目 录

简介

MAPLE IR 是支持程序编译和执行的内部程序表示。由于源程序中的任何信息都可能有助于程序分析和优化，MAPLE IR 旨在提供尽可能完整的源程序信息。

程序信息由两部分组成: 定义程序结构的声明部分和可执行程序代码，前者通常称为符号表。

MAPLE IR 是独立于目标平台的，它不针对任何特定的目标处理器或处理器特性进行预处理。

MAPLE-IR 可以看作是虚拟机（VM）的 ISA，MAPLE VM 可以看作是一个以 MAPLE IR 为输入，直接执行 MAPLE IR 程序部分的通用处理器，MAPLE VM 可以看作是 MAPLE-IR 的第一个接收者，MAPLE-IR 编译得到的程序可以在 MAPLE VM 上执行，不需要完成对任何目标处理器的机器指令的编译。

MAPLE IR 是不同编程语言在编译过程中的通用表示形式，包括 C 语言、C++语言和 Java 语言等通用语言。MAPLE IR 是可扩展的: 随着编译到 MAPLE IR 的语言（包括领域特定语言）的增加，MAPLE IR 将增加更多的结构用来表示不同语言特有的结构。

MAPLE-IR 能够支持所有已知的程序分析和优化，因为它能够灵活地表示不同语义级别的程序代码。MAPLE IR 在最高级别的程序表示具有以下特点:

- 多种语言结构
- 短码序列
- 分层结构
- 保留完整程序信息

特定于语言的分析和优化最好在高层执行。随着编译的进行，MAPLE IR 的层次逐渐降低，以便其操作的粒度更接近于通用处理器的指令。在较低的层次上执行通用优化，特别是在最低层次上，MAPLE IR 指令在大多数情况下都是一对一地映射到主流处理器的 ISA 的机器指令的。这能够最大程度地对 IR 代码进行优化，因为在层次逐渐降低地过程中每个操作的消除都会产生目标机器相关的影响。在最低级别的 IR 中，所有的操作（包括类型转换操作），都会在 IR 级别中显式表示。

MAPLE IR 用树的形式表示程序代码。在最高层次上，通过树来表示程序的层次，尊重源程序的层次结构; 同时尊重语言定义的抽象操作。随着编译的进行，抽象操作被转换成需要更长代码序列的通用操作。由于通用处理器按顺序执行指令序列，程序结构也变得更加扁平。

由于 MAPLE-IR 是一个可以存在于多个语义级别的 IR，MAPLE-IR

程序的级别由它所遵循的约束决定，这些约束包括以下两个类型:

- 操作码约束

IR 的级别越高，允许的操作码类型越多（包括仅从特定语言生成的操作码）；在最低级别，只允许和通用处理器中的操作一一对应的操作码

- 程序结构约束

在较高的 IR 级别上，程序结构是分层的。随着 IR 级别降低，分层结构越来越少，在最低的级别上，程序结构是扁平的：由通用处理器使用的原始指令序列组成。

尽管 MAPLE IR 在最高级别上是目标独立的，但是在级别降低过程中变得依赖于目标。

程序表示

在实际实现中还存在其他的设计标准。为了方便编译器开发人员，MAPLE IR 提供 ASCII 和二进制两种格式。两种格式之间的转换可以看作汇编和反汇编。因此，可以将 ASCII 格式的 IR 视为 MAPLE VM 指令的汇编语言。ASCII 格式是可编辑的，这意味着可以直接在 MAPLE IR 中编程。MAPLE IR 的 ASCII 格式是以一个典型的 C 程序为模型的，由以下几个部分组成:

- 声明语句：代表符号表信息
- 可执行语句：代表可执行程序代码

但是 ASCII MAPLE IR 并不打算遵循 C 的语法。

编译器前端将源文件编译成 MAPLE IR 文件，每个 MAPLE IR 文件对应一个编译单元（CU）。一个编译单元由全局范围内的声明组成，声明中包括函数（程序单元，PU）。在程序单元内部是其函数内部范围内的声明，声明后跟随可执行代码。

在 MAPLE IR 中有三种可执行的节点:

- 叶节点

也称为终端节点，这些节点表示在执行时的值，该值可以是常数或者存储单元的值。

- 表达式节点

表达式节点对其操作数执行计算得到运算结果。其结果是其操作数的函数。每个操作数可以是叶节点或者另一个表达式节点。表达式节点是表达式树的内部节点。表达式节点中的“类型”字段提供和操作结果相关联的类型。

- 语句节点

这些节点表示控制流。执行从函数的入口开始，并且按照语句顺序执行，直到执行控制流语句。除了修改控制流之外，语句还可以修改程序中的数据存储。语句节点的操作树可以是叶节点、表达式节点以及语句节点。

在所有的可执行节点中，操作码字段指定节点的操作，然后是与操作码相关的附加字段说明。节点的操作数在括号内指定，用逗号分隔。一般形式为:

opcode fields (opnd0, opnd1, opnd2)

例如，C 语句“a=b”使用直接赋值操作码 dassign，它将右侧的操作数 b 赋给 a，对应的 IR 形式为:

```
dassign $a (dread i32 $b)
```

为了 ASCII IR 中的可视化，每当操作数不是一个叶节点时，操作数换行并缩进两个空格，因此语句”a=b+c”对应结构为:

```
dassign $a (
  add i32(dread i32 $b, dread i32 $c))
```

语句 “a=b+c-d” 的结构为:

```
dassign $a (
  sub i32(
    add i32(dread i32 $b, dread i32 $c),
    dread i32 $d))
```

关于换行的通用规则如下:

每个表达式或语句节点必须占用自己的行，并且每行不能包含多个表达式或者语句节点

如果至少有一个操作数不是叶节点，则必须在单独的新行中指定当前的表达式或语句的所有操作数，也包括叶节点的操作数。

可以通过字符 “#” 指定注释，该字符可以被 IR 解析器视为行末字符。

对于人工编辑的 MAPLE IR 文件，表达式并不强制换行，因为换行不会影响程序的正确性，因为操作数规范的结尾由右括号表示。但是每行不能有多个语句节点，因为我们不使用'; '分隔语句边界的字符。

符号表

具有声明性质的程序信息最好存储在 IR 的符号表部分。让可执行指令引用符号表可以减少存储在可执行指令中的信息量。对每个声明的作用域，都有一个名为 SymbolTable 的主表，用来管理这个作用域中所有用户定义的项。这意味着文件中声明的每个函数都有一个全局符号表和一个局部符号表。符号表中不同类型的项对应于编程语言支持的各种

类型的声明:

- 存储变量
- 类型
- 程序单元（函数或函数原型）

程序单元的符号表只存在全局作用域下。

在 ASCII 形式的 IR 中，IR 指令使用名称来引用各种符号。在二进制形式下，需要在指令中编码合适的作用域+索引来引用。

原始类型

原始类型可以看作执行机器可以直接操作的预定义类型，原始类型名称中的数字表示存储大小（以位为单位）。

原始类型包括:

- 无类型:void
- 有符号整型:i8,i16,i32,i64
- 无符号整型:u8,u16,u32,u64
- 布尔类型:u1
- 地址类型:ptr, ref, a32, a64
- 浮点数:f32,f64
- 复数:c64,c128
- Javascript 类型:
  - dynany
  - dynu32
  - dyni32
  - dynundef
  - dynnull
  - dynhole
  - dynhole
  - dynbool
  - dynptr
  - dynf64
  - dynf32
  - dynstr
  - dynobj
- SIMD 类型：待定义
- 未知类型 unknown

生成或者操作值的指令必须在指令中指定原始类型，因为操作码中不一定包含这一信息。生成的结果的类型和操作数的类型之间存在区别。

结果的类型可以看作是值的类型，因为它驻留在机器寄存器中，因为主流处理器架构中的算术操作大多基于寄存器。当指令只指定一种类型时，指定的类型同时适用于操作数和结果。如果指令的操作数和结果类型可能不同，则指定的类型是结果类型，第二个字段指定操作数类型。

