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 always read from standard input.

OPTIONS

−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 a specific prefix to match.

−−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.

−version

Show the version number of this program.

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&gt;* %A, double %B) {
      %tmp3 = load <2 x double&gt;* %A, align 16
      %tmp7 = insertelement <2 x double&gt; undef, double %B, i32 0
      %tmp9 = shufflevector <2 x double&gt; %tmp3,
                             <2 x double&gt; %tmp7,
                             <2 x i32&gt; < i32 0, i32 2 &gt;
      store <2 x double&gt; %tmp9, <2 x double&gt;* %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 an the previous directive. A CHECK-NEXT 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
}
```

FileCheck Pattern Matching Syntax

The CHECK: and CHECK–NOT: directives both 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}}**. 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.

FileCheck Variables

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 allows named 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 variable "REGISTER". The second line verifies that whatever is in REGISTER occurs later in the file after an "andw". FileCheck variable references are always contained in **[[ ]]** pairs, are named, and their 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, if not, it is a use.

FileCheck variables can be defined multiple times, and uses always get the latest value. Note that variables are all read at the start of a ''CHECK'' line and are all defined at the end. This means that if you have something like "**CHECK: [[XYZ:.*]]x[[XYZ]]**", the check line will read the previous value of the XYZ variable and define a new one after the match is performed. If you need to do something like this you can probably take advantage of the fact that FileCheck is not actually line-oriented when it matches, this allows you to define two separate CHECK lines that match on the same line.

AUTHORS

Maintained by The LLVM Team (<http://llvm.org/>).