NAME

FileCheck - Flexible pattern matching file verifier

SYNOPSIS

FileCheck *match-filename* [*--check-prefix=XXX*] [*--strict-whitespace*]

DESCRIPTION

FileCheck reads two files (one from standard input, and one specified on the command line) and uses one to verify the other. This behavior is particularly useful for the testsuite, which wants to verify that the output of some tool (e.g. **llc**) contains the expected information (for example, a movsd from esp or whatever is interesting). This is similar to using **grep**, but it is optimized for matching multiple different inputs in one file in a specific order.

The **match-filename** file specifies the file that contains the patterns to match. The file to verify is read from standard input unless the *--input-file* option is used.

OPTIONS

Options are parsed from the environment variable **FILECHECK_OPTS** and from the command line.

-help

Print a summary of command line options.

--check-prefix prefix

FileCheck searches the contents of **match-filename** for patterns to match. By default, these patterns are prefixed with "**CHECK:**". If you'd like to use a different prefix (e.g. because the same input file is checking multiple different tool or options), the *--check-prefix* argument allows you to specify (without the trailing "**:**") one or more prefixes to match. Multiple prefixes are useful for tests which might change for different run options, but most lines remain the same.

FileCheck does not permit duplicate prefixes, even if one is a check prefix and one is a comment prefix (see *--comment-prefixes* below).

--check-prefixes prefix1,prefix2,...

An alias of *--check-prefix* that allows multiple prefixes to be specified as a comma separated list.

--comment-prefixes prefix1,prefix2,...

By default, FileCheck ignores any occurrence in **match-filename** of any check prefix if it is preceded on the same line by "**COM:**" or "**RUN:**". See the section *The "COM:" directive* for usage details.

These default comment prefixes can be overridden by *--comment-prefixes* if they are not appropriate for your testing environment. However, doing so is not recommended in LLVM's LIT-based test suites, which should be easier to maintain if they all follow a consistent comment style. In that case, consider proposing a change to the default comment prefixes instead.

--allow-unused-prefixes

This option controls the behavior when using more than one prefix as specified by *--check-prefix* or *--check-prefixes*, and some of these prefixes are missing in the test file. If true, this is allowed, if false, FileCheck will report an error, listing the missing prefixes. The default value is false.

--input-file filename

File to check (defaults to stdin).

--match-full-lines

By default, FileCheck allows matches of anywhere on a line. This option will require all positive matches to cover an entire line. Leading and trailing whitespace is ignored, unless *--strict-whitespace* is also specified. (Note: negative matches from **CHECK-NOT** are not affected by this option!)

Passing this option is equivalent to inserting **{{^ *}}** or **{{^}}** before, and **{{ *$}}** or **{{$}}** after every positive check pattern.

--strict-whitespace

By default, FileCheck canonicalizes input horizontal whitespace (spaces and tabs) which causes it to ignore these differences (a space will match a tab). The *--strict-whitespace* argument disables this behavior. End-of-line sequences are canonicalized to UNIX-style **\n** in all modes.

--ignore-case

By default, FileCheck uses case-sensitive matching. This option causes FileCheck to use case-insensitive matching.

--implicit-check-not check-pattern

Adds implicit negative checks for the specified patterns between positive checks. The option allows writing stricter tests without stuffing them with **CHECK-NOT**s.

For example, "**--implicit-check-not warning:**" can be useful when testing diagnostic messages from tools that don't have an option similar to **clang -verify**. With this option FileCheck will verify that input does not contain warnings not covered by any **CHECK:** patterns.

--dump-input <value>

Dump input to stderr, adding annotations representing currently enabled diagnostics. When there are multiple occurrences of this option, the **<value>** that appears earliest in the list below has precedence. The default is **fail**.

- **help** - Explain input dump and quit
- **always** - Always dump input
- **fail** - Dump input on failure
- **never** - Never dump input

--dump-input-context <N>

In the dump requested by **--dump-input**, print **<N>** input lines before and **<N>** input lines after any lines specified by **--dump-input-filter**. When there are multiple occurrences of this option, the largest specified **<N>** has precedence. The default is 5.

