HAAGA-HELIA
University of Applied Sciences

Smart syntax highlighting for dynamic language
Case: Common Lisp in Emacs

Mikhail Novikov

Bachelor’s Thesis
DP in Business Information Technology
2008

HAAGA-HELIA
University of Applied Sciences

Abstract

30 May 2010

Business Information Technology

Authors Mikhail Novikov	**Group** BIT
Smart syntax highlighting for dynamically typed languages Case study: Common Lisp in Emacs	**Number of pages and appendices** 27+33
Supervisors Tero Karvinen	

Today, Java, C++ and C# being the most popular programming languages, the majority of integrated development environments are tailored for them, and, in general, for statically-typed languages. However, dynamically-typed languages are gaining more and more popularity; thus, integrated development environments must evolve to support them. The purpose of this thesis was to improve the situation by creating an advanced syntax highlighting system for dynamically-typed languages.

This study developed a general framework for implementing an advanced syntax highlighting system for integrated development environments. A proof-of-concept implementation was developed in Common Lisp language using Emacs.

The developed framework makes it possible to create syntax highlighting system that provides sophisticated scope detection and symbol extraction systems allowing an integrated development environment for dynamically-typed languages to possess the same language analysis features as the ones designed for statically-typed languages.

The study concludes that there are still many unexplored, potential improvements that could be made for integrated development environments in order to increase a programmer's productivity.

Keywords
dynamic languages, syntax highlighting, development environments, Common Lisp, Emacs

Table of Contents

List of Figures

Glossary

Integrated Development Environment Software application that provides set of tools to aid computer programmers in one package.

S-expression Symbolic expression. List based data-structure that represents semi-structured data, that can be nested. Best known for it's use in Lisp family of languages.

Symbol Lisp primitive named data structure. Throughout the thesis it is used as synonym to an identifier - a name that can be used as a variable, function, class, etc, name.

Reflection Programming language features that let's it observe and modify it's structure and behavior

Static (code) analysis Code analysis that is done without executing the code

Homoiconic Homoiconic languages are the languages in which their source code is also primitive type in them

Metaprogramming Programming that aims at manipulating other programs

Multiple dispatch Feature of some object-oriented languages that allows function or method to be dispatched on type of more than one of their arguments

RPC Remote Procedure Call - a network protocol that allows a computer program running on one host to cause code to be executed on another host

Dynamic language Programming language which is primarily dynamically-typed

1 Introduction

The popularity of dynamically typed languages have been increasing in recent years. Success of Python, that has been used by corporations like Google, Amazon NASA (Python Software Foundation, 2010), and Ruby, whose Ruby on Rails platform became "the model" web framework of today (Bächle and Kirchberg, 2007), has increased interest to concept of dynamically typed language as a whole. And even though dynamic typing have been around since as yearly as 1960, when first LISP has been created by McCarty (McCarty, 1978), for long time they were considered slow and unsuitable for modern development. Nowadays, with computers getting faster and cheaper, and programmer's time getting more expensive, rapid prototyping became very important (Mcclendon et al., 1996)(Grimm, 2005), and this is where dynamic languages generally excel (Kindborg, 2010)(Norvig, 1992). Performance of dynamic programming languages also have been on increase, with some Common Lisp implementations being able to compete with C++ and Java in speed (Norvig, 2002)(Gat, 2000) and some experimental Python implementations have reported 10 fold increase in speed comparing to CPython reference implementation(PyPy Project website, 2010).

One place where dynamically-typed languages generally lost to a more widespread static languages, such as Java and C++, is Integrated Development Environments field. Static code analysis, which is easily available for static languages, can easily provide features like auto-completion and "smart" syntax highlighting, that can provide hints to the programmer and simplify programming process (Visual Studio, 2010)(Eclipse, 2010). Dynamically-typed languages are certainly catching up, with, for example, full fledged Python support coming to Eclipse(PyDev, 2010) and Visual Studio(IronPython, 2010), that provide some advanced features such as auto-completion. One feature is generally overlooked, though - "smart" syntax highlighting - dynamically updating syntax highlighting which attempts to analyze the code in such way, that symbols in code are colored in according to their semantical meaning (for example variables and functions are colored differently).

1.1 Objectives and scope of the work

During this thesis work we would attempt to define a clear method to provide such "smart" syntax highlighting for a given dynamic language in a arbitrary extensible Integrated Development Environment. We would also attempt to create a proof of concept of such method, by implementing such "smart" syntax highlighting for a Common Lisp language using Emacs

Integrated Development Environment.

We define “smart” syntax highlighting as syntax highlighting that distinguishes not only set of keywords, predefined by author of the IDE (or plugin to IDE, where applicable), but that detects, for example, variables and functions defined either in a library that user uses or by user himself. This kind of syntax highlighting should also detect local variables or functions and variable shadowing and distinguish them correctly.

Scope of the work only covers developing a proposed methodology and proof of concept creation, the actual feasibility of this project in area of increased programmer’s productivity is out of scope as we do not possess resources or time to execute any formal tests of the resulting product. Therefore we can only speculate on the usefulness of such “smart” highlighting, and the usefulness would definitely vary from programmer to programmer, as syntax highlighting is highly personal topic and some programmers do not perceive even basic kind of syntax highlighting as useful(Åkesson, 2007).

Proof of concept should provide a functional version of “smart” syntax highlighting, which is integrated with existing syntax highlighting tools of chosen IDE. Success or failure of proof of concept would indicate the soundness of proposed approach and would, hopefully, pave the road for future implementors of similar systems for other Integrated Development Environments and languages.

2 Basis

In this section we derive a proposed designed for a system that performs smart highlighting. Different possibilities are considered and different considerations are listed.

2.1 Syntax highlighting

Syntax highlighting is process of distinguishing certain parts of source code in the editor based on some criteria. First syntax highlighting system was developed for VM operating system for the needs of Oxford English Dictionary computerization. It was called LEXX (Live Parsing Editor) (Cowlinshaw, 1987). Since then most major Integrated Development Environments provide syntax highlighting and many text editors do that too, nowadays (Wikipedia, 2010). Most of the time pattern-matching is used to determine coloration of particular symbol, but sometimes static code analysis is used for these needs. Some languages have strict variable syntax rules (for examples in Erlang all variables start with a capital letter (see figure)), which simplifies the highlighting process, while other (for example Common Lisp) have more uniform syntax, meaning pattern matching is harder to do (see figure).

Visual Code Assist X tool for Microsoft Visual Studio provide advanced syntax highlighting capabilities to default Visual Studio syntax highlighting (colorizer in Visual Studio terms). In addition to Visual Studio default syntax highlighter it provides highlighting for additional code elements, like local symbols(Visual Code Assist X, 2010).

2.2 Auto-competition

Auto-completion (also Code Completion and Microsoft Visual Studio trademarked IntelliSense) is a tool that assist programmer in writing code by providing 'auto-completion possibilities' when user requires them, for example by pressing some key combination. This is usually done after part of some keyword or function name is printed already and programmer wants program to complete it for him, to avoid extra typing and to provide fast way to browse API. For example, lisp symbol *multiple-value-bind* is very long, and programmer might be inclined to type *multiple-* and then press completion combination (M-Tab in Emacs/Slime) and auto-completion will show possible combinations, from which programmer can choose the correct one.

Primitive auto-completion provides only keyword based options, similarly to primitive syntax-highlighting. Auto-completion in modern Integrated Development Environments provide "smart" syntax highlighting, that provides options based on libraries that user uses and by user created code.

We can see that auto-completion used to have limitations similar to syntax highlighting, but nowadays it provides a more advanced options. I believe this can be reused to improve syntax highlighting capabilities as well.

3 Design of smart syntax coloration

As already discussed before, smart syntax highlighting should colorize not only according to the set of keywords, but according to set of symbols defined in scope that user is working in and by taking scoping into account. Therefore syntax highlighting should colorize semantically and therefore different approaches are needed compared to pattern matching method.

In order for the system to be aware of symbols that are currently in the user's scope, system should be able to get symbols somehow, either through language reflection or through some static analysis too. Backend that is used to get possibilities in auto-completion can be used for it. We will call this step "extraction".

As our system has to take scope into account, syntax highlighting system should be able to detect symbols that are defined in local scope (in contrast with extraction that detects "top-level" symbols), such as arguments to the function inside that function definition. We will call that step "scope detection".

And last step is to perform actual coloration. Mainly this should be done using existing tools for syntax highlighting, that are built into the Integrated Development Environment for which smart syntax highlighting is developed.

