("RC low-pass") => named %title`	`在 (_, _) 有 @文本("RC 低通") => 记作 %title`
	管道	`… => on them there is a @TEXT("${w}", key="size")`	`… => 其上有 @文本("${w}", 引用名="size")`

- 给点命名，如 `%j = (_, _)`，不加词。

B.6 定义与套用

关键字	情况	英文	中文
DEFINE		`$PAIR = a definition, @DEFINE(roles=["a", "b"], parameters=[["gap", 2]]) {…}`	`$PAIR = 一个普通定义， @定义(角色表=["a", "b"], 参数表=[["gap", 2]]) {…}`
AS	两个角色	`#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR`	`#M1(正输入) 和 #M2(负输入) @视作 $"差分对"`
	收集合的角色	`{#M5 #M7 #M8} are treated @AS $MATCHED`	`{#M5 #M7 #M8} @视作 $"匹配组"`
	一个对象	`#X1:3 is treated @AS $SUPPLY_HIGH`	`#X1:3 @视作 $"高压侧电源"`
	三个角色，带参数	`#M1(a), #M2(b) and %ax(axis) are treated @AS(priority="high") $MIRRORED`	`#M1(甲)、#M2(乙) 和 %ax(轴) @视作(优先级="高") $"镜像对称"`
	带普通定义的参数	`{#R1 {%rc} #M1} are treated @AS(direction="top-to-bottom") $FLOW`	`{#R1 {%rc} #M1} @视作(方向="从上到下") $"信号流向"`
	填用例	`%dp(pair) and %cm(load) are treated @AS $OTA`	`%dp(pair) 和 %cm(load) @视作 $OTA`
	命名	`%dp = #M1(in_p) and #M2(in_n) treated @AS $DIFFERENTIAL_PAIR`	`%dp = #M1(正输入) 和 #M2(负输入) @视作 $"差分对"`
	管道里命名、又加对象	`#M1(in_p) and #M2(in_n) are treated @AS $DIFFERENTIAL_PAIR => named %dp => it and %cm are treated @AS $OTA`	`#M1(正输入) 和 #M2(负输入) @视作 $"差分对" => 记作 %dp => 其与 %cm @视作 $OTA`
ATTACH	网络上	`$GROUND is @ATTACH to &vss`	`$"地" @附着 在 &vss 上`
	端子上	`$NO_CONNECT is @ATTACH to #M1:4`	`$"无连接" @附着 在 #M1:4 上`
	点上	`$JUNCTION is @ATTACH to %j`	`$"交汇点" @附着 在 %j 上`
	两个角色	`$DIFF_PORT is @ATTACH to &inp(a) and &inn(b)`	`$DIFF_PORT @附着 在 &inp(a) 和 &inn(b) 上`
	后面接别的、命名	`$GROUND is @ATTACH to &vss => it is at @PLACE(24, 38) => named %g0`	`$"地" @附着 在 &vss 上 => 其 @位于(24, 38) => 记作 %g0`
	添在每个对象上	`$SIZE_LABEL is @ATTACH to {#M1 #M2 #M3}`	`$SIZE_LABEL @附着 在 {#M1 #M2 #M3} 上`
	补位（集合）	`$GROUND_STUB is @ATTACH to $GROUND_FILL_SET`	`$"接地桩" @附着 在 $"接地补位集" 上`
	补位（管道）	`among the terminals at each level in # and below, @SELECT-TERMINAL(net="vss", wired=false) => $GROUND_STUB is @ATTACH(gap=3) to them`	`从本层 # 及以下每一层的端子中 @选出端子(网络="vss", 已接线=否) => $"接地桩" @附着(间距=3) 在其上`

- 对象合起来构成这个东西时用 AS，如电路结构、摆法，或 `$RC_STAGE` 这样的局部画法；这个东西添在对象上时用 ATTACH，如地、端口符号、无连接标记、汇合圆点、每个元件一个的标签、补位。两个关键字做的是同一件事，只是定义写的位置不同（9.4）。

- `gap` 是 `$GROUND_STUB` 的参数：从端子往下到地符号的最小距离，默认 2。它和角色名 `in_p`、`axis` 一样，是定义里起的名字，不是标准词。

