Automatic syntax highlighter generation

Allen, S. T.; Williams, S. R.

December 9, 2005

Abstract

Autohighlight is a tool that produces syntax highlighting modules for user-defined BNF grammars and lexical specifications for Emacs and VIM. Autohighlight can color all literals in a sample document, plus a strictly-defined subset of the nonterminal symbols defined in the BNF grammar.

1 Definitions

Autohighlight is a meta-level tool, so the terms used to talk about it are sometimes confusing. Autohighlight produces code that helps the editor color the words in a file, called the *sample*, which is a conforming example of a given *language*. The *language spec* tells you the syntax and lexical properties of the language.

Autohighlight is interested in two parts of the language spec, the BNF grammar of the language (the *grammar*), and the lexical specification of the basic symbols of the language (the *lexical spec*). The lexical specification specifies the regular syntax of *lexical symbols*.

The BNF grammar of a language is made of a list of productions, which name a *non-terminal* symbol on the left-hand side of a rule and tell on the right-hand side of the rule what symbols (both *terminals* (*literals* or lexical symbols) and non-terminals) the non-terminal may expand to in the sample.

Expanding a symbol s to terminals on a non-terminal symbol s is accomplished by expanding all symbols in all the productions of s until only terminals remain. Note that not all non-terminal symbols *can* be expanded to terminals—expanding the root of a useful language should produce all the possible samples of that language, which is infinite in many cases.

Within the grammar, Autohighlight distinguishes among different classes of non-terminal symbols. A non-terminal would be *terminal-equivalent* if it produces a single terminal when expanded to terminals. The *terminal-equivalent regex* of a terminal-equivalent symbol s is a regular expression matching the terminals that the terminal-equivalent symbol s may expand to. A closely related action is to find the *left-most or right-most expansion regex* of a given non-terminal s. This regex matches the end of all productions of s when s is expanded to terminals.

Lastly, Autohighlight employs a few more terms related to the BNF grammar. The *total context* of a terminal-equivalent symbol s is a set of pairs of expansion regexes that match the symbol's context when it appears in the sample.

2 Motivation

When we were first introduced to the Eli compiler generation system, Scott immediately hacked out an Emacs mode for highlighting Eli's FunnelWeb files. Writing syntax highlighting code is painful, because a lot of effort is required to keep the highlighter in sync with the language spec. Additionally, lots of iterations are required to produce a highlighter that deals with coloring a symbol which has the same lexical specification as a symbol which should be colored differently, as the context of the symbol must be taken into account. Since the syntax highlighting specification is written by intuition and not by a particular algorithm, there's always the danger that a context will be missed simply because the author of the syntax highlighting spec has never generated a sample containing that particular context.

3 Analysis flow

Autohighlight follows the general analysis flow of any compiler:

1. Tokenization
2. Parsing
3. Context-checking
   (a) Name analysis
   (b) Type analysis
   (c) Colorability analysis
4. Output

We'll delve further into the details of all the items than would be necessary in an Eli-generated compiler because we implemented our parser using Python. Despite using a different technology, many of the same underlying concepts from Eli apply to our Python implementation.

We began our implementation using Eli, abandoning the method when we reached item 3, but the grammar of the Autohighlight specifications didn't change. In order to provide a basis for understanding the Autohighlight tokenizer and parser, figure 3 gives the concrete grammar and lexical specification we wrote for Eli.

3.1 Tokenization

The tokenizer is a finite state machine encapsulated in a Python generator. Because it is a generator, to use it is a matter of initializing it (using a stream or a filename), and then simply calling next(). Each call to next will return the next token. It is also possible to use it as a source list in a for-in loop. A generator must have a method `__iter__` which produces an object that has a `next` method. The next method is called repeatedly to get the next element in the sequence the generator represents until the `next` method raises a `StopIteration` exception.

```
class Tokenizer:
    ...
    def next(self):
        self.token, self.sline, self.scol, self.char = "", self.line, self.col, ''

        while True:
            self.char = self.stream.read(1)
            retval = self.transition()
            if self.char == '\n':
                self.setCursor(self.line + 1, 0)
            else: self.setCursor(self.line, self.col + 1)
            if retval: return retval
            if self.char == '': raise StopIteration()
```

