Advanced Compiler Design using Version-Fenced Lexemes

@MarkDavidRaley

In traditional build systems a set of files is pulled from source control and then compiled and linked. A single specific instance of each file needed is selected from among one or more revisions by a label or tagging mechanism. Thus, from many revisions a specific executable is derived that implicitly supports a single version of that software. Source control defines the version. See figure 1.

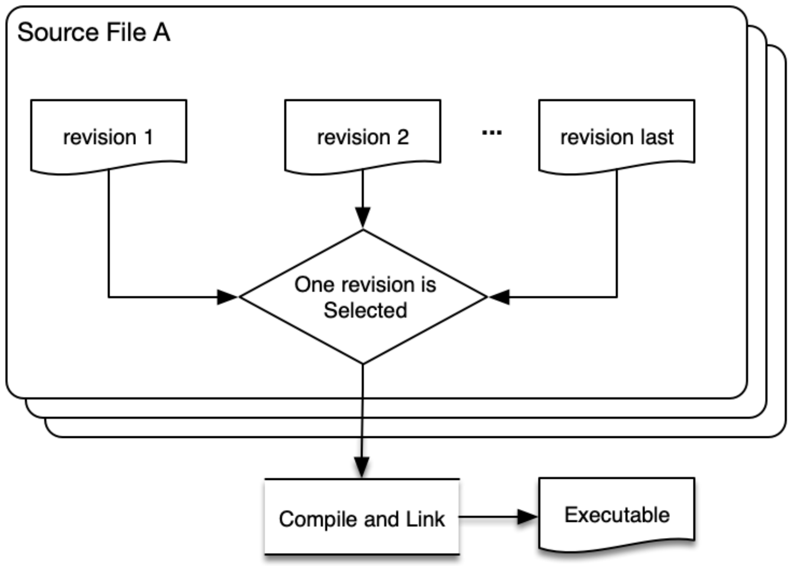


Figure 1 - Traditional Build - a single revision of each needed source is selected and used.

Alternatively, consider source code that encodes versioning information at the lexeme level and is intended to be fed to advanced “multiple parse stream capable” compilers. This kind of compiler is designed to knit these different versions together at the abstract syntax tree level into an enhanced executable which can run any version specified. See figure 2. Version-Fenced Lexemes describes the mechanism and process of assigning versions to lexemes.