B.7 选择

选择一律写成在候选里选：英文 `among <candidates>, @SELECT-…(conditions)`，中文 `从<候选>中 @选出…(条件)`。候选的说法按范围的种类和要选的种类定：

范围	要选的	英文的候选	中文的候选
`#`	任何种类	`the <kind> at each level in # and below`	`本层 # 及以下每一层的<种类>`
`#`， `recursive=false`	任何种类	`the <kind> in # only`	`本层 # 的<种类>`， 句首加"只"：`只从本层 # 的<种类>中`
同一种类的集合（筛）	同种类	`{…}`	`{…}`
网络	元件	`the instances connected to &en`	`接在 &en 上的元件`
网络	端子	`the terminals connected to &en`	`接在 &en 上的端子`
网络	导线	`the wires on &en`	`&en 上的导线`
元件	端子	`the terminals of #R1`	`#R1 的端子`
元件	网络	`the nets connected to #M1`	`与 #M1 相连的网络`
端子	元件	`the instances owning #M1:g`	`#M1:g 所属的元件`
端子	网络	`the nets connected to #M1:g`	`#M1:g 所在的网络`
导线	网络	`the nets of %w13`	`%w13 所在的网络`
元件、端子、导线、网络、图元	标签	`the labels attached to #M1`	`附着在 #M1 上的标签`
用例（成员里没有用例）	任何种类	`the members of %dp`	`%dp 的成员`
用例（成员里有用例，或不知道）	任何种类	`all members of %st1 at any depth`	`%st1 的各级成员`
用例	用例（SELECT-USAGE）	`%st1 itself and the usages inside it`	`%st1 自身及其内部的用例`
任何对象	所在的用例（SELECT-ANCESTOR-USAGE）	`the usages containing #M1`	`包含 #M1 的用例`
几种对象混在一起的集合（种类算不出）		write `for %mixed_set, @SELECT-…` instead	改写成 `对于 %mixed_set， @选出…`

- `<kind>` 按关键字写：instances、terminals、nets、wires、labels、annotations、usages；中文是元件、端子、网络、导线、标签、标注、用例。
- 管道中间的选择，候选里的范围英文写 `them`，中文写"其"，如 `=> among all members of them at any depth,`、`=> 从其各级成员中`。写句子时不知道前一段选出的用例的成员里有没有用例，所以写 `at any depth`、"各级"。
- 用例的成员写 `at any depth`、"各级"，不写 `at any level`、"各层"：层指电路的层，如 `at each level in # and below`、"本层 # 及以下每一层"。
- 包里的规则作用于用到这个包的每一层，所以在包里，`at each level in # and below` 改成 `at every level that uses this package, #`，中文"本层 # 及以下每一层"改成"用到本包的每一层 #"。

关键字	情况	英文	中文
SELECT-INSTANCE	本层及以下	`among the instances at each level in # and below, @SELECT-INSTANCE(model="nch") => they @SET-SYMBOL to $NMOS`	`从本层 # 及以下每一层的元件中 @选出元件(模型="nch") => 其 @使用符号 $NMOS`
	只在本层	`among the instances in # only, @SELECT-INSTANCE(type="pmos", recursive=false) => …`	`只从本层 # 的元件中 @选出元件(类型="pmos", 含下层=否) => …`