--dump-input-filter <value>

In the dump requested by **--dump-input**, print only input lines of kind **<value>** plus any context specified by **--dump-input-context**. When there are multiple occurrences of this option, the **<value>** that appears earliest in the list below has precedence. The default is **error** when **--dump-input=fail**, and it's **all** when **--dump-input=always**.

- **all** - All input lines
- **annotation-full** - Input lines with annotations
- **annotation** - Input lines with starting points of annotations
- **error** - Input lines with starting points of error annotations

--enable-var-scope

Enables scope for regex variables.

Variables with names that start with **$** are considered global and remain set throughout the file.

All other variables get undefined after each encountered **CHECK-LABEL**.

-D<VAR=VALUE>

Sets a filecheck pattern variable **VAR** with value **VALUE** that can be used in **CHECK:** lines.

-D#<FMT>,<NUMVAR>=<NUMERIC EXPRESSION>
Sets a filecheck numeric variable **NUMVAR** of matching format **FMT** to the result of evaluating **<NUMERIC EXPRESSION>** that can be used in **CHECK:** lines. See section **FileCheck Numeric Variables and Expressions** for details on supported numeric expressions.

-version
Show the version number of this program.

-v Print good directive pattern matches. However, if **-dump-input=fail** or **-dump-input=always**, add those matches as input annotations instead.

-vv Print information helpful in diagnosing internal FileCheck issues, such as discarded overlapping **CHECK-DAG:** matches, implicit EOF pattern matches, and **CHECK-NOT:** patterns that do not have matches. Implies **-v**. However, if **-dump-input=fail** or **-dump-input=always**, just add that information as input annotations instead.

--allow-deprecated-dag-overlap
Enable overlapping among matches in a group of consecutive **CHECK-DAG:** directives. This option is deprecated and is only provided for convenience as old tests are migrated to the new non-overlapping **CHECK-DAG:** implementation.

--allow-empty
Allow checking empty input. By default, empty input is rejected.

--color
Use colors in output (autodetected by default).

EXIT STATUS

If **FileCheck** verifies that the file matches the expected contents, it exits with 0. Otherwise, if not, or if an error occurs, it will exit with a non-zero value.

TUTORIAL

FileCheck is typically used from LLVM regression tests, being invoked on the RUN line of the test. A simple example of using FileCheck from a RUN line looks like this:

```
; RUN: llvm-as < %s | llc -march=x86-64 | FileCheck %s
```

This syntax says to pipe the current file ("**%s**") into **llvm-as**, pipe that into **llc**, then pipe the output of **llc** into **FileCheck**. This means that FileCheck will be verifying its standard input (the llc output) against the filename argument specified (the original **.ll** file specified by "**%s**"). To

see how this works, let's look at the rest of the **.ll** file (after the RUN line):

```
define void @sub1(i32* %p, i32 %v) {
entry:
; CHECK: sub1:
; CHECK: subl
        %0 = tail call i32 @llvm.atomic.load.sub.i32.p0i32(i32* %p, i32 %v)
        ret void
}

define void @inc4(i64* %p) {
entry:
; CHECK: inc4:
; CHECK: incq
        %0 = tail call i64 @llvm.atomic.load.add.i64.p0i64(i64* %p, i64 1)
        ret void
}
```

Here you can see some "**CHECK:**" lines specified in comments. Now you can see how the file is piped into **llvm-as**, then **llc**, and the machine code output is what we are verifying. FileCheck checks the machine code output to verify that it matches what the "**CHECK:**" lines specify.

The syntax of the "**CHECK:**" lines is very simple: they are fixed strings that must occur in order. FileCheck defaults to ignoring horizontal whitespace differences (e.g. a space is allowed to match a tab) but otherwise, the contents of the "**CHECK:**" line is required to match some thing in the test file exactly.

One nice thing about FileCheck (compared to grep) is that it allows merging test cases together into logical groups. For example, because the test above is checking for the "**sub1:**" and "**inc4:**" labels, it will not match unless there is a "**subl**" in between those labels. If it existed somewhere else in the file, that would not count: "**grep subl**" matches if "**subl**" exists anywhere in the file.

