TableGen入门与实践

刘子康

2021年 9月29日

目录

TableGen是什么

- 是一个领域特定语言（DSL）
- 广泛用于LLVM项目（包括Clang和MLIR等）
- 类似C++类描述的语法，声明式地描述数据和数据结构
- 用于自动化生成其他语言文件（主要是C++用的.inc）
- 减少重复工作量：将例如架构信息的内容整合后生成，不需要在其他文件中重复声明
- 前后端分离，同一个前端解析文件，**不同的后端**生成不同内容

TableGen在LLVM-project主要用途

- 在前端（clang, flang等）中用作诊断（Diagnostics）和属性（Attributes）
- **在LLVM中大量用于代码生成器中描述生成目标(Target)**
- 在MLIR中用于描述Dialect等
- 少部分用于其他模块，通常包含字符串信息等，如调试信息

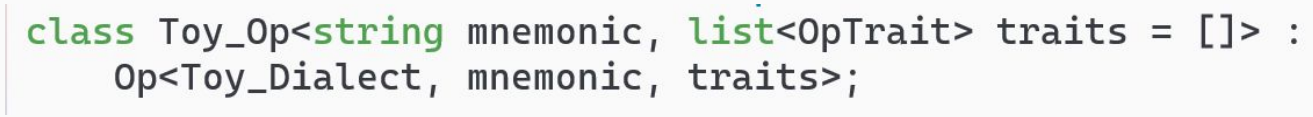
TableGen在Clang

clang/include/clang/Basic/DiagnosticCommonKinds.td

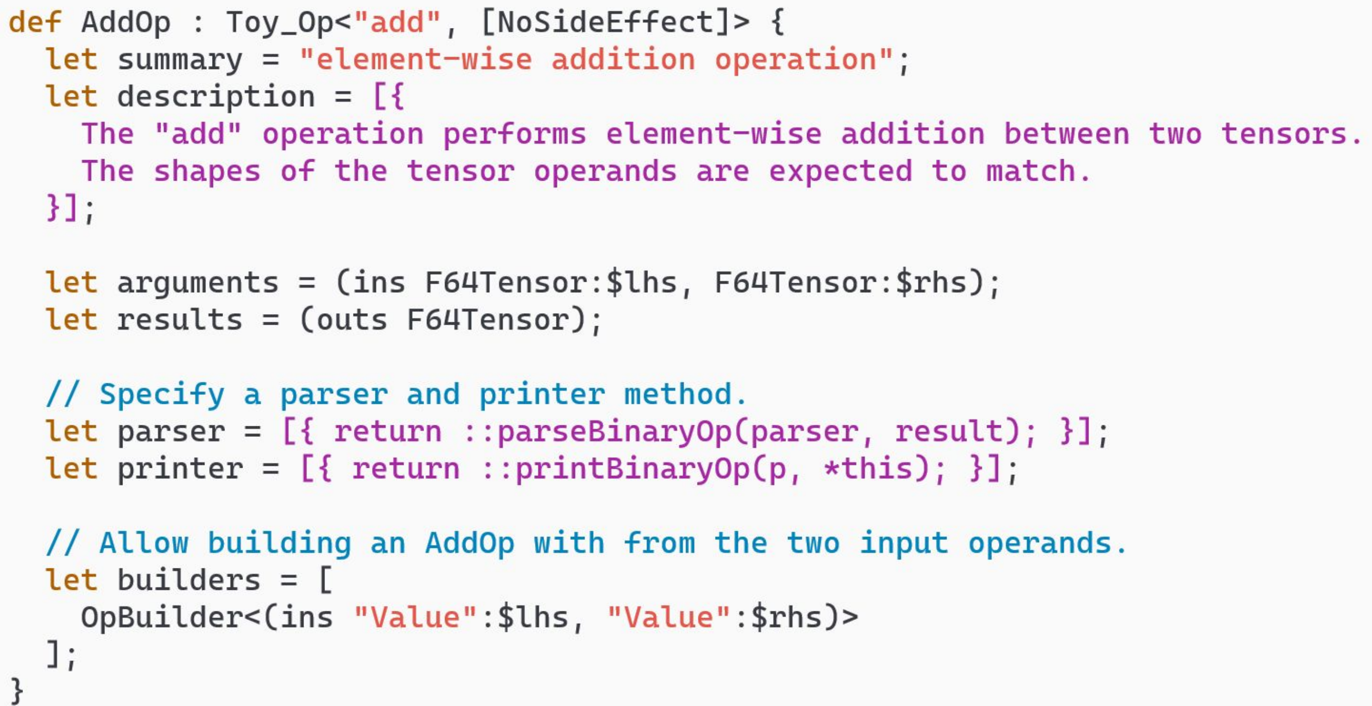
```
let Component = "Common" in {

// Substitutions.

def select_constexpr_spec_kind : TextSubstitution<
  "%select{<ERROR>|constexpr|consteval|constinit}0">;

// Basic.

def fatal_too_many_errors
  : Error<"too many errors emitted, stopping now">, DefaultFatal;

def warn_stack_exhausted : Warning<
  "stack nearly exhausted; compilation time may suffer, and "
  "crashes due to stack overflow are likely">,
  InGroup<DiagGroup<"stack-exhausted">>, NoSFINAE;
```

TableGen在MLIR

/mlir/examples/toy/Ch3/include/toy/Ops.td (教程示例代码）

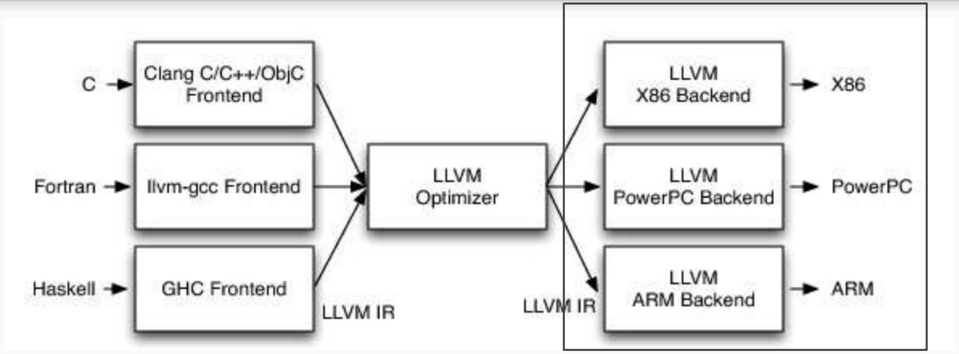


```
class Toy_Op<string mnemonic, list<OpTrait> traits = []> :
    Op<Toy_Dialect, mnemonic, traits>;
```



```
def AddOp : Toy_Op<"add", [NoSideEffect]> {
  let summary = "element-wise addition operation";
  let description = [{
    The "add" operation performs element-wise addition between two tensors.
    The shapes of the tensor operands are expected to match.
  }];

  let arguments = (ins F64Tensor:$lhs, F64Tensor:$rhs);
  let results = (outs F64Tensor);

  // Specify a parser and printer method.
  let parser = [{ return ::parseBinaryOp(parser, result); }];
  let printer = [{ return ::printBinaryOp(p, *this); }];

  // Allow building an AddOp with from the two input operands.
  let builders = [
    OpBuilder<(ins "Value":$lhs, "Value":$rhs)>
  ];
}
```

为什么要使用TableGen?



LLVM Code Generator

- 每个生成器都专为特定目标服务
- 生成器之间有相似问题需要解决：给寄存器赋值，尽管各平台寄存器不一样，算法大同小异
- LLVM的代码生成器分为多阶段：指令选择，寄存器分配，调度，layout优化，汇编生成；默认提供了很多passes，开发者可采取默认值或覆盖操作
    - 多目标共用的组件需要“了解”它们生成目标的特征
    - 每个目标都使用一系列描述文件来声明它们的特征

*.td : *Target Description*

Brown, Amy, and Greg Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*.

题外话：TableGen高亮

- TableGen支持Vim和Emacs的词法高亮

  llvm　　/utils/vim（复制所有文件到~/.vim）

  llvm/utils/emacs

  VS Code：目前有且仅有 Jakob Erzar的LLVM TableGen一个插件可高亮

TableGen特性和基本概念：语法

最基本的两种类型：

- Class（Abstract record）：抽象描述一种数据（Record）的集合
- Record （Concrete record）：具体的，实例化的数据内容

每个广义record（class或record）都有一个独一无二的名字（name），可指定或自动生成（匿名Record）。

名字后跟随一系列的成员属性（fields），fields是后端主要处理的内容。fields的含义由后端决定。

不同后端可生成不同文件，例如生成”.inc”文件供C++文件include，也可生成JSON文件等。

TableGen特性和基本概念：语法

llvm/lib/Target/RISCV/RISCVInstrFormats.td

```
class InstFormat<bits<5> val> {
  bits<5> Value = val;
}
def InstFormatPseudo : InstFormat<0>;
def InstFormatR      : InstFormat<1>;
def InstFormatR4     : InstFormat<2>;
def InstFormatI      : InstFormat<3>;
```