关键字	情况	英文	中文
	筛	`among {#M1 #M2 #R1}, @SELECT-INSTANCE(name="M*") => they @ORIENT("MY")`	`从 {#M1 #M2 #R1} 中 @选出元件(名字="M*") => 其 @朝向("MY")`
	网络上的	`among the instances connected to &en, @SELECT-INSTANCE => they @SET-STYLE to $grey`	`从接在 &en 上的元件中 @选出元件 => 其 @使用样式 $"灰色"`
	端子所属的	`among the instances owning #M1:g, @SELECT-INSTANCE => they are on @LAYER("top")`	`从 #M1:g 所属的元件中 @选出元件 => 其处于 @图层("顶")`
	用例的成员	`among all members of %st1 at any depth, @SELECT-INSTANCE(role="pair")`	`从 %st1 的各级成员中 @选出元件(角色="pair")`
	单层的用例	`among the members of %dp, @SELECT-INSTANCE`	`从 %dp 的成员中 @选出元件`
	管道中间	`among the usages at each level in # and below, @SELECT-USAGE of $DIFFERENTIAL_PAIR => among all members of them at any depth, @SELECT-INSTANCE => they @SET-STYLE to $signal`	`从本层 # 及以下每一层的用例中 @选出用例 $"差分对" => 从其各级成员中 @选出元件 => 其 @使用样式 $"信号"`
SELECT-TERMINAL	本层及以下	`among the terminals at each level in # and below, @SELECT-TERMINAL(net="vss", wired=false) => $GROUND_STUB is @ATTACH(gap=3) to them`	`从本层 # 及以下每一层的端子中 @选出端子(网络="vss", 已接线=否) => $"接地桩" @附着(间距=3) 在其上`
	元件的	`among the terminals of #R1, @SELECT-TERMINAL => on them there is a @TEXT("${net}")`	`从 #R1 的端子中 @选出端子 => 其上有 @文本("${网络}")`
	网络上的	`among the terminals connected to $GROUND_NET_SET, @SELECT-TERMINAL(wired=false) => $GROUND_STUB is @ATTACH to them`	`从接在 $"地网络集" 上的端子中 @选出端子(已接线=否) => $"接地桩" @附着 在其上`
	筛	`among {#M1:s #M2:s}, @SELECT-TERMINAL(wired=false) => $GROUND is @ATTACH to them`	`从 {#M1:s #M2:s} 中 @选出端子(已接线=否) => $"地" @附着 在其上`
SELECT-NET	本层及以下	`among the nets at each level in # and below, @SELECT-NET(port=true) => $IO_PORT is @ATTACH to them`	`从本层 # 及以下每一层的网络中 @选出网络(端口=是) => $"端口" @附着 在其上`
	元件接的	`among the nets connected to #M1, @SELECT-NET => named %m1_net_set`	`从与 #M1 相连的网络中 @选出网络 => 记作 %m1_net_set`
	端子所在的	`among the nets connected to #M1:g, @SELECT-NET => they are fully @ROUTE`	`从 #M1:g 所在的网络中 @选出网络 => 其全部 @布线`
	导线所在的	`among the nets of %w13, @SELECT-NET => they are fully @ROUTE`	`从 %w13 所在的网络中 @选出网络 => 其全部 @布线`
	筛	`among {&a &b}, @SELECT-NET(wired=false) => they are fully @ROUTE`	`从 {&a &b} 中 @选出网络(已接线=否) => 其全部 @布线`
SELECT-WIRE	本层及以下	`among the wires at each level in # and below, @SELECT-WIRE(bus=true) => they @SET-STYLE to $bus`	`从本层 # 及以下每一层的导线中 @选出导线(总线=是) => 其 @使用样式 $"总线"`
	网络上的	`among the wires on &clk, @SELECT-WIRE => they @SET-STYLE to $red`	`从 &clk 上的导线中 @选出导线 => 其 @使用样式 $red`
SELECT-LABEL	本层及以下	`among the labels at each level in # and below, @SELECT-LABEL(key="wl") => they @SET-STYLE to $grey`	`从本层 # 及以下每一层的标签中 @选出标签(引用名="wl") => 其 @使用样式 $"灰色"`
	附着在对象上的	`among the labels attached to #M1, @SELECT-LABEL => they @SET-STYLE to $grey`	`从附着在 #M1 上的标签中 @选出标签 => 其 @使用样式 $"灰色"`
SELECT-ANNOTATION	本层及以下	`among the annotations at each level in # and below, @SELECT-ANNOTATION => they @SET-STYLE to $base`	`从本层 # 及以下每一层的标注中 @选出标注 => 其 @使用样式 $"基础"`
	筛	`among {%n1 %f1 #M1}, @SELECT-ANNOTATION`	`从 {%n1 %f1 #M1} 中 @选出标注`