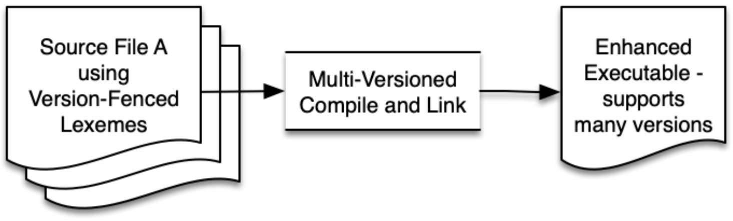
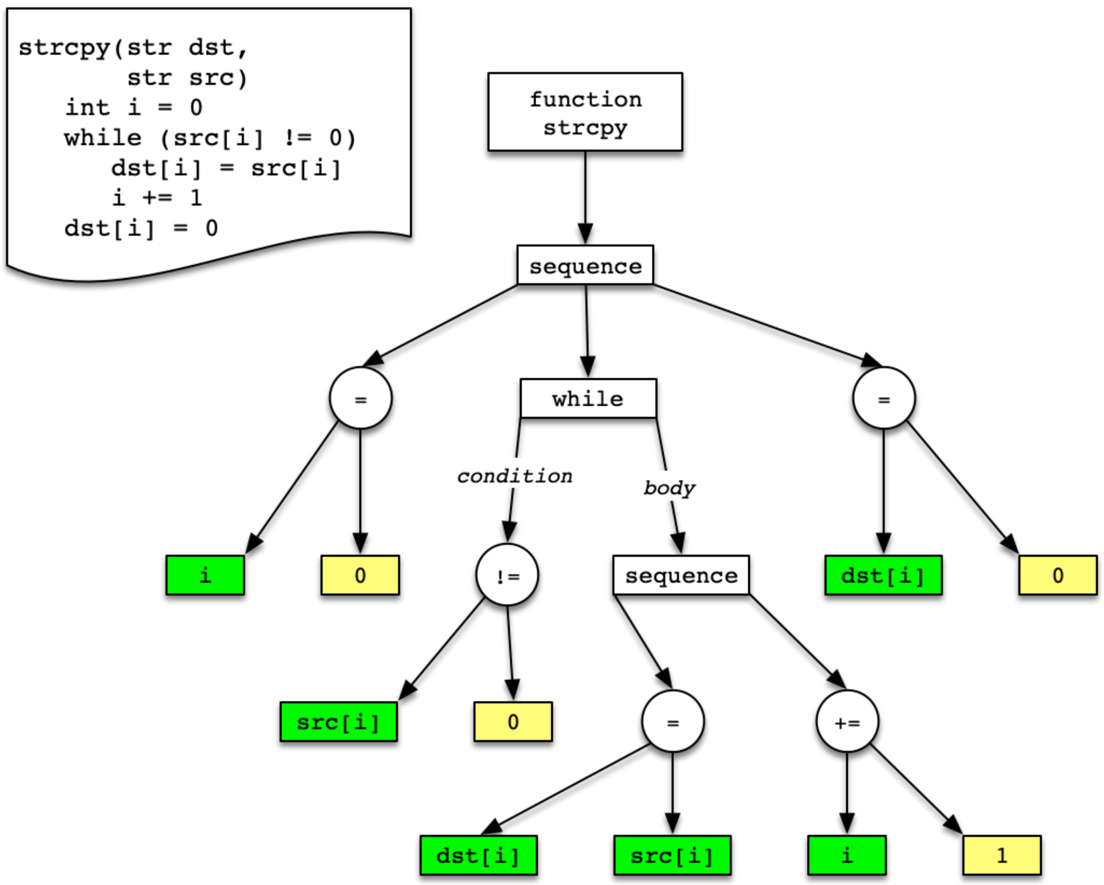

Source File A using Version-Fenced Lexemes
Multi-Versioned Compile and Link
Enhanced Executable - supports many versions

Figure 2 - Multi-Versioned Builds - Source uses version fenced lexemes (lexical tokens) to define multiple versions within a unified executable.

Version-Fenced Lexemes

Lexical tokens, or lexemes, are the building blocks of all computer languages. Version-fenced lexemes are lexemes that have been assigned version information at the source level. This is accomplished by introducing a meta-grammar to any language which consists of four elements - A start meta-delimiter, a version test, a lexeme stream specific to the host language, and end meta delimiter. Consider the following historical example of real code evolution. The elements used are chosen for clarity and many details are elided, such as error checking and return values, which would otherwise be present in production code.