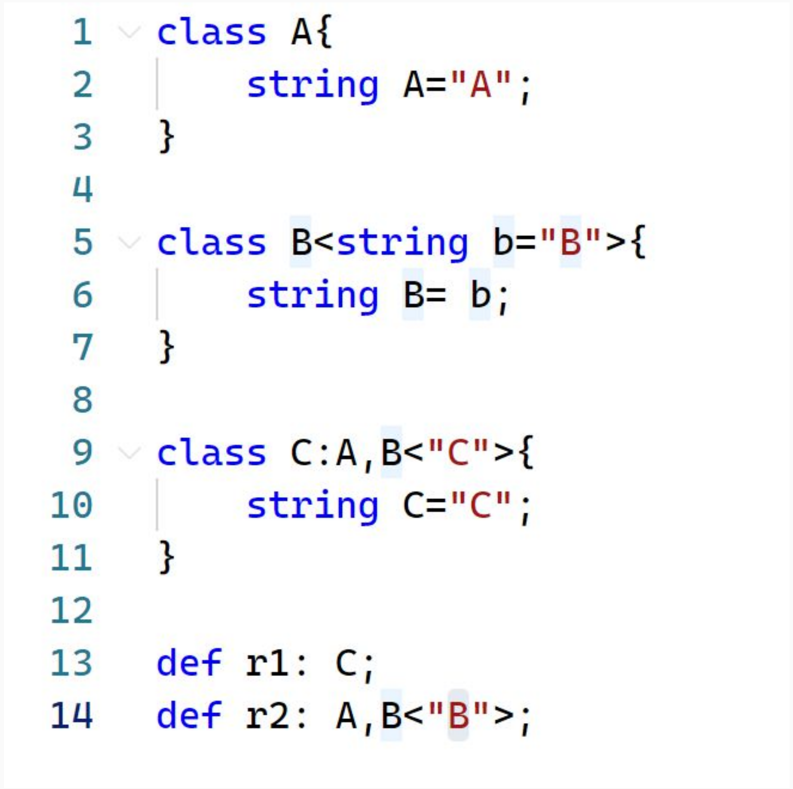
参数列表

数据类型

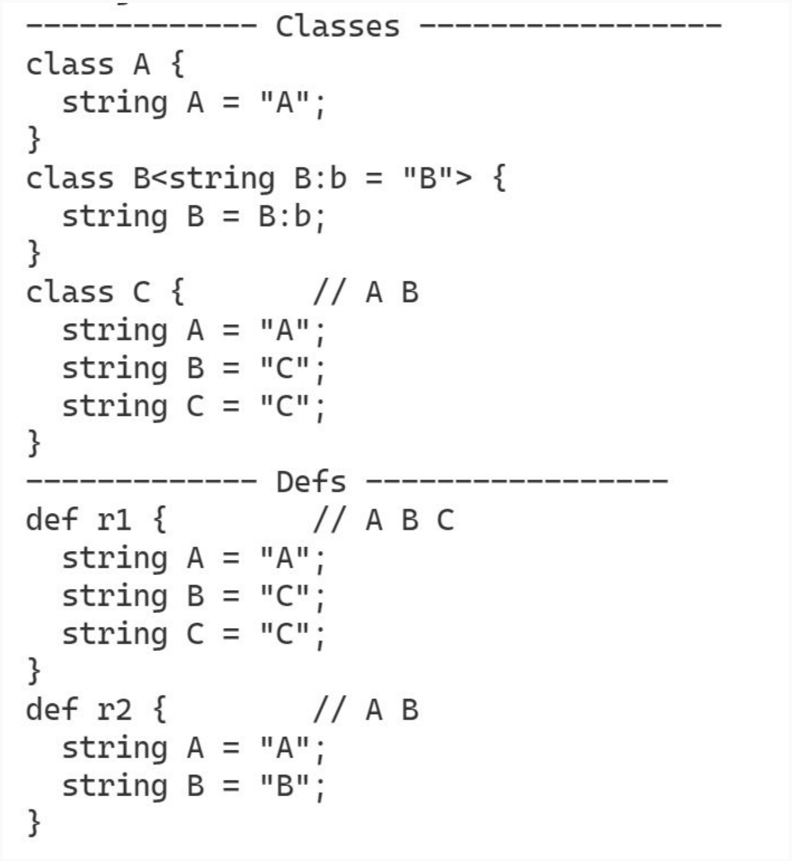
使用def关键字将class具体化为一个record

TableGen特性和基本概念：语法

- Class的继承和Record的多Class实例化



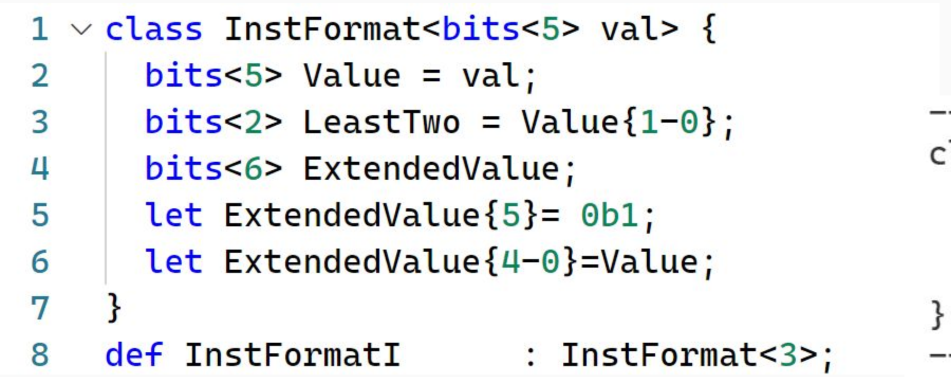
```
class A{
    string A="A";
}

class B<string b="B">{
    string B= b;
}

class C:A,B<"C">{
    string C="C";
}

def r1: C;
def r2: A,B<"B">;
```



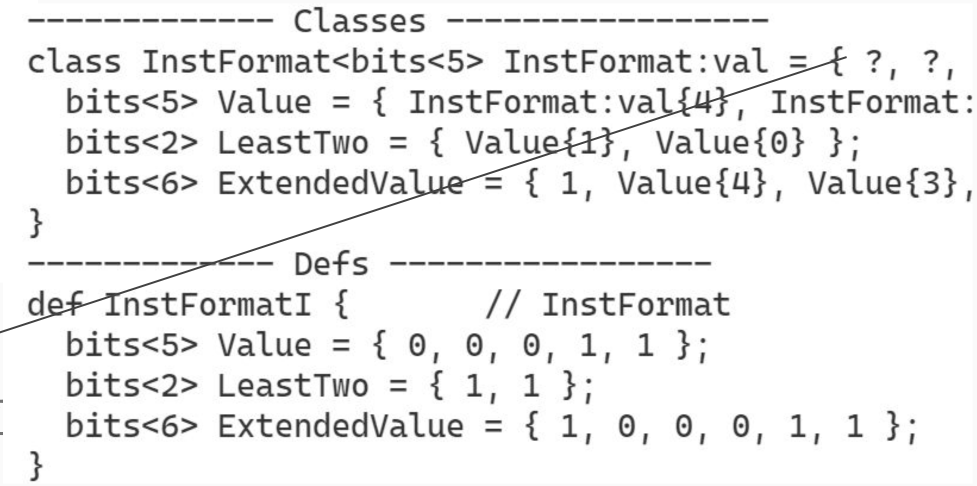
```
------------- Classes -----------------
class A {
  string A = "A";
}
class B<string B:b = "B"> {
  string B = B:b;
}
class C {        // A B
  string A = "A";
  string B = "C";
  string C = "C";
}
------------- Defs -----------------
def r1 {        // A B C
  string A = "A";
  string B = "C";
  string C = "C";
}
def r2 {        // A B
  string A = "A";
  string B = "B";
}
```

TableGen特性和基本概念：数据类型

- bit, bits<N>, int(64 bits)
  - 隐式类型转换
  - 比特序列：5 = {0, 0, 1, 0, 1}
  - 可slice（如定长指令划分操作数和寄存器地址）： {Start-End}，不能对int用



```
class InstFormat<bits<5> val> {
    bits<5> Value = val;
    bits<2> LeastTwo = Value{1-0};
    bits<6> ExtendedValue;
    let ExtendedValue{5}= 0b1;
    let ExtendedValue{4-0}=Value;
}
def InstFormatI       : InstFormat<3>;
```

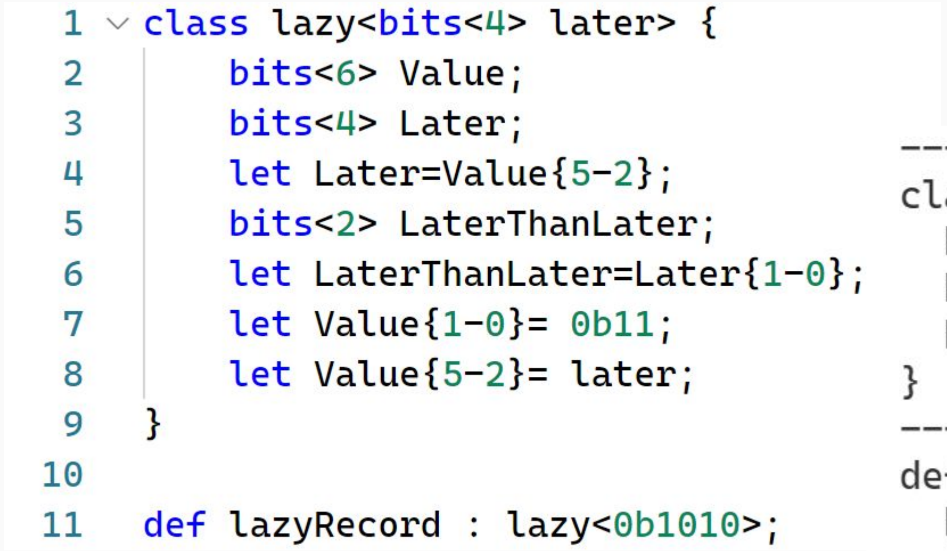


```
------------- Classes -----------------
class InstFormat<bits<5> InstFormat:val = { ?, ?,
  bits<5> Value = { InstFormat:val{4}, InstFormat:
  bits<2> LeastTwo = { Value{1}, Value{0} };
  bits<6> ExtendedValue = { 1, Value{4}, Value{3},
}
------------- Defs -----------------
def InstFormatI {       // InstFormat
  bits<5> Value = { 0, 0, 0, 1, 1 };
  bits<2> LeastTwo = { 1, 1 };
  bits<6> ExtendedValue = { 1, 0, 0, 0, 1, 1 };
}
```

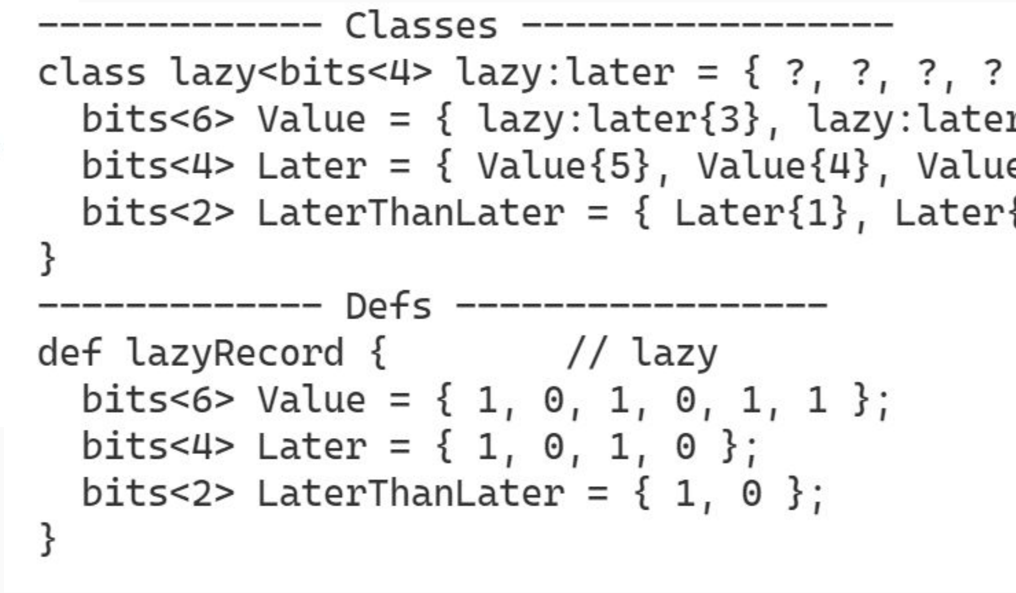
- Let：赋值已经存在的成员变量
- unset：（？）未定值，惰性取值

TableGen特性和基本概念：数据类型

- unset：（？）未定值，惰性取值



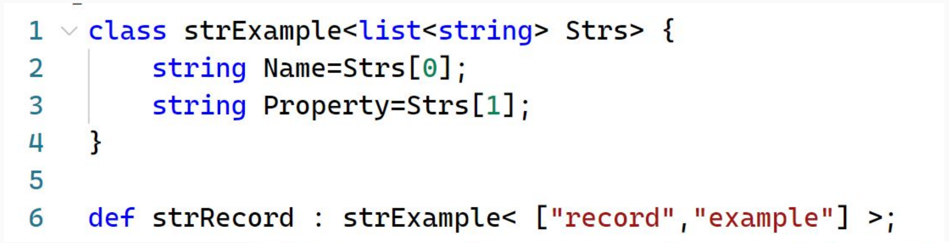
```
class lazy<bits<4> later> {
    bits<6> Value;
    bits<4> Later;
    let Later=Value{5-2};
    bits<2> LaterThanLater;
    let LaterThanLater=Later{1-0};
    let Value{1-0}= 0b11;
    let Value{5-2}= later;
}

def lazyRecord : lazy<0b1010>;
```



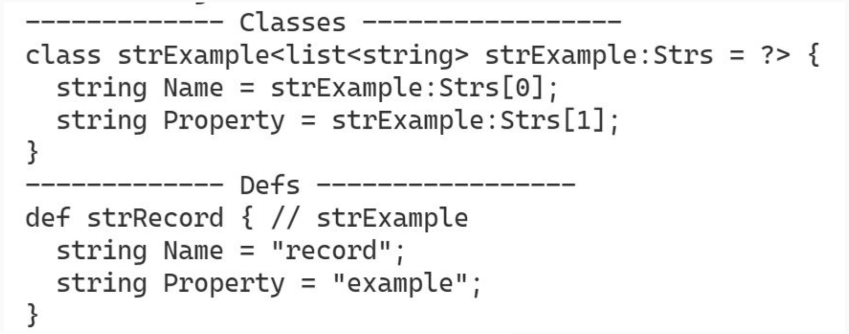
```
------------- Classes -----------------
class lazy<bits<4> lazy:later = { ?, ?, ?, ?
  bits<6> Value = { lazy:later{3}, lazy:later
  bits<4> Later = { Value{5}, Value{4}, Value
  bits<2> LaterThanLater = { Later{1}, Later{
}
------------- Defs -----------------
def lazyRecord {        // lazy
  bits<6> Value = { 1, 0, 1, 0, 1, 1 };
  bits<4> Later = { 1, 0, 1, 0 };
  bits<2> LaterThanLater = { 1, 0 };
}
```

TableGen特性和基本概念：数据类型

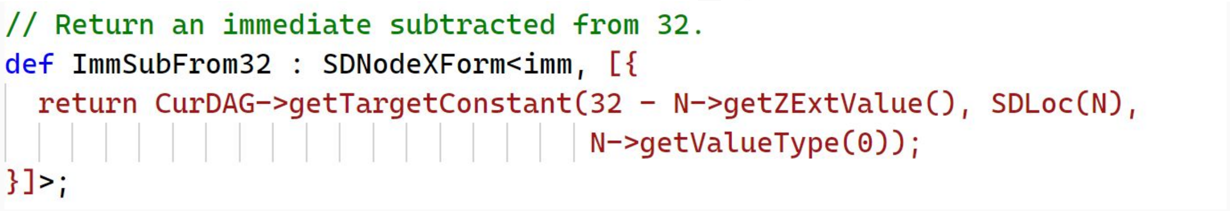
- string, code
  - string用于存储字符串 “This is an example string”
  - Code用于指示一个string是code [{ This.IsCode(); }]
- list<T>
  - list<string>: [“Hello”, “World”, “!”]
  - 下标访问，下标只能是常量



```
class strExample<list<string> Strs> {
    string Name=Strs[0];
    string Property=Strs[1];
}

def strRecord : strExample< ["record","example"] >;
```



```
------------- Classes -----------------
class strExample<list<string> strExample:Strs = ?> {
  string Name = strExample:Strs[0];
  string Property = strExample:Strs[1];
}
------------- Defs -----------------
def strRecord { // strExample
  string Name = "record";
  string Property = "example";
}
```

