TableGen Language Reference ¶

Section author: Sean Silva <silvas@purdue.edu>

Warning

This document is extremely rough. If you find something lacking, please fix it, file a documentation bug, or ask about it on llvmdev.

Notation

The lexical and syntax notation used here is intended to imitate Python's. In particular, for lexical definitions, the productions operate at the character level and there is no implied whitespace between elements. The syntax definitions operate at the token level, so there is implied whitespace between tokens.

Lexical Analysis

TableGen supports BCPL (`// ...`) and nestable C-style (`/* ... */`) comments.

The following is a listing of the basic punctuation tokens:

```
- + [ ] { } ( ) < > : ; .  = ? #
```

Numeric literals take one of the following forms:

```
TokInteger     ::=  DecimalInteger | HexInteger | BinInteger
DecimalInteger ::=  ["+" | "-"] ("0"..."9")+
HexInteger     ::=  "0x" ("0"..."9" | "a"..."f" | "A"..."F")+
BinInteger     ::=  "0b" ("0" | "1")+
```

One aspect to note is that the DecimalInteger token *includes* the + or -, as opposed to having + and - be unary operators as most languages do.

TableGen has identifier-like tokens:

```
ualpha        ::=  "a"..."z" | "A"..."Z" | "_"
TokIdentifier ::=  ("0"..."9")* ualpha (ualpha | "0"..."9")*
TokVarName    ::=  "$" ualpha (ualpha |  "0"..."9")*
```

Note that unlike most languages, TableGen allows TokIdentifier to begin with a number. In case of ambiguity, a token will be interpreted as a numeric literal rather than an identifier.

TableGen also has two string-like literals:

```
TokString       ::=  '"' <non-'"' characters and C-like escapes> '"'
TokCodeFragment ::=  "[{" <shortest text not containing "}]"> "}]"
```

TableGen also has the following keywords:

```
bit   bits      class   code         dag
def   foreach   defm    field        in
int   let       list    multiclass   string
```

TableGen also has “bang operators” which have a wide variety of meanings:

```
!eq     !if      !head    !tail      !con
!shl    !sra     !srl
!cast   !empty   !subst   !foreach   !strconcat
```

Syntax

TableGen has an `include` mechanism. It does not play a role in the syntax per se, since it is lexically replaced with the contents of the included file.

```
IncludeDirective ::=  "include" TokString
```

TableGen’s top-level production consists of “objects”.

```
TableGenFile ::=  Object*
Object       ::=  Class | Def | Defm | Let | MultiClass | Foreach
```

`class`es

```
Class ::=  "class" TokIdentifier [TemplateArgList] ObjectBody
```

A `class` declaration creates a record which other records can inherit from. A class can be parametrized by a list of “template arguments”, whose values can be used in the class body.

A given class can only be defined once. A `class` declaration is considered to define the class if one of the following is true:

1. The TemplateArgList is present.
2. The Body in the ObjectBody is present and is not empty.

3. The `BaseClassList` in the `ObjectBody` is present.

You can declare an empty class by giving and empty `TemplateArgList` and an empty `ObjectBody`. This can serve as a restricted form of forward declaration: note that records deriving from the forward-declared class will inherit no fields from it since the record expansion is done when the record is parsed.

```
TemplateArgList ::=  "<" Declaration ("," Declaration)* ">"
```

Declarations

The declaration syntax is pretty much what you would expect as a C++ programmer.

```
Declaration ::=  Type TokIdentifier ["=" Value]
```

It assigns the value to the identifer.

Types

```
Type    ::=  "string" | "code" | "bit" | "int" | "dag"
           | "bits" "<" TokInteger ">"
           | "list" "<" Type ">"
           | ClassID
ClassID ::=  TokIdentifier
```

Both `string` and `code` correspond to the string type; the difference is purely to indicate programmer intention.

The `ClassID` must identify a class that has been previously declared or defined.

Values

```
Value       ::=  SimpleValue ValueSuffix*
ValueSuffix ::=  "{" RangeList "}"
               | "[" RangeList "]"
               | "." TokIdentifier
RangeList   ::=  RangePiece ("," RangePiece)*
RangePiece  ::=  TokInteger
               | TokInteger "-" TokInteger
               | TokInteger TokInteger
```

The peculiar last form of `RangePiece` is due to the fact that the `-` is included in the `TokInteger`, hence `1-5` gets lexed as two consecutive `TokInteger`'s, with values `1` and `-5`, instead of "1", "-", and "5". The `RangeList` can be thought of as specifying "list slice" in some contexts.