All these steps can be performed in several different ways and all of them have different advantages and disadvantages. It is important not only to choose the best solution in general, but best solution for each language. This is especially the case since Lisp-like language are homoiconic, meaning that they represent code in the same way they represent data, meaning it is really easier to parse them, but the same is not true for Python or C-like languages, which require much more complex tools to parse their code. On the other hand in Common Lisp there are no special syntax for forms and generic syntax is used in most of the cases, meaning it is not possible to get any "shortcuts" and define rules for special forms as user can easily create more special forms through "macros". This makes scope detection hard. In Python and C-like languages special forms are mostly predefined meaning that this task is easier.

3.1 Extraction

In normal syntax highlighting extraction is done by defining a set of keywords and patterns through which forms are found. This is a good solution if a language have limited set of keywords that are used in one way or for a language that has strict syntax rules, but it won't work for a more "freely" defined language.

Methods that are used to get possible variants for a auto-completion can be used to extract needed symbols.

One possibility is to extract list of current global symbols through language reflection or by using that language running image (for example using running Lisp in case of Common Lisp - this is how SLIME works (Rittweiler, 2008)(SLIME, 2010b)). It is already done when possible completions for a symbol is gathered, both in SLIME and in any of three completion possibilities for python-mode and therefore this step is unlikely to cause any problems.

Sometimes refactoring tools can be used to provide auto-completion possibilities(ropemacs, 2010). Static analysis of the code can be used to extract “tags” from libraries that user make us off and then they can be used to auto-completion and so it can also be used for syntax-highlighting(CTAGS, 2010).

3.2 Scope detection

In languages with lexical scoping, when programmer defines a local variable, it’s definition “shadows” the possible global (or one scoped higher than local scope) definition with itself, meaning that programmer will access it, rather than global variable. This should be taken into account by smart syntax highlighting, as symbols like *type* often have a built-in top level function associated, but they also have a broad English meaning, which means that they might be used a local variable a lot.

In order to detect the correct scope, syntax highlighting has to detect the variables that are shadowed at each certain position. In languages where finite set of special forms define local scope, this shouldn’t cause significant problems, but languages with rich meta-programming capabilities which allow definition of new special forms, this can be a more complex task.

One feasible way to perform scope detection is to go parse the source code from top to bottom, extending the scope when some scope defining forms are encountered and performing coloration when symbol that requires coloration is encountered.

3.3 Coloration

The Integrated Development Environment built-in tools to expand syntax highlighting should be used to implement the coloration of the code. Syntax highlighting should be expandable in a way that allows to instruct colorizer to colorize particular symbols in particular locations, so it should be possible to have some other matching mechanism than simple pattern matching. This is required so that scope can be detected correctly.

4 Proof of concept implementation

Overview of the proof of concept implementation is presented here - Common Lisp smart highlighting in Emacs Integrated Development Environment.

4.1 Basis

The rationale of the choice of Common Lisp and Emacs is explained in this section and libraries and system that are going to be used in the system are reviewed and their operations is briefly described.

4.1.1 Common Lisp

Common Lisp is the most popular actively used Lisp language implementation nowadays. Developed as a successor to Maclisp and Zetalisp, it was supposed to be new Lisp language standard when it was designed. It was standardized by ANSI in 1994(INCITS, 1994).

Several Common Lisp implementations exists, both commercial and open-source and it has been used in many industrial projects(Weinreb, 2010).

Common Lisp is a dynamic language in it's core, but it is possible to provide compiler hints to declare types for variables and function arguments(LispWorks, 2010). Steel Bank Common Lisp compiler Python (not to be mixed with Python programming language) provides complex type inference in addition to that(SBCL, 2010). Many Lisp implementations compile to native code and some of them achieve quite impressive results for a dynamic language, by having performance that is comparable to one of Java(Gat, 2000). Usually types are declared only late in development, when the code have finalized, meaning type declarations are not idiomatic to do on early stage of development.

Common Lisp also features large number of advanced features that are not always found in other languages. One is a macro facility, which allows defining functions that are expanded into code on compile time, for example to provide new control structures. They are idiomatic part of Lisp family of languages and they provide immense meta-programming facilities(McIlroy, 1960)(Steele, 1994)(Graham, 1995)(Norvig, 1992).

Lisp languages are also known for their homoiconicity, meaning data and code is represented the same in Lisp. This is part of the macro power in Lisp, as this allows manipulating code as lists(McIlroy, 1960).

In addition to these features Lisp includes a powerful and innovative object system called CLOS - Common Lisp Object System, which allows multiple inheritance, multiple dispatch and is highly dynamic and customizable. Through so-called Meta-Object Protocol CLOS can be heavily expanded and it's behavior can be heavily modified(Kiczales et al., 1991).

All these features make Lisp a highly interesting language as a study subject. It's homoiconicity and balanced syntax makes parsing easier, and macro system adds challenge to the task of making smart highlighting for it.

4.1.2 Emacs

According to GNU/Emacs website, Emacs is the extensible, customizable, self-documenting real-time display editor(GNU Project, 2010). It is usually characterized for it's almost unlimited extensibility.

Emacs is customizable through so called *modes*. There are major and minor modes and while only one major mode can be active at a time, multiple minor modes can be active. Most programming languages have their own major modes and there are some special major modes, such as major mode for reading email. Major modes provide can provide their own key bindings, menus, behavior and syntax highlighting rules. Minor modes usually add additional features to major mode, for example flyspell-mode can enable spell checking in any other mode(Stallman, 2010).

Another important emacs concept is a *buffer*. For every file opened or created in emacs a buffer is created, also there might be many additional buffers, such as a buffer for directory structure. Only one buffer can be active, or *current* at the same time (even though it is possible to have several of them displayed). When buffer is active, Emacs behaves in accordance to this buffer's setting, and this usually includes buffer mode. In case of file editing, buffer displays file contains and it is possible to save changes to the buffer to opened file or to a new file. *Point* is a location of single character inside the buffer. It is a integer and is determined by counting all characters before the point character. A group of characters between two points are called a *region*, a group of characters that are not separated by space are called a *word*(Stallman, 2010).

Emacs is expanded using Emacs Lisp langauge, which is a high level lisp-like language tailored specifically to be used as an expansion langauge for Emacs. It shares many features with Lisp family of languages, including parenthesized syntax, powerful symbolic manipulations, linked-list based data structures and macro facility. Extremely sophisticated additions to the Emacs have been made using Emacs Lisp language, including a news and mail reader, IRC client, directory manager. Some people jokingly call Emacs an operating system, not an editor, just because of the immense customization capabilities that Emacs provides(Stallman, 2010)(Lewis et al., 1990).

It is important to understand that Emacs is state-machine based in a sense that many operations are using concept of *current-point* - a literate pointer that user uses to edit or input buffer text for example. Therefore often programming style is different from classic functional languages, as it always involves global state. Therefore portability of functions is sometimes harder than one could expect. Several functions are included to save the position of the pointer, so that an addition can manipulate them without hassle to the user(Stallman, 2010)(Cameron et al., 1996).

4.1.3 Font-lock mode for Emacs

Font lock is a emacs minor mode included in all emacs distributions, that provide syntax highlighting capabilities. It provides all the generic tools for major modes authors to create syntax highlighting rules for them(Stallman, 2010).

The basics of font-lock operation is based on 'keywords' - set of special symbols that are meant to be highlighted(Stallman, 2010). Coloration is done in three “passes” - syntactic keyword pass, syntactic pass and regular expression based keyword pass. Syntactic passes are based on Emacs syntax tables and are primarily used for comments and strings, while regular expression based keywords colorize actual language constructs(Font-Lock, 2010).

During the syntactic keyword pass special rules are applied to syntactic elements, that may have variable meaning. For instance quote can begin a string line and also perform some other function. Syntactic keywords are meant to allow mode authors to have more flexibility, as many language are not ambiguous on all their special elements. After syntactic keyword pass, faces are applied to buffer text in according to buffer syntax text syntax table. It uses syntax parsing to determine context of different regions. This is important as strings or comments can span multiple lines. Font-lock documentation claims that this is not usually appropriate for syntactic passes as it works only within given region(Font-Lock, 2010).

Keyword pass applies certain face to some parts of the buffer. It searches with a regular expression or with a given function, withing certain region. This fontifies keywords, reserved words and other mode defined constructs(Font-Lock, 2010).