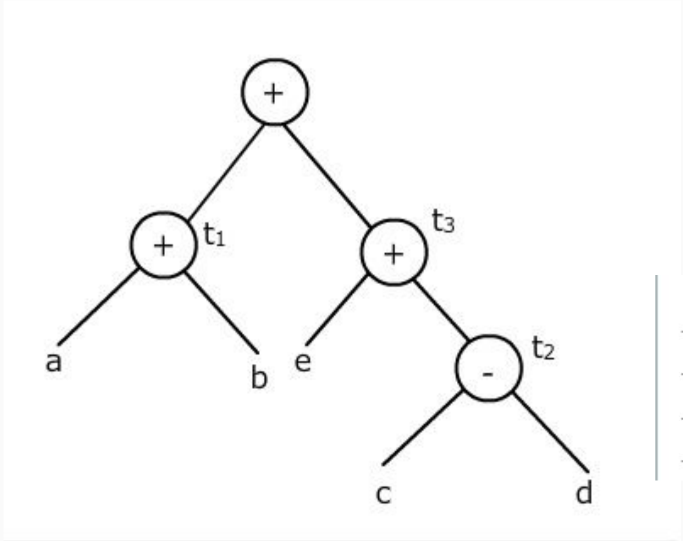


```
// Return an immediate subtracted from 32.
def ImmSubFrom32 : SDNodeXForm<imm, [{
  return CurDAG->getTargetConstant(32 - N->getZExtValue(), SDLoc(N),
                                   N->getValueType(0));
}]>;
```

TableGen特性和基本概念：数据类型

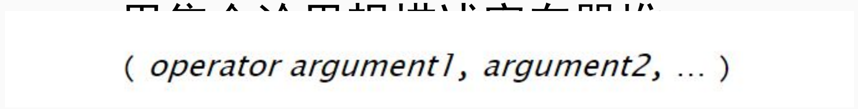
- DAG：Directed Acyclic Graph 有向无环图

编译原理用的DAG：一种表示法

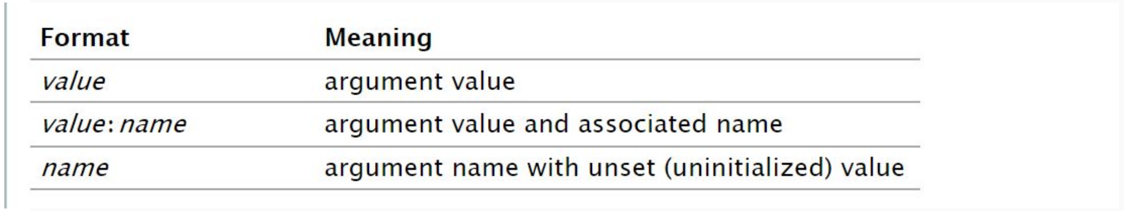


DAG for *(a + b) + (e + (c - d))*

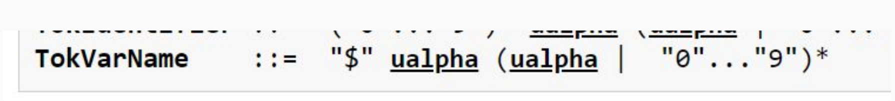
三种用处：
- 描述机器指令的操作数
- **描述指令选择（ISel）的模式（Pattern）**



(*operator argument1*, *argument2*, …)



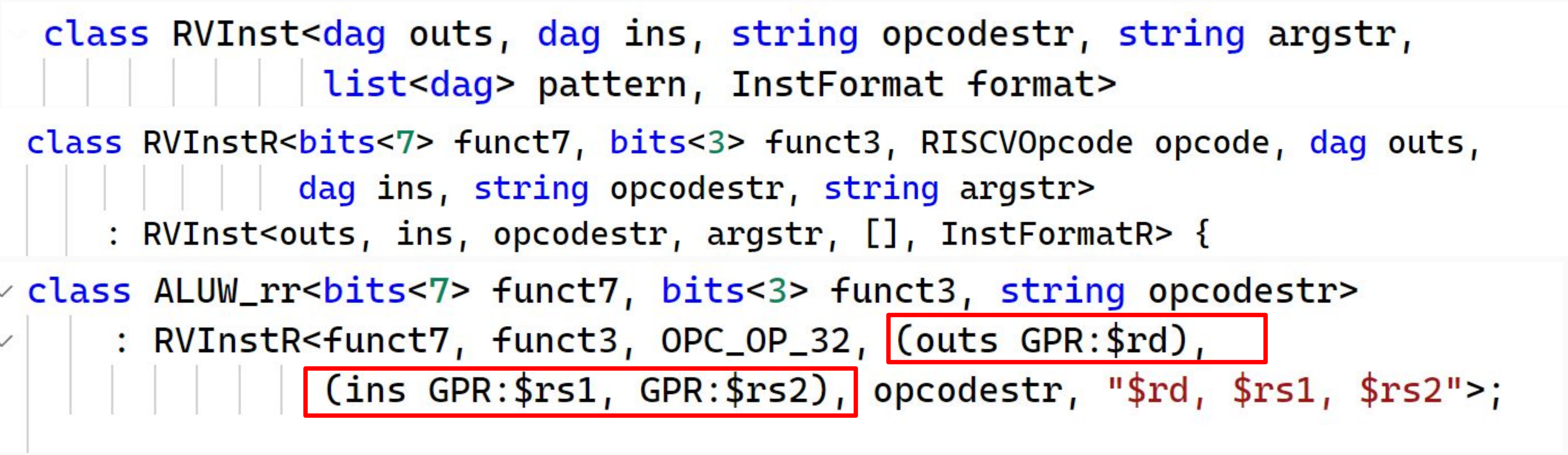
Format	Meaning
value	argument value
value:*name*	argument value and associated name
name	argument name with unset (uninitialized) value



```
TokVarName    ::=  "$" ualpha (ualpha |  "0"..."9")*
```

TableGen特性和基本概念：数据类型，DAG

- 描述机器指令的操作数



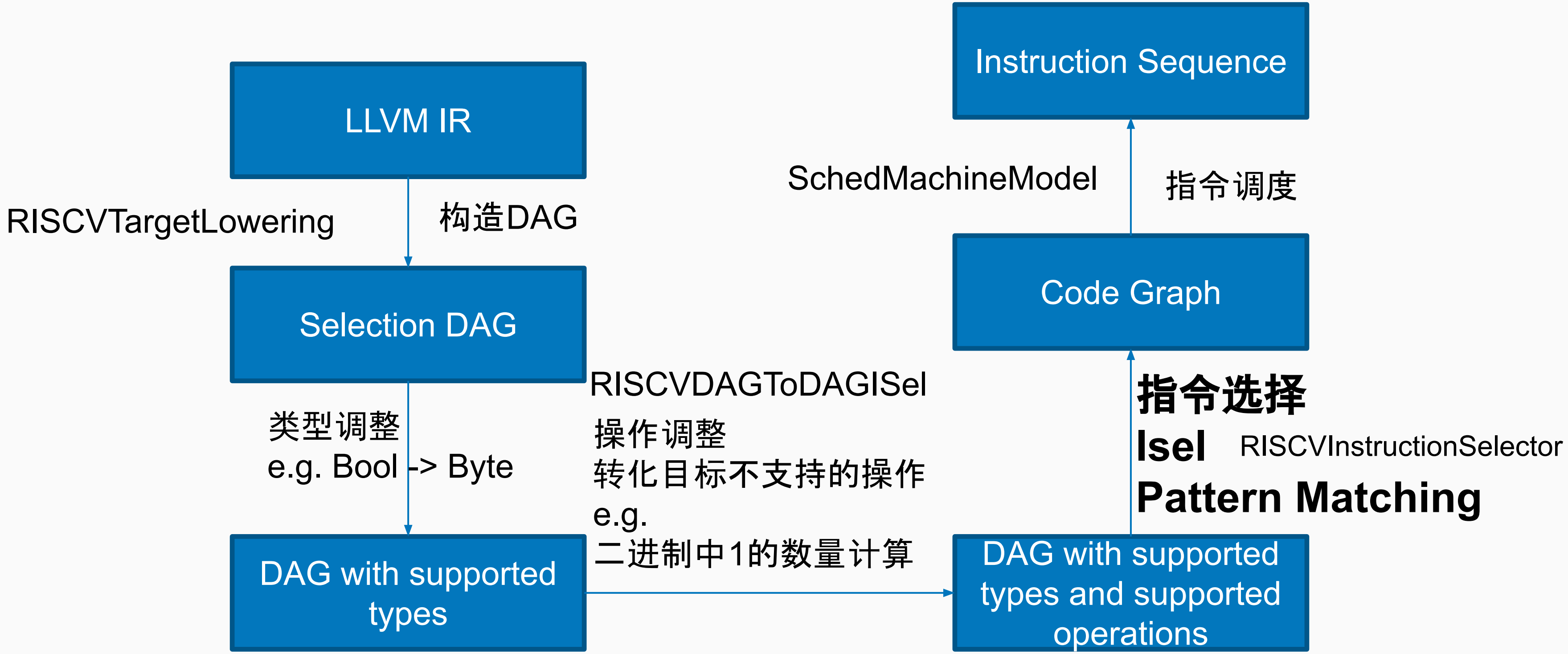
```
class RVInst<dag outs, dag ins, string opcodestr, string argstr,
             list<dag> pattern, InstFormat format>
```

```
class RVInstR<bits<7> funct7, bits<3> funct3, RISCVOpcode opcode, dag outs,
              dag ins, string opcodestr, string argstr>
    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
```

```
class ALUW_rr<bits<7> funct7, bits<3> funct3, string opcodestr>
    : RVInstR<funct7, funct3, OPC_OP_32, (outs GPR:$rd),
              (ins GPR:$rs1, GPR:$rs2), opcodestr, "$rd, $rs1, $rs2">;
```

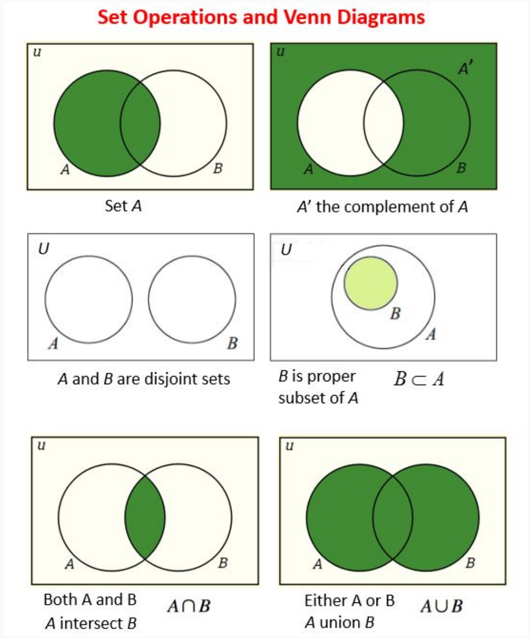
TableGen特性和基本概念：数据类型，DAG

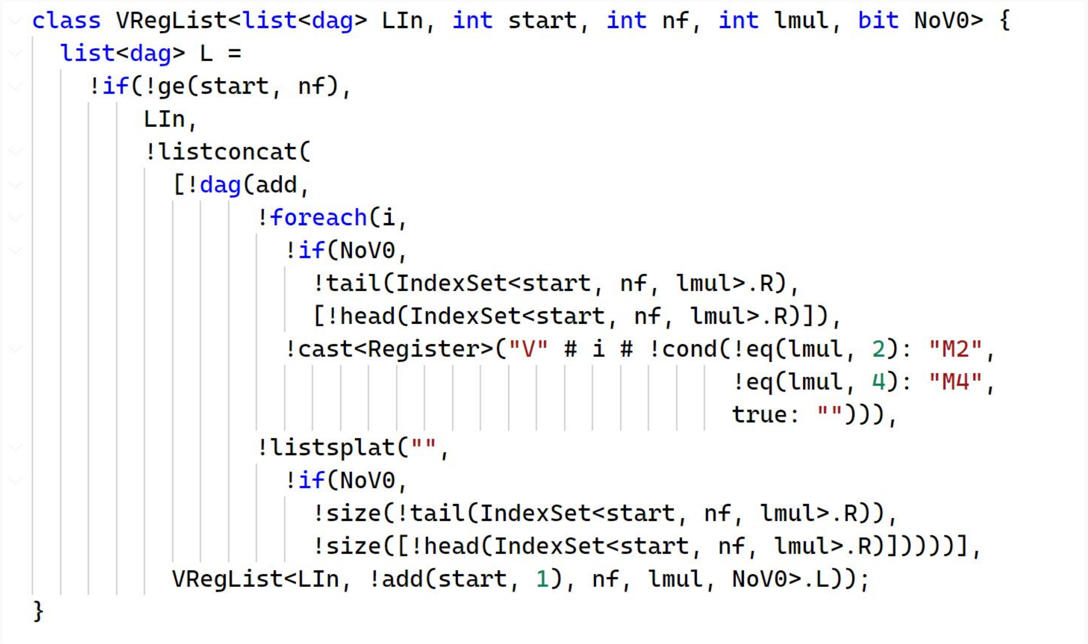
- 描述指令选择（Instruction Selection）的模式（Pattern）



TableGen特性和基本概念：数据类型，DAG

- 用集合论思想描述寄存器堆(?)