The FileCheck -check-prefix option

The FileCheck *-check-prefix* option allows multiple test configurations to be driven from one *.ll* file. This is useful in many circumstances, for example, testing different architectural variants with **llc**. Here's a simple example:

```
; RUN: llvm-as < %s | llc -mtriple=i686-apple-darwin9 -mattr=sse41 \
; RUN:              | FileCheck %s -check-prefix=X32
; RUN: llvm-as < %s | llc -mtriple=x86_64-apple-darwin9 -mattr=sse41 \
```

```
; RUN:            | FileCheck %s -check-prefix=X64

define <4 x i32> @pinsrd_1(i32 %s, <4 x i32> %tmp) nounwind {
      %tmp1 = insertelement <4 x i32>; %tmp, i32 %s, i32 1
      ret <4 x i32> %tmp1
; X32: pinsrd_1:
; X32:    pinsrd $1, 4(%esp), %xmm0

; X64: pinsrd_1:
; X64:    pinsrd $1, %edi, %xmm0
}
```

In this case, we're testing that we get the expected code generation with both 32-bit and 64-bit code generation.

The "COM:" directive

Sometimes you want to disable a FileCheck directive without removing it entirely, or you want to write comments that mention a directive by name. The "**COM:**" directive makes it easy to do this. For example, you might have:

```
; X32: pinsrd_1:
; X32:    pinsrd $1, 4(%esp), %xmm0

; COM: FIXME: X64 isn't working correctly yet for this part of codegen, but
; COM: X64 will have something similar to X32:
; COM:
; COM:   X64: pinsrd_1:
; COM:   X64:    pinsrd $1, %edi, %xmm0
```

Without "**COM:**", you would need to use some combination of rewording and directive syntax mangling to prevent FileCheck from recognizing the commented occurrences of "**X32:**" and "**X64:**" above as directives. Moreover, FileCheck diagnostics have been proposed that might complain about the above occurrences of "**X64**" that don't have the trailing "**:**" because they look like directive typos. Dodging all these problems can be tedious for a test author, and directive syntax mangling can make the purpose of test code unclear. "**COM:**" avoids all these problems.

A few important usage notes:

- "**COM:**" within another directive's pattern does *not* comment out the remainder of the pattern. For

example:

```
; X32: pinsrd $1, 4(%esp), %xmm0 COM: This is part of the X32 pattern!
```

If you need to temporarily comment out part of a directive's pattern, move it to another line. The reason is that FileCheck parses "**COM:**" in the same manner as any other directive: only the first directive on the line is recognized as a directive.

- For the sake of LIT, FileCheck treats "**RUN:**" just like "**COM:**". If this is not suitable for your test environment, see *--comment-prefixes*.

- FileCheck does not recognize "**COM**", "**RUN**", or any user-defined comment prefix as a comment directive if it's combined with one of the usual check directive suffixes, such as "**-NEXT:**" or "**-NOT:**", discussed below. FileCheck treats such a combination as plain text instead. If it needs to act as a comment directive for your test environment, define it as such with *--comment-prefixes*.

The "CHECK-NEXT:" directive

Sometimes you want to match lines and would like to verify that matches happen on exactly consecutive lines with no other lines in between them. In this case, you can use "**CHECK:**" and "**CHECK-NEXT:**" directives to specify this. If you specified a custom check prefix, just use "**<PREFIX>-NEXT:**". For example, something like this works as you'd expect:

```
define void @t2(<2 x double>* %r, <2 x double>* %A, double %B) {
    %tmp3 = load <2 x double>* %A, align 16
    %tmp7 = insertelement <2 x double> undef, double %B, i32 0
    %tmp9 = shufflevector <2 x double> %tmp3,
                        <2 x double> %tmp7,
                        <2 x i32> < i32 0, i32 2 >
    store <2 x double> %tmp9, <2 x double>* %r, align 16
    ret void

; CHECK:          t2:
; CHECK:            movl    8(%esp), %eax
; CHECK-NEXT:       movapd  (%eax), %xmm0
; CHECK-NEXT:       movhpd  12(%esp), %xmm0
; CHECK-NEXT:       movl    4(%esp), %eax
; CHECK-NEXT:       movapd  %xmm0, (%eax)
; CHECK-NEXT:       ret
}
```

"**CHECK-NEXT:**" directives reject the input unless there is exactly one newline between it and the previous directive. A "**CHECK-NEXT:**" cannot be the first directive in a file.