Here I present the general schema of how functions of font-lock mode are executed when font-lock is performed. We ignore the actual invocation of font-lock and focus on actual font-lock performed. Some functions are omitted; exclamation marks shows functions that are overridable through font-lock configuration.

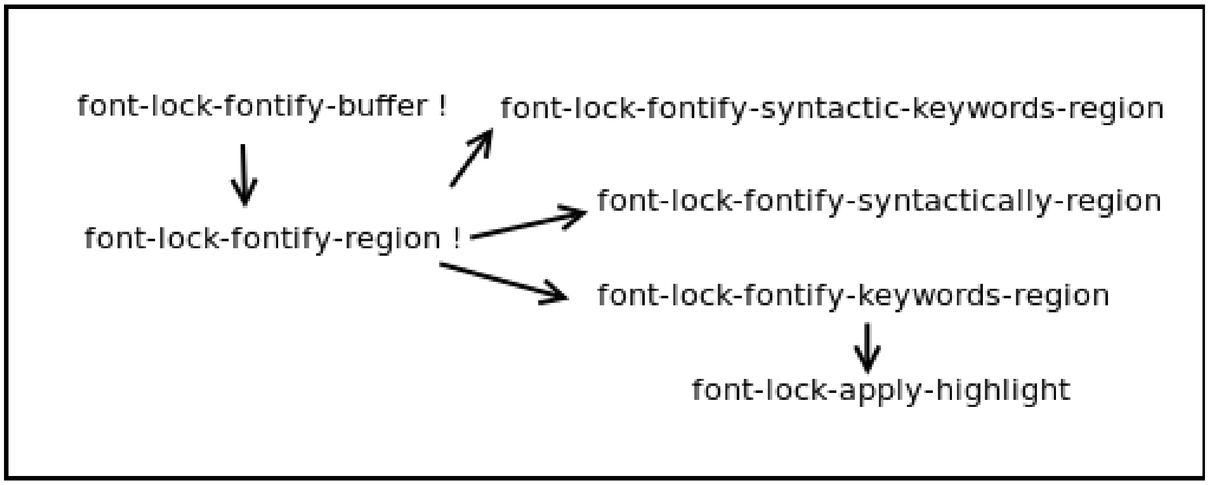

font-lock-fontify-buffer !
font-lock-fontify-syntactic-keywords-region
font-lock-fontify-syntactically-region
font-lock-fontify-region !
font-lock-fontify-keywords-region
font-lock-apply-highlight

Figure 1: Font-lock function flow

Font-lock is customizable through font-lock-defaults variable, that can be overridden per mode. Most of them time just keywords are overridden and font-lock is performed through defaults. A more advanced way to do this is to provide font-lock a searching function, that will itself find the keyword to colorize. If this depth of customization is not enough then even more in-depth modifications can be made by overriding many of the functions that font-lock uses. Unfortunately this set is limited, but we would still later use this possibility to expanded the font-lock to include our system(Stallman, 2010).

4.1.4 Slime

SLIME - Superior Lisp Interaction Mode for Emacs - is an advanced mode for Emacs to improve Common Lisp programming experience on that platform(SLIME, 2010a). It was originally started at 2003 and as of 2008, had more than 40000 lines of code and more than 100 contributors(Rittweiler, 2008).

SLIME provides many features for it's users, such as dynamic compilation and interpretation of code, using the user's Common Lisp platform, display argument list of a function, dynamic symbol auto-competition, Read-Evaluate-Print Loop of the underlying Common Lisp, inspector, debugger, macroexpansion, code disassembling and many other features. It is widely used in Common Lisp community (as well as for some other Lisp dialects) and is de-facto standard as a Common Lisp IDE.

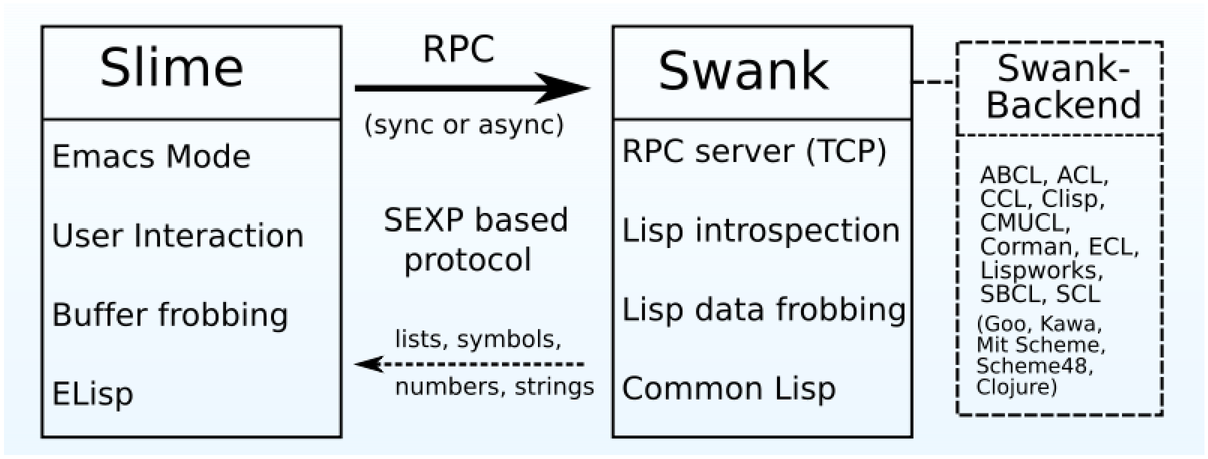


Figure 2: SLIME architecture. (Rittweiler, 2008)

SLIME consists of three distinct system. The Emacs mode itself, SWANK server, which is RPC server, that SLIME calls in order to get some information out of running Lisp, that returns lists, symbols, numbers or strings and SWANK back ends, which are platform dependent back ends for various Common Lisp (and other Lisp dialects, such as Clojure and Scheme)(Rittweiler, 2008).

This architecture allows SLIME to exchange information with Lisp image and therefore retrieve dynamic information out of there for, for example, auto completition. This differs from auto-completion in popular IDE-s, such as Visual Studio, as it doesn't use static code analysis(SLIME, 2010b).

SLIME can be expandable through so called 'slime-contribs' - user contributed extension that can be loaded on SLIME startup. SWANK also provides possibility of expansion through special macro that defines function that is callable from emacs lisp SLIME(SLIME, 2010a).

4.2 Application of general design to use case

Implementation design is presented in this section and general use case is applied to Common Lisp and Emacs with SLIME.

4.2.1 General overview

As we have seen previously most of the syntax highlighting consists of two distinct steps: getting information about what symbols to colorize (I will call it "extraction") and finding these symbols and coloring them ("coloring"). I am going to add one more step to those two - detection of the current scope in order to see if global symbol that is found should be colored or not (I will call this step "scope detection").

Our general approach would be to extract the global environment through extraction and then scan all s-expressions symbol-by-symbol, detecting local scope where applicable and colorizing where applicable too. This approach would combine scope detection with coloration.

4.2.2 S-expression motion

As our approach includes scanning (parsing) of s-expressions, we would need to find out the way to reliably scan every element of s-expression.

Emacs has several built-in functions to move through s-expressions and balanced lists - functions to go forward (next element), backward (previous element), down (inside next s-expression), up (outside current s-expression). Unfortunately they are all not fail-safe and will signal and error in any unexpected condition - forward would fail if we are located at the last element of the list, for example. Therefore we need to catch these errors and interpret them correctly, in order to parse the s-expression correctly.

4.2.3 Extraction

Usually in Emacs modes symbol extraction is done by mode creator, who supplies regular expressions that are used to find keywords in code and apply coloration to it. This is a good solution if a language have limited set of keywords that are used in one way, but it can fail even in simple situations.

With SLIME it would be possible to get list of symbols currently loaded into the running Lisp image and use them to do the coloration. SLIME can also provide classification for the symbol (for example if they are functions, classes or macros). By default SLIME returns fully qualified symbols, meaning symbols with package identifier. Symbol extractor would have to separate package definition from the symbols.

However symbols would only get into the Lisp image when they are interpreted or compiled, meaning that system should take that into the account and colorize on compilation or SLIME connection so that the results would be correct. Before code is loaded, *:common-lisp* package, which is a default package for Common Lisp would be used and before the SLIME connection built-in font-lock can be used.

Environment would be presented as an association list, where symbol names are keys and list of it's properties are values. For example ("list" (:fboundb :boundp)) would mean that symbol *list* is a bound as a function and as a variable in this environment.