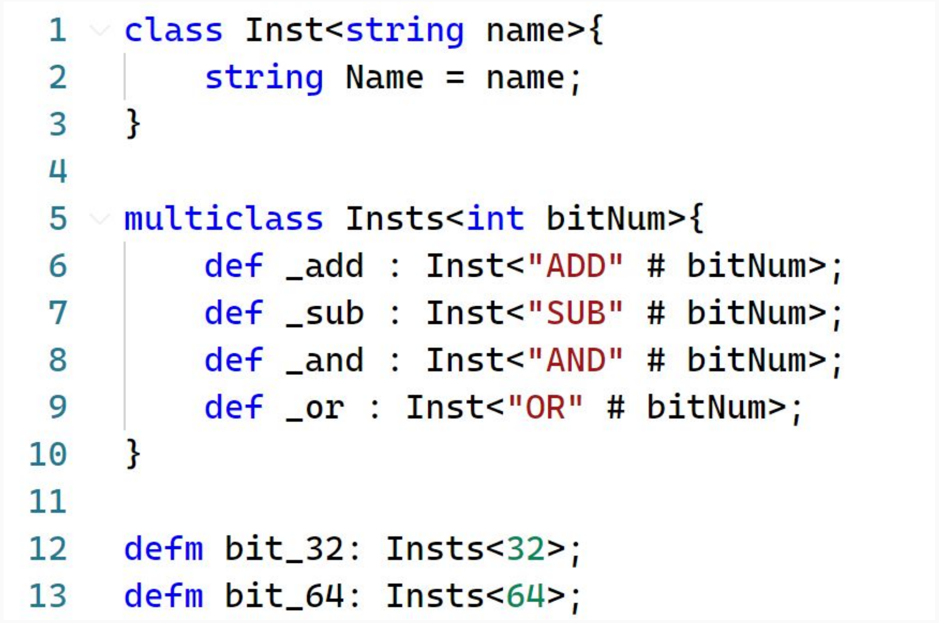




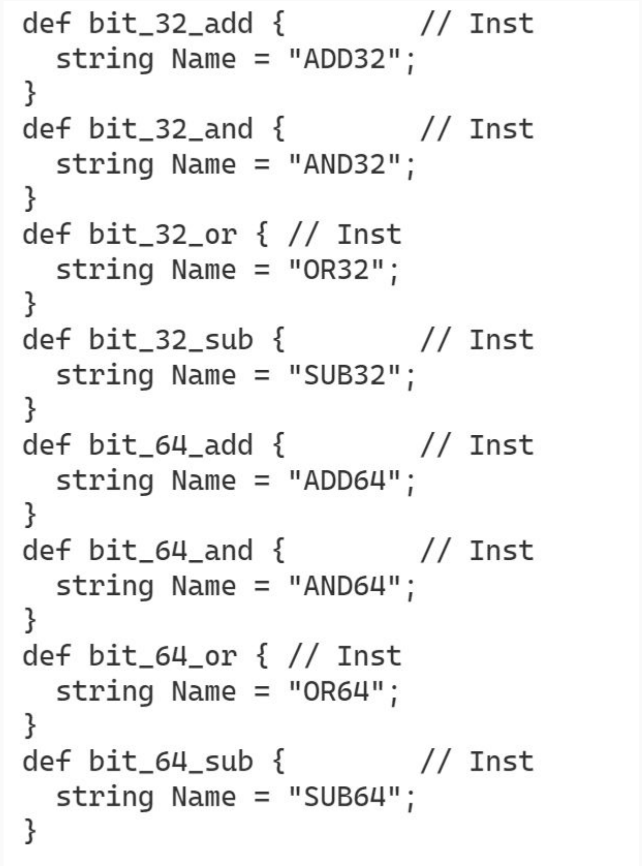
```
class VRegList<list<dag> LIn, int start, int nf, int lmul, bit NoV0> {
  list<dag> L =
    !if(!ge(start, nf),
        LIn,
        !listconcat(
          [!dag(add,
                !foreach(i,
                  !if(NoV0,
                    !tail(IndexSet<start, nf, lmul>.R),
                    [!head(IndexSet<start, nf, lmul>.R)]),
                  !cast<Register>("V" # i # !cond(!eq(lmul, 2): "M2",
                                                  !eq(lmul, 4): "M4",
                                                  true: ""))),
                !listsplat("",
                  !if(NoV0,
                    !size(!tail(IndexSet<start, nf, lmul>.R)),
                    !size([!head(IndexSet<start, nf, lmul>.R)]))))],
          VRegList<LIn, !add(start, 1), nf, lmul, NoV0>.L));
}
```

TableGen特性和基本概念：语法

- Multiclass
    - 用于批量生成具有某种共性的records



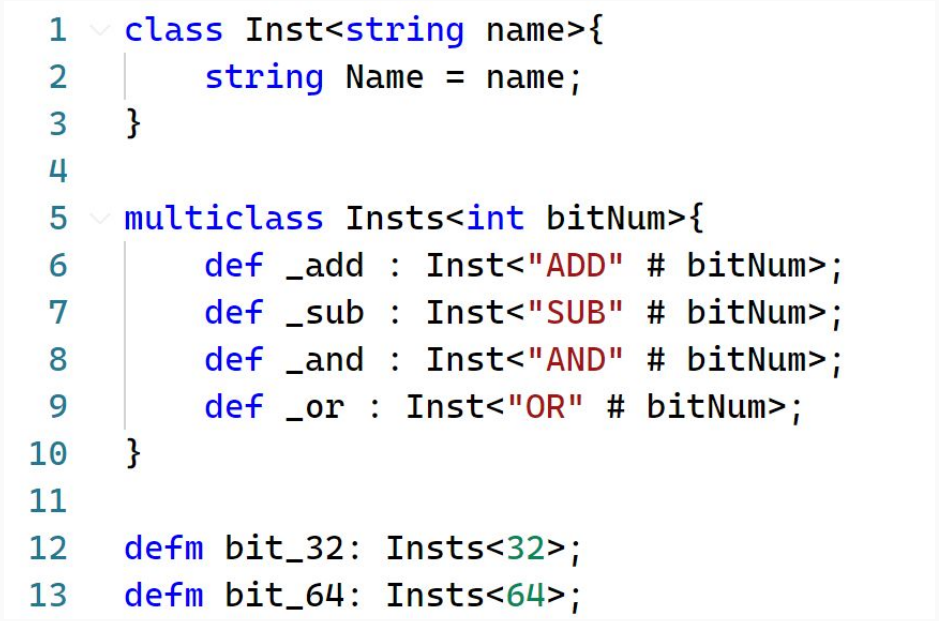
```
class Inst<string name>{
    string Name = name;
}

multiclass Insts<int bitNum>{
    def _add : Inst<"ADD" # bitNum>;
    def _sub : Inst<"SUB" # bitNum>;
    def _and : Inst<"AND" # bitNum>;
    def _or : Inst<"OR" # bitNum>;
}

defm bit_32: Insts<32>;
defm bit_64: Insts<64>;
```



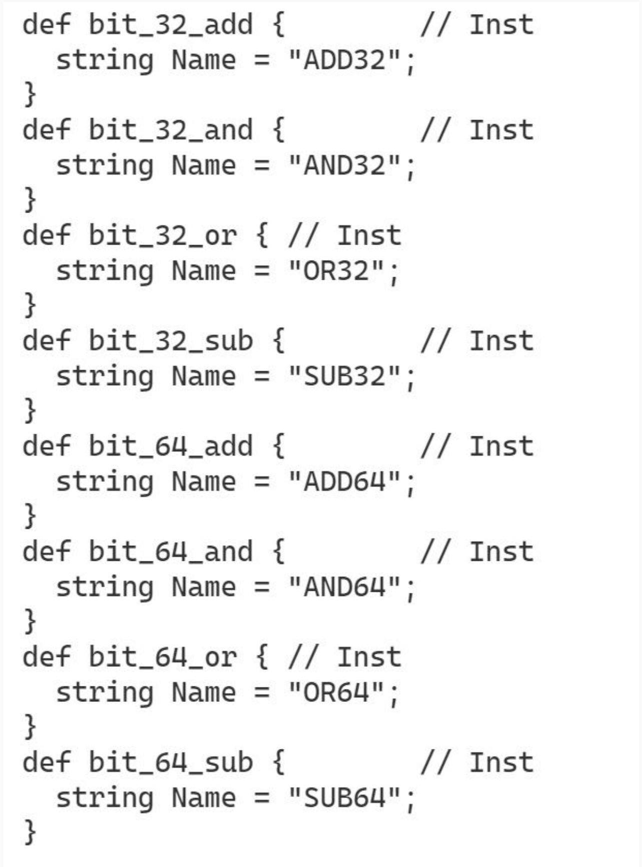
```
def bit_32_add {          // Inst
  string Name = "ADD32";
}
def bit_32_and {          // Inst
  string Name = "AND32";
}
def bit_32_or { // Inst
  string Name = "OR32";
}
def bit_32_sub {          // Inst
  string Name = "SUB32";
}
def bit_64_add {          // Inst
  string Name = "ADD64";
}
def bit_64_and {          // Inst
  string Name = "AND64";
}
def bit_64_or { // Inst
  string Name = "OR64";
}
def bit_64_sub {          // Inst
  string Name = "SUB64";
}
```

TableGen特性和基本概念：语法

- Multiclass和defm
  - 用于批量生成具有某种共性的records



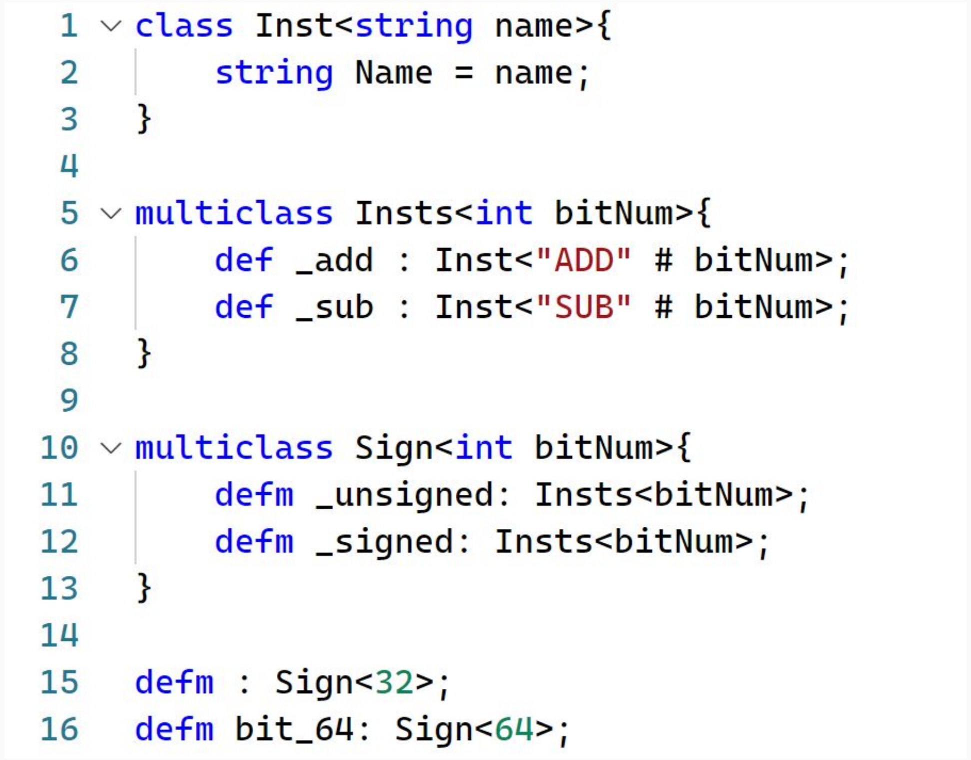
```
class Inst<string name>{
    string Name = name;
}

multiclass Insts<int bitNum>{
    def _add : Inst<"ADD" # bitNum>;
    def _sub : Inst<"SUB" # bitNum>;
    def _and : Inst<"AND" # bitNum>;
    def _or : Inst<"OR" # bitNum>;
}

defm bit_32: Insts<32>;
defm bit_64: Insts<64>;
```