关键字	情况	英文	中文
SELECT-USAGE	本层及以下	`among the usages at each level in # and below, @SELECT-USAGE of $DIFFERENTIAL_PAIR => they have @PRIORITY("disabled")`	`从本层 # 及以下每一层的用例中 @选出用例 $"差分对" => 其 @优先级("停用")`
	不写定义	`among the usages at each level in # and below, @SELECT-USAGE => they @SET-STYLE to $base`	`从本层 # 及以下每一层的用例中 @选出用例 => 其 @使用样式 $"基础"`
	用例自身和里面	`among %st1 itself and the usages inside it, @SELECT-USAGE of $DIFFERENTIAL_PAIR`	`从 %st1 自身及其内部的用例中 @选出用例 $"差分对"`
SELECT-ANCESTOR-USAGE		`among the usages containing #M1, @SELECT-ANCESTOR-USAGE of $DIFFERENTIAL_PAIR => named %dp_set`	`从包含 #M1 的用例中 @选出所处用例 $"差分对" => 记作 %dp_set`
	端子	`among the usages containing #M1:3, @SELECT-ANCESTOR-USAGE of $GROUND => they @ORIENT("R180")`	`从包含 #M1:3 的用例中 @选出所处用例 $"地" => 其 @朝向("R180")`
any selection		`for %mixed_set, @SELECT-TERMINAL => …`	`对于 %mixed_set, @选出端子 => …`

B.8 集合与其他关键字

关键字	情况	英文	中文
ADD-TO		`&vdd @ADD-TO $SUPPLY_NET_SET`	`&vdd @加入 $"电源网络集"`
	管道	`… => they @ADD-TO $SUPPLY_NET_SET`	`… => 其 @加入 $"电源网络集"`
REMOVE-FROM		`{&vdd &vss} @REMOVE-FROM $PORT_FILL_SET`	`{&vdd &vss} @移出 $"端口补位集"`
UNION		`%u_set = %a_set and %b_set @UNION`	`%u_set = %a_set 和 %b_set 的 @并集`
BUNDLE		`#XI<3:0> is drawn as a @BUNDLE => named %xi`	`#XI<3:0> @叠画 => 记作 %xi`
CURRENT	流入	`#M5:1 has @CURRENT("in")`	`#M5:1 有 @电流("流入")`
	流出	`#:iout has @CURRENT("out")`	`#:iout 有 @电流("流出")`
	不定	`#M5:2 has @CURRENT("unspecified")`	`#M5:2 的 @电流("不定")`

B.9 头段记录与包

不加词。头段记录按标准拼写读，不用代称（16.5）。

关键字	情况	英文	中文
VERSION		`@VERSION("0.3")`	`@VERSION("0.3")`
ALIAS		`@ALIAS("@位于", "@PLACE")`	`@ALIAS("@位于", "@PLACE")`
USE-ALIASES		`@USE-ALIASES(name="rnl-zh", version="0.2")`	`@USE-ALIASES(name="rnl-zh", version="0.2")`
DEVICE		`@DEVICE(…)`	`@DEVICE(…)`
PACKAGE		`@PACKAGE(name="acme-style", version="1.2")`	`@PACKAGE(name="acme-style", version="1.2")`
USE-PACKAGE	头段记录里；包文件的头段记录里引用别的包也这样写	`@USE-PACKAGE(name="rnl-std", version="0.2")`	`@USE-PACKAGE(name="rnl-std", version="0.2")`
	写在层里	`@USE-PACKAGE(name="acme-style", version="1.2")`	`@使用包(名字="acme-style", 版本="1.2")`