The "CHECK-SAME:" directive

Sometimes you want to match lines and would like to verify that matches happen on the same line as the previous match. In this case, you can use "**CHECK:**" and "**CHECK-SAME:**" directives to specify this. If you specified a custom check prefix, just use "**<PREFIX>-SAME:**".

"**CHECK-SAME:**" is particularly powerful in conjunction with "**CHECK-NOT:**" (described below).

For example, the following works like you'd expect:

```
!0 = !DILocation(line: 5, scope: !1, inlinedAt: !2)

; CHECK:       !DILocation(line: 5,
; CHECK-NOT:               column:
; CHECK-SAME:              scope: ![[SCOPE:[0-9]+]]
```

"**CHECK-SAME:**" directives reject the input if there are any newlines between it and the previous directive.

"**CHECK-SAME:**" is also useful to avoid writing matchers for irrelevant fields. For example, suppose you're writing a test which parses a tool that generates output like this:

```
Name: foo
Field1: ...
Field2: ...
Field3: ...
Value: 1

Name: bar
Field1: ...
Field2: ...
Field3: ...
Value: 2

Name: baz
Field1: ...
Field2: ...
Field3: ...
```

Value: 1

To write a test that verifies **foo** has the value **1**, you might first write this:

```
CHECK: Name: foo
CHECK: Value: 1{{$}}
```

However, this would be a bad test: if the value for **foo** changes, the test would still pass because the "**CHECK: Value: 1**" line would match the value from **baz**. To fix this, you could add **CHECK-NEXT** matchers for every **FieldN:** line, but that would be verbose, and need to be updated when **Field4** is added. A more succinct way to write the test using the "**CHECK-SAME:**" matcher would be as follows:

```
CHECK:      Name: foo
CHECK:      Value:
CHECK-SAME:       {{ 1$}}
```

This verifies that the *next* time "**Value:**" appears in the output, it has the value **1**.

Note: a "**CHECK-SAME:**" cannot be the first directive in a file.

The "CHECK-EMPTY:" directive

If you need to check that the next line has nothing on it, not even whitespace, you can use the "**CHECK-EMPTY:**" directive.

```
declare void @foo()

declare void @bar()
; CHECK: foo
; CHECK-EMPTY:
; CHECK-NEXT: bar
```

Just like "**CHECK-NEXT:**" the directive will fail if there is more than one newline before it finds the next blank line, and it cannot be the first directive in a file.

The "CHECK-NOT:" directive

The "**CHECK-NOT:**" directive is used to verify that a string doesn't occur between two matches (or before the first match, or after the last match). For example, to verify that a load is removed by a transformation, a test like this can be used:

```
define i8 @coerce_offset0(i32 %V, i32* %P) {
  store i32 %V, i32* %P

  %P2 = bitcast i32* %P to i8*
  %P3 = getelementptr i8* %P2, i32 2

  %A = load i8* %P3
  ret i8 %A
; CHECK: @coerce_offset0
; CHECK-NOT: load
; CHECK: ret i8
}
```

The "CHECK-COUNT:" directive

If you need to match multiple lines with the same pattern over and over again you can repeat a plain **CHECK:** as many times as needed. If that looks too boring you can instead use a counted check "**CHECK-COUNT-<num>:**", where **<num>** is a positive decimal number. It will match the pattern exactly **<num>** times, no more and no less. If you specified a custom check prefix, just use "**<PREFIX>-COUNT-<num>:**" for the same effect. Here is a simple example:

```
Loop at depth 1
Loop at depth 1
Loop at depth 1
Loop at depth 1
  Loop at depth 2
    Loop at depth 3

; CHECK-COUNT-6: Loop at depth {{[0-9]+}}
; CHECK-NOT:     Loop at depth {{[0-9]+}}
```