有些操作码适用于派生类型。比如在聚合赋值中，应当使用派生类型，可以用到 agg。这种情况下可以通过符号类型查找数据的大小。

ptr 和 ref 是目标独立的地址类型。ref 附加对堆中运行时管理的内存块或对象引用的语义。使用 ptr 或 ref 而不是 a32 或 a64 允许 IR 独立于目标机器，在后续的目标相关编译阶段之前不显示地址的大小。

当对象中字段的类型由于完整定义位于不同的编译单元中而未完全解析时，将使用未知类型。

常量

MAPLE IR 中的常量总是原始类型。

整数和地址(指针)类型可以使用十进制和十六进制表达(0x 后缀)。

浮点类型可以用十六进制表达，也可以用标准 C 中的浮点数指定。

单引号括起来的单个字符用于表示 i8 和 u8 常量。

字符串文字用双引号括起。

对复数和 SIMD 类型，值组用逗号分隔在方括号中。

标识符

在 ASCII 形式的 MAPLE IR 中，独立的标识符名称被视作 IR 语言的关键字。为了引用符号表中的条目，必须在标识符名称前面加上前缀。前缀为'$'的标识符是全局变量，将在全局符号表中查找；前缀为'%'的标识符为局部变量，将在局部符号表中查找。前缀为'&'的标识符是函数名，将在函数表中查找。这些前缀的作用是避免标识符名称和 IR 中的关键字（如操作码名称）发生冲突。

伪寄存器

伪寄存器可以被看作一个原始类型的局部变量，它的地址永远不会被接受。可以通过外观辨认伪寄存器。伪寄存器和原始类型相关联。相同大小的整数和地址类型也可以和伪寄存器关联。

因为伪寄存器只能创建来存储原始类型，所以不会使用聚合类型的字段 ID。伪寄存器由前缀“%”后跟数字引用。这使它们区别于其他不是伪寄存器的局部变量，因为它们的名称不能以数字开头。

编译器将把变量提升为伪寄存器。为了避免在将变量提升为伪寄存器时丢失高级类型信息,使用 reg 声明提供与伪寄存器关联的类型信息。

特殊寄存器

特殊寄存器是具有特殊含义的寄存器。它们使用‘%%’前缀指定。%%SP 是堆栈指针,%%FP 是引用当前函数堆栈帧的帧指针。%%GP 是用于寻址全局变量的全局指针。

特殊寄存器%retval0、%retval1、%retval2 等用于获取调用返回的多个值。它们被每个调用覆盖,每次调用后最多应读取一次。它们可以假设返回值为任意类型。

语句标签

标签名称的前缀是“@”,用于标识它们。任何以标签名称开头的语句都将该标签定义为引用该文本位置。标签仅由 goto 和 branch 语句在本地引用。

存储访问

由于 MAPLE IR 是目标独立的,所以它不显示任何关于如何为程序变量分配存储的预处理,它只应用由语言定义的关于存储的规则。

一般来说,有两种不同的存储访问:直接访问和间接访问。直接访问不需要任何运行时计算来确定确切的地址位置。在确定位置之前,间接访问需要用到地址算法。间接访问与指针解引用和数组关联,直接访问与结构中的标量变量和固定字段关联。

如果变量或字段没有别名,则可以将直接访问映射到伪寄存器。除非计算地址不变,否则间接访问不能映射到伪寄存器。但是,由于间接访问可以表示数组和矩阵元素,因此有许多优化只与间接存储访问相关。

在 MAPLE IR 中,dassign 和 dread 是直接赋值和直接引用的操作码;iassign 和 iread 是间接赋值和间接引用的操作码。

聚合

聚合(或复合)可以是结构或数组。它们都指定了一组存储元素。在结构中,存储元素由字段名指定,可以是不同类型和大小的。在本文档中,结构设计包括面向对象编程语言中的类和对象。在数组中,同一个存储元素重复多次,并通过索引(或下标)访问这些元素。

数组

数组下标指定地址的计算。因为使下标突出有助于数据依赖性分析和其他嵌套循环优化，MAPLE IR 使用特殊操作码 array 表示数组下标运算，该运算用于计算下标对应的地址并返回。例如，”a[i]=i”对应的 IR 为：

```
iassign<*i32>(
   array a32 <* [10] i32> (addrof a32 $a, dread i32 $i),
   dread i32 $i)
```

（注： <* [10] i32> 表示包含 10 个 int 类型元素的数组指针）

“x=a[i,j]” 对应：

```
dassign $x (
   iread i32 <* i32>(
    array a32 <* [10] [10] i32> (addrof a32 a, dread i32 i,dread i32 $j)))
```

（注： <* [10] [10] i32 表示 10*10 矩阵的指针）

结构

结构中的字段可以直接访问，但在结构上使用 dassign 或 dread 会将整个结构作为聚合引用。因此，我们扩展 dassign、dread、iassign 和 iread 接受名为 field-ID 的附加参数。

通常，对于顶级结构，可以为结构中包含的所有字段分配唯一的字段 ID。顶层的结构的字段 ID 值为 0（如果不是结构体字段 ID 也为 0）. 访问每个字段时，字段 ID 值加 1。如果一个字段是结构，这个结构被赋值为一个特殊的字段 ID，然后字段 ID 分配继续执行嵌套结构中的字段。如果一个字段是数组，则该数组只分配一个字段 ID。

注意，如果一个结构既独立又嵌套在另一个结构中，那么该结构中的同一个字段将被分配不同的字段 ID，因为字段 ID 分配总是从顶层结构开始。

支持三种结构：struct、class 和 interface:

struct 对应于 C 中的 struct 类型，由 struct 关键字后跟由大括号括起的字段声明列表指定，如：

```
struct{
      @f1 i32,
      @f2 <structz>}
```

类对应于 Java 中的 Class，以提供单个继承。语法与 struct 相同，

只是在类名后指定其继承的类。父类中的字段也通过字段 id 引用，就好像派生类的第一个字段是父类一样，换句话说，父类被当作子结构处理。

class <classz>{
    @f1 i32,
    @f2 f32 }

与存储无关，结构可以包含成员函数原型。成员函数原型的列表必须在指定所有字段后出现。每个成员函数名以&开头，表示它是一个函数原型。原型规范遵循与普通函数原型相同的语法。

接口对应于 Java 中的接口类型，其形式与类相同，只是它不能通过变量声明进行实例化，并且内部声明的字段总是静态分配的。

指令详述

在 ASCII MAPLE IR 中，我们使用括号和大括号将操作数与指令的其他字段区分开来，以便于可视化 MAPLE IR 的嵌套结构。每条指令的表达式操作数总是用括号括起来，用逗号分隔操作数。语句操作数由大括号括起，每个语句都从新行开始。MAPLE IR 不允许使用分号来指示每个语句的结尾。每个语句必须以新行开头。

存储访问指令

存储访问指令要么将存储器中的值加载到寄存器以便进一步处理，要么将值从寄存器存储到存储器。对于加载指令，指令中给出的结果类型是驻留在寄存器中的加载值的类型。如果内存位置的大小小于寄存器大小，则加载的值必须是整数类型，并且根据结果类型的有符号性，将有隐式的零或符号扩展。

dassign 语法:

dassign <var-name> <field-id> (<rhs-expr>)

<rhs-expr>通过计算返回一个值，该值分配给变量<var-name>。如果<field-id>不为 0，<var-name>必须是一个结构，并且只将值分配给<field-id>指定的字段。如果<rhs-expr>是聚合类型，那么结构的大小必须匹配；如果是原始整数类型，被分配的变量可能更小，导致发生数据截断。如果未指定<field-id>,则假定其值为 0。

dread 语法:

dread <prim-type> <var-name> <field-id>

从变量<var-name>存储位置读取值。如果是结构类型，<prim-type>应该指定为 agg。如果<field-id>非 0，变量必须为结构，并且只读取<field-id>指定的字段而非整个变量。如果字段本身也是一个结构，也应该用<prim-type>指定为 agg。如果<field-id>未指定，则假定值为 0。

iassign 语法：

iassign <type> <field-id> (<addr-expr>, <rhs-expr>)

