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 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.

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

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

--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 `<mode>`

Dump input to stderr, adding annotations representing currently enabled diagnostics. Do this either 'always', on 'fail', or 'never'. Specify 'help' to explain the dump format and quit.

--dump-input-on-failure

When the check fails, dump all of the original input. This option is deprecated in favor of *`–dump-input=fail`*.

--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#<NUMVAR>`=<NUMERIC EXPRESSION>`

Sets a filecheck numeric variable `NUMVAR` 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 `-input-dump=fail` or `-input-dump=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 `-input-dump=fail` or `-input-dump=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.

--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 "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. 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.

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 define a numeric variable is `[[#<NUMVAR>:]]` where `<NUMVAR>` is the name of the numeric variable to define to the matching value.

For example:

```
; CHECK: mov r[[#REG:]], 42
```

would match `mov r5, 42` and set `REG` to the value `5`.

The syntax of a numeric substitution is `[[#<expr>]]` where `<expr>` is an expression. An expression is 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, or an integer literal. The supported operators are `+` and `-`. Spaces are accepted before, after and between any of these elements.

For example:

```
; CHECK: load r[[#REG:]], [r0]
; CHECK: load r[[#REG+1]], [r1]
```

The above example would match the text:

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

but would not match the text:

```
load r5, [r0]
load r7, [r1]
```

due to `7` being unequal to `5 + 1`.

The syntax also supports an empty expression, equivalent to writing {{[0-9]+}}, for cases where the input must contain a numeric value but the value itself does not matter:

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

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

A numeric variable can also be defined to the result of a numeric expression, in which case the numeric expression is checked and if verified the variable is assigned to the value. The unified syntax for both defining numeric variables and checking a numeric expression is thus `[[#<NUMVAR>: <expr>]]` with each element as described previously.

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:]].*}}"
```

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`".