4.2.4 Scope detection

We are going to parse the s-expressions ourself, through s-expression motion functions of Emacs. The environment would be carried along as an argument to functions making coloration and functions would recourse when they encounter a new s-expression, meaning that environment can be expanded in case special form would be detected. Environment would be expanding by adding more keywords to the environment association list. Coloring would also be performed at this stage, and would be invoked when symbol that is suspected to be colorizable is encountered.

The reason why I think that syntactic parsing is a superior option is that it can detect scope while it does it, so there is no need for separate scope checker, it can be built into parser. As syntactic parser parses from top level, it is possible to track what is a current scope level and which symbols are shadowed on it. This is much harder to implement with “normal” keyword parsing because there is no guarantee it will pass whole file to the function at once. It might be possible to use closures to track of state, but as Emacs Lisp have dynamic scope [explain what it means] it might produce different problems.

4.2.5 Coloring

When symbol is encountered, system would attempt to detect whether this system is a special form and if it is not then it would attempt to colorize it using the environment. Local properties would have precedence over global ones, to take symbol shadowing into account. Position of the symbol is a s-exp would indicate whether it should “tried” as a function or as a symbol.

In background, some font-lock functions would be overridden. Unfortunately font-lock only provides graceful overrides for some functions and it doesn’t provide such over ride for keyword function only - one can either supply *font-lock-apply-region-function* or provide a search function for it. Former doesn’t satisfy us because we would want to reuse the syntactic pass and latter doesn’t satisfy us because our system has to maintain state in order to keep track of local scope. So we would have to re-implement the function and it would be used only when slime-mode is on. We would have to override the *font-lock-keyword-region* and apply the highlight manually with *font-lock-apply-highlight*.

4.3 Implementation overview

Functionally system has several distinct parts in code. There are functions that work on environment - it can do the extraction of environment, expand environment with new symbols and get symbols from environment. This section provide extraction part and part of scope detection part. Extracts of the codes would be presented in this chapter to improve understanding and full source code is available in Appendix 2.

Extraction step is implemented through a small addition to SWANK server interface - new interface function *package-symbols* was created that returns an association list of symbols that are interned into current package along with it’s classification. Function *slime-dfl-global-env* in the actual system code uses that interface to get the list of package symbols in current function (see figure).

The s-expression scanning part is the main part of the program, which performs actual s-expression scanning. It has functions to parse normal s-expression and also lambda list, lambda form and defining forms (such as defun). The entry point is function *slime-dfl-scan-region*, which takes two arguments - *beg* (beginning) and *end* and, starting from beginning, calls *scan-dlf-scan-sexp* with global environment. When point reaches end or file ends, the function exits.

Figure 3: package-symbols and slime-dfl-global-env

```
;; swank-dynamic-font-lock.lisp
(defslimefun package-symbols (package)
   (let ((result '())
         (package (or (parse-package package) cl-package)))
    (do-symbols (s package)
      (push (cons s (classify-symbol s)) result))
    (remove-duplicates result)))

;; slime-dynamic-font-lock.el
(defun slime-dfl-global-env ()
  (let ((symbols (slime-dfl-package-symbols (slime-current-package))))
    (dolist (symbol symbols symbols)
      (rplaca symbol
              (cadr (split-string
                     (symbol-name (car symbol)) "::\\|:"))))))
```

To capture typical case needed for s-expression scanner - scan each element of the list, while taking empty expression possibility into account and exiting gracefully when the expression end macro *slime-dlf-with-sexp-scanning* was introduced. It wraps all the forms that it is given as an argument into a while loop that loops until the s-expression finishes. It allows provides local variable *first-element* for body forms to use, if they need it, and it doesn't start the while loop in case empty expression is being scanned. Every time body forms would be executed, the point would be at the new element of the list.

Figure 4: slime-dfl-with-sexp-scanning

```
(defmacro slime-dfl-with-sexp-scanning (&rest body)
  `(let ((continue 't)
         (first-element 't))
     (condition-case empty-list
         (forward-sexp)
       (error
        (setf continue '())
        (up-list)))
     (while continue
       ,@body
       (condition-case end-of-list
           (when continue
             (forward-sexp))
         (error
          (setf continue '())
          (up-list)))
       (setf first-element '()))))
```

slime-dfl-scan-sexp just goes down the list (so inside next s-expression) and then calls *slime-dfl-scan-rest-sexp*, which performs actual s-expression scanning. This scanning uses pattern captured in the above-mentioned macro, and body of the form takes element of the list at current point and if it is a list it would go into this list by recursively calling *slime-dfl-scan-sexp*. If it is indeed an atom, then if this atom is not a first element of current s-expression, then the system tries to colorize it as a variable, and if it is a first-item, system first checks whether it is one of predefined special forms, and if it is it colorizes the atom as a function and calls *slime-dfl-scan-special-form* to parse the rest of special form and if it is not a special form, then it just colorizes the atom as a function.

Figure 5: slime-dfl-scan-rest-sexp

```
(defun slime-dfl-scan-rest-sexp (env)
  (slime-dfl-with-sexp-scanning
   (let ((current (slime-dfl-sexp-at-point)))
     (if (slime-dfl-atom current)
         (if first-element
             (let ((special-form (slime-dfl-find-special-form current)))
               (if special-form
                   (progn
                     (slime-dfl-colorize env current :fboundp)
                     (setf continue '())
                     (slime-dfl-scan-special-form env special-form))
                 (slime-dfl-colorize env current :fboundp)))
           (slime-dfl-colorize env current :boundp))
       (progn
         (backward-sexp)
         (slime-dfl-scan-sexp env))))))
```

slime-dfl-scan-special-form just checks which kind of special form this is and then calls it. Most special form parsing functions just get their lambda list, use *slime-dfl-scan-lambda-list* to extract locally scoped variables that they introduce and then use *slime-dfl-scan-rest-sexp* with expanded environment to parse the remainder of the special form (see figure). In addition they may colorize some standard elements of the special form, for example in case of *defun* special form, the name of defined function.

Figure 6: Example of the special form scanning function

```
(defun slime-dfl-scan-defining-form (env)
  (forward-sexp)
  (let ((current (slime-dfl-sexp-at-point)))
    (slime-dfl-apply-face current :defined-name))
  (slime-dfl-scan-lambda-form env))
```

slime-dfl-scan-lambda-list is a very complex function, as it has to at the same time scan and colorize lambda list and expand the environment. Lambda list follow quite strict semantics [HYPERSPEC link], therefore it is possible to take care of all possible corner cases. As with *slime-dfl-scan-rest-sexp*, the function uses *slime-dfl-with-sexp-scanning* to do the s-expression scanning, but in it's body forms it expands the environment together with coloring and it takes some specifics of lambda-lists into detail - for example lambda-list keywords - symbols starting with &, are treated separately (see figure).

Figure 7: Lambda list parsing

```
(defun slime-dfl-scan-lambda-list (env)
  (down-list)
  (let ((new-env '()))
    (slime-dfl-with-sexp-scanning
     (let ((current (slime-dfl-sexp-at-point)))
       (if (slime-dfl-atom current)
           (if (char-equal (elt current 0) ?\&)
               (slime-dfl-apply-face current :lambdalistkeyword)
             (setf new-env (slime-dfl-extend-env new-env current :lexboundp)))
         (progn
           (backward-sexp)
           (down-list)
           (forward-sexp)
           (let ((current (slime-dfl-sexp-at-point)))
             (setf new-env (slime-dfl-extend-env new-env current :lexboundp)))
           (slime-dfl-scan-rest-sexp env)))))
    new-env))
```

Colorizing function *slime-dfl-colorize* accepts 3 arguments - the atom, current environment and the way to colorize it. When it is asked to colorize as a function or a variable it first checks this function is defined as a local function or variable in environment and if it is not then looks if it is defined as a global function or variable. Then it calls *slime-dfl-apply-face* that calls font-lock *font-lock-apply-highlight* to perform actual coloration.

System is integrated with font-lock and slime through slime-autoloads, which allow adding contribs to SLIME easily through configuration. In addition to that re-definable settings are provided for faces to be used to colorize different symbols and for special forms that should be detected. From font-lock side system just overrides default font-lock functions. Most code of the function is the same, but relevant changes are made. Unfortunately font-lock doesn't provide any more low-level functions to be overridden and therefore some code just had to be reused without the change from font-lock mode, which hurts re-usability and maintainability of the code.