<addr-expr>计算返回一个地址。<type>给出了<addr-expr>的高级类型并且必须是一个指针。<rhs-expr>计算返回一个值，随后将该值赋给<addr-expr>指定的地址。如果<field-id>不为 0，则计算的地址必须与结构相对应，并且赋值仅赋值给指定的字段。如果<rhs-expr>是复合类型，则结构的大小必须匹配。被赋值的位置的大小由<type>指向的对象决定。如果<rhs-expr>是原始整型，则<type>指定的位置可能比整型更小，从而导致截断的发生。如果<field-id>未指定，其默认值为 0。

iread 语法：

iread <prim-type> <type> <field-id> (<addr-expr>)

<addr-expr>是地址表达式，计算得到一个地址值，该地址的内容被当作给定的通用类型返回。<type>提供高级类型，并且必须是指针类型。如果地址指定的内容是结构，那么<prim-type>应该指定为复合类型。如果<field-id>不为 0，<type>必须指定为指向结构的指针。并且只是读取指定字段的值而非整个结构。如果字段本身也是结构，那么<prim-type>也应该指定为复合类型。<field-id>不指定默认值为 0。

iassignoff 语法：

iassignoff <prim-type> <offset> (<addr-expr>, <rhs-expr>)

<rhs-expr>计算返回一个标量值，随后该值被赋值给<addr-expr>返回的地址值加上<offset>指定的值（单位字节）对应的存储地址。<prim-type>提供存储位置的类型，指定了受到影响的内存位置的大小。

iassignfpoff 语法：

iassignfpoff <prim-types> <offset> (<rhs-expr>)

<rhs-expr>计算返回一个标量值，随后该值被赋值给%%FP 指定的地址值加上<offset>指定的值（单位字节）对应的存储地址。<prim-type>提供存储位置的类型，指定了受到影响的内存位置的大小。相当于 iassignoff 中的<addr-expr>为%%FP。

ireadoff 语法：

ireadoff <prim-type> <offset> (<addr-expr>)

<prim-type>必须为标量类型。<offset>的字节值加上<addr-expr>指定的地址值作为读取的存储位置，读取为指定的标量类型。

ireadfpoff 语法:

ireadfpoff <prim-type> <offset>

<prim-type>必须为标量类型。<offset>的字节值加上%%FP 指定的地址值作为读取的存储位置，读取为指定的标量类型。相当于 ireadoff 中<addr-expr>为%%FP。

regassign 语法:

regassign <prim-type> <register> (<rhs-expr>)

<rhs-expr>计算返回一个标量值，赋值给<register>指定的伪寄存器或特殊寄存器。<prim-type>给定寄存器的类型，指定了被赋值的值的大小。

regread 语法:

regread <prim-type> <register>

<register>指定的伪寄存器或特殊寄存器中的值被当作<prim-type>指定的类型读取。

叶节点操作码

dread 和 regread 是叶节点操作码，用于读取变量的内容。以下是其他的叶节点操作码:

addrof 语法:

addrof <prim-type> <var-name> <field-id>

返回<var-name>的变量地址，<prim-type>必须为 ptr,a32,a64 中的一种。如果<field-id>不为 0，则变量必须为结构，并返回指定的结构字段的地址。

addroflabel 语法:

addroflabel <prim-type> <label>

<label>的文本地址被返回，<prim-type>必须为 a32,a64 中的一种。

addroffunc 语法:

addroffunc <prim-type> <function-name>

<function-name>的文本地址被返回，<prim-type>必须为 a32，a64 中

的一种。

conststr 语法:

conststr <prim-type> <string literal>

返回<string literal>的地址，<prim-type>必须为 ptr,a32,a64 中的一种。字符串必须存储在只读存储空间内。

conststr16 语法:

conststr16 <prim-type> <string literal>

返回由 16 位宽字符组成的字符串文本的地址。<prim-type>必须为 ptr,a32,a64 中的一种。字符串必须存储在只读存储空间内。

constval 语法:

constval <prim-type> <const-value>

返回给定原始类型的指定常量值。由于浮点值用 ASCII 形式表示无法保证不丢失精度，因此可以用十六进制形式指定，在这种情况下<prim-type>表示浮点类型。

sizeoftype 语法:

sizeoftype <int-prim-type> <type>

返回<type>字节大小的整型值。由于类型大小通常依赖于目标机器，因此使用本操作码可以保证程序代码的目标独立性。

一元表达式操作码

abs 语法:

abs <prim-type> (<opnd0>)

返回操作数的绝对值。

bnot 语法:

bnot <prim-type> (<opnd0>)

反转操作数的每一位，返回得到的数值。

extractbits 语法:

extractbits <int-type> <boffset> <bsize> (<opnd0>)

提取操作数从<boffset>位开始<bsize>个位的比特域，随后符号扩展或者零扩展形成给定的<int-type>原始整型。操作数必须为整型并且保证位数包含指定的比特域。

iaddrof 语法:

iaddrof <prim-type> <type> <field-id>(<addr-expr>)

<type>给定<addr-expr>的高层次类型，并且必须是指针类型。返回指向项的地址。<prim-type>为 ptr,a32,a64 中的一种。如果<field-id>不为 0，<type>必须指定指向结构的指针，返回结构体中指定字段的地址。Iaddrof 在<field-id>值为 0 时没有意义，直接返回<addr-expr>的值。

lnot 语法:

lnot <prim-type> (<opnd0>)

如果操作数不为 0，返回 0，如果操作数为 0，返回 1。

neg 语法:

neg <prim-type> (<opnd0>)

操作数值取反并返回。

recip 语法:

recip <prim-type> (<opnd0>)

返回操作数的倒数，<prim-type>必须为浮点类型。

sext 语法:

sext <signed-int-type> <bsize> (<opnd0>)

符号扩展操作数对应的整型值，被（截取）扩展的整型的位宽由<bsize>指定。相当于 extractbits 指令里的比特域为最低的一部分比特。<signed-int-type>保持一致。

sqrt 语法:

sqrt <prim-type> (<opnd0>)

返回操作数的平方根。<prim-type>必须为浮点类型。

zext 语法:

zext <unsigned-int-type> <bsize> (<opnd0>)

零扩展操作数对应的整型值,被(截取)扩展的整型的位宽由<bsize>指定。相当于 extractbits 指令里的比特域为最低的一部分比特。<unsigned-int-type>保持一致。

类型转换表达式操作码

类型转换操作码本质上是一元的。除了 retype 之外，它们都要求在指令中同时指定 from 类型和 to 类型。不同大小的整数类型之间的转换需要使用 cvt 操作码。同样大小的有符号整数和无符号整数之间的转换不需要任何操作，甚至不需要 retype。

ceil 语法:
ceil <prim-type> <float-type> (<opnd0>)
浮点数向正无穷大取整。

cvt 语法:
cvt <to-type> <from-type> (<opnd0>)
将操作数的值从<from-type>转换为<to-type>。如果两个类型的大小相同，则转换必须包含位的改变。

floor 语法:
floor <prim-type> <float-type> (<opnd0>)
浮点数向负无穷大取整。

retype 语法:
retype <prim-type> <type> (<opnd0>)
<opnd0>不改变任何位转换至<prim-type>，<prim-type>衍生自<type>。<opnd0>和<prim-type>的大小必须相同，<opnd0>可以是复合类型。

round 语法:
round <prim-type> <float-type> (<opnd0>)
浮点数四舍五入为最接近的整数。

trunc 语法:
trunc <prim-type> <float-type> (<opnd0>)
浮点数转换为整数，忽略浮点值。

二元表达式操作码

add 语法:

add <prim-type> (<opnd0>, <opnd1>)
执行两个操作数的加法。

ashr 语法:
ashr <int-type> (<opnd0>, <opnd1>)
返回<opnd1>右移<opnd0>个比特后的值。高位用符号位补充。

band 语法:
band <int-type> (<opnd0>, <opnd1>)
返回 opnd0 和 opnd1 的位与运算。

bior 语法:
bior <int-type> (<opnd0>, <opnd1>)
返回 opnd0 和 opnd1 的位或运算。

bxor 语法:
bxor <int-type> (<opnd0>, <opnd1>)
返回 opnd0 和 opnd1 的位异或运算。

cand 语法:
cand <int-type> (<opnd0>, <opnd1>)
通过短路执行逻辑与运算。如果<opnd0>的值是 0 就不进行后续运算。结果为 0 或者 1。

cior 语法:
cior <int-type> (<opnd0>, <opnd1>)
通过短路执行逻辑或运算。如果<opnd0>的值是 1 就不进行后续运算。结果为 0 或者 1。

cmp 语法:
cmp <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
比较两个操作数，如果相同返回 0，如果<opnd0>小于<opnd1>返回-1,其他情况返回 1。

cmpg 语法:
cmpg <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
如果存在操作数是 NaN 返回 1，其他和 cmp 相同。

cmpl 语法:
cmpl <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
如果存在操作数是 NaN 返回-1，其他和 cmp 相同。

depositbits 语法:
depositbits <int-type> <boffset> <bsize> (<opnd0>, <opnd1>)
通过把<opnd1>的值放在<opnd0>从<boffset>位开始的<bsize>比特域内创造一个新整数值。<opnd0>必须足够大以包含指定的比特域。根据<opnd1>相对于比特域的大小，可能发生截断。<opnd0>剩下的位保持不变。

div 语法:
div <prim-type> (<opnd0>, <opnd1>)
返回<opnd0>除以<opnd1>的值。

eq 语法:
eq <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
操作数相等返回 1，否则返回 0。

ge 语法:
ge <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
<opnd0>大于等于<opnd1>，返回 1，否则返回 0.

gt 语法:
ge <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
<opnd0>大于<opnd1>，返回 1，否则返回 0.

land 语法:
land <int-type> (<opnd0>, <opnd1>)
逻辑与运算，返回 0 或者 1.

lior 语法:
lior <int-type> (<opnd0>, <opnd1>)
逻辑或运算，返回 0 或者 1.

le 语法:
le <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
<opnd0>小于等于<opnd1>，返回 1，否则返回 0.

lshr 语法:
lshr <int-type> (<opnd0>, <opnd1>)
<opnd0>右移<opnd1>位，高位补 0.

lt 语法:
lt <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
<opnd0>小于<opnd1>，返回 1，否则返回 0.

max 语法:
max <prim-type> (<opnd0>, <opnd1>)
返回两个操作数中的较大值。

min 语法:
min <prim-type> (<opnd0>, <opnd1>)
返回两个操作数的较小值。

mul 语法:
mul <prim-type> (<opnd0>, <opnd1>)
返回两个操作数的乘积。

ne 语法:
ne <int-type> <opnd-prim-type> (<opnd0>, <opnd1>)
如果两个操作数不相等，返回 1，否则返回 0.

rem 语法:
rem <prim-type> (<opnd0>, <opnd1>)
返回<opnd0>除以<opnd1>的余数。

shl 语法:
shl <int-type> (<opnd0>, <opnd1>)
<opnd0>左移<opnd1>位，低位补 0 并返回。

sub 语法:
sub <prim-type> (<opnd0>, <opnd1>)
返回<opnd0>减去<opnd1>的值。

三元表达式操作码

select 语法:

select <prim-type> (<opnd0>, <opnd1>, <opnd2>)

<opnd0>必须是整型，<opnd1>和<opnd2>必须是<prim-type>指定的类型。如果<opnd0>不是 0，返回<opnd1>，否则返回<opnd2>。

N 元表达式操作码

array 语法:

array <check-flag> <addr-type> <array-type> (<opnd0>, <opnd1>, . . . , <opndn>)

<opnd0>是数组在存储空间中的基址，<check-flag>值为 0 或者 1，表示不需要/需要分别执行边界检查。<array-type>给出了指向数组的指针的高级类型。返回行主序多维索引操作产生的地址，第一维索引从<opnd1>开始。

intrinsicop 语法:

intrinsicop <prim-type> <intrinsic> (<opnd0>, ..., <opndn>)

<intrinsic>表示没有副作用的内部函数，其返回值仅依赖于参数(纯函数)，因此可以表示成表达式操作码。

intrinsicopwithtype 语法:

intrinsicopwithtype <prim-type> <type> <intrinsic> (<opnd0>, ..., <opndn>)

与 intrinsicop 相同，只是 intrinsicopwithtype 接受由<type>指定的额外的高级类型参数。

控制流语句

程序控制流可以用层次化的语句结构或一个简单的语句序列来表示。层次结构的语句结构主要来自源语言中的结构。语句序列更接近于处理器指令。

因此，层次结构语句只存在于高级 MAPLE IR 中。它们在编译过程中被转换成控制流语句序列。

语句块由括在‘{’和‘}’内的多个语句组成，它可以出现在允许语句出现的任何位置。在层次结构语句中，嵌套语句由这样的语句块指定。

在 MAPLE IR 中，每个语句都必须以新行开头，不需要分号，甚至不允许分号来表示语句的结尾。

分层控制流语句

doloop 语法:

doloop <do-var> (<start-expr>, <cont-expr>, <incr-amt>) {
    <body-stmts> }

<do-var>指定不带别名的局部整数标量变量。<incr-amt>必须是一个整型表达式。<do-var>由<start-expr>初始化，<cont-expr>必须是表示终止循环的单个比较操作。循环体由<body-stmts>表示，它指定只要<cont-expr>为真的执行的语句列表。循环体每次执行之后，<do-var>由<incr-amt>改变并再次测试循环是否终止。

dowhile 语法:

dowhile {
  <body-stmts>} (
    <cond-expr>)

执行由<body-stmts>表示的循环体，当<cond-expr>计算结果非 0 时，继续执行<body-stmts>。因为是在循环体之后进行测试，因此循环体至少执行一次。

foreachelem 语法:

foreach <elem-var> <collection-var> {
    <body-stmts> }

这是一个抽象的循环表示，其中<collection-var>是类似数组的变量，表示统一元素的集合，<elem-var>指定一个对应元素类型的变量。<body-stmts>表示循环体，表示每个元素要执行的语句列表。该语句将根据<collection-var>的类型降低为更具体的循环形式。

if 语法:

if (<cond-expr>) {
 <then-part> }
else {

<else-part>}

如果<cond-expr>的值不为 0,控制流传递到<then-part>语句,否则,控制流传递到<else-part>语句。如果没有 else 的部分,“else { <else-part> }”可以被省略。

while 语法:

while (<cond-expr>) {
<body-stmts>}

表示 while 循环。当<cond-expr>值不为 0,<body-stmts>表示的语句列表被重复执行。因为<cond-expr>在循环体之前被检查,因此循环可能执行 0 次。

平坦控制流语句

brfalse 语法:

brfalse <label> (<opnd0>)

如果<opnd0>值为 0,跳转至<label>。

brtrue 语法:

brtrue <label> (<opnd0>)

如果<opnd0>值不为 0,跳转<label>。

goto 语法:

goto <label>

无条件跳转至<label>。

multiway 语法:

multiway (<opnd0>) <default-label> {
(<expr0>): goto <label0>
(<expr1>): goto <label1>
. . .
(<exprn>): goto <labeln> }

<opnd0>的类型必须可转换为整数或字符串。<default-label>之后是表达式和标签的列表。执行时,计算<opnd0>的值并在列表中按顺序搜索每个<expri>的值。一旦匹配,控制流转到对应的<labeli>。如果匹配失败,则控制转移到<default-label>。表达式<expri>的计算不能产生副