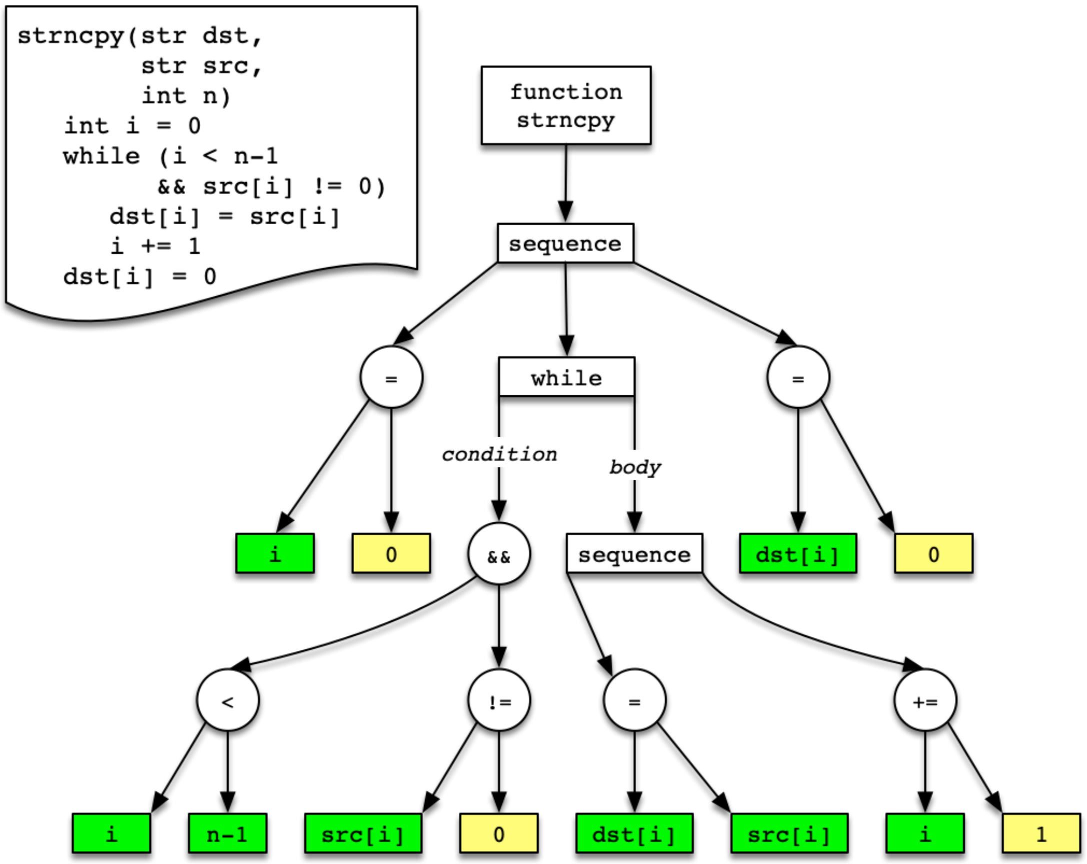
Decades ago a function named strcpy() was introduced to the "C" library. It moved characters from a source zero-terminated string to a destination string, including the zero terminator. Figure 3 presents a pseudo-code implementation of this, where "str" may be thought of as a string pointer or array of characters.



```
strcpy(str dst,
       str src)
   int i = 0
   while (src[i] != 0)
      dst[i] = src[i]
      i += 1
   dst[i] = 0
```

Figure 3 - Function strcpy, source and AST.

Because buffer overflows (intentional or otherwise) became a common problem with strcpy() calls, it was deprecated and strncpy() was introduced that added an extra parameter - the maximum number of array elements that could be safely moved to the destination buffer. See figure 4.