附录 C 留给以后版本的内容（说明性）

本附录列出本版标准没有收入、并说明以后的版本可能加入的内容，以及提到它们的节。本版没有为这些内容预留关键字或写法（5.3、16.4），以后的版本也不一定加入。

留给以后版本的	见
更多宿主的载体，包括块注释（如 Eldo 的 `#com … #endcom`），以及由此带来的宿主注释结束符的处理	2.2、2.5
记录 ID；工具按 `type`、`author` 和语句的键找自己的记录	2.8
表达式	3.7
说明一个定义是对整个集合套用一次还是对每个成员各套用一次的标准例句（`each`）	4.5
可选槽位；没有可选槽位，选择总要写范围，至少写 `#`	5.3、10.1、13.7
一个引用里展开多处下标，以及更多的下标写法，如 `<*2>A`、`A<0:2*2>`、`D[0..7]`	6.11
标准的诊断码	7.6
在文件里压制诊断的写法和严格模式，例如作为记录头的字段	7.6
几条事实组成的单位：定义或用例展开出的事实在冲突取舍里一起保留或一起丢弃。还没定的有：这种单位的优先级、嵌套的用例、从外面经 `::` 写的优先级	8.5
符号和元件的缩放	9.2.1、11.7
导线两端的箭头	9.2.5
更多的线型	9.3
颜色名和 `rgb()`	9.3
图案填充、渐变和阴影	9.3
断言语句，用于检查如两个角色不能填同一个对象这样的条件	9.5
在一条语句里求集合的交集	10.1
按端子数或参数给原生元件选符号。每种方言里有几个原生元件，类别由模拟器按行决定，现有的 `name`、`model`、`type` 条件分不出来：Spectre 的 `switch` 接两个端子是单刀单掷，接三个是单刀双掷；SPICE 的 B 源写 `V=` 是电压源，写 `I=` 是电流源。子电路和模型不会这样，它们的端口和模型类型都是固定的。一种做法是给 SELECT-INSTANCE 加通用条件，如 `terminals=`（端子数）、`param=`（行上有这个参数）。另一种是由方言的适配器给这些原生元件一个区分用的类型（6.12），如 B 的 `"current"`、三端 switch 的 `"spdt"`，用 `type=` 选，这时它也比较适配器给原生元件的类型。在那之前，标准包照现在的规则画这两类	10.1、6.12
按元件所用的语言模式选符号。Spectre 文件中间可以有 `simulator lang=spice` 的段落，段里写着要画的元件，如 `I1 a b 1m`；这个文件是 Spectre 方言（16.3），所以这些元件按 Spectre 的规则找符号，可能画成方框。用 USE-PACKAGE 用上 SPICE 的规则，又会把整个文件里像 `I0` 这样命名的 Spectre 元件画成电流源。要按段区分，规则得能看到元件是用哪种语法写的。这和上一条按端子数、按参数选符号是同一类问题，以后一起考虑。被 include 的另一种方言的文件在子电路外面写的元件，属于 include 它的层（6.1），也是这样	16.3、10.1
用 ORIENT 把整个集合当作一块旋转	11.2
一个宿主对象画几次，各是有自己名字的独立对象	11.4
框住一组对象的图元	13.4
锚在导线上的标注：沿导线的位置（如 `t`、`offset`），以及锚在导线中间的符号或箭头	13.3、15.2
文字里更多的 TeX，含写作 `\(…\)` 的数学模式	13.6
`${…}` 引用别的对象的属性，如 KiCad 的 `${R3:VALUE}`	13.6
约束的对角投影 `"x+y"`、`"x-y"`	14.1
文件里放实现的设置（如求解器的设置）的地方，作为识别但不解释的内容	14.8、16.4
只用于本层的 USE-PACKAGE	16.2
分在几个文件里的包	16.2
网表和它的包文件的清单格式	16.2
包的内容摘要，如 USE-PACKAGE 的 `digest=` 参数	16.2
包只给自己用的引用，与转给使用者的引用分开	16.2

留给以后版本的	见
包限定名、给包另起的名字和层级标签，如 `$acme.DIFFERENTIAL_PAIR` 或 `as=`，写法不用名字后面紧跟 `.`	16.2
扩展机制：带厂商前缀的关键字，如 `@ACME.ROUTE`，在头段记录里声明为必需或可选的扩展	2.8、16.4
精确 RNL，即文件不经求解器就能画出的条件，以及往返的判据；这些也可以改由工具之间约定	第 17 章

附录 D 与 0.2 版相比的改动（说明性）

0.3 版让求解器把自己定下的每个决定都写回，使不求解的查看器画出的就是求解器画的图。0.x 各版本之间不承诺兼容（16.1）；本附录列出改了什么，以及文件怎样迁到 0.3 版。