The state machine has several transition actions when switching between states. By looking at the transition actions listed below, you can tell the state machine is not "clean" and some of the state is kept in the tokenizer object as member data instead of encapsulated in the state number. This is for practical reasons.

```
class Tokenizer:
    ...
    def add(self):
```

```
@O@<test.gla@>==@{@-
Identifier: C_IDENTIFIER [mkidn]
Literal: MODULA2_LITERALSQ [mkidn]
Integer: C_INTEGER [mkidn]
RegularExpression: $\$[^\040]+ [mkidn]
@}
@O@<.lido@>==@{@-
RULE: Document ::= '{' GlaFile '}' '{' ConFile '}' '{' AhFile '}' END;

RULE: SymbolDef ::= Identifier END;
RULE: GlaSymbolDef ::= Identifier END;
RULE: LiteralDef ::= Literal END;
RULE: LiteralUse ::= Literal END;
RULE: ColorDef ::= Identifier END;
RULE: ColorUse ::= Identifier END;
RULE: PreDefPatternUse ::= Identifier END;

RULE: GlaFile LISTOF Specification END;
RULE: Specification ::= GlaSymbolDef ':' RegularExpression '.' END;
RULE: Specification ::= GlaSymbolDef ':' PreDefPatternUse '.' END;

RULE: ConFile LISTOF Production END;
RULE: Production ::= SymbolDef ':' Elements '.' END;
RULE: Elements LISTOF Element END;
RULE: ConSymbol ::= SymbolUse END;
RULE: ConSymbol ::= LiteralDef END;
RULE: Element ::= ConSymbol END;
RULE: Element ::= '&' ConSymbol END;
RULE: Element ::= '@@' ConSymbol END;
RULE: Element ::= '$' ConSymbol END;
RULE: AhFile LISTOF Statement COMPUTE
        AhFile.done = CONSTITUENTS ColorDef.defined;
END;
RULE: Statement ::= SyntaxGroupRule END;
RULE: Statement ::= MappingRule END;
RULE: SyntaxGroupRule ::= ColorDef '{' ColorAttrs '}' END;
RULE: ColorAttrs LISTOF ColorAttr END;
RULE: ColorAttr ::= 'font-face' ':' Literal ';' END;
RULE: ColorAttr ::= 'font-size' ':' Integer ';' END;
RULE: MappingRule ::= ColorUse ':' RuleRefs '.' END;
RULE: RuleRefs LISTOF RuleRef END;
RULE: RuleRef ::= SymbolUse END;
RULE: RuleRef ::= LiteralUse END;
@}
```

Figure 1: The Eli specification file for our generator.

```
        self.token += self.char
        return None
    def tok(self): return Token(self.sline, self.scol, self.token)
    def push(self):
        if self.char == '': return
        self.setCursor(self.line, self.col - 1)
        if self.col < 0 or self.char == '\n': self.setCursor(self.line - 1, 0)
        self.stream.seek(-1, 1)
    def stop(self): raise StopIteration()
    def noop(self): return None
    def reset(self):
        self.sline, self.scol = self.line, self.col
        return None
    def strangechar(self): raise UnexpectedCharacter(self.line, self.col, self.char)
    def endinstr(self): raise EofInString(self.sline, self.scol)
```