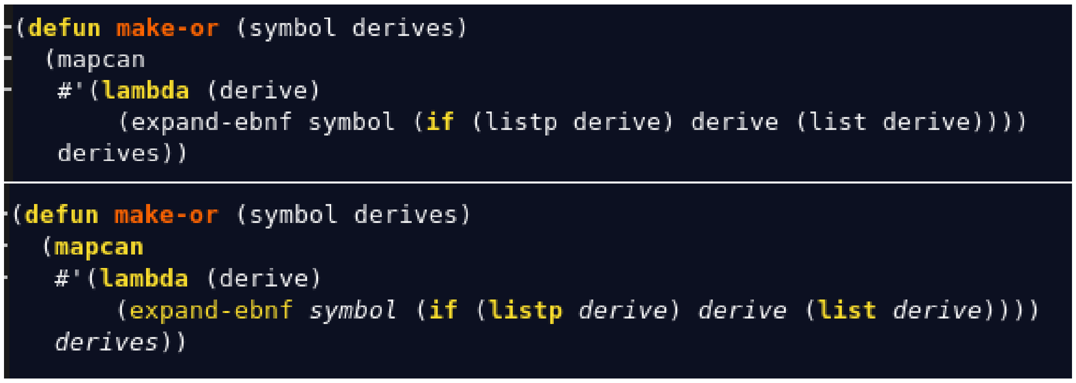
Function *slime-dfl-fontify-region* is a function that overrides font-lock default *font-lock-fontify-region* function. It checks if SLIME has successfully connected the list and if setting *slime-dfl-*

colorize-dynamically is true and then calls *slime-dfl-scan-region*. Otherwise it just calls default font-lock function for keyword coloration is called.

4.4 Implementation results

The system was implemented in according to the plan. System integrates with Emacs font-lock and SLIME-mode and provide dynamic coloration with scope detection.

Basic comparison of standard font-locked code and code with implemented smart highlighting.

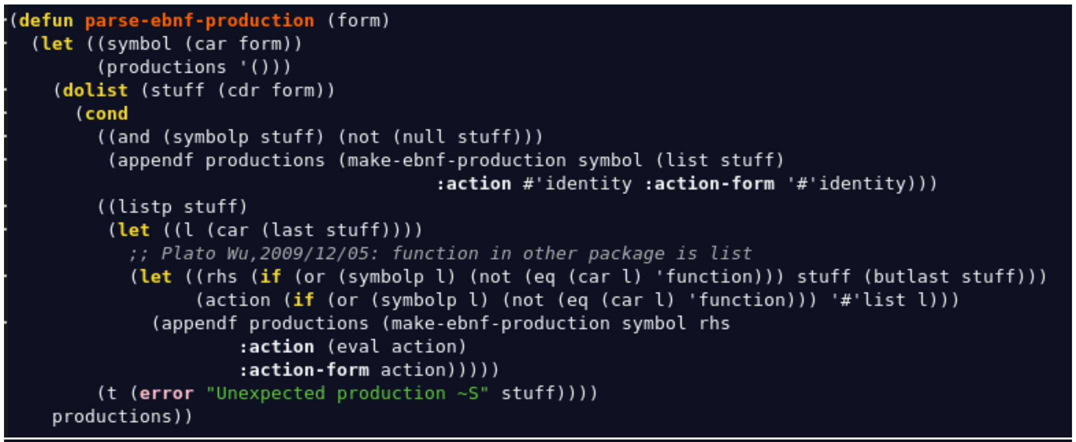


```
(defun make-or (symbol derives)
  (mapcan
   #'(lambda (derive)
       (expand-ebnf symbol (if (listp derive) derive (list derive))))
   derives))
```

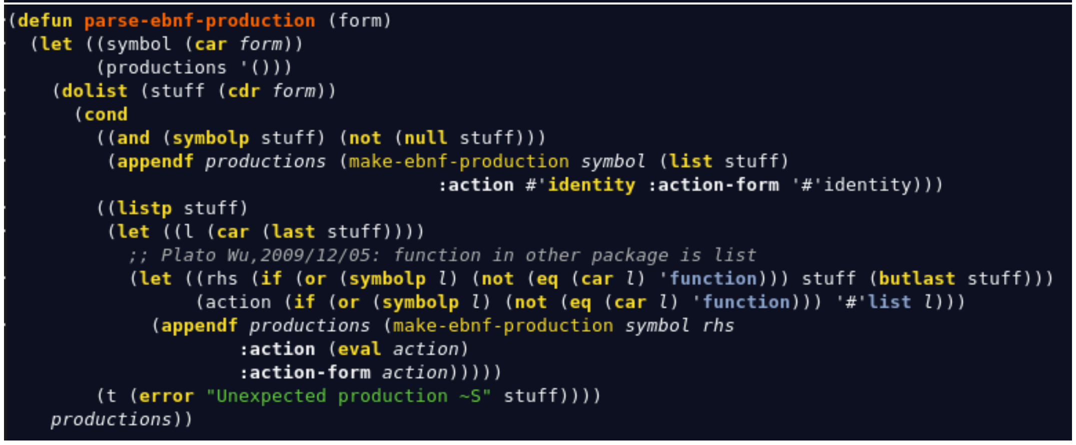
```
(defun make-or (symbol derives)
  (mapcan
   #'(lambda (derive)
       (expand-ebnf symbol (if (listp derive) derive (list derive))))
   derives))
```

As you see in normal mode even many lisp functions from base package are not colored. Not only new system colorizes those, but it also detected function local to current package *expand-ebnf* and colorized it accordingly and colorized function arguments *symbol* and *derives* correctly inside the code.

Here I present a more complex example which illustrates how little original mode does in the coloration comparing to the new mode.



```
(defun parse-ebnf-production (form)
  (let ((symbol (car form))
        (productions '()))
    (dolist (stuff (cdr form))
      (cond
        ((and (symbolp stuff) (not (null stuff)))
         (appendf productions (make-ebnf-production symbol (list stuff)
                                                    :action #'identity :action-form '#'identity)))
        ((listp stuff)
         (let ((l (car (last stuff))))
           ;; Plato Wu,2009/12/05: function in other package is list
           (let ((rhs (if (or (symbolp l) (not (eq (car l) 'function))) stuff (butlast stuff)))
                 (action (if (or (symbolp l) (not (eq (car l) 'function))) '#'list l)))
             (appendf productions (make-ebnf-production symbol rhs
                                                        :action (eval action)
                                                        :action-form action)))))
        (t (error "Unexpected production ~S" stuff))))
    productions))
```



```
(defun parse-ebnf-production (form)
  (let ((symbol (car form))
        (productions '()))
    (dolist (stuff (cdr form))
      (cond
        ((and (symbolp stuff) (not (null stuff)))
         (appendf productions (make-ebnf-production symbol (list stuff)
                                                    :action #'identity :action-form '#'identity)))
        ((listp stuff)
         (let ((l (car (last stuff))))
           ;; Plato Wu,2009/12/05: function in other package is list
           (let ((rhs (if (or (symbolp l) (not (eq (car l) 'function))) stuff (butlast stuff)))
                 (action (if (or (symbolp l) (not (eq (car l) 'function))) '#'list l)))
             (appendf productions (make-ebnf-production symbol rhs
                                                        :action (eval action)
                                                        :action-form action)))))
        (t (error "Unexpected production ~S" stuff))))
    productions))
```

4.5 Evaluation

The main goal have been reached - the proof of concept has been created that provides smart syntax highlighting for dynamic languages through the method that matches described in theoretical part of this paper. Unfortunately this is only a proof of concept and it is not recommended for production.

The main problem is performance - unfortunately smart highlighting noticeably slows down the editor, because optimization was out of scope and font-lock updates on any buffer change, which of course slows down the editor, as s-expressions have to be rescanned every time. It might be possible to optimize by disabling default *on-buffer-change* update and providing updates on buffer save or code compile. Unfortunately this includes even bigger modification of base font-lock functions and we have decided to leave this out of scope.

Unfortunately system failed to auto-detect forms that create local scope in Lisp, as it proved to be more complex than we have considered. Current system of default forms with user defined parts works well enough for most of the cases, and auto-detection of special forms is seldom needed in implementations for non-lisp languages, so we have decided to leave it out of the project. SLIME indentation contrib might provide a good base for such projects.

Some advanced Lisp macros and special forms have a very complex syntax that is hard to parse through the means provided. Therefore they are left out of scope of the project and they are parsed in a very naïveway. Also we tried to include as many special forms to the special form definition, but it we inevitably missed some of them.