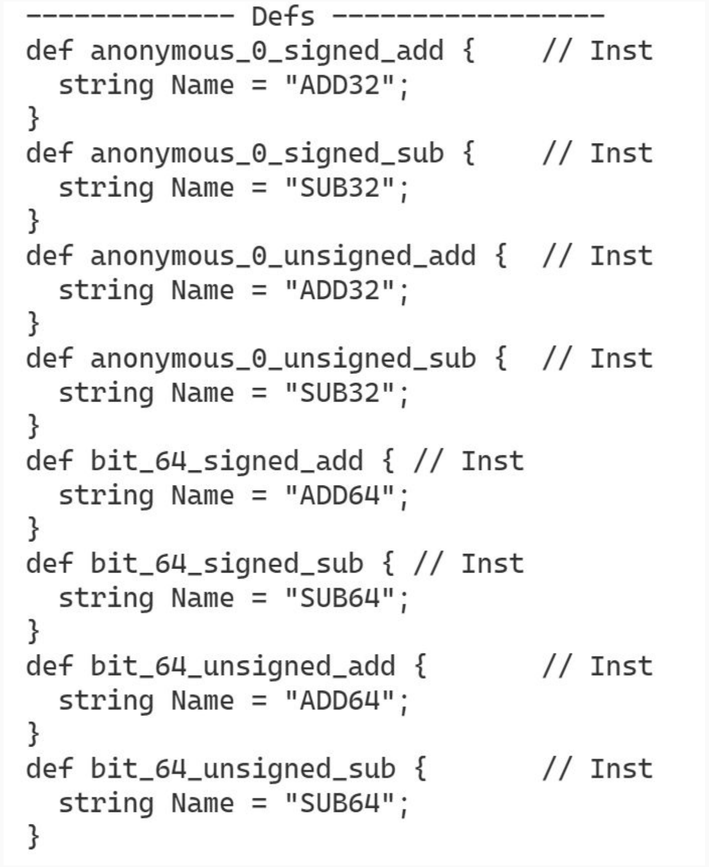
```
def bit_32_add {        // Inst
  string Name = "ADD32";
}
def bit_32_and {        // Inst
  string Name = "AND32";
}
def bit_32_or { // Inst
  string Name = "OR32";
}
def bit_32_sub {        // Inst
  string Name = "SUB32";
}
def bit_64_add {        // Inst
  string Name = "ADD64";
}
def bit_64_and {        // Inst
  string Name = "AND64";
}
def bit_64_or { // Inst
  string Name = "OR64";
}
def bit_64_sub {        // Inst
  string Name = "SUB64";
}
```

TableGen特性和基本概念：语法

- 嵌套multiclass和匿名record



```
class Inst<string name>{
    string Name = name;
}

multiclass Insts<int bitNum>{
    def _add : Inst<"ADD" # bitNum>;
    def _sub : Inst<"SUB" # bitNum>;
}

multiclass Sign<int bitNum>{
    defm _unsigned: Insts<bitNum>;
    defm _signed: Insts<bitNum>;
}

defm : Sign<32>;
defm bit_64: Sign<64>;
```

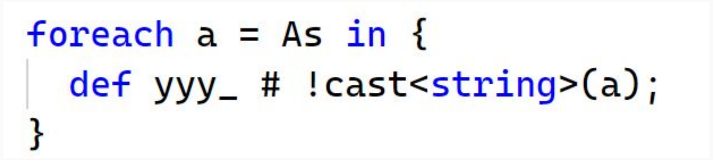


```
------------- Defs -----------------
def anonymous_0_signed_add {    // Inst
  string Name = "ADD32";
}
def anonymous_0_signed_sub {    // Inst
  string Name = "SUB32";
}
def anonymous_0_unsigned_add {  // Inst
  string Name = "ADD32";
}
def anonymous_0_unsigned_sub {  // Inst
  string Name = "SUB32";
}
def bit_64_signed_add { // Inst
  string Name = "ADD64";
}
def bit_64_signed_sub { // Inst
  string Name = "SUB64";
}
def bit_64_unsigned_add {       // Inst
  string Name = "ADD64";
}
def bit_64_unsigned_sub {       // Inst
  string Name = "SUB64";
}
```

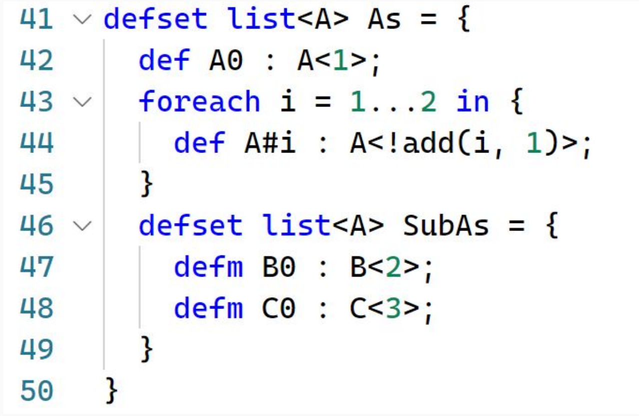
TableGen特性和基本概念：语法

- Defset
  - 将一系列record收集到一个全局的list里
  - 少见
- Foreach
  - 遍历列表并做某事
  - 多层的迭代结构

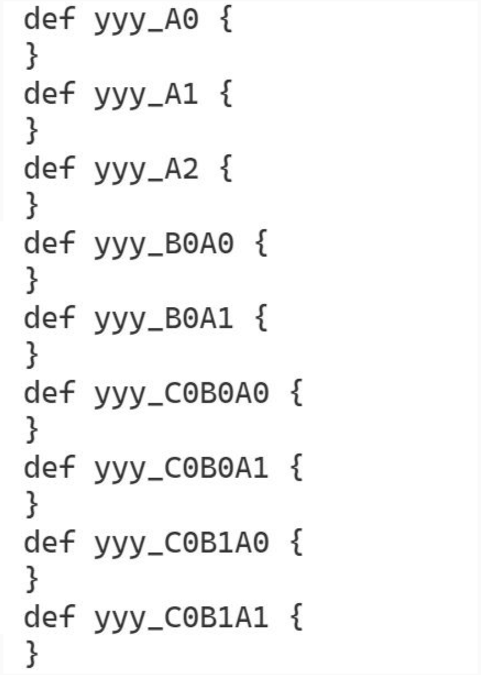
    可简化写法



```
foreach a = As in {
  def yyy_ # !cast<string>(a);
}
```



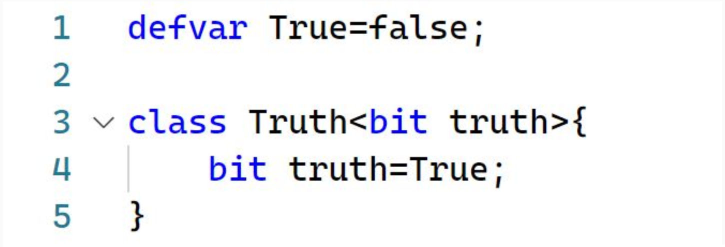
```
defset list<A> As = {
  def A0 : A<1>;
  foreach i = 1...2 in {
    def A#i : A<!add(i, 1)>;
  }
  defset list<A> SubAs = {
    defm B0 : B<2>;
    defm C0 : C<3>;
  }
}
```



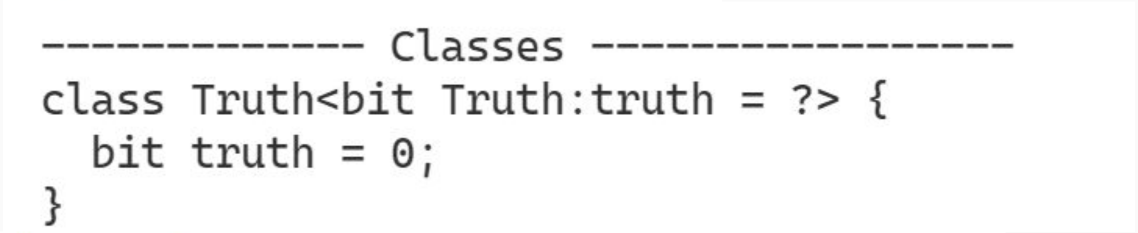
```
def yyy_A0 {
}
def yyy_A1 {
}
def yyy_A2 {
}
def yyy_B0A0 {
}
def yyy_B0A1 {
}
def yyy_C0B0A0 {
}
def yyy_C0B0A1 {
}
def yyy_C0B1A0 {
}
def yyy_C0B1A1 {
}
```

TableGen特性和基本概念：语法

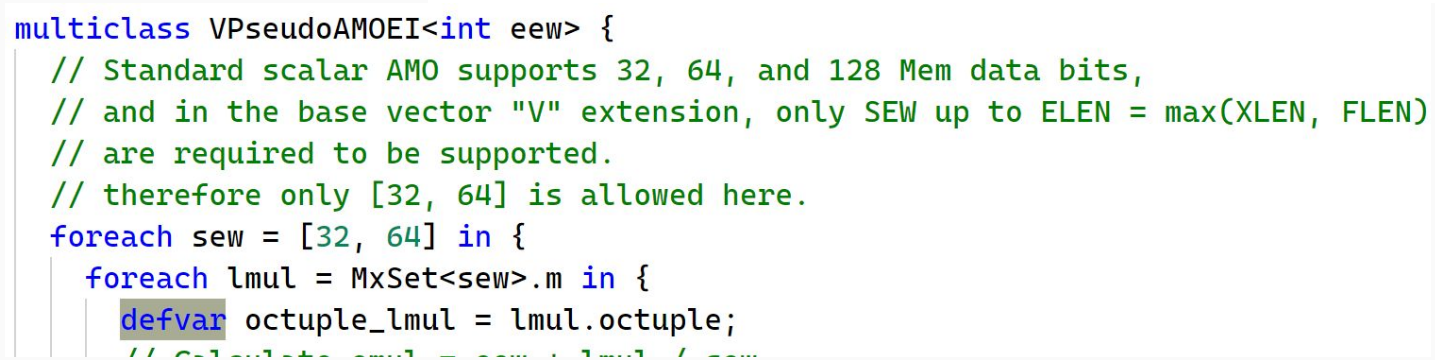
- Defvar
  - 声明全局常量
  - 起别名



```
defvar True=false;

class Truth<bit truth>{
    bit truth=True;
}
```



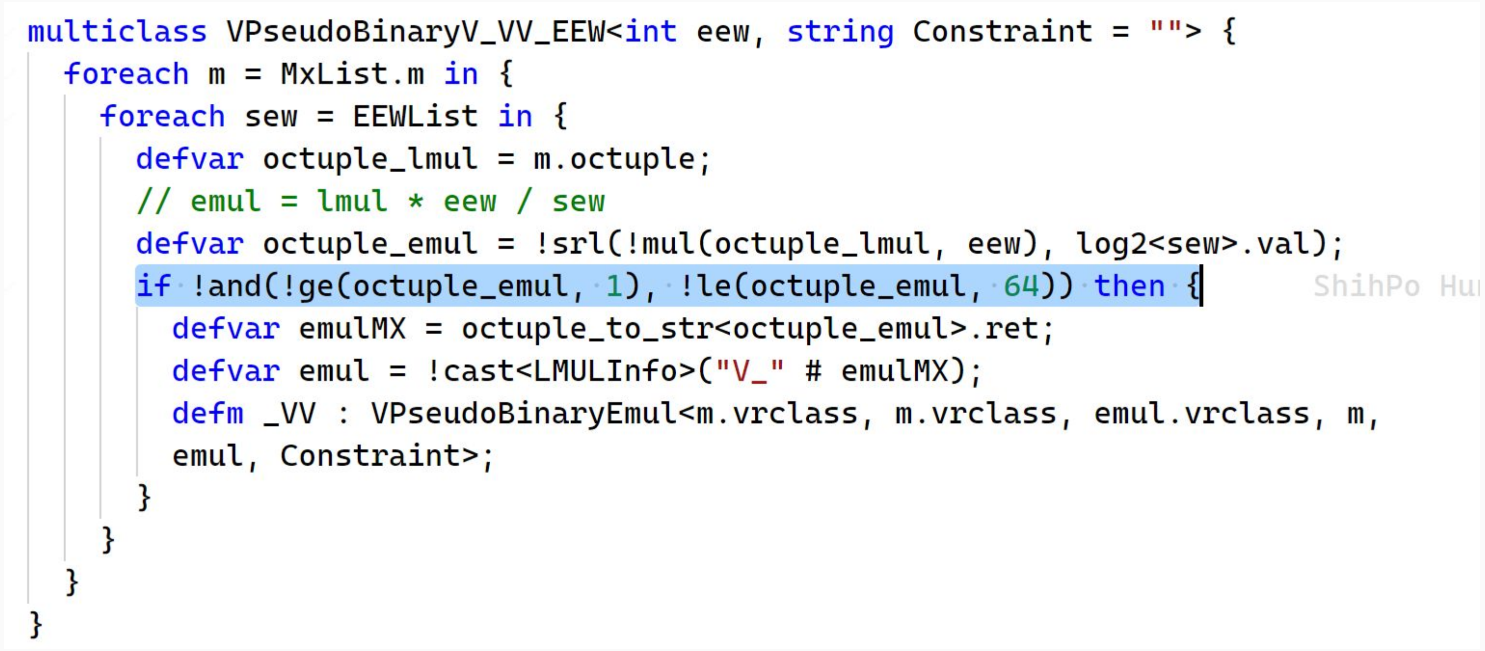
```
------------- Classes -----------------
class Truth<bit Truth:truth = ?> {
  bit truth = 0;
}
```



```
multiclass VPseudoAMOEI<int eew> {
  // Standard scalar AMO supports 32, 64, and 128 Mem data bits,
  // and in the base vector "V" extension, only SEW up to ELEN = max(XLEN, FLEN)
  // are required to be supported.
  // therefore only [32, 64] is allowed here.
  foreach sew = [32, 64] in {
    foreach lmul = MxSet<sew>.m in {
      defvar octuple_lmul = lmul.octuple;
```