The state transition table is an array, where each element corresponds to a state. Each element is a list of transitions out of the state. Each transition is a tuple whose head is either a string to match exactly or a regex to match that indicates whether this transition should be used. The first transition whose head matches is used. The second element in the transition tuple indicates the destination state number, and the remainder of the tuple is a list of actions to take on leaving. The `c` function is a helper function that compiles its argument into a regular expression object.

```
class Tokenizer:
    ...
  transitions = [ \
      # state 0: initial state \
      [ (c("[A-Za-z]"), 1, reset, add), ('', 0, stop), (c('[0-9]'), 5, reset, add), \
        ('$', 2, reset, add), ("'", 3, reset, add), \
        (c('[][{};:,.]'), 0, reset, add, tok), (c('\s'), 0, noop), \
        (c('.'), 0, strangechar) ], \
      # state 1: accumulating identifiers \
      [ (c("[-A-Za-z0-9_]"), 1, add), ('', 0, tok), (c('.'), 0, push, tok) ], \
      # state 2: accumulating regexes \
      [ ('', 0, push, tok), (c('\s'), 0, tok), (c('.'), 2, add) ], \
      # state 3: accumulating strings \
      [ ('\\',␣4,␣add),␣("'",␣0,␣add,␣tok),␣('',␣0,␣endinstr),␣(c('.'),␣3,␣add)␣],␣\
␣␣␣␣␣␣#␣state␣4:␣escaping␣strings␣\
␣␣␣␣␣␣[␣('',␣0,␣endinstr),␣(c('.'),␣3,␣add)␣],␣\
␣␣␣␣␣␣#␣state␣5:␣integers␣\
␣␣␣␣␣␣[␣(c("[0-9]"),␣5,␣add),␣('',␣0,␣tok),␣(c('.'),␣0,␣push,␣tok)␣]␣]
␣␣
```

The transition notation is complex, so a member function `transition` exists that takes the input character and the current state, determines the transition, takes the transition actions (accumulating their output).

```
class Tokenizer:
    ...
    def transition(self):
        statedef = self.transitions[self.state]
        for path in statedef:
            pat, dest = path[:2]
            if type(pat).__name__ == "str" and pat == self.char or \
               type(pat).__name__ == "SRE_Pattern" and pat.match(self.char):
```

```
            for action in path[2:]:
                retval = action(self)
            self.state = dest
            return retval
        raise Exception("No matching path for char %s from state %d."\
                        % (self.char, self.state))
```

3.2 Parsing and basic analysis

Our parsing routine can perform basic name and type analysis for simple cases, so the two are integrated. See `Autohighlight::parse` for details on the highest level of parsing and simple analysis. Once the `parse` method has finished running, the `Autohighlight` object has a `ColorDefinitions` hash containing entities representing colors defined in the input file. It also contains an `OrderedColorMappings` list representing the color requests the user made. This is an ordered list, so that color mappings will be output in the order specified by the user, to allow the user to specify more specific color mappings first, and more general color mappings last, to help the editors properly cope with ambiguous colorings. Additionally, the object also contains a `GlobalSymbolDict` hash containing all the grammar and lexical symbols defined. After `parse` is run, the only remaining step in the analysis phase is to determine colorability, that is, to determine whether any coloring requests the user has given meet the criteria for colorable symbols.

3.2.1 Parsing the lexical section

The method `parse_lexical_symbols` illustrated below works by building up a stack until a complete lexical specification is built up. When a period is found, it checks that the symbol isn't being redefined, and then inserts it into the `GlobalSymbolDict` hash.

```
class Autohighlight:
    ...
    def parse_lexical_symbols(self):
        stack = []
        self.tokenizer.next().must_be('{')
        for token in self.tokenizer:
            stack += [ token ]
            if token.text == ".":
                stack[0].assert_symbol_name()
                stack[1].must_be(':')
                stack[2].must_match('^\\$', "regular expression")
                ## Name analysis
                if stack[0].text in self.GlobalSymbolDict:
                    originalDef = self.GlobalSymbolDict[stack[0].text].defining_token
                    raise Exception("Symbol %s redefined at %d,%d. Originally at %d,%d"

                s = Symbol(stack[0])
                s.is_gla = True
                s.regex = stack[2].text[1:]
                self.GlobalSymbolDict[stack[0].text] = s
                stack = []
            elif token.text == "{":
                raise Exception("Unexpected %s" % token)
            elif token.text == "}":
                if len(stack) > 1: raise Exception("Unfinished lexical specification beg
                #pp = pprint.PrettyPrinter()
```

```
            #pp.pprint(self.GlobalSymbolDict)
            return
        else: pass
```

The other parsing tasks are similar.

3.2.2 Analysis tasks

- An error must be signaled if a lexical symbol is defined more than once.
- An error must be signaled if a lexical symbol is used on the left-hand side of a production.
- An error must be signaled if a color name is redefined.
- An error must be signaled if a symbol on the right-hand side of a production is not a defined lexical symbol, a literal, or a symbol appearing on the left-hand side of another production

3.3 Colorability analysis

There are further, more difficult analysis tasks handled separately from the parser.