In general system proved that concept is a viable one and I believe it greatly increases the productivity of the programmer, especially in a language such as Lisp. Most probably this would be of less use for languages with stricter syntax, but it is a great asset for languages like Lisp.

5 Recommendations and further research

Concept have proved to be a viable one and we believe that it would be worthy to make this system production ready and to implement similar system for languages like Ruby and Python. It also would be an interesting project to implement such system in other IDE, such as Eclipse.

Current project should be improved to detect Lisp special forms and provide a better interface for customization for use in situation where automatic detection does not work. As already mentioned above, SLIME indentation contrib is likely to provide a good base for such development.

SLIME contribs can also be studied more in order to improve the parse process. There is a defunct now parser implementations in SLIME contribs that possibly could be remade and then used in this system. In addition to that definition of special forms in bulky and cumbersome - basically one have to redefine special form parsing function when new type of special forms is to be parsed. Some sort of parser generator based on abstract grammar DSL can be used to improve that.

Performance problems should be solved, both through optimization of existing code and modification of update rate, so that s-expressions are not rescanned too much. It might be that techniques such as caching or memoization could improve the performance too.

Study about actual effect of this system on programmers' productivity should be made in order to see if the any of these further research are worth it.

6 Conclusions

The system to do smart highlighting have been implemented, confirming the validity of the proposed design. This design can be later used to improve syntax highlighting for other languages and in other IDEs and can be basis for further research on this topic, like the research of the usefulness of this addition for programmer's productivity.

This study also showed how important reflective features of the languages are. Reflection can not only be used as a method to solve programming problems, but also as a method to significantly improve the environment programmer is working is and the more such systems are implemented the more, hopefully, productive the programmer would be.

I hope that this thesis would push other programmers forward, to develop new, powerful IDEs for dynamic languages or adding support for them in existing ones, thus improving user experience with such languages. Rapid prototyping and development are getting more and more important in the evolving programming world and I believe that dynamic languages will play even bigger role in them, thus demand for development tools tailored for them would be even higher than now.

References

Linus Åkesson. A case against syntax highlighting. 2007. URL http://www.linusakesson.net/programming/syntaxhighlighting/index.php.

M Bächle and P Kirchberg. Ruby on rails. *IEEE SOFTWARE*, 24, 2007.

D Cameron, B Rosenblatt, and E Raymond. *Learning GNU Emacs, Second Edition*. O'Reilly, 1996.

M Cowlinshaw. Lexx - a programmable structured editor. *IBM Journal of Research and Development*, 31, 1987.

CTAGS. Exuberant ctags website, 2010.

Eclipse. Eclipse project website. 2010. URL http://www.eclipse.org/.

Font-Lock. font-lock mode source code, 2010.

E Gat. Point of view: Lisp as an alternative to java. *Intelligence*, 11, 2000.

GNU Project. Gnu emacs website. 2010. URL http://www.gnu.org/software/emacs/.

P Graham. *ANSI Common Lisp*. Prentice Hall, 1995.

T Grimm. Rapid, efficient and easy prototyping. *Time-Compression Technologies*, 2005.

INCITS. *ANSI/INCITS 226-1994 Programming Language Common Lisp*. ANSI, 1994.

IronPython. Ironpython studio. 2010. URL http://ironpythonstudio.codeplex.com/.

G Kiczales, D Bobrow, and J des Rivieres. *The Art of Metaobject Protocol*. MIT Press, 1991.

M Kindborg. Better software with dynamic languages. 2010.

B Lewis, D LaLiberte, and R Stallman. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, 1990.

LispWorks. *Common Lisp HyperSpec*. LispWorks, 2010. URL http://www.lispworks.com/documentation/common-lisp.html.

J McCarty. *History of LISP*. ACM, 1978.

M Mcclendon, L Regot, and G Akers. The analysis and prototyping of effective graphical user interfaces. 1996.

D McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3, 1960.

P Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.

P Norvig. A retrospective on paradigms of ai programming. 2002. URL `http://www.norvig.com/Lisp-retro.html`.

PyDev. Pydev project website. 2010. URL `http://pydev.org/`.

PyPy Project website. Performance overview. 2010. URL `http://speed.pypy.org/overview/`.

Python Software Foundation. Python success stories. 2010. URL `http://python.org/about/success/`.

T Rittweiler. Slime - superior lisp interaction mode for emacs presentation. Munich Lisp Group, 2008.

ropemacs. ropemacs, rope in emacs website, 2010. URL `http://rope.sourceforge.net/ropemacs.html`.

SBCL. Steel bank common lisp website. 2010. URL `http://sbcl.sourceforge.net/`.

SLIME. Slime project website. 2010a. URL `http://common-lisp.net/project/slime/`.

SLIME. Slime source code, 2010b.

R Stallman. *GNU Emacs Manual*. Free Software Foundation, 2010.

G Steele. *Common Lisp the Language, 2nd Edition*. Digital Press Bedford, MA, 1994.

Visual Code Assist X. Visual code assist x features. 2010. URL `http://www.wholetomato.com/products/default.asp`.

Visual Studio. Visual studio feature list. 2010. URL `http://www.microsoft.com/visualstudio/`.

Daniel Weinreb. Common lisp implementations: A survey. 2010. URL `http://common-lisp.net/~dlw/LispSurvey.html`.

Wikipedia. Comparison of text editors. 2010. URL `http://en.wikipedia.org/wiki/Comparison_of_text_editors#Programming_features`.

Appendices

Appendix 1: End user documentation

Slime-dfl is a SLIME contrib that provides smart syntax highlighting for Common Lisp code. It gets symbols interned into package through SWANK server and then parses the code and colorizes things, while taking local scope into account. System is highly experimental and is not yes tested thoroughly.

Installation

Either add

```
(add-to-list 'load-path "<directory-of-this-file>")
(add-hook 'slime-load-hook (lambda () (require 'slime-dfl)))
```

to your .emacs or add *slime-dfl* to *slime-setup*.

Configuration

Several custom variables are provided that can be modified by the user to tailor the system to himself.

slime-dfl-colorize-dynamically *Boolean*
: If this variable is nil then smart highlighting won't be performed. Useful for temporarily disabling smart coloration.

slime-dfl-use-advanced-faces *Boolean*
: slime-dfl provides a set of additional faces to cover all types of symbols that can be colored. If this is non-nil that those advanced faces are used, but as they are non-standard there is no definition for them in normal color-themes, so user is responsible for adding them himself. If this is nil then basic font-lock faces are used, but this means that much coloration would be of the same color.

slime-dfl-faces *Alist*

Alist of (symbol-type face-name). This is used when *slime-dfl-advanced-faces* is nil.

slime-dfl-advanced-faces *Alist*

Alist of (symbol-type face-name). This is used when *slime-dfl-advanced-faces* is non-nil. Allowed symbol-types - :global-function, :global-variable, :global-type, :global-package-function, :global-package-variable, :global-package-type, :local-variable, :local-function, :defined-function, :defined-variable, :defined-type, :lambdakeyword, :keyword.

slime-dfl-special-forms *Alist*

Alist of (special-form special-form-type). Special form is a string with a symbol that works as this special form. This is used to parse special forms and macros. User can add new special-forms/macros here to make system parse them correctly. Allowed special-form-types - :defining, :type, :constant, :lambda, :lambdaf, :double-lambda.