TableGen特性和基本概念：语法

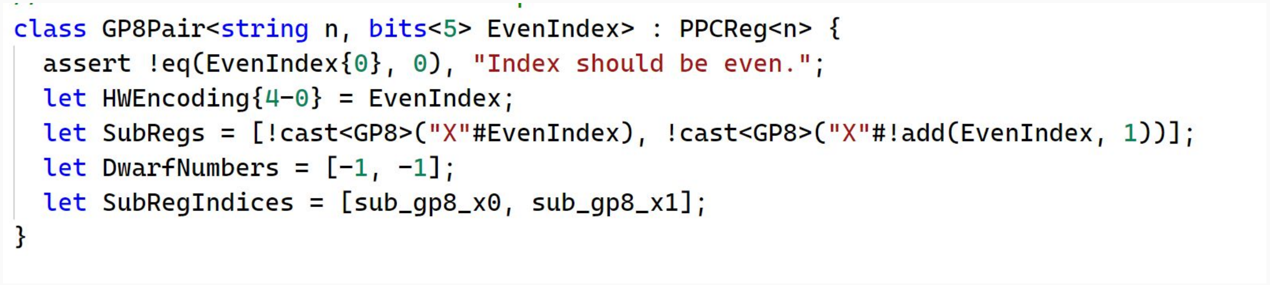
- If…then…else
  - 类似三目运算符，但可以省略else



```
multiclass VPseudoBinaryV_VV_EEW<int eew, string Constraint = ""> {
  foreach m = MxList.m in {
    foreach sew = EEWList in {
      defvar octuple_lmul = m.octuple;
      // emul = lmul * eew / sew
      defvar octuple_emul = !srl(!mul(octuple_lmul, eew), log2<sew>.val);
      if !and(!ge(octuple_emul, 1), !le(octuple_emul, 64)) then {
        defvar emulMX = octuple_to_str<octuple_emul>.ret;
        defvar emul = !cast<LMULInfo>("V_" # emulMX);
        defm _VV : VPseudoBinaryEmul<m.vrclass, m.vrclass, emul.vrclass, m,
        emul, Constraint>;
      }
    }
  }
}
```

TableGen特性和基本概念：语法

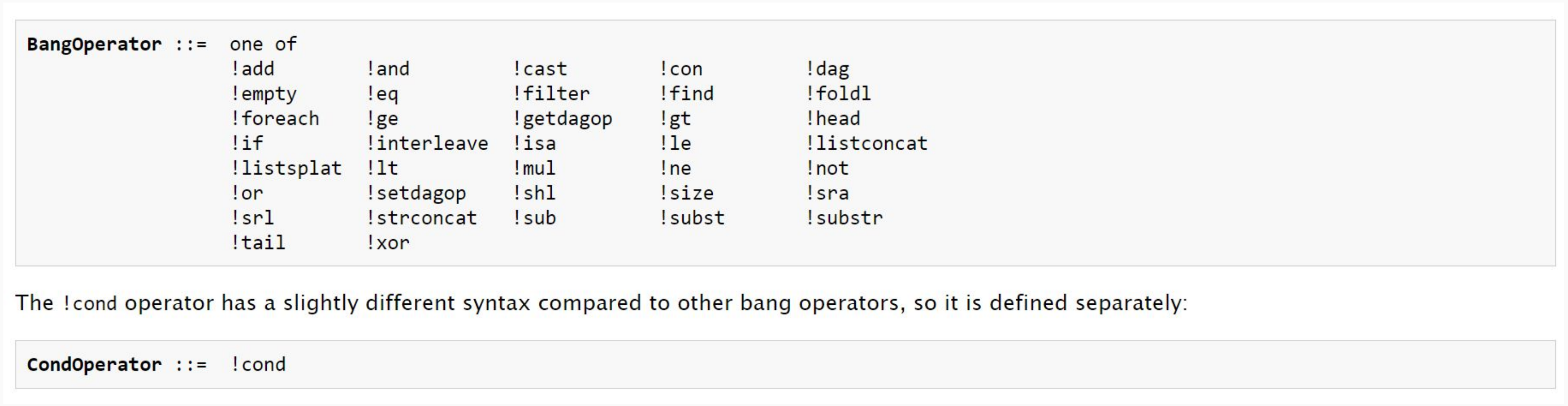
- assert
  - 断言，如果表达式为0则产生一条类似warning的error信息（不影响输出）
  - 在全局层面则立即被检查
  - 在record中，record完全实例化后再检查
  - 在class中，被所有派生类和实例化record继承，record实例化后再检查
  - 在multiclass中，multiclass被实例化的时候检查
  - 在Target中有且仅有PowerPC中一处出现



```
class GP8Pair<string n, bits<5> EvenIndex> : PPCReg<n> {
  assert !eq(EvenIndex{0}, 0), "Index should be even.";
  let HWEncoding{4-0} = EvenIndex;
  let SubRegs = [!cast<GP8>("X"#EvenIndex), !cast<GP8>("X"#!add(EvenIndex, 1))];
  let DwarfNumbers = [-1, -1];
  let SubRegIndices = [sub_gp8_x0, sub_gp8_x1];
}
```

TableGen特性和基本概念：函数

- 内建函数（Bang Operators）
  - 以一个感叹号开头
  - 用户不能自己定义函数



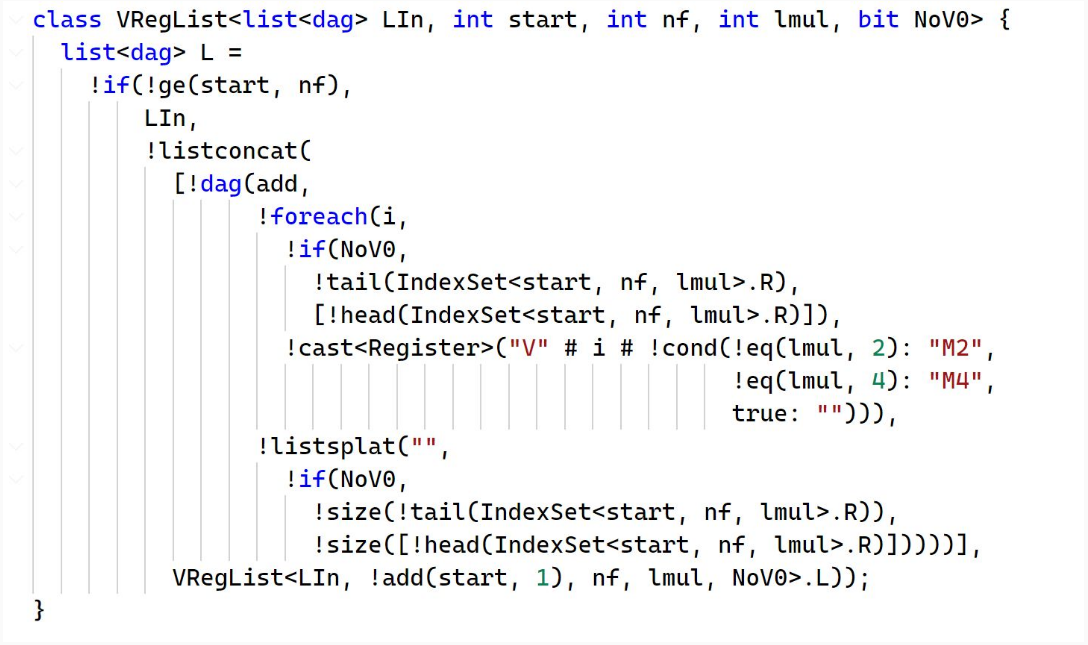
```
BangOperator ::=  one of
                  !add        !and         !cast       !con        !dag
                  !empty      !eq          !filter     !find       !foldl
                  !foreach    !ge          !getdagop   !gt         !head
                  !if         !interleave  !isa        !le         !listconcat
                  !listsplat  !lt          !mul        !ne         !not
                  !or         !setdagop    !shl        !size       !sra
                  !srl        !strconcat   !sub        !subst      !substr
                  !tail       !xor
```

The `!cond` operator has a slightly different syntax compared to other bang operators, so it is defined separately:

```
CondOperator ::=  !cond
```

TableGen特性和基本概念：函数

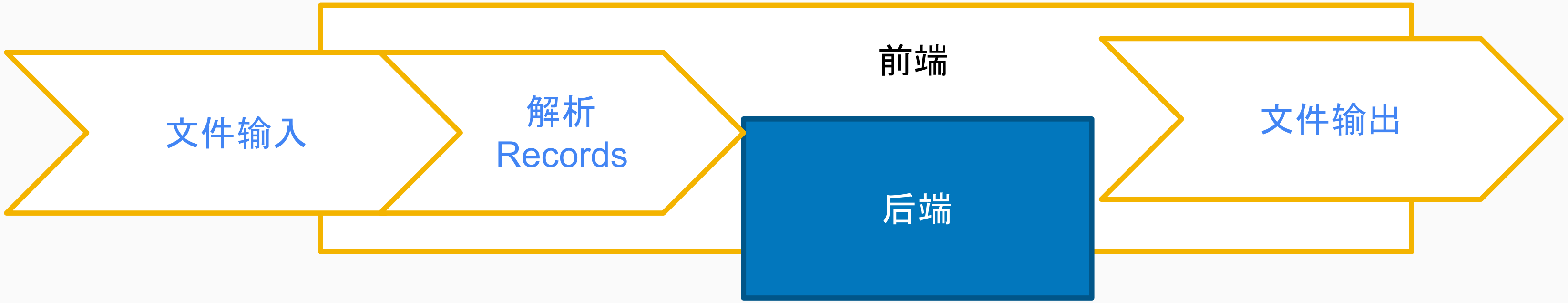
- 对应关键词的区别
  - !dag(op, arguments, names)
  - !foreach(var, sequence, expr)
  - !if(test, then, else)
  - 结合其他operator工作
  - 如：foreach生成多条记录
  - 而!foreach产生一条记录的内容



```
class VRegList<list<dag> LIn, int start, int nf, int lmul, bit NoV0> {
  list<dag> L =
    !if(!ge(start, nf),
        LIn,
        !listconcat(
          [!dag(add,
                !foreach(i,
                  !if(NoV0,
                    !tail(IndexSet<start, nf, lmul>.R),
                    [!head(IndexSet<start, nf, lmul>.R)]),
                  !cast<Register>("V" # i # !cond(!eq(lmul, 2): "M2",
                                                  !eq(lmul, 4): "M4",
                                                  true: ""))),
                !listsplat("",
                  !if(NoV0,
                    !size(!tail(IndexSet<start, nf, lmul>.R)),
                    !size([!head(IndexSet<start, nf, lmul>.R)]))))],
          VRegList<LIn, !add(start, 1), nf, lmul, NoV0>.L));
}
```

TableGen的工具链

- Llvm-tblgen
  - 编译一份llvm代码得到
  - Llvm/build/bin/llvm-tblgen
  - 软件源中的tablegen（如apt安装llvm附带）可能无法正常工作（如无法识别true和false）
  - 前端：llvm/lib/TableGen
  - 后端：llvm/utils/TableGen



TableGen的工具链:前端

- Main处理参数，检验文件合法性，文件目录交给RecordKeeper
- RecordKeeper读入文件
- TGParser解析文件
    - 调用词法分析器Lex对每个Token分情况处理
        - RecordKeeper.addClass(std::unique_ptr<Record> R)
        - RecordKeeper.addDef(std::unique_ptr<Record> R)
        - …
- 将RecordKeeper和输出流交给后端函数，后端解析后返回输出至文件