The "CHECK-DAG:" directive

If it’s necessary to match strings that don’t occur in a strictly sequential order, "**CHECK-DAG:**" could be used to verify them between two matches (or before the first match, or after the last match). For example, clang emits vtable globals in reverse order. Using **CHECK-DAG:**, we can keep the checks in the natural order:

```
// RUN: %clang_cc1 %s -emit-llvm -o - | FileCheck %s

struct Foo { virtual void method(); };
Foo f;  // emit vtable
```

```
// CHECK-DAG: @_ZTV3Foo =

struct Bar { virtual void method(); };
Bar b;
// CHECK-DAG: @_ZTV3Bar =
```

CHECK-NOT: directives could be mixed with **CHECK-DAG:** directives to exclude strings between the surrounding **CHECK-DAG:** directives. As a result, the surrounding **CHECK-DAG:** directives cannot be reordered, i.e. all occurrences matching **CHECK-DAG:** before **CHECK-NOT:** must not fall behind occurrences matching **CHECK-DAG:** after **CHECK-NOT:**. For example,

```
; CHECK-DAG: BEFORE
; CHECK-NOT: NOT
; CHECK-DAG: AFTER
```

This case will reject input strings where **BEFORE** occurs after **AFTER**.

With captured variables, **CHECK-DAG:** is able to match valid topological orderings of a DAG with edges from the definition of a variable to its use. It's useful, e.g., when your test cases need to match different output sequences from the instruction scheduler. For example,

```
; CHECK-DAG: add [[REG1:r[0-9]+]], r1, r2
; CHECK-DAG: add [[REG2:r[0-9]+]], r3, r4
; CHECK:     mul r5, [[REG1]], [[REG2]]
```

In this case, any order of that two **add** instructions will be allowed.

If you are defining *and* using variables in the same **CHECK-DAG:** block, be aware that the definition rule can match *after* its use.

So, for instance, the code below will pass:

```
; CHECK-DAG: vmov.32 [[REG2:d[0-9]+]][0]
; CHECK-DAG: vmov.32 [[REG2]][1]
vmov.32 d0[1]
vmov.32 d0[0]
```

While this other code, will not:

```
; CHECK-DAG: vmov.32 [[REG2:d[0-9]+]][0]
; CHECK-DAG: vmov.32 [[REG2]][1]
vmov.32 d1[1]
vmov.32 d0[0]
```

While this can be very useful, it's also dangerous, because in the case of register sequence, you must have a strong order (read before write, copy before use, etc). If the definition your test is looking for doesn't match (because of a bug in the compiler), it may match further away from the use, and mask real bugs away.

In those cases, to enforce the order, use a non-DAG directive between DAG-blocks.

A **CHECK-DAG:** directive skips matches that overlap the matches of any preceding **CHECK-DAG:** directives in the same **CHECK-DAG:** block. Not only is this non-overlapping behavior consistent with other directives, but it's also necessary to handle sets of non-unique strings or patterns. For example, the following directives look for unordered log entries for two tasks in a parallel program, such as the OpenMP runtime:

```
// CHECK-DAG: [[THREAD_ID:[0-9]+]]: task_begin
// CHECK-DAG: [[THREAD_ID]]: task_end
//
// CHECK-DAG: [[THREAD_ID:[0-9]+]]: task_begin
// CHECK-DAG: [[THREAD_ID]]: task_end
```

The second pair of directives is guaranteed not to match the same log entries as the first pair even though the patterns are identical and even if the text of the log entries is identical because the thread ID manages to be reused.

The "CHECK-LABEL:" directive

Sometimes in a file containing multiple tests divided into logical blocks, one or more **CHECK:** directives may inadvertently succeed by matching lines in a later block. While an error will usually eventually be generated, the check flagged as causing the error may not actually bear any relationship to the actual source of the problem.