改动	见
经名字找不到的对象有编号，如 `%#3`，只用在求解器的记录里。记录头字段 `numbering` 是编号的校验值；对不上时，用到编号的语句不生效	5.9、2.8、3.2、5.7、7.6、9.4、13.1、13.3
WIRE 收到的不是网络而是一根导线时，改写这根导线；求解器这样写回 `"auto"` 导线的走法	13.1、5.1、5.3、5.8、10.2
求解器应把自己定下的每个决定都写回，交给别的工具的结果只用求解器的记录能含有的形式	14.8、1.6、8.2、8.5、12.1、13.3、17
求解时，`type="generated"` 的记录不算进解释；此外怎样用它们，由求解器决定。0.2 版说求解器不应读它们	2.8、14.8、17
澄清：用例的优先级管住它的定义体里的 PRIORITY 语句；从外面经 `::` 写在局部用例上的 PRIORITY 管住这个用例展开出的一切；停用的用例里的分组可以从外面重新打开	14.8、11.5
澄清：`scope` 记录绑定的名字发布到目标层，在那里再绑定到同一个对象是可以的	2.6
关于怎样取舍冲突的一条注	8.5
澄清：有解，是指有实数的坐标使事实在容差内成立，整数格点只是目标；写不成有限小数的值，写回时写足够多的位数	8.5、11.7
实现可以限定检查有没有解所花的时间；检查没做完的事实丢弃，并给警告。这个上限和容差都是求解器的选项，用户可以设定；容差大于 0	8.5、7.6、11.7
澄清：求解器写回的对象，只在规则把已经写出的去掉时，才取代规则生成的对象，标准包的补位就是这样	10.5
被丢弃的事实的提示列出与它冲突的每一组事实；这些组里的语句也会得知因它而丢弃的事实；priority 为 `"low"` 的事实，这些提示可以不给	8.5
澄清：单独就无解的事实是自相矛盾，失效，不参加取舍；0.2 版的 9.5 已经是这个读法	8.5、9.5
澄清：管道的每一段在文本里有自己的位置，凡是书写位置起作用的地方，后面的段都算后写，如按键覆盖、外观队列和取舍次序	5.5、1.5、8.3、8.4、8.5、13.1
编辑器怎样改求解器写出的对象	17
0.3 版的文件隐式去掉另一种方言的规则包：SPICE 方言的文件不用 `rnl-std-spectre`，Spectre 方言的文件不用 `rnl-std-spice`。文件的方言看它第一条语句的语言模式；层链上写了这两个包任一个的 USE-PACKAGE，那里就留下它	16.3
T、O 的默认端子名（`p1`、`n1`、`p2`、`n2`），U 的（`a`、`b`、`ref`），Z 的（`d`、`g`、`s`），按元件那一行所在文件的版本	6.12
列出了 Spectre 的默认端子名：原生器件的、`switch` 的，以及模型卡的元件按模型的器件族给的，还有选择条件 `type` 比较的类型	6.12、10.1
澄清：经 `.lib` 读入的一节按一个文件算，它的头段记录是节里的第一条记录	2.8
澄清：文件里的名字和较新版本新加的标准词拼写相同时，升版本给警告，版本照样升	16.1
记录头的优先级不给 AS、ATTACH 作默认值，`"disabled"` 除外：用例没被给优先级时，体保持各自的优先级；关掉的记录仍连同其中的用例和体一起关掉。求解器写出规则生成的用例时，照原样带上规则在它上面明写的优先级，或者从关掉的记录继承的 `"disabled"`，此外什么都不写	14.8、2.8、10.5
隐式标准包随文件的语言版本：0.2 版的文件用标准系列 0.2，0.3 版的文件用标准系列 0.3；钉住的版本优先	16.3、16.1

改动	见
0.3 版取代 0.2 版，工具仍可以读 0.2 版：写出的是 0.3 版，写进 0.2 版的文件时把它升到 0.3 版，并钉住标准系列 0.2，使文件的名字和画法不变	16.1、16.3