作用。根据<opnd0>的类型，本语句将降低为 switch 语句或者一系列 if 语句。

return 语法:

return (<opnd0>, . . ., <opndn>)

使用操作数给定的多个返回值从当前程序单元返回。操作数列表可以是空，表示没有返回值。<opndi>的类型必须和程序单元中的声明或者函数原型中定义的返回类型列表兼容。

switch 语法:

```
switch (<opnd0>) <default-label> {
  <intconst0>: goto <label0>
     <intconst1>: goto <label1>
        . . .
     <intconstn>: goto <labeln> }
```

<opnd0>必须是整型。在<default-value>之后，指定了一个由整型标志和标签的列表。当执行时，搜索列表找到匹配到<opnd0>的<intconsti>，随后控制转移至对应的<labeli>。如果匹配失败，控制转移至<default-label>。常量整数值不能重复。在分析了表中的整型标签的分布之后，编译器后端才会决定如何生成代码。

rangegoto 语法:

```
rangegoto (<opnd0> <tag-offset> {
  <intconst0>: goto <label0>
  <intconst1>: goto <label1>
       . . .
     <intconstn>: goto <labeln> }
```

这个是 switch 的低级表示，它明确指定了由跳转表机制指定的执行。<opnd0>一定是整型。在<tag-offset>后跟随了由整型数值和标签组成的列表。在执行期间用<opnd0>的值减去<tag-offset>的值进行匹配。列表中的常量整数值中不能有间隙，并且在指定的常量范围内能保证成功匹配，这意味着代码生成器可以忽略掉范围之外的检查。常量整数值不能有重复项。

indexgoto 语法:

```
indexgoto (<opnd0> <varnamel>
```

只有编译器在降低 switch 语句时生成。<varname>是编译器生成的符号的名称，该符号指定静态数组或跳转表，该数组或跳转表被静态初

始化以存储标签。每个标签标记了 switch 语句中 case 对应的代码。在该指令执行过程中，使用<opnd0>的值到数组/跳转表中索引并将控制转移至对应的标签如果计算值小于 0 或者超过跳转表中的条目数，则该行为是未定义的。

调用语句

存在各种类型的过程调用。他们只指定实际传递的参数。任何返回值都需要通过特殊寄存器%%retval0,%%retval1,%%retval2…来获取。

call 语法:

call <PU-name> (<opnd0>, ..., <opndn>)

<opndi>提供参数，调用程序单元。

callinstant 语法:

callinstant <generic-PU-name> <instant-vector> (<opnd0>, ..., <opndn>)

根据<instant-vector>指定的实例化向量实例化给定的泛型函数，然后传递操作数给定的参数调用函数。请参阅有关泛型函数实例化的初始化部分。

icall 语法:

icall (<PU-ptr>, <opnd0>, ..., <opndn>)

调用由<PU-ptr>间接指定的程序单元，传递<opndi>提供的参数。

intrinsiccall 语法:

intrinsiccall <intrinsic> (<opnd0>, ..., <opndn>)

调用编译器定义的内部函数。

xintrinsiccall 语法:

xintrinsiccall <user-intrinsic-index> (<opnd0>, ..., <opndn>)

根据<user-intrinsic-index>索引到用户定义的内部函数表中找到函数并传参调用。

Java 调用语句

以下语句用域表示尚未解析的 Java 成员函数调用。

virtualcall 语法:

virtualcall <method-name> (<object-ptr>, <opnd0>, ..., <opndn>)

<object-ptr>是指向一个类对象的指针。使用<method-name>搜索类层次结构以找到要调用的虚方法。其余操作数为参数。

superclasscall 语法:

superclasscall <method-name> (<object-ptr>, <opnd0>, ..., <opndn>)

和 virtualcall 相同，只是它不会使用类本身的虚方法，而是使用和类最接近的超类中定义的虚方法。

interfacecall 语法:

interfacecall <method-name> (<object-ptr>, <opnd0>, ..., <opndn>)

<method-name>是接口中定义的方法。<object-ptr>是一个指向实现了该接口的类的对象。使用<method-name>搜索类找到要调用的方法。其余操作数为参数。

还有 virtualcallinstant、superclasscallinstant 和 interfacecallinstant，用于在使用指定的实例化向量实例化后调用方法的泛型版本。和 callinstant 相同，实例化向量在<method-name>之后指定。

分配返回值的调用

上述所有调用操作都有一个对应的高级抽象变量，这样单个调用操作也可以指定如何分配多个函数返回值，不依赖单独的语句来读取%%retval 寄存器。只允许分配给标量变量或符合变量中的标量字段。这些操作的名称在上节对应的指令后添加后缀：“assigned”。他们是：callassigned,callinstantassigned, icallassigned,intrinsiccallassigned,intrinsiccallwithtypeassigned,xintrinsiccallassigned,virtualcallassigned,virtualcallinstantassigned,superclasscallassigned,superclasscallinstantassigned,interfacecallassigned and interfacecallinstantassigned。下面只给出 callassigned 的定义，同样的扩展也适用于其他此类调用操作的定义。

callassigned 语法:

```
callassigned <PU-name> (<opnd0>, ..., <opndn>) {
  dassign <var-name0> <field-id0>
     dassign <var-name1> <field-id1>
     . . .
     dassign <var-namen> <field-idn> }
```

通过传递操作数给定的参数调用程序单元。调用返回后，将多个返回值按照顺序分配给顺序列出的标量变量。如果某个<field-id>不为 0，则对应的<var-name0>必须是聚合变量，并且把返回值分配给<field-id>指定的字段。如果没有<field-id>，默认值为 0。如果<var-namei>为空，表示忽略对应的返回值。如果函数没有返回值，则不需要列出 dassign。

在编译过程中，可以降低这些调用指令以使用特殊寄存器%retval0、%retval1、%retval2，指示如何获取和使用它们的返回值。这些特殊寄存器被每次调用所覆盖。同一个特殊寄存器可以任意假定返回值的类型。每个特殊寄存器在每次调用后只能读取一次。

异常处理

本节介绍各种异常处理构造和操作。try 语句标记 try 块的入口。catch 语句标记 catch 块的入口。finally 语句标记 finally 块的入口。endtry 语句标记以 try 开头的复合异常处理构造的结尾。此外，还有两种特殊类型的标签，处理程序标签放在 catch 语句之前，最后标签放在 finally 语句之前。处理程序标签通过前缀“@h@”与普通标签区别开来，而最终标签使用前缀“@f@”。 这些特殊标签在每个 try-catch-finally 组合中显式地显示 try、catch 和 finally 之间的对应关系，而不依赖于块嵌套。特殊寄存器%%thrownval 包含要抛出的值，该值是引发当前异常的 throw 操作的操作数。

try 语法:

try <handler-label> <finally-label>

执行此语句表示进入 try 块。<handler-label>为 0 表示没有和 try 关联的 catch 块。<finally-label>为 0 表示没有和 try 相关的 finally 块。除非输入了另一个嵌套的 try 块，否则在此 try 块中引发的任何异常都会将控制权传递给这些标签。无论是否引发异常，都必须执行 finally 块（如果存在）以结束 try 复合结构的执行。

根据 try-catch-finally 组合的编写方式，有三种可能的情况:

- try-catch
- try-finally

- try-catch-finally

对于情况 1，如果在 try 块中引发异常，则将控制传输到标记 catch 语句的处理程序标签，并将该异常视为已被处理。程序流最终使用 goto 语句退出 try 块，并返回到标记 endtry 语句的标签。如果未引发异常，try 块最终通过 goto 语句退出到标记 endtry 语句的标签。

对于情况 2，如果在 try 块中引发异常，则控制权将转移到标记 finally 语句的 finally 标签。但该异常被视为尚未处理，并且将开始搜索该抛出的上层处理程序。如果 try 块中未引发异常，程序流最终使用 gosub 语句退出 try 块并返回 finally 块。finally 块中的执行以 retsub 结束，retsub 返回 try 块，然后通过 agoto 语句退出 try 块并返回标记 endtry 语句的标签。