In order to produce better error messages in these cases, the "**CHECK-LABEL:**" directive can be used. It is treated identically to a normal **CHECK** directive except that FileCheck makes an additional assumption that a line matched by the directive cannot also be matched by any other check present in **match-filename**; this is intended to be used for lines containing labels or other unique identifiers. Conceptually, the presence of **CHECK-LABEL** divides the input stream into separate blocks, each of which is processed independently, preventing a **CHECK:** directive in one block matching a line in

another block. If **--enable-var-scope** is in effect, all local variables are cleared at the beginning of the block.

For example,

```
define %struct.C* @C_ctor_base(%struct.C* %this, i32 %x) {
entry:
; CHECK-LABEL: C_ctor_base:
; CHECK: mov [[SAVETHIS:r[0-9]+]], r0
; CHECK: bl A_ctor_base
; CHECK: mov r0, [[SAVETHIS]]
  %0 = bitcast %struct.C* %this to %struct.A*
  %call = tail call %struct.A* @A_ctor_base(%struct.A* %0)
  %1 = bitcast %struct.C* %this to %struct.B*
  %call2 = tail call %struct.B* @B_ctor_base(%struct.B* %1, i32 %x)
  ret %struct.C* %this
}

define %struct.D* @D_ctor_base(%struct.D* %this, i32 %x) {
entry:
; CHECK-LABEL: D_ctor_base:
```

The use of **CHECK-LABEL:** directives in this case ensures that the three **CHECK:** directives only accept lines corresponding to the body of the **@C_ctor_base** function, even if the patterns match lines found later in the file. Furthermore, if one of these three **CHECK:** directives fail, FileCheck will recover by continuing to the next block, allowing multiple test failures to be detected in a single invocation.

There is no requirement that **CHECK-LABEL:** directives contain strings that correspond to actual syntactic labels in a source or output language: they must simply uniquely match a single line in the file being verified.

CHECK-LABEL: directives cannot contain variable definitions or uses.

Directive modifiers

A directive modifier can be append to a directive by following the directive with **{<modifier>}** where the only supported value for **<modifier>** is **LITERAL**.

The **LITERAL** directive modifier can be used to perform a literal match. The modifier results in the directive not recognizing any syntax to perform regex matching, variable capture or any substitutions.

This is useful when the text to match would require excessive escaping otherwise. For example, the following will perform literal matches rather than considering these as regular expressions:

```
Input: [[[10, 20]], [[30, 40]]]
Output %r10: [[10, 20]]
Output %r10: [[30, 40]]

; CHECK{LITERAL}: [[[10, 20]], [[30, 40]]]
; CHECK-DAG{LITERAL}: [[30, 40]]
; CHECK-DAG{LITERAL}: [[10, 20]]
```

FileCheck Regex Matching Syntax

All FileCheck directives take a pattern to match. For most uses of FileCheck, fixed string matching is perfectly sufficient. For some things, a more flexible form of matching is desired. To support this, FileCheck allows you to specify regular expressions in matching strings, surrounded by double braces: **{{yourregex}}**. FileCheck implements a POSIX regular expression matcher; it supports Extended POSIX regular expressions (ERE). Because we want to use fixed string matching for a majority of what we do, FileCheck has been designed to support mixing and matching fixed string matching with regular expressions. This allows you to write things like this:

```
; CHECK: movhpd      {{[0-9]+}}(%esp), {{%xmm[0-7]}}
```

In this case, any offset from the ESP register will be allowed, and any xmm register will be allowed.

Because regular expressions are enclosed with double braces, they are visually distinct, and you don’t need to use escape characters within the double braces like you would in C. In the rare case that you want to match double braces explicitly from the input, you can use something ugly like **{{[}][}]}** as your pattern. Or if you are using the repetition count syntax, for example **[[:xdigit:]]{8}** to match exactly 8 hex digits, you would need to add parentheses like this **{{([[:xdigit:]]{8})}}** to avoid confusion with FileCheck’s closing double-brace.