文件从 0.2 版迁到 0.3 版：把头段记录里的版本改成 0.3（16.1）；工具写进 0.2 版的文件时自己会这样做，并钉住系列 0.2（16.1）。0.3 版加了一个标准词，即记录头字段 `numbering`；在 0.2 版里它是不认识的字段。升版本时，与它同拼写的名字会给警告（16.1）；在 0.3 版里，与它同拼写的代称是错误（16.5）。为 0.2 版写的求解器记录不用编号，照样有效；下次求解时，0.2 版容纳不下的决定也会写回。有几处结果可能与工具对 0.2 版的理解不同：从外面写在局部用例上的优先级，对这个用例展开出的一切说了算（14.8）；文件不再用另一种方言的规则包，所以在 Spectre 网表里，`I0` 这样的元件不再画成电流源（16.3）；T、O、U、Z 有了默认的端子名，文件里的 DEVICE 照样可以改写它们（6.12）。没钉标准包的文件升到 0.3 版后，也从标准系列 0.2 换到 0.3，有些定义和符号改了名；要沿用 0.2 的名字，就在头段记录里写 `@USE-PACKAGE(name="rnl-std", version="0.2")` 钉住它（16.3）；见下面的标准系列 0.3。

标准系列 0.3：标准包不属于本标准（16.3），它的系列 0.3 改了什么，标准包的 README 里列全了。下面是文件从系列 0.2 换到系列 0.3 时要知道的。

改了名的：0.3 的名字写全称，虚拟元件的符号是它的定义名后面加 `_SYMBOL`。角色名和参数名不变（`$CHAIN` 的角色 `series` 除外，见下），只换下面这些名字：

系列 0.2	系列 0.3	种类
`$GND`	`$GROUND`	定义
`$IOPORT`	`$IO_PORT`	定义
`$NC`	`$NO_CONNECT`	定义
`$DIFFPAIR`	`$DIFFERENTIAL_PAIR`	定义
`$DIFFPAIR_TAIL`	`$DIFFERENTIAL_PAIR`，填上它的 `tail`	定义
`$MIRROR_PAIR`	`$MIRRORED`	定义
`$GND_STUB`	`$GROUND_STUB`	定义
`$GND_FILL_SET`	`$GROUND_FILL_SET`	集合
`$LABEL_TERM_SET`	`$LABEL_TERMINAL_SET`	集合
`$GROUND`	`$GROUND_SYMBOL`	符号
`$SUPPLY_BAR`	`$SUPPLY_SYMBOL`	符号
`$PORT_TAG`	`$IO_PORT_SYMBOL`	符号
`$NC_CROSS`	`$NO_CONNECT_SYMBOL`	符号
`$JUNCTION_DOT`	`$JUNCTION_SYMBOL`	符号
`$CURRENT_ARROW_SYM`	`$CURRENT_ARROW_SYMBOL`	符号
`$ARROW_HEAD`	`$ARROW_SYMBOL`	符号
`$RES`	`$RESISTOR`	符号
`$CAP`	`$CAPACITOR`	符号
`$IND`	`$INDUCTOR`	符号
`$VSRC`	`$VOLTAGE_SOURCE`	符号
`$ISRC`	`$CURRENT_SOURCE`	符号
`$NMOS4`	`$NMOS_4TERM`	符号
`$PMOS4`	`$PMOS_4TERM`	符号

`$GROUND` 两个系列都有，含义不同：在 0.2 里是地的符号，在 0.3 里是地这个定义，即 0.2 的 `$GND`。所以文件先改 `$GROUND`，再改 `$GND`，而且只改整个名字：`$GND_STUB` 里的 `$GND` 不是一个名字。