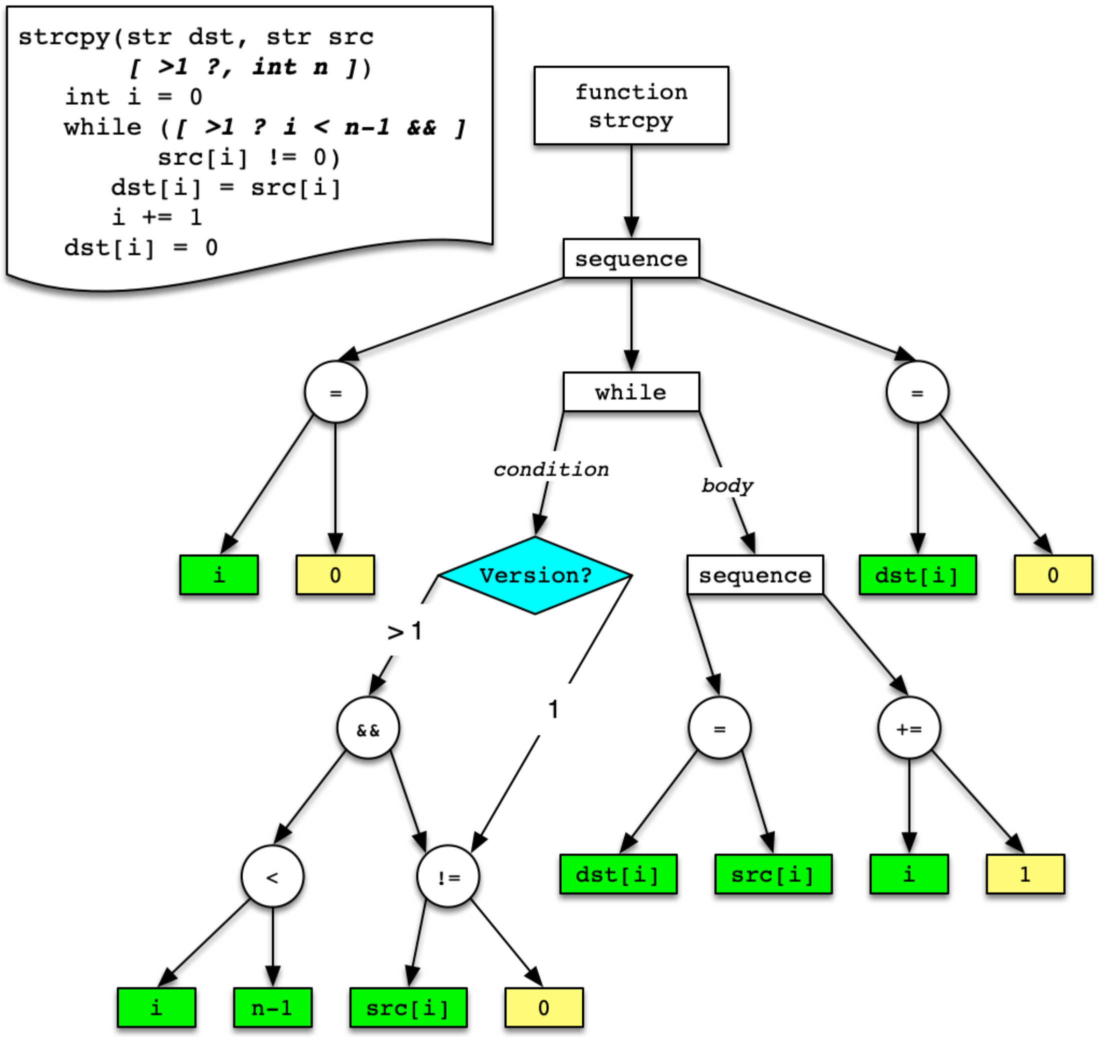
```
strncpy(str dst,
        str src,
        int n)
   int i = 0
   while (i < n-1
         && src[i] != 0)
      dst[i] = src[i]
      i += 1
   dst[i] = 0
```

Figure 4 - Function strncpy, source code and AST.

Using version-fenced lexemes, strcpy() can be re-coded to support both versions. The lexeme sequence

[<version-test> ? <host-language-lexemes>]

Directs the compiler to apply the delimited (fenced) host language lexemes to certain versions but not to others. In the strncpy() example, a new parameter "n" and associated loop test are applied to versions greater than 1 which reproduce the figure 4 code, and the original unfenced lexemes apply only to version 1 and reproduce the base figure 3 code. To accomplish this the compiler and supporting runtime uses a new type of vertex within the AST (colored blue) that takes a different path based on a (meta) version test. See figure 5.



```
strcpy(str dst, str src
       [ >1 ?, int n ])
   int i = 0
   while ([ >1 ? i < n-1 && ]
         src[i] != 0)
      dst[i] = src[i]
      i += 1
   dst[i] = 0
```

Figure 5 – Function strcpy with strncpy behavior implemented with version-fenced lexemes, source and versioned AST.