FileCheck String Substitution Blocks

It is often useful to match a pattern and then verify that it occurs again later in the file. For codegen tests, this can be useful to allow any register, but verify that that register is used consistently later. To do this, **FileCheck** supports string substitution blocks that allow string variables to be defined and substituted into patterns. Here is a simple example:

```
; CHECK: test5:
; CHECK:    notw     [[REGISTER:%[a-z]+]]
```

```
; CHECK:    andw     {{.*}}[[REGISTER]]
```

The first check line matches a regex **%[a-z]+** and captures it into the string variable **REGISTER**. The second line verifies that whatever is in **REGISTER** occurs later in the file after an "**andw**". **FileCheck** string substitution blocks are always contained in **[[ ]]** pairs, and string variable names can be formed with the regex **[a-zA-Z_][a-zA-Z0-9_]***. If a colon follows the name, then it is a definition of the variable; otherwise, it is a substitution.

FileCheck variables can be defined multiple times, and substitutions always get the latest value. Variables can also be substituted later on the same line they were defined on. For example:

```
; CHECK: op [[REG:r[0-9]+]], [[REG]]
```

Can be useful if you want the operands of **op** to be the same register, and don't care exactly which register it is.

If **--enable-var-scope** is in effect, variables with names that start with **$** are considered to be global. All others variables are local. All local variables get undefined at the beginning of each CHECK-LABEL block. Global variables are not affected by CHECK-LABEL. This makes it easier to ensure that individual tests are not affected by variables set in preceding tests.

FileCheck Numeric Substitution Blocks

FileCheck also supports numeric substitution blocks that allow defining numeric variables and checking for numeric values that satisfy a numeric expression constraint based on those variables via a numeric substitution. This allows **CHECK:** directives to verify a numeric relation between two numbers, such as the need for consecutive registers to be used.

The syntax to capture a numeric value is **[[#%<fmtspec>,<NUMVAR>:]]** where:

- **%<fmtspec>,** is an optional format specifier to indicate what number format to match and the minimum number of digits to expect.

- **<NUMVAR>:** is an optional definition of variable **<NUMVAR>** from the captured value.

  The syntax of **<fmtspec>** is: **#.<precision><conversion specifier>** where:

- **#** is an optional flag available for hex values (see **<conversion specifier>** below) which requires the value matched to be prefixed by **0x**.

- **.<precision>** is an optional printf-style precision specifier in which **<precision>** indicates the

minimum number of digits that the value matched must have, expecting leading zeros if needed.

- **<conversion specifier>** is an optional scanf-style conversion specifier to indicate what number format to match (e.g. hex number). Currently accepted format specifiers are **%u**, **%d**, **%x** and **%X**. If absent, the format specifier defaults to **%u**.

  For example:

```
; CHECK: mov r[[#REG:]], 0x[[#%.8X,ADDR:]]
```

  would match **mov r5, 0x0000FEFE** and set **REG** to the value **5** and **ADDR** to the value **0xFEFE**. Note that due to the precision it would fail to match **mov r5, 0xFEFE**.

  As a result of the numeric variable definition being optional, it is possible to only check that a numeric value is present in a given format. This can be useful when the value itself is not useful, for instance:

```
; CHECK-NOT: mov r0, r[[#]]
```

  to check that a value is synthesized rather than moved around.

  The syntax of a numeric substitution is **[[#%<fmtspec>, <constraint> <expr>]]** where:

- **<fmtspec>** is the same format specifier as for defining a variable but in this context indicating how a numeric expression value should be matched against. If absent, both components of the format specifier are inferred from the matching format of the numeric variable(s) used by the expression constraint if any, and defaults to **%u** if no numeric variable is used, denoting that the value should be unsigned with no leading zeros. In case of conflict between format specifiers of several numeric variables, the conversion specifier becomes mandatory but the precision specifier remains optional.

- **<constraint>** is the constraint describing how the value to match must relate to the value of the numeric expression. The only currently accepted constraint is **==** for an exact match and is the default if **<constraint>** is not provided. No matching constraint must be specified when the **<expr>** is empty.

- **<expr>** is an expression. An expression is in turn recursively defined as:

  - a numeric operand, or

  - an expression followed by an operator and a numeric operand.