Appedix 2: Source code listing

```
-------------------- swank-dfl.lisp --------------------
;;; swank-dfl.lisp --- Smart syntax highlighting for SLIME
;;
;; Authors: Mikhail Novikov <freiksenet@gmail.com>
;;
;; License: Public Domain
;;
;;;

(in-package :swank)

(defslimefun package-symbols (package)
  "Gets and classifies symbols from given package or cl-package if given package does not exist."
  (let ((result '())
        (package (or (parse-package package) cl-package)))
    (do-symbols (s package)
      (push (cons s (classify-symbol s)) result))
    (remove-duplicates result)))
------------------------- EOF -------------------------



--------------------- slime-dfl.el ---------------------
;;; slime-dfl.el --- Smart syntax highlighting for SLIME
;;
```

```
;; Authors: Mikhail Novikov <freiksenet@gmail.com>
;;          some code is from font-lock.el of GNU Emacs
;;
;; License: GNU GPL (same license as Emacs)
;;
;;; Installation
;;
;; Add this to your .emacs:
;;
;;   (add-to-list 'load-path "<directory-of-this-file>")
;;   (add-hook 'slime-load-hook (lambda () (require 'slime-dfl)))
;;
;;;

;;; Customization

(defcustom slime-dfl-colorize-dynamically t
  "Determines if dynamic coloration should be used."
  :group 'slime-mode
  :type 'boolean)

(defcustom slime-dfl-use-advanced-faces ()
  "Determines whether to use advanced faces or not."
  :group 'slime-mode
  :type 'boolean)

(defcustom slime-dfl-faces
  '((:global-function font-lock-keyword-face)
    (:global-variable font-lock-constant-face)
    (:global-type font-lock-type-face)
    (:global-package-function font-lock-keyword-face)
    (:global-package-variable font-lock-constant-face)
    (:global-package-type font-lock-type-face)
    (:local-variable font-lock-variable-name-face)
    (:local-function font-lock-function-name-face)
    (:defined-function font-lock-function-name-face)
    (:defined-variable font-lock-variable-name-face)
    (:defined-type font-lock-type-face)
    (:lambdakeyword font-lock-type-face)
    (:keyword font-lock-builtin-face))
  "Determines faces to use to colorize each type of variable."
  :group 'slime-mode
  :type '(alist :key-type symbol :value-type face))

(defcustom slime-dfl-advanced-faces
  '((:global-function slime-dfl-faces-global-function)
    (:global-variable slime-dfl-faces-global-variable)
    (:global-type slime-dfl-faces-global-type)
    (:global-package-function slime-dfl-faces-global-package-function)
    (:global-package-variable slime-dfl-faces-global-package-variable)
    (:global-package-type slime-dfl-faces-global-package-type)
    (:local-variable slime-dfl-faces-local-variable)
    (:local-function slime-dfl-faces-local-function)
    (:defined-function slime-dfl-faces-defined-function)
    (:defined-variable slime-dfl-faces-defined-variable)
    (:defined-type slime-dfl-faces-defined-type)
    (:lambdakeyword slime-dfl-faces-lambdakeyword)
    (:keyword slime-dfl-faces-keyword))
  "Determines faces to use in advanced mode."
  :group 'slime-mode
  :type '(alist :key-type symbol :value-type face))

(defcustom slime-dfl-special-forms
  '(("defun" :defining)
    ("defmacro" :defining)
    ("defgeneric" :defining)
    ("defmethod" :defining)
    ("define-compiler-macro" :defining)
    ("define-modify-macro" :defining)
    ("define-setf-expander" :defining)
    ("defsetf" :defining)
    ("deftype" :type)
    ("defpackage" :type)
    ("define-condition" :type)
    ("defstruct" :type)
    ("defclass" :type)
    ("defconstant" :constant)
    ("defparameter" :constant)
    ("defvar" :constant)
    ("let" :lambda)
    ("let*" :lambda)
    ("flet" :lambdaf)
    ("labels" :lambdaf)
    ("lambda" :lambda)
    ("multiple-value-bind" :double-lambda))
  "Determines special forms and their types. Valid types are :defining, :type,
:constant, :lambda, :lambdaf, :double-lambda."
  :group 'slime-mode
  :type '(alist :key-type string :value-type symbol))

;;; Faces definiton and face variables

(defgroup slime-dfl-faces '()
  "Group for slime-dfl faces"
  :group 'slime)
```

```
(defface slime-dfl-faces-global-function '()
  "Face for highlighting global function symbols."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-global-variable '()
  "Face for highlighting global variable symbols."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-global-type '()
  "Face for highlighting global class or type symbols."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-global-package-function '()
  "Face for highlighting global function symbols that are defined in current package."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-global-package-variable '()
  "Face for highlighting global variable symbols that are defined in current package."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-global-package-type '()
  "Face for highlighting global types or classes that are defined in current package."
  :group 'slime-dfl-faces)
```

```
(defface slime-dfl-faces-local-function '()
  "Face for highlighting local function symbols."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-local-variable '()
  "Face for highlighting local variable symbols."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-defined-function '()
  "Face for highlighting names of functions being defined."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-defined-variable '()
  "Face for highlighting names of variables being defined."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-defined-type '()
  "Face for highlighting names of types being defined."
  :group 'slime-dfl-faces)

(defface slime-dfl-faces-lambdakeyword '()
  "Face for highlighting lambda list keywords."
  :group 'slime-dfl-faces)
```

```
(defface slime-dfl-faces-keyword '()
  "Face for highlighting keywords."
  :group 'slime-dfl-faces)

(defvar slime-dfl-faces-global-function 'slime-dfl-faces-global-function
  "Face for highlighting global function symbols.")

(defvar slime-dfl-faces-global-variable 'slime-dfl-faces-global-variable
  "Face for highlighting global variable symbols.")

(defvar slime-dfl-faces-global-type 'slime-dfl-faces-global-type
  "Face for highlighting global class or type symbols.")

(defvar slime-dfl-faces-global-package-function 'slime-dfl-faces-global-package-function
  "Face for highlighting global function symbols that are defined in current package.")

(defvar slime-dfl-faces-global-package-variable 'slime-dfl-faces-global-package-variable
  "Face for highlighting global variable symbols that are defined in current package.")

(defvar slime-dfl-faces-global-package-type 'slime-dfl-faces-global-package-type
  "Face for highlighting global types or classes that are defined in current package.")

(defvar slime-dfl-faces-local-function 'slime-dfl-faces-local-function
  "Face for highlighting local function symbols.")
```

```
(defvar slime-dfl-faces-local-variable 'slime-dfl-faces-local-variable
  "Face for highlighting local variable symbols.")

(defvar slime-dfl-faces-defined-function 'slime-dfl-faces-defined-function
  "Face for highlighting names of functions being defined.")

(defvar slime-dfl-faces-defined-variable 'slime-dfl-faces-defined-variable
  "Face for highlighting names of variables being defined.")

(defvar slime-dfl-faces-defined-type 'slime-dfl-faces-defined-type
  "Face for highlighting names of types being defined.")

(defvar slime-dfl-faces-lambdakeyword 'slime-dfl-faces-lambdakeyword
  "Face for highlighting lambda list keywords.")

(defvar slime-dfl-faces-keyword 'slime-dfl-faces-keyword
  "Face for highlighting keywords.")

;;; Global variables

(defvar slime-dfl-used-faces '()
  "Faces that are used for coloration now.")
```

```
;;; Utilities

(defun slime-dfl-find-special-form (name)
  "Returns special form definition if one exists."
  (assoc name slime-dfl-special-forms))

(defun slime-dfl-string-trim-left (string chars)
  "Trims string of any of 'char' from the left."
  (let ((trimmed string))
    (while (and (member (aref trimmed 0) chars)
                (> (length trimmed) 1))
      (setf trimmed (substring trimmed 1)))
    trimmed))

(defun slime-dfl-sexp-at-point ()
  "Gets current s-expression or atom at the point."
  (save-excursion
    (backward-sexp)
    (let ((sexp (slime-sexp-at-point)))
      (if sexp
          sexp
        ; AVDH - A Very Dirty Hack
        "!----UNREADABLE-SEXP----!"))))
```

```
(defun slime-dfl-atom (string)
  "Checks if string is an s-expressions or an atom."
  (let ((trimmed (slime-dfl-string-trim-left string '(?\# ?\' ?\` ?\,))))
    (not (and
          (> (length trimmed) 1)
          (char-equal (aref trimmed 0) ?\( )
          (char-equal (aref trimmed (1- (length trimmed))) ?\) )))))

(defun slime-dfl-function (string)
  "Checks if string has a function prefix."
  (and
   (char-equal (aref string 0) ?\#)
   (char-equal (aref string 1) ?\')))

(defun slime-dfl-keyword (string)
  "Checks if string is a keyword."
  (char-equal (aref string 0) ?\:))

;;; Environment

(defun slime-dfl-global-env ()
  "Returns global envieronment for current buffer package."
  (slime-dfl-get-symbols (slime-current-package)))
```

```
(defun slime-dfl-get-symbols (package)