对于情况 3，如果在 try 块中引发异常，则将控件传输到将 catch 语句标记为情况 1 的处理程序标签。catch 块中的执行以 gosub 语句结束，以 finally 块结束。finally 块中的执行以 retsub 结束，返回到 catch 块，然后通过 goto 语句将 catch 块退出到标记 endtry 语句的标签中。如果 try 块中没有抛出异常，则程序流最终会使用 gosub 语句将 try 块退出到 finally 块，在 finally 块中继续执行，直到执行 retsub，此时返回 try 块，然后 try 块通过 goto 语句退出到标记 endtry 语句的标签。

throw 语法：

throw (<opnd0>)

使用给定的异常值引发用户定义的异常。如果此语句嵌套在 try 块中，则控制将传输到与 try 关联的标签，该标签是 catch 语句或 finally 语句。如果此 throw 语句嵌套在 catch 块中，控件首先传输到与 catch 关联的 finally（如果有），在这种情况下将执行 finally 块。在 finally 块完成执行后，将开始搜索 throw 的上层处理程序。如果此 throw 语句嵌套在 finally 块中，立即开始搜索抛出的上层处理程序。如果该抛出未嵌套在函数中的任何 try 块中，则系统将通过展开调用堆栈来查找第一个封闭的 try 块。如果未找到 try 块，则程序将终止。在 catch 块中，抛出的异常值可以使用特殊寄存器%%thrownval 访问。

catch 语法：

<handler-label> catch

这将标记与 try 关联的 catch 块的开始。与此 catch 块关联的 try 块被视为已退出，异常被视为正在处理。如果 catch 块中没有抛出异常，则退出 catch 块将由 gosub 语句影响到 finally 标签（如果存在 finally 块或转移至 endtry 的 goto 语句。

finally 语法:

<finally-label> finally

这标志着 finally 块的开始。finally 块可以通过执行 gosub 语句进入，或者 finally 对应的没有 catch 块的 try 块的 throw 语句，或者在 finally 对应的 catch 块中的 throw 语句。finally 块的退出可以通过在 finally 块中执行 retsub 或 throw 语句来实现。如果退出是通过 retsub 的，并且有未完成的抛出尚未处理，则继续搜索该抛出的上层处理程序。

cleanuptry 语法:

cleanuptry

此语句是在控制将通过与异常处理无关的跳转过早离开 try-catch-finally 的情况下生成的。此语句影响与当前 try-catch-finally 的异常处理相关的清理工作。

endtry 语法:

<label> endtry

这标志着每个 try-catch-finally 组合的结束或每个 javatry 块的结束。

gosub 语法:

gosub <finally-label>

控制转移至<finally-label>对应的 finally 块。还具有退出此语句所属的 try 块或 catch 块的效果。它类似于 goto，只是保存了下一条指令。执行 retsub 语句时，执行将返回到下一条指令。这也可以看作是调用，只是它使用标签名而不是函数名，并且不传递参数或返回值。

retsub 语法:

retsub

这只能作为 finally 块中的最后一条指令出现。如果没有未完成的 throw，则将控制权传输回最后一个被执行的 gosub 指令的下一条指令。否则，将继续搜索上层异常处理程序。

存储分配和释放

以下指令与程序执行期间动态内存的分配和释放有关。以“gc”为前缀的指令与具有管理运行时环境的语言关联。

alloca 语法:

alloca <prim-type> (<opnd0>)

返回指向通过调整函数堆栈指针%SP 分配的未初始化内存块的指针,大小由<opnd0>指定，单位为位。此指令只能显示为赋值操作的右侧。

decref 语法:

decref (<opnd0>)

<opnd0>必须是指向堆的运行时管理部分中分配的对象的指针的 dread 或 iread。它将指向对象的引用计数减少 1。<opnd0>必须是原始类型 ref。

decrefwithcheck 语法:

decrefwithcheck (<opnd0>)

<opnd0>必须是指向堆的运行时管理部分中分配的对象的指针的 dread 或 iread。它检查<opnd0>的值是否为空。如果不为空，则假定为有效指针,并将指向对象的引用计数递减 1。<opnd0>必须是原始类型 ref。

free 语法:

free (<opnd0>)

<opnd0>指向的内存块被释放，以便系统可以重新分配给其他用途。

gcmalloc 语法:

gcmalloc <pointer prim-type> <type>

要求内存管理器根据托管运行时的要求，为类型<type>的对象分配关联的元数据。对象的大小必须在编译时固定。由于这将返回指向已分配块的指针，因此此指令只能显示为赋值操作的右侧。托管运行时负责其最终释放。

gcmallocjarray 语法:

gcmallocjarray <pointer prim-type> <java-array-type> (<opnd0>)

这要求内存管理器分配一个<java-array-type>的 java 数组对象。分配的存储空间必须足够大，以存储<opnd0>中给定的数组元素数。由于这将返回指向分配块的指针，因此此指令只能出现在赋值操作的右侧。托管运行时负责其最终释放，并且块的大小在其生命周期内必须保持固定。

incref 语法:

incref (<opnd0>)

<opnd0>必须是指向堆的运行时管理部分中分配的对象的指针的

dread 或 iread。它将指向对象的引用计数增加 1。<opnd0>必须是原始类型 ref。

malloc 语法:

malloc <pointer prim-type> (<opnd0>)

这要求系统分配一块未初始化内存，其大小由<opnd0>给定。由于这将返回指向已分配块的指针，因此此指令只能出现在赋值操作的右侧。内存块在通过 free 指令显式释放之前不可重用。

其他语句

assertge 语法:

assertge (<opnd0>, <opnd1>)

如果<opnd0>小于<opnd1>，则引发异常。这用于检查执行期间数组索引是否在范围内。<opnd0>和<opnd1>必须属于同一类型。

assertlt 语法:

assertlt (<opnd0>, <opnd1>)

如果<opnd0>不小于<opnd1>，则引发异常。这用于检查执行期间数组索引是否在范围内。<opnd0>和<opnd1>必须属于同一类型。

assertnonnull 语法:

assertnonnull (<opnd0>)

如果<opnd0>是对应值为 0 的空指针，引发异常。

eval 语法:

eval (<opnd0>)

<opnd0>被求值，但结果被丢弃。如果<opnd0>包含易失性引用，则无法优化此语句。

membaracquire 语法:

membaracquire

此指令同时充当加载到加载指令和加载到存储指令之间的屏障：必须严格遵循它之前的任何加载指令与之后的任何加载指令之间的顺序，以及它之前的任何加载指令与之后的任何存储指令之间的顺序。

membarrelease 语法:

membarrelease

此指令同时充当加载到存储指令和存储到存储指令之间的屏障：必须严格遵循它之前的任何加载指令与之后的任何存储指令之间的顺序，以及它之前的任何存储指令与之后的任何存储指令之间的顺序。

membarfull 语法：

membarfull

此指令对之前的任何加载或存储指令以及之后的任何加载或存储指令起到屏障作用。

syncenter 语法：

syncenter (<opnd0>)

此指令表示<opnd0>指针指向的对象需要同步以实现 Java 多线程的区域的入口。这意味着在任何时候，在同一对象的同步区域中不能有多个线程正在执行。任何其他试图进入同一对象的同步区域的线程都将被阻止。对于编译器来说，这意味着对访问该对象的任何操作的向后移动（与控制流相反）存在障碍。

syncexit 语法：

syncexit (<opnd0>)

此指令表示从为 Java 多线程同步的<opnd0>指针指向的对象的区域中退出。对于编译器来说，它意味着访问该对象的任何操作向前移动（沿着控制流）的障碍。

声明详述

模块声明

每个 Maple IR 文件代表一个程序模块，也称为编译单元，它由全局范围内的各种声明组成。以下指令出现在 Maple IR 文件的开头，并提供有关该模块的信息：

entryfunc 语法：

entryfunc <func-name>

给出了在模块中定义的函数的名称，该函数将作为模块的单个入口点。

flavor 语法:

flavor <IR-flavor>

IR flavor 提供了关于如何生成 IR 的信息，同时指示了编译过程的状态。

globalmemmap 语法:

globalmemmap = [<initialization-values>]