名字没变、画法或含义变了的：

- `$VCVS`、`$VCCS` 只画输出的两个端子。四个端子都画的符号是 `$VCVS_4TERM`、`$VCCS_4TERM`，E、G 元件和 Spectre 的 master `vcvs`、`vccs` 默认配它们。文件自己给元件配 `$VCVS` 或 `$VCCS` 的，这个元件现在只有两个引脚，接到它第 3、4 端子的导线失效（9.2.4）；文件应改配带 `_4TERM` 的符号。
- 符号照 Analog Canvas 的重画了，许多符号的引脚换了位置：`$RESISTOR` 这样的两端元件，两个引脚相距 4，不是 6；二极管横着放；`$OPAMP` 的 + 输入，即端子 1，在上方。线宽是 0.16，0.2 画的是细线。为 0.2 的符号写回的位置和走线因此可能对不上引脚；这样的文件要重新求解，或钉住系列 0.2。
- `$DIFFERENTIAL_PAIR` 多了第三个角色，集合角色 `tail`，可以不填，所以照 0.2 写的差分对含义不变。
- `$CURRENT_MIRROR` 的两个角色都是集合角色：有几个参考管或输出管的电流镜是一个用例，而 `#M3(ref) {#M4 #M6}(out)` 在 0.2 是两个用例。
- `$CHAIN` 的集合角色 `series` 改叫 `chain`。
- `$DIFFERENTIAL_PAIR` 体里一条优先级为 `"low"` 的约束把尾管放在两管中间（9.5），所以没摆尾管的文件，画出来和 0.2 不同；`%dp1 @PRIORITY("disabled")` 能单独关掉一对（14.8）。
- 标准包的定义体里的约束，优先级一律是 `"low"`（9.5）。`$MIRRORED` 让 a 在 b 的左边（14.7）；`$BRANCH` 让成员同一列，`$CHAIN` 让源、链条、负载从左到右。
- `$CASCADE` 的四个角色都是集合角色：由几个对象组成的一级是一个集合，而 `{#RZ #CC}(feedback)` 在 0.2 是每个成员各一个用例。
- `$OPAMP` 不在 `rnl-std` 里了，在 `rnl-std-functional` 里；文件在头段记录里写这个包，单写或经 `rnl-std-full` 都可以（16.3）。
- `$GROUND_STUB`、`$SUPPLY_STUB` 只给离引脚的最小距离 `gap`，不再是确定的距离，求解器可以把导线画得更长；它们还让地朝下、电源朝上，`@ORIENT("R0")`；这些优先级都是 `"low"`（10.5）。
- 补位不在地网络和电源网络上加端口符号；0 以外的地网络（如加进 `$GROUND_NET_SET` 的 vss）上的地用 `$NAMED_GROUND_SYMBOL` 画，它标出网络名（10.5）。
- 系列 0.2 没有配专门符号的元件，现在默认有了：S 配 `$VOLTAGE_CONTROLLED_SWITCH`，W 配 `$SWITCH`，B 配 `$BEHAVIORAL_SOURCE`，SPICE 的 F、H、J、Z、T、O、U 也配了各自的符号，Spectre 里这些元件的 master 同样，如 `relay`、`switch`。

去掉的：`$INPUT_STAGE`：把它的尾管写进对应差分对的角色 `tail`，如 `#M1(in_p) #M2(in_n) #M5(tail) @AS $DIFFERENTIAL_PAIR`。`$TAIL_CENTER`：标准包自己会把每个差分对的尾管放在中间；要让别的对象居中，文件自己写一个定义，如 9.4 的 `$TAIL_MID`。`rnl-std-semantics` 和 `rnl-std-layout`：合成了一个包 `rnl-std-structures`（16.3）；原来去掉 `rnl-std-layout` 来不要默认布局的文件，改为对每个定义写一条规则，如 `# @SELECT-USAGE $MIRRORED => @PRIORITY("disabled")`（14.8）。

新加的：`rnl-std-basic`、`rnl-std-semiconductors` 加了符号，如开关、电流控制的源、传输线、JFET。系列还加了 `rnl-std` 不带进来的包：`rnl-std-functional`、`rnl-std-rf`、`rnl-std-logic`、`rnl-std-signal-flow`，把它们和 `rnl-std` 合在一起的集合 `rnl-std-full`，以及可选的 `rnl-std-no-labels`（16.3）。文件自己定义了、系列 0.3 又新加的名字，如 `$SWITCH`，遮蔽标准包的那个并给提示（6.8），文件用的仍是它自己的。

修订 2：编辑性的修订，语言没有改动。1.3 和 5.9 的例子写上了晶体管的模型卡 `.model nch nmos level=1`，让标准包认得出它们的极性、给它们符号。

后续编辑性澄清：13.3 明确，文字位置的两个轴都由场景给定时，文字不继承附着对象的朝向，对齐按场景的方向解释。语言没有改动。