A numeric operand is a previously defined numeric variable, an integer literal, or a function. Spaces are accepted before, after and between any of these elements. Numeric operands have 64-bit precision. Overflow and underflow are rejected. There is no support for operator precedence, but parentheses can be used to change the evaluation order.

The supported operators are:

- **+** - Returns the sum of its two operands.
- **-** - Returns the difference of its two operands.

The syntax of a function call is **<name>(<arguments>)** where:

- **name** is a predefined string literal. Accepted values are:
  - add - Returns the sum of its two operands.
  - div - Returns the quotient of its two operands.
  - max - Returns the largest of its two operands.
  - min - Returns the smallest of its two operands.
  - mul - Returns the product of its two operands.
  - sub - Returns the difference of its two operands.
- **<arguments>** is a comma separated list of expressions.

For example:

```
; CHECK: load r[[#REG:]], [r0]
; CHECK: load r[[#REG+1]], [r1]
; CHECK: Loading from 0x[[#%x,ADDR:]]
; CHECK-SAME: to 0x[[#ADDR + 7]]
```

The above example would match the text:

```
load r5, [r0]
load r6, [r1]
```

Loading from 0xa0463440 to 0xa0463447

but would not match the text:

load r5, [r0]
load r7, [r1]
Loading from 0xa0463440 to 0xa0463443

Due to **7** being unequal to **5 + 1** and **a0463443** being unequal to **a0463440 + 7**.

A numeric variable can also be defined to the result of a numeric expression, in which case the numeric expression constraint is checked and if verified the variable is assigned to the value. The unified syntax for both checking a numeric expression and capturing its value into a numeric variable is thus **[[#%<fmtspec>,<NUMVAR>: <constraint> <expr>]]** with each element as described previously. One can use this syntax to make a testcase more self-describing by using variables instead of values:

; CHECK: mov r[[#REG_OFFSET:]], 0x[[#%X,FIELD_OFFSET:12]]
; CHECK-NEXT: load r[[#]], [r[[#REG_BASE:]], r[[#REG_OFFSET]]]

which would match:

mov r4, 0xC
load r6, [r5, r4]

The **--enable-var-scope** option has the same effect on numeric variables as on string variables.

Important note: In its current implementation, an expression cannot use a numeric variable defined earlier in the same CHECK directive.

FileCheck Pseudo Numeric Variables

Sometimes there’s a need to verify output that contains line numbers of the match file, e.g. when testing compiler diagnostics. This introduces a certain fragility of the match file structure, as "**CHECK:**" lines contain absolute line numbers in the same file, which have to be updated whenever line numbers change due to text addition or deletion.

To support this case, FileCheck expressions understand the **@LINE** pseudo numeric variable which evaluates to the line number of the CHECK pattern where it is found.

This way match patterns can be put near the relevant test lines and include relative line number

references, for example:

```
// CHECK: test.cpp:[[# @LINE + 4]]:6: error: expected ';' after top level declarator
// CHECK-NEXT: {{^int a}}
// CHECK-NEXT: {{^     \^}}
// CHECK-NEXT: {{^     ;}}
int a
```

To support legacy uses of **@LINE** as a special string variable, **FileCheck** also accepts the following uses of **@LINE** with string substitution block syntax: **[[@LINE]]**, **[[@LINE+<offset>]]** and **[[@LINE-<offset>]]** without any spaces inside the brackets and where **offset** is an integer.

Matching Newline Characters

To match newline characters in regular expressions the character class **[[:space:]]** can be used. For example, the following pattern:

```
// CHECK: DW_AT_location [DW_FORM_sec_offset] ([[DLOC:0x[0-9a-f]+]]){{[[:space:]].*}}"intd"
```

matches output of the form (from llvm-dwarfdump):

```
DW_AT_location [DW_FORM_sec_offset]   (0x00000233)
DW_AT_name [DW_FORM_strp]  ( .debug_str[0x000000c9] = "intd")
```

letting us set the **FileCheck** variable **DLOC** to the desired value **0x00000233**, extracted from the line immediately preceding "**intd**".

AUTHOR

Maintained by the LLVM Team (https://llvm.org/).

COPYRIGHT