这将全局内存块的静态初始化值指定为以空格分隔的 32 位整数常量的列表。初始化的数量应与 globalmemsize 给定的内存大小相对应。

globalmemsize 语法:

globalmemsize <size-in-bytes>

给出了存储所有全局静态变量的全局内存块的大小。

globalwordstypetagged 语法:

globalwordstypetagged = [<word-values>]

指定一个位向量，该位向量初始化为由空格分隔的 32 位整型常量列表指定的值。如果 globalmemap 中的第 N 个字具有类型标记，则此位向量中的第 N 个位设置为 1，在这种情况下，类型标记位于第（N+1）个字处。

globalwordsrefcounted 语法:

globalwordsrefcounted = [<word-values>]

这将指定一个位向量，该位向量初始化为 32 位整数常量列表指定的值。如果 globalmemap 中的第 n 个字是指向动态分配内存块的引用计数的指针，则该位向量中的第 n 个位设置为 1

id 语法:

id <id-number>

这将给出分配给模块的唯一模块 id。此 id 使 Maple 虚拟机能够处理来自多个 Maple IR 模块的程序代码的执行。

import 语法:

import "<filename>"

<filename>是 MAPLE 类型文件的路径名，后缀为.mplt，将导入此文件的内容。在.mplt 文件中只允许类型声明。这允许在多个文件中共享同一类型声明，并允许按文件组织大量类型声明。只允许一级导入，

as.mplt 文件不允许有 import 语句。只有在完全解析所有类型和符号名之后才使用。

importpath 语法:

importpath "<path-name>"

此选项指定编译器查找完成编译所需的导入 MAPLE IR 文件的目录路径。此选项仅在所有类型和符号名称完全解析之前的早期编译阶段使用。每个仅指定一个特定路径.

numfuncs 语法:

numfuncs <integer>

给出了模块中函数定义的数量，不包括函数原型。

srclang 语法:

srclang <language>

源语言类型。

变量声明

语法:

var <id-name> <storage-class> <type> <type-attributes>

关键字“var”指定变量的声明语句。<id-name>指出变量名字，根据其作用域是全局的还是本地的，指定其名称，前缀为“$”或“%”。<storage-class>是可选的，可以是 extern、fstatic 或 pstatic。<type>是类型规范，<type-attributes>是可选的，并指定其他属性，如 volatile、const、alignment 等。例如:

var $x extern f32 volatile static

伪寄存器声明

语法:

reg <preg-name> <type>

关键字“reg”指定伪寄存器的声明语句。<preg-name>指定前缀为“%”的伪寄存器。<type>是高层类型信息。如果伪寄存器是原始类型，则其声明是可选的。

类型规范

类型既可以是原始类型，也可以是派生类型。派生类型使用类 C 的词组表示，并且是右结合的，遵循严格的从左到右的顺序。派生类型与原始类型的区别在于被括在尖括号中。派生类型也可以被视为高级类型。示例：

```
var %p <* i32>              # 指向 32 位整数的指针
var %a <[10] i32>           #包含 10 个 32 位整数的数组
var %foo <func(i32) i32>                # 指向函数的指针，函数参数伪一个 32 位整数类型，返回值为为 32 位整型。其中 func 为内置关键字。
```

不需要额外的嵌套尖括号，因为右结合规则不会产生歧义。但是可以选择插入嵌套尖括号以帮助可读性。因此，以下两个示例是等效的：

```
var %q <* <[10] i32>>           #指向包含 10 个 32 位整型的数组
var %q <* [10] i32>          #指向包含 10 个 32 位整型的数组
```

在结构声明中，字段名的前缀是@。虽然标签名也使用@作为前缀，但由于结构字段声明和标签用法之间的使用上下文是不同的，因此不存在歧义。例如：

```
var %s <struct{@f1 i32,
            @f2 f64,
            @f3:3 i32}>          # 3 位的比特域
```

联合 Union 声明的语法与 struct 相同。

结构的最后一个字段可以是灵活的数组成员，这是一个具有可变元素数的数组。它用空方括号指定，如下所示：

```
var %p <* struct{@f1 i32,
            @f2 <[] u16>}>          #元素类型为 16 位的 unsigned 整型的可变长数组
```

以可变长数组成员作为最后一个字段的结构只能动态分配，其实际大小仅在执行期间分配时固定，在其生命周期内不能更改。具有可变长

数组成员的结构不能嵌套在另一个聚合中。在编译过程中，可变长数组被视为大小为零。由于其使用通常与托管运行时关联，语言处理器可能会引入与数组关联的其他元数据。特别是，必须有一些依赖于语言的方案来跟踪执行期间数组的大小。

当类型需要由 const、volatile、restrict 和各种对齐的其他属性限定时，它们将遵循它们限定的类型。这些属性不视为类型的一部分。如果这些属性应用于派生类型，则它们必须遵循类型尖括号外的属性。示例：

```
var %x f64 volatile align(16)
# %s 是 f64 类型的值且是 volatile，16 字节边界对齐
var %p <* f32> const volatile
    # %p 是一个指向 f32 值的指针，%p 本身是 const 和 volatile 的
```

对齐是以字节为单位指定的，并且必须是 2 的幂。对齐属性只能用于增加类型的自然对齐，以使对齐更加严格。对于减少对齐，MAPLE IR 的生成器必须使用较小的类型来实现打包的效果，而不是依赖 align 属性。

不完整类型规范

像 Java 这样的语言允许引用任何对象的内容，而不需要完整的对象定义。它们的完整内容将在以后的编译阶段从附加的输入文件中解析。MAPLE IR 允许不完整地声明结构、类和接口，以便可以引用它们的特定内容。应该使用 structcomplete、classincomplete 和 interfaceincomplete 代替 struct、class 和 interface 关键字。

类型声明

语法：

```
type <id-name> <derived-type>
```

根据作用域是全局作用域还是本地作用域，类型名的前缀也为“$”或“%”。示例：

```
type $i32ptr <* i32>          # $i32ptr 是指向 i32 的指针
```

不允许为原始类型指定其他类型名。

类型声明中不允许使用属性。

定义类型名后，指定类型名等同于指定它所代表的派生类型。因此，类型名的使用应始终括在尖括号中。

Java 类和接口声明

Java 类指定的不仅仅是一个类型，因为类名还带有该类声明的属性。因此 Java 类按照以下语法声明：

javaclass <id-name> <class-type> <attributes>

<id-name>必须以“$”作为前缀，因为类名始终具有全局作用域。例如：

javaclass $Puppy <class [{@color](mailto:%7B@color) i32}> public final

一个名为“Puppy”的 java 类，只有一个字段“color”，属性 public 和 final 函数声明

不应将 java 类名视为类型名称，因为它包含其他属性信息。不能将其括在尖括号中，因为它不能被称为类型。

java 接口的形式与类类型相同，可以扩展另一个接口，但与类不同，一个接口可以扩展多个接口，与类的另一个区别是接口不能被实例化，没有实例化，接口中的数据字段总是静态分配的。例如：

```
interface <$interfaceA> {      #这个接口继承了 interfaceA
 @s1 int32,                  # 接口中的数据字段总是静态分配的
&method1(int32) f32 }        # 一个方法声明
```

MAPLE IR 将接口声明作为类型声明处理。因此，可以在要与类型名称关联的类型关键字之后指定上述内容。另外，javainterface 关键字声明与接口关联的符号：

语法：

javainterface <id-name> <interface-type> <atteributes>

<id-name>必须以“$”作为前缀，因为接口名称始终具有全局作用域。例如：

```
javainterface $IFA <interface{&amethod(void) int32}> public static
# $IFA 是一个具有单个方法&amethod 的接口
```