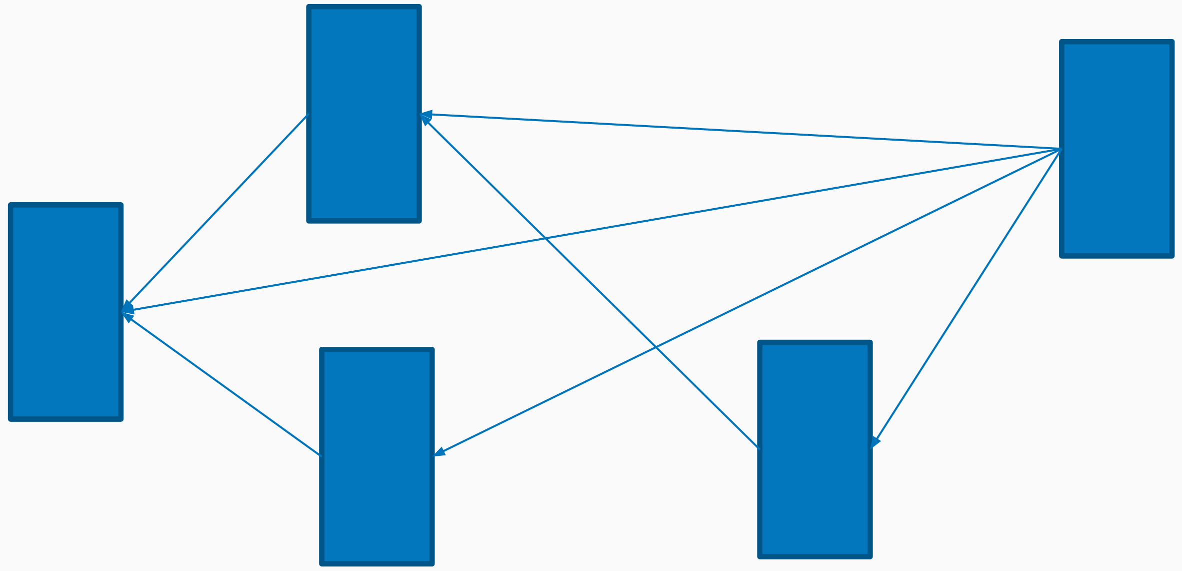
TableGen的工具链:后端

- 不同后端做不同的事
- Llvm-tblgen独有的后端
- 所有TableGen都有的后端
    - -null-backend ：不跑后端不输出任何东西，计时或者检验文件是否合法
    - -dump-json（该后端和前端在同一文件夹）：输出json文件包含所有record
    - -print-records：打印所有records，不加参数默认运行这个
    - …

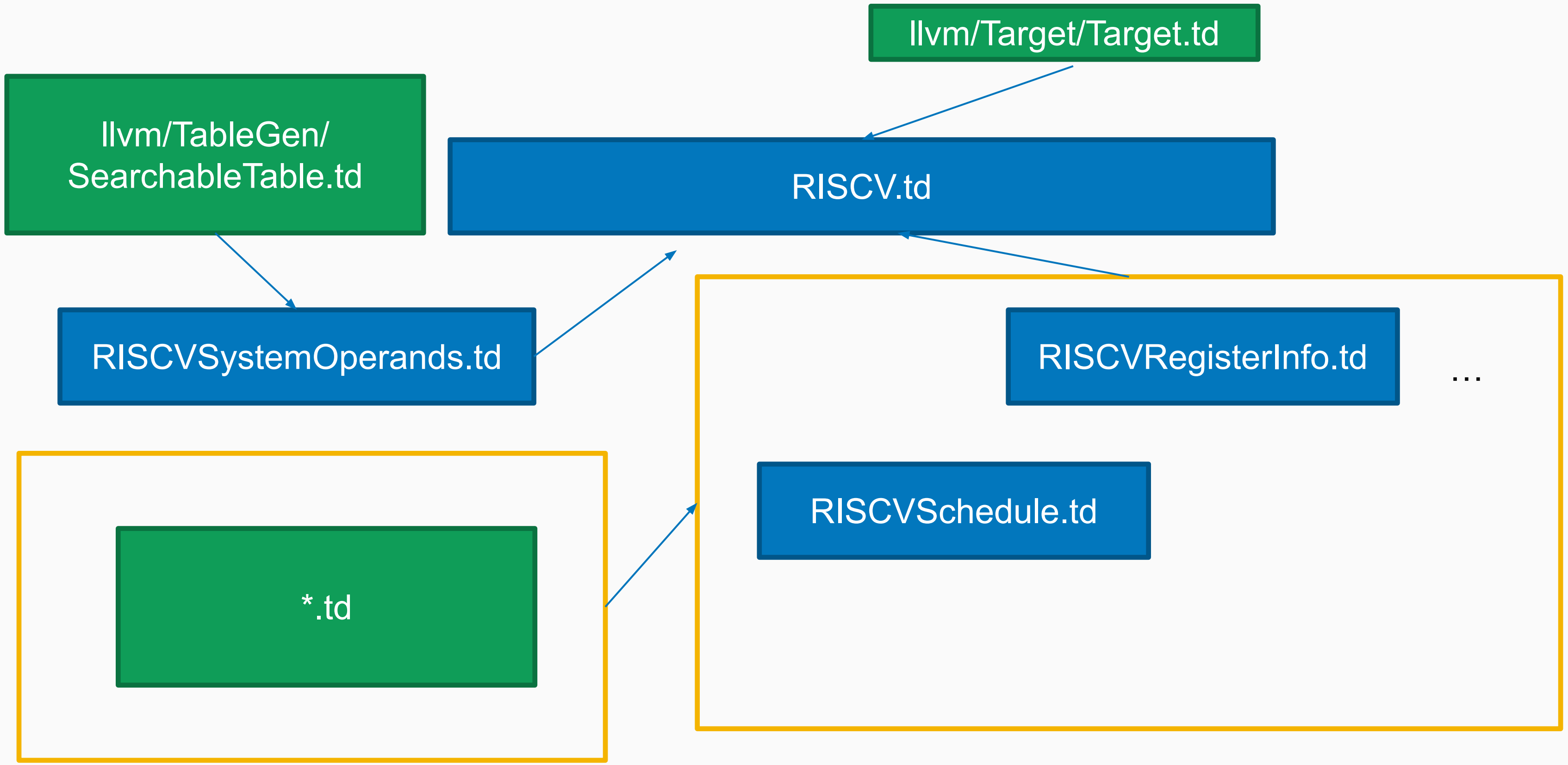
https://llvm.org/docs/CommandGuide/tblgen.html

范例：RISCV后端中的TableGen

- 目标平台文件位置：Llvm/lib/Target/
- 如RISCV/RISCV.td为RISCV架构的顶层文件
- 不同于C++的include结构，没有include guard (#ifdef endif)



范例：RISCV后端中的TableGen：文件架构



范例：RISCV后端中的TableGen：文件架构

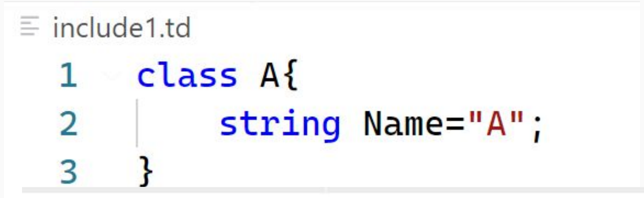
- 在C++中，要用到某个include文件里声明定义的东西，做好guard直接include

（合理调整结构可避免修改一个h重新编译几百个cpp）

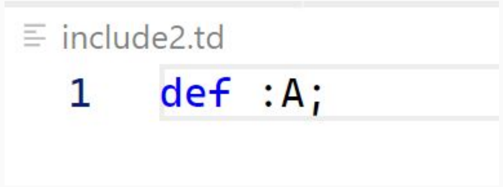
- 在TableGen中

对方文件只会在 该文件被使用:就地 include

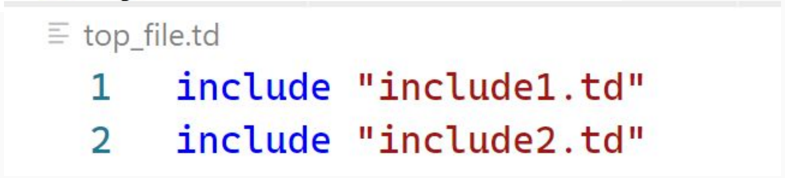
有共同使用文件的模 块:include放在模块的最高共同祖先 处
将include看作是直接把整个文件替 换进入**include声明处**


include1.td

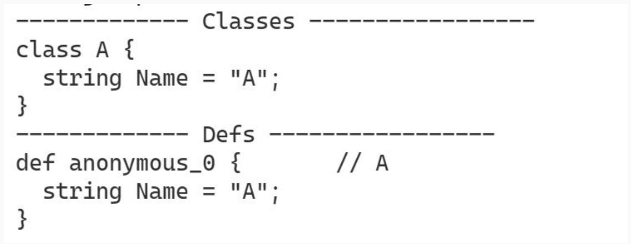
```
class A{
    string Name="A";
}
```


include2.td

```
def :A;
```


top_file.td

```
include "include1.td"
include "include2.td"
```



```
------------- Classes -----------------
class A {
  string Name = "A";
}
------------- Defs -----------------
def anonymous_0 {       // A
  string Name = "A";
}
```

范例：RISCV后端中的TableGen：从td到目标文件

llvm/lib/Target/RISCV/CMakeLists.txt

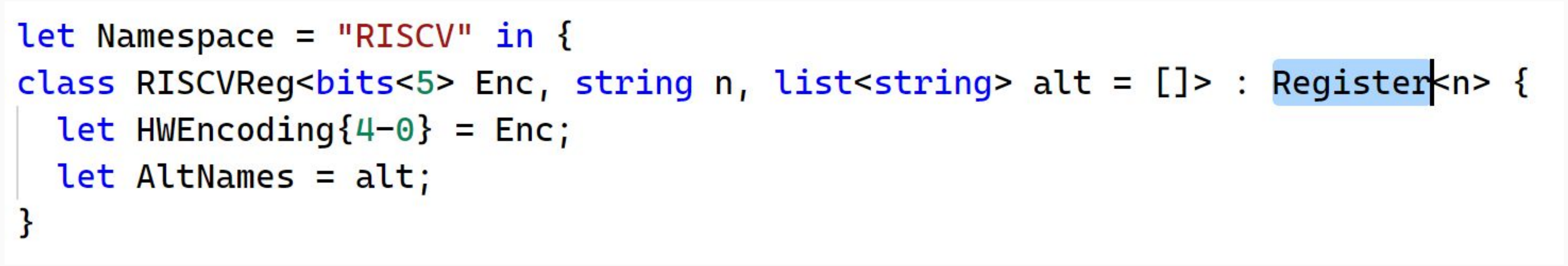
不同的后端

```
tablegen(LLVM RISCVGenAsmMatcher.inc -gen-asm-matcher)
tablegen(LLVM RISCVGenAsmWriter.inc -gen-asm-writer)
tablegen(LLVM RISCVGenCompressInstEmitter.inc -gen-compress-inst-emitter)
tablegen(LLVM RISCVGenDAGISel.inc -gen-dag-isel)
tablegen(LLVM RISCVGenDisassemblerTables.inc -gen-disassembler)
tablegen(LLVM RISCVGenGlobalISel.inc -gen-global-isel)
tablegen(LLVM RISCVGenInstrInfo.inc -gen-instr-info)
tablegen(LLVM RISCVGenMCCodeEmitter.inc -gen-emitter)
tablegen(LLVM RISCVGenMCPseudoLowering.inc -gen-pseudo-lowering)
tablegen(LLVM RISCVGenRegisterBank.inc -gen-register-bank)
tablegen(LLVM RISCVGenRegisterInfo.inc -gen-register-info)
tablegen(LLVM RISCVGenSearchableTables.inc -gen-searchable-tables)
tablegen(LLVM RISCVGenSubtargetInfo.inc -gen-subtarget)
```