- The user may only color terminal-equivalent symbols.

In addition to the relatively simple task of determining whether a symbol is terminal-equivalent, the output generation step requires knowing the total context of a symbol, a much hairier process.

3.3.1 Determining terminal-equivalence

Terminal equivalence works by looking down the expansion path for a symbol. If at any time it recurses, it is not terminal equivalent. If at any time it produces more than one symbol in a context (zero is ok) it is not terminal equivalent.

3.3.2 Determining context

The context-finding algorithm is heart of the program. It figures out what literals may occur on either side of a given symbol. To do this, we first collect all the productions a symbol s occurs in. In each production, we look to the left and the right of s. If a symbol exists on either side, then finding the context in this production is easy: find the regex that matches the leftmost portion of the symbol on the right-hand side and the regex that matches the rightmost of the symbol on the left-hand side.

If there is no symbol on a particular side, the operation becomes more complex. In this case, it is necessary to look at the context of the symbol on the left hand side of the given production rule. The matching side of the parent context may be substituted for a lack of a symbol on either side of the current symbol.

This algorithm must also account for recursion. It skips productions that it has already looked at once for the given symbol in a recursive call. This allows it to still find all the other contexts within the recursion, producing the correct result.

Finally, we take some rudimentary steps to ensure that there are no duplicate contexts, as symbols may occur in different productions which yield the same immediate context.

3.4 Output generation

We applied the Strategy pattern in order to divest the Autohighlight main module from knowing the details Emacs and VIM output generation.

3.5 User Manual

To use autohighlighter, first you must generate an autohighlighter file, with extension `.ah`. This file specifies not only the specifics of the language you wish to color, but how you would like that language to be colored. There are three main sections to the autohighlighter file: the gla section, which contains regular expressions for terminal symbols, the concrete syntax tree section, which contains the BNF grammar for the language, and the coloring section, which provides definitions of custom colors, and dictates which portions of the language will be colored which color.

The gla section should consist of lines of the following format:

```
identifier: $regularexpression .
```

The BNF grammar section should consist of lines similar to the following format:

```
symbol: symbol 'literal' symbol .
```

where the right hand side may be any combination of symbols and literals, with literals enclosed in single quotes.

The coloring section should consist of color definitions, and color mappings. Color definitions are of the form:

```
colorname {
    color: red;
    background: blue;
    text-decoration: underline;
}
```

The complete list of attributes, with possible values are:

```
font-family: <fontname>
font-style: normal, italic
font-weight: bold, normal
font-size: <points>
text-decoration: underline, overline, line-through, inverse

color: Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta,
Brown, DarkYellow, LightGray, LightGrey, Gray, Grey, DarkGray,
DarkGrey, Blue, LightBlue, Green, LightGreen, Cyan, LightCyan, Red,
LightRed, Magenta, LightMagenta, Yellow, LightYellow, White

background-color: Black, DarkBlue, DarkGreen, DarkCyan, DarkRed,
DarkMagenta, Brown, DarkYellow, LightGray, LightGrey, Gray, Grey,
DarkGray, DarkGrey, Blue, LightBlue, Green, LightGreen, Cyan,
LightCyan, Red, LightRed, Magenta, LightMagenta, Yellow, LightYellow,
White
```

Color mappings are of the form:

```
colorname: symbol 'literal' symbol .
```

There are several predefined color names that map appropriately to the built-in recommended highlighting colors of both editors. These are:

Comment Use this to highlight anything

Constant For integers and other constants

String For strings

VariableName For identifiers

FunctionName For function identifiers

Keyword For distinguished and useful literals listed in the grammar.

Type For type identifiers

None Don't color

Error Should be flagged as an error

Once the autohighlighter file has been generated, it should be given a `.ah` file suffix. Then, the compiler should be run. The full command line syntax is as follows:

```
Usage: python ah.py [OPTION]... [FILE]
Generates the specified syntax highlighting files from the given input FILE.

Options:
  -h, --help  Prints this help
      --vim   Generates a vim syntax highlighting file
      --emacs Generates an emacs font locking file
      --error-checking Highlight all symbols not currently being colored as errors (curr
```