  "Returns list of symbols with classification for given package."
  (mapcar #'slime-dfl-process-symbol
          (slime-dfl-package-symbols package)))

(defun slime-dfl-package-symbols (package)
  "Returns unprocessed list of symbols for given package."
  (let ((slime-current-thread t))
    (slime-eval
     `(swank:package-symbols ,package))))

(defun slime-dfl-process-symbol (symbol-def)
  "Processes symbol - cuts package off and classifies it in according to system definitions."
  (let ((symbol (symbol-name (car symbol-def)))
        (defs (cdr symbol-def)))
    (let* ((local (slime-dfl-local-symbol-p symbol))
           (symbol (cadr (split-string symbol (if local "::" ":")))))
      (cons symbol
            (mapcar #'(lambda (def)
                        (slime-dfl-process-def def local))
                    defs)))))

(defun slime-dfl-process-def (def local)
  "Processes on classification for a symbol."
  (cond
   ((or (eq def :class) (eq def :typespec))
    (if local :global-package-type :global-type))
   ((or (eq def :fboundp) (eq def :macro) (eq def :generic-function) (eq def :special-operator))
    (if local :global-package-function :global-function))
   ((or (eq def :boundp) (eq def :constant))
    (if local :global-package-variable :global-variable))))

(defun slime-dfl-local-symbol-p (symbol)
  "Checks is symbol is local at the package."
  (string-match-p "\\(\\(|.+|\\)\\|\\([^:]+\\)\\)::\\(\\(|.+|\\)\\|\\([^:]+\\)\\)" symbol))

(defun slime-dfl-get-from-env (env symbol)
  "Gets symbol from environment."
  (assoc symbol env))

(defun slime-dfl-extend-env (env symbol role)
  "Extends symbol in envieronment by adding a new role to it."
  (let* ((new-env env)
         (roles (assoc symbol new-env)))
    (if roles
        (rplacd roles (cons role (cdr roles)))
      (setf new-env (append (list (list symbol role)) new-env)))
    new-env))
```

```
;;; Sexp scanning

(defmacro slime-dfl-with-sexp-scanning (count-first &rest body)
  "Macro that abstracts common pattern for s-expression scanning.

It handles empty lists gracefully, keeps track of first element if needed
 and exists gracefully when s-expression ends."
  `(let ((continue 't)
         (first-element ,count-first))
     (condition-case empty-list
         (forward-sexp)
       (error
        (setf continue '())
        (up-list)))
     (while continue
       ,@body
       (condition-case end-of-list
           (when continue
             (forward-sexp))
         (error
          (setf continue '())
          (up-list)))
       (setf first-element '()))))
```

```
(defun slime-dfl-scan-region (beg end)
  "Scans region from beg to end or until buffer ends."
  (save-excursion
    (goto-char beg)
    (let ((continue 't)
          (global-env (slime-dfl-global-env)))
      (while (and (< (point) end) continue)
        (condition-case end-of-buffer
            (slime-dfl-scan-sexp global-env)
          (error
           (setf continue '()))))))))

(defun slime-dfl-scan-sexp (env)
  "Scans one full s-expression with given environment env."
  (down-list)
  (slime-dfl-scan-rest-sexp env))

(defun slime-dfl-scan-rest-sexp (env &optional ignore-first)
  "Scans rest of s-expression with given environment.

If ignore-first is non-nil then it doesn't keep track of the first symbol
 of the s-expression (if it is started in the middle of one, for example."
  (slime-dfl-with-sexp-scanning (if ignore-first '() 't)
   (let ((current (slime-dfl-sexp-at-point)))
     (if (slime-dfl-atom current)
         (if first-element
             (let ((special-form (slime-dfl-find-special-form current)))
               (if special-form
                   (progn
                     (slime-dfl-colorize env current :function)
                     (setf continue '())
                     (slime-dfl-scan-special-form env special-form))
                 (slime-dfl-colorize env current :function)))
           (if (slime-dfl-function current)
               (slime-dfl-colorize env current :function)
             (slime-dfl-colorize env current :variable)))
       (progn
         (backward-sexp)
         (slime-dfl-scan-sexp env))))))

(defun slime-dfl-scan-special-form (env form)
  "Calls special form scanning function in accordance to the type of that form."
  (let ((type (cadr form)))
    (cond
     ((eq type :type)
      (slime-dfl-scan-simple-form env :defined-type))
     ((eq type :constant)
      (slime-dfl-scan-simple-form env :defined-variable))
     ((eq type :defining)
      (slime-dfl-scan-defining-form env))
     ((eq type :lambda)
      (slime-dfl-scan-lambda-form env :local-variable))
     ((eq type :lambdaf)
      (slime-dfl-scan-lambda-form env :local-function))
     ((eq type :double-lambda)
      (slime-dfl-scan-double-lambda env)))))

(defun slime-dfl-scan-simple-form (env as)
  "Simple form scanning function.
Used when no local scope is created and when form has too complex grammar to parse."
  (forward-sexp)
  (let ((current (slime-dfl-sexp-at-point)))
    (if (slime-dfl-atom current)
        (slime-dfl-apply-face current as)
      (progn
        (backward-sexp)
        (down-list)
        (forward-sexp)
        (let ((current (slime-dfl-sexp-at-point)))
          (slime-dfl-apply-face current as))
        (slime-dfl-scan-rest-sexp env)))))
  (slime-dfl-scan-rest-sexp '()))

(defun slime-dfl-scan-defining-form (env)
  "Scans form that defines some symbol with a lambda list, for example defun or defmacro."
  (forward-sexp)
  (let ((current (slime-dfl-sexp-at-point)))
    (slime-dfl-apply-face current :defined-function))
  (slime-dfl-scan-lambda-form env :local-variable))

(defun slime-dfl-scan-lambda-form (env as)
  "Scans form that defines something with lambda list anonymously."
  (let ((new-bindings (slime-dfl-scan-lambda-list env as)))
    (slime-dfl-scan-rest-sexp new-bindings)))

(defun slime-dfl-scan-double-lambda (env)
  "Scans form that defines something with lambda list anonymously and
 has an additional form after lambda list. For example multiple-value-bind."
  (let ((new-bindings (slime-dfl-scan-lambda-list env :local-variable)))
    (slime-dfl-scan-sexp env)
    (slime-dfl-scan-rest-sexp new-bindings)))

(defun slime-dfl-scan-lambda-list (env as)
  "Scans lambda list and returns extended envieronment."
  (down-list)
 (let ((new-env env))
   (slime-dfl-with-sexp-scanning '()
    (let ((current (slime-dfl-sexp-at-point)))
      (if (slime-dfl-atom current)
          (if (char-equal (elt current 0) ?\&)
              (slime-dfl-apply-face current :lambdakeyword)
              (setf new-env (slime-dfl-extend-env new-env current as)))
        (progn
          (backward-sexp)
          (down-list)
          (forward-sexp)
          (let ((current (slime-dfl-sexp-at-point)))
            (setf new-env (slime-dfl-extend-env new-env current as)))
          (slime-dfl-scan-rest-sexp env 't)))))
   new-env))

;;; Colorizing

(defun slime-dfl-colorize (env current as)
  "Colorizes current atom with envieronemnt as function or variable.
 Local definitions are given priority over package defitions and package
 definitions are given priority over global ones."
  (let ((current (slime-dfl-string-trim-left current '(?\# ?\' ?\` ?\, ?\@))))
    (if (slime-dfl-keyword current)
        (slime-dfl-apply-face current :keyword)
      (let ((defs (cdr (slime-dfl-get-from-env env current))))
        (when defs
          (if (eq as :function)
              (slime-dfl-colorize-check current defs '(:local-function
                                                       :global-package-function
                                                       :global-function))
            (when (eq as :variable)
              (slime-dfl-colorize-check current defs '(:local-variable
                                                       :global-package-variable
                                                       :global-variable
                                                       :global-package-type
                                                       :global-type)))))))))

(defun slime-dfl-colorize-check (current defs check-for)
  "Utility function to handle definition priorities."
  (if (null check-for)
      '()
    (let ((check (car check-for)))
      (if (member check defs)
          (slime-dfl-apply-face current check)
        (slime-dfl-colorize-check current defs (cdr check-for))))))

;; Colorizes current atom
```

```
(defun slime-dfl-apply-face (current-word as)
  "Low-level function to apply face to the buffer element."
  (save-excursion
    (re-search-backward (regexp-opt (list current-word)))
    (font-lock-apply-highlight (list 0 (cadr (assoc as slime-dfl-used-faces))))))

;; Basic keywords from font-lock mode

(defconst lisp-font-lock-keywords-1
  (eval-when-compile
    `(;; Definitions.
      (,(concat "(\\(def\\("
;; Function declarations.
"\\(advice\\|alias\\|generic\\|macro\\*?\\|method\\|"
"setf\\|subst\\*?\\|un\\*?\\|"
"ine-\\(condition\\|"
"\\(?:derived\\|\\(?:global\\(?:ized\\)?-\\)?minor\\|generic\\)-mode\\|"
"method-combination\\|setf-expander\\|skeleton\\|widget\\|"
"function\\|\\(compiler\\|modify