范例：RISCV后端中的TableGen：从td到目标文件

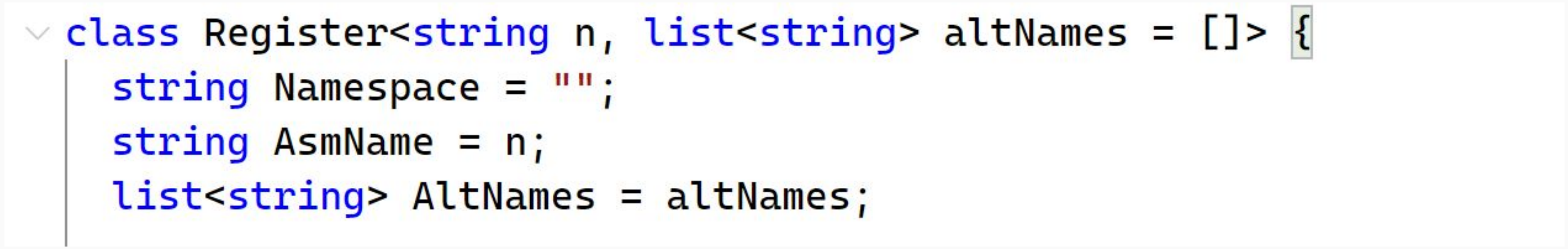
- 所有内容都在顶层模块里，后端怎么知道哪部分用于对应生成？

llvm/lib/Target/RISCV/RISCVRegisterInfo.td



```
let Namespace = "RISCV" in {
class RISCVReg<bits<5> Enc, string n, list<string> alt = []> : Register<n> {
  let HWEncoding{4-0} = Enc;
  let AltNames = alt;
}
```

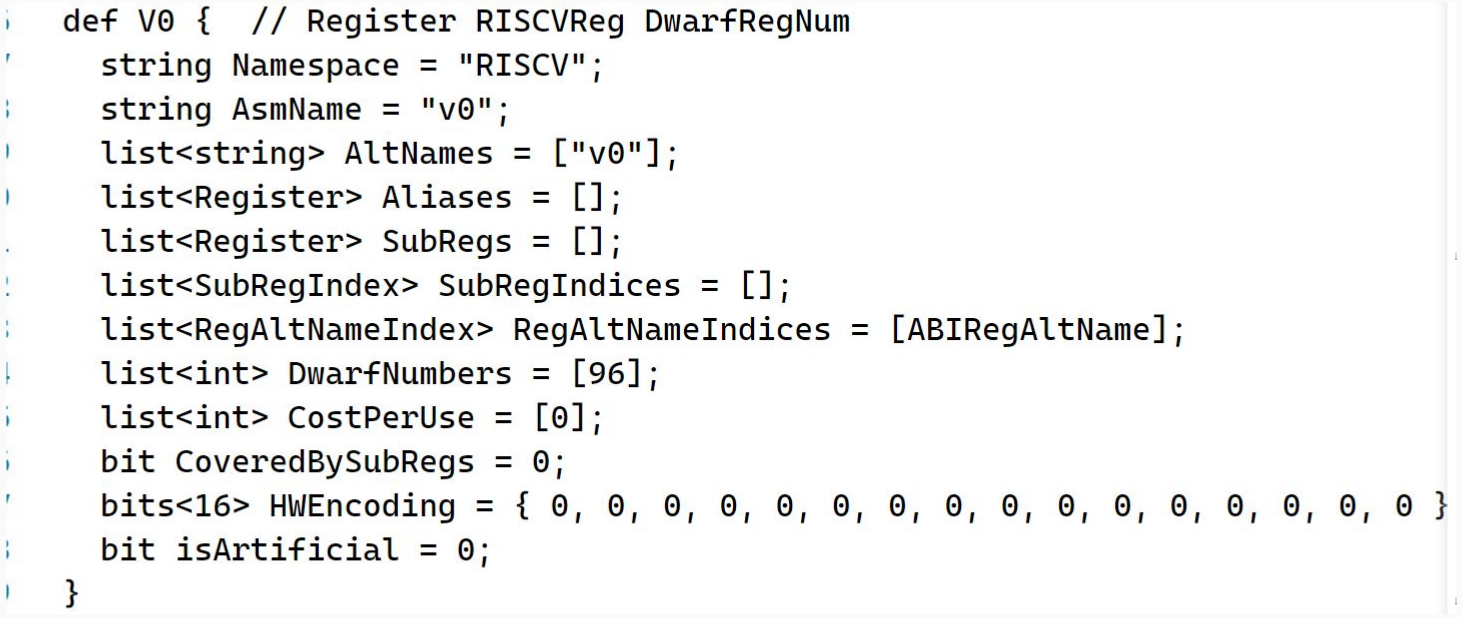
llvm/include/llvm/Target/Target.td



```
class Register<string n, list<string> altNames = []> {
  string Namespace = "";
  string AsmName = n;
  list<string> AltNames = altNames;
```

范例：RISCV后端中的TableGen：从td到目标文件，寻找匹配的后端

前端解析出的原始record



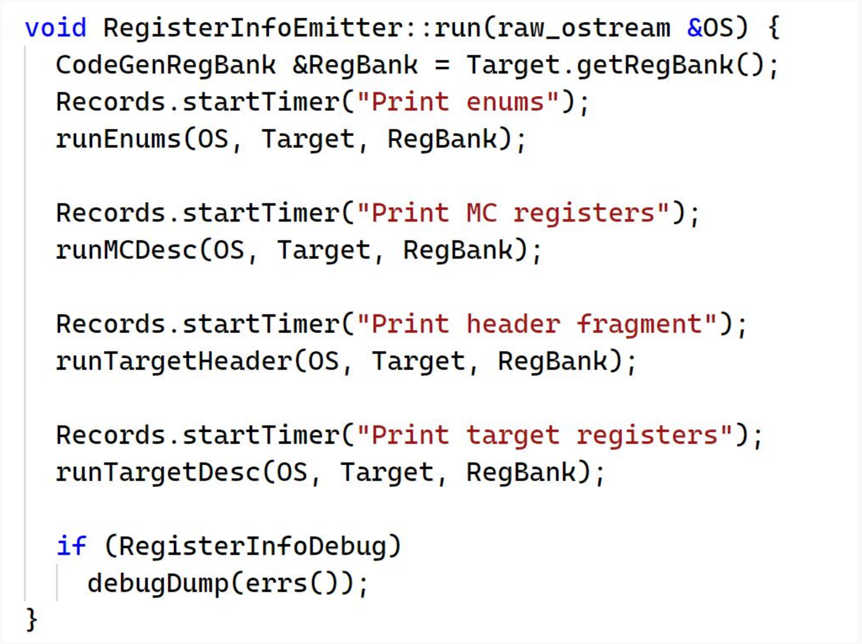
```
def V0 {  // Register RISCVReg DwarfRegNum
  string Namespace = "RISCV";
  string AsmName = "v0";
  list<string> AltNames = ["v0"];
  list<Register> Aliases = [];
  list<Register> SubRegs = [];
  list<SubRegIndex> SubRegIndices = [];
  list<RegAltNameIndex> RegAltNameIndices = [ABIRegAltName];
  list<int> DwarfNumbers = [96];
  list<int> CostPerUse = [0];
  bit CoveredBySubRegs = 0;
  bits<16> HWEncoding = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
  bit isArtificial = 0;
}
```

范例：RISCV后端中的TableGen：从td到目标文件，寻找匹配的后端并送入

后端会将records通过bool isSubClassOf(StringRef Name) const

判断是否为某个类的子类（此处为Register基类）

同一类型的records被送入对应后端并生成对应内容



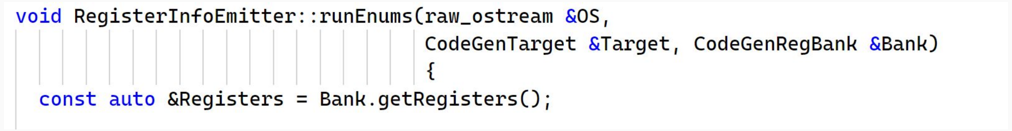
```
void RegisterInfoEmitter::run(raw_ostream &OS) {
  CodeGenRegBank &RegBank = Target.getRegBank();
  Records.startTimer("Print enums");
  runEnums(OS, Target, RegBank);

  Records.startTimer("Print MC registers");
  runMCDesc(OS, Target, RegBank);

  Records.startTimer("Print header fragment");
  runTargetHeader(OS, Target, RegBank);

  Records.startTimer("Print target registers");
  runTargetDesc(OS, Target, RegBank);

  if (RegisterInfoDebug)
    debugDump(errs());
}
```

范例：RISCV后端中的TableGen：从td到目标文件，寻找匹配的后端

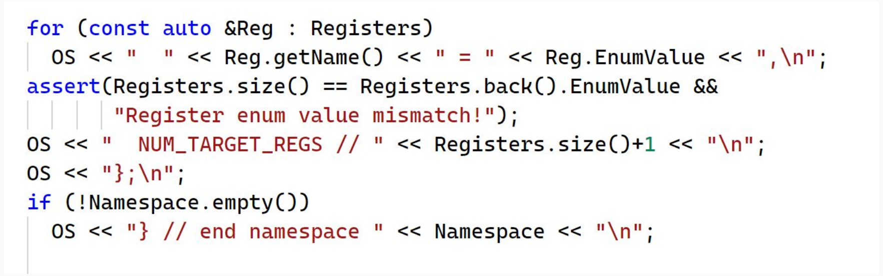
后端会将records通过bool isSubClassOf(StringRef Name) const

判断是否为某个类的子类（此处为Register基类）

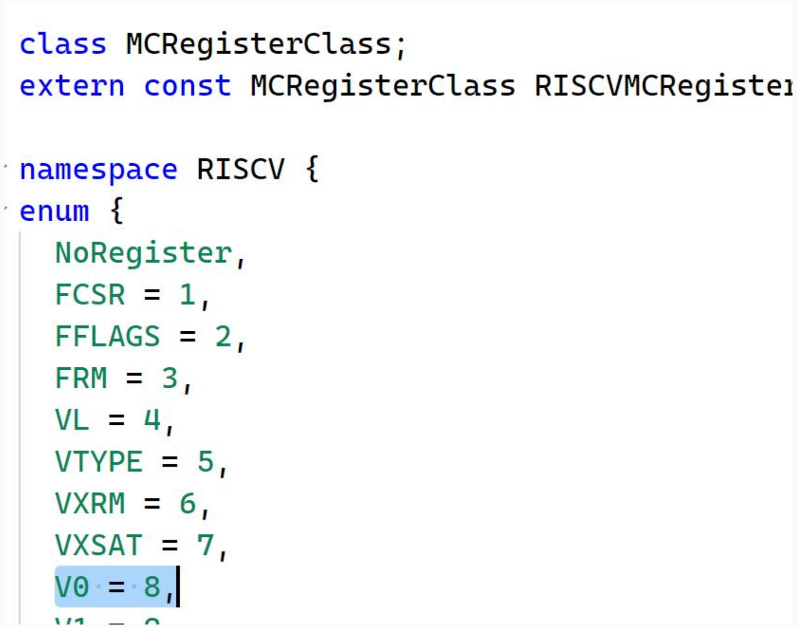
同一类型的records被送入对应后端并生成对应内容



```
void RegisterInfoEmitter::runEnums(raw_ostream &OS,
                                  CodeGenTarget &Target, CodeGenRegBank &Bank)
                                  {
  const auto &Registers = Bank.getRegisters();
```



```
for (const auto &Reg : Registers)
  OS << "  " << Reg.getName() << " = " << Reg.EnumValue << ",\n";
assert(Registers.size() == Registers.back().EnumValue &&
       "Register enum value mismatch!");
OS << "  NUM_TARGET_REGS // " << Registers.size()+1 << "\n";
OS << "};\n";
if (!Namespace.empty())
  OS << "} // end namespace " << Namespace << "\n";
```



```
class MCRegisterClass;
extern const MCRegisterClass RISCVMCRegiste

namespace RISCV {
enum {
  NoRegister,
  FCSR = 1,
  FFLAGS = 2,
  FRM = 3,
  VL = 4,
  VTYPE = 5,
  VXRM = 6,
  VXSAT = 7,
  V0 = 8,
```

思考与总结

- 强大的功能，减少繁重工作量
  - 可通过嵌套multiclass或者foreach数行代码生成大量同类
  - 甚至可通过内置函数实现函数式编程
- 对开发者不够友善
  - 只有简单的高亮支持
  - Include方式难以快速找到上级定义
  - 复杂的多层嵌套容易出错
- 混乱的命名：
  - Class，又名Abstract Record；Record，又名Concrete Record，或是Definition
  - Record有时指Concrete Record，有时包括Class

参考资料

- LLVM Documents

https://llvm.org/docs/GettingStarted.html
https://llvm.org/docs/TableGen/index.html
https://llvm.org/docs/CommandGuide/tblgen.html
https://llvm.org/docs/TableGen/BackEnds.html
https://llvm.org/docs/TableGen/BackGuide.html
https://llvm.org/docs/TableGen/ProgRef.html

- Blogs

Tagebuch eines Interplanetaren Botschafters: TableGen Series

- Slides

LESSONS IN TABLEGEN by NICOLAI HÄHNLE
Building an LLVM Backend by Codeplay Software

- Books

Howto: Implementing LLVM Integrated Assembler
The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks.

感谢观看，
烦请批评指
正。