`SimpleValue` has a number of forms:

```
SimpleValue ::=  TokIdentifier
```

The value will be the variable referenced by the identifier. It can be one of:

- name of a `def`, such as the use of `Bar` in:

```
def Bar : SomeClass {
  int X = 5;
}
```

```
def Foo {
  SomeClass Baz = Bar;
}
```

- value local to a `def`, such as the use of `Bar` in:

```
def Foo {
  int Bar = 5;
  int Baz = Bar;
}
```

- a template arg of a `class`, such as the use of `Bar` in:

```
class Foo<int Bar> {
  int Baz = Bar;
}
```

- value local to a `multiclass`, such as the use of `Bar` in:

```
multiclass Foo {
  int Bar = 5;
  int Baz = Bar;
}
```

- a template arg to a `multiclass`, such as the use of `Bar` in:

```
multiclass Foo<int Bar> {
  int Baz = Bar;
}
```

```
SimpleValue ::=  TokInteger
```

This represents the numeric value of the integer.

```
SimpleValue ::=  TokString+
```

Multiple adjacent string literals are concatenated like in C/C++. The value is the concatenation of the strings.

```
SimpleValue ::=  TokCodeFragment
```

The value is the string value of the code fragment.

```
SimpleValue ::=  "?"
```

`?` represents an “unset” initializer.

```
SimpleValue ::=  "{" ValueList "}"
ValueList   ::=  [ValueListNE]
ValueListNE ::=  Value ("," Value)*
```

This represents a sequence of bits, as would be used to initialize a `bits<n>` field (where `n` is the number of bits).

```
SimpleValue ::=  ClassID "<" ValueListNE ">"
```

This generates a new anonymous record definition (as would be created by an unnamed `def` inheriting from the given class with the given template arguments) and the value is the value of that record definition.

```
SimpleValue ::=  "[" ValueList "]" ["<" Type ">"]
```

A list initializer. The optional `Type` can be used to indicate a specific element type, otherwise the element type will be deduced from the given values.

```
SimpleValue ::=  "(" DagArg DagArgList ")"
DagArgList  ::=  DagArg ("," DagArg)*
DagArg      ::=  Value [":" TokVarName]
```

The initial `DagArg` is called the "operator" of the dag.

```
SimpleValue ::=  BangOperator ["<" Type ">"] "(" ValueListNE ")"
```

Bodies

```
ObjectBody      ::=  BaseClassList Body
BaseClassList   ::=  [BaseClassListNE]
BaseClassListNE ::=  SubClassRef ("," SubClassRef)*
SubClassRef     ::=  (ClassID | DefmID) ["<" ValueList ">"]
DefmID          ::=  TokIdentifier
```

The version with the `DefmID` is only valid in the `BaseClassList` of a `defm`. The `DefmID` should be the name of a `multiclass`.

It is after parsing the base class list that the "let stack" is applied.

```
Body     ::=  ";" | "{" BodyList "}"
BodyList ::=  BodyItem*
BodyItem ::=  Declaration ";"
            | "let" TokIdentifier [RangeList] "=" Value ";"
```

The `let` form allows overriding the value of an inherited field.

def

```
Def ::=  "def" TokIdentifier ObjectBody
```

Defines a record whose name is given by the `TokIdentifier`. The fields of the record are inherited from the base classes and defined in the body.

Special handling occurs if this `def` appears inside a `multiclass` or a `foreach`.

defm

```
Defm ::=  "defm" TokIdentifier ":" BaseClassList ";"
```

Note that in the `BaseClassList`, all of the `multiclass`'s must precede any `class`'s that appear.

foreach

```
Foreach ::=  "foreach" Declaration "in" "{" Object* "}"
           | "foreach" Declaration "in" Object
```

The value assigned to the variable in the declaration is iterated over and the object or object list is reevaluated with the variable set at each iterated value.

Top-Level `let`

```
Let     ::=   "let" LetList "in" "{" Object* "}"
            | "let" LetList "in" Object
LetList ::=  LetItem ("," LetItem)*
LetItem ::=  TokIdentifier [RangeList] "=" Value
```

This is effectively equivalent to `let` inside the body of a record except that it applies to multiple records at a time. The bindings are applied at the end of parsing the base classes of a record.

`multiclass`

```
MultiClass         ::=  "multiclass" TokIdentifier [TemplateArgList]
                        [":" BaseMultiClassList] "{" MultiClassDef+ "}"
BaseMultiClassList ::=  MultiClassID ("," MultiClassID)*
MultiClassID       ::=  TokIdentifier
```