同样，不应将 javainterface 名称视为类型名称，因为它是符号名称。当类实现接口时，它将 javainterface 名称指定为其逗号分隔内容的一部分，如下所示：

class <$PP> { &amethod(void) int32,
#这个类扩展了$PP 类，%amethod 是这个类的一个成员函数
$IFA } # 这个类实现了$IFA 接口

函数声明

语法：
func <func-name> <attributes> (
var <parm0> <type>,
. . .
var <parmn> <type>)<ret-type0>, . . .
<ret_typen> {
<stmt0>
. . .
<stmtn> }
<arrtibutes>提供有关函数的各种属性，如 static、extern 等。左括号之后是参数声明，可以为空。每个<parmi>都是每个传入参数的 var 或 reg 声明的形式。如果最后一个参数指定为“...”，则表示不定参数。参数声明之后是用逗号分隔的多个返回类型的列表。如果没有返回值,<ret-type0>应该设置为 void。每个<ret-typei>可以是原始类型，也可以是派生类型。如果大括号后没有语句，那么只是一个声明原型。

funcsize 语法：
funcsize <size-in-bytes>
此指令出现在函数块内，给出函数的 Maple IR 代码大小。

framesize 语法：
framesize <size-in-bytes>
此指令出现在函数块内，以给出函数的堆栈帧大小

moduleid 语法：
moduleid <id>
此指令出现在函数内部，给出函数所属模块的唯一 id

upformalsize 语法:

upformalsize <size-in-bytes>

此指令出现在函数块中，给出通过栈指针%%FP 传递的形参所在的段（upformal）的大小。

formalwordstypetagged 语法:

formalwordstypetagged = [<word-values>]

它指定一个位向量，该位向量初始化为 32 位整型常量列表指定的值。如果 upformal 段中的第 N 个字有类型标记，则该位向量中的第 N 位设置为 1，在这种情况下，类型标记位于第（N+1）个字。

formalwordsrefcounted 语法:

formalwordsrefcounted = [<word-values>]

它指定一个位向量，该位向量初始化为由 32 位整数常量的空格分隔列表指定的值。如果 upformal 段中的第 n 个字是指向动态分配内存块的引用计数的指针，则该位向量中的第 n 个位设置为 1。

localwordstypetagged 语法:

localwordstypetagged = [<word-values>]

指定一个位向量,该位向量初始化为32位整型常量列表指定的值。如果本地堆栈帧中的第 N 个字具有类型标记，则该位向量中的第 N 位设置为 1，在这种情况下，类型标记位于第（-N+1）个字处。

localwordsrefcounted 语法:

localwordsrefcounted = [<word-values>]

指定一个位向量，该位向量初始化为空间分隔的 32 位整数常量列表指定的值。如果本地堆栈帧中的第 n 个字是指向动态分配内存块的引用计数的指针，则该位向量中的第 n 个位设置为 1。

初始化

如果有与 var 声明相关联的初始化，则在 var 声明后有“=”，后跟初始化值。对于聚合，初始化值列表由括号“[”和“]”括起，值由逗号分隔。在数组中，数组元素的初始化值将逐个列出，并且必须使用嵌套括号来对应每个低阶维度中的元素。

在为结构指定初始化时，必须在括号内使用字段 ID 后跟“=”来显式指定每个字段的值。字段的初始化值可以按任意顺序列出。对于嵌套结

构，嵌套方括号的用法是可选的，因为顶级的字段 ID 可以唯一地指定嵌套结构中的字段。但是，如果使用嵌套方括号，则嵌套方括号中的字段 ID 用法与相应级别的子结构相对应。例如：

```
type %SS <struct { @g1 f64, @g2 f64 }>
var %s [struct{@f1i32,
@f2 <%SS>,
@f3:4 i32} = [ 1 = 99,
          2 = [1= 10.0],
        4 = 22.2,
          5 = 15 ]
```

类型参数

类型参数也称为泛型，允许在编写派生类型和函数时不必在其部分内容中指定确切的类型。类型参数可以稍后实例化为不同的特定类型，从而使代码更广泛地应用，从而促进软件重用。

类型参数及其实例化完全可以由语言前端处理。MAPLE IR 为泛型类型和泛型函数及其实例化提供表示，以减少语言前端的工作量。带有类型参数的 MAPLE IR 文件需要前端降低阶段来取消泛型化在其他 MAPLE IR 组件可以处理代码之前的 IR。

类型参数是前缀为“！”的符号名，并且可以出现在类型可以出现的任何位置。每个类型或函数定义可以有多个类型参数，并且每个类型参数可以出现多次。由于类型参数也是类型，因此它们只能出现在尖括号“<”和“>”中，例如：<！T> 。当派生类型的定义包含任何类型参数时，该类型将成为泛型类型。当函数的定义包含任何类型参数时，该函数将成为泛型函数。函数原型不能包含泛型类型。

泛型类型或泛型函数使用泛型属性进行标记，以便更容易识别它们。

泛型类型或函数通过为其每个类型参数分配特定的非泛型类型来实例化。实例化是由一个用逗号分隔的此类赋值列表指定的。我们将其称为实例化向量，它在大括号“{”和“}”中指定。对于泛型类型的实例化，在类型参数后面紧跟实例化向量。例如：

```
type $apair <struct {@f1 <!T>, @f2 <!T>}>
var $x <$apair{!T=f32}>
```

泛型函数通过使用实例化向量调用它来实例化。实例化向量紧跟泛型函数的名称。由于实例化向量被视为类型信息，因此它被进一步封装在尖括号“<”和“>”中。泛型函数的调用必须通过操作码 callinstant 和

callinstantassigned 进行，这两个操作码分别对应于 call 和 callassigned, 例如:

```
func &swap (var %x <!UU>, var %y <!UU>) void {
var %z <!UU>
dassign %z (dread agg %x)
dassiign %x (dread agg %y)
dassign %y (dread agg %z)
 return
}

 . . .

callinstant &swap<{!UU=<$apair{!T=i32}>}> (
dread agg %a,
 dread agg %b)
```

样例

样例 1

函数定义和算术运算:

C 代码:

```
int foo(int i,int j){
return (i + j) * -998;
}
```

```
MapleIR:
func &foo (var %i i32, var %j i32) i32 {
return (
      mul i32 (
            add i32 (dread i32 %i, dread i32 %j),
            constval i32 -998))}
```

样例 2

循环和数组。

C 代码:

```
float a[10];
void init(void){
int i;
for(i=0; i<10; i++)
        a[i]=i*3;
}
```

MapleIR:

```
var $a <[10] f32>
func &init() void{
var %i i32
dassign %i(constval i32 0)
while(
lt i32 i32(dread i32 %i, constval i32 10)){
        iassign<*[10] f32>(
                array a32<*[10] f32>(addrof a32 $a, dread i32 %i),
                mul i32(dread i32 %i, constval i32 3))
        dassign %i(
                add i32(dread i32 %i, constval i32 1))}}
```

样例 3

类型和结构体。

C 代码:

```
typedef struct SS{
int f1;
char f2:6;
char f3:2;
}SS;
SS foo(SS x){
x.f2=33;
return x;
}
```

MapleIR:

```
type $SS <struct {@f1 i32, @f2:6 i8,@f3:2 i8}>
func &foo (var %x <$SS>) <$SS> {
dassign %x 2 (constval i32 32 )
return (dread agg %x) }
```

样例 4

If-then-else，函数调用以及块结构。

C 代码:

```
int fact(int n) {
if(n!=1)
    return n*fact(n-1);
else return 1;
}
```

MapleIR:

```
func &fact (var %n i32) i32 {
if (ne i32 (dread i32 %n, constval i32 1)) {
    call &fact (sub i32 (dread i32 %n, constval i32 1))
    return (mul i32 (dread i32 %n, regread i32 %%retval))
}
return(constval i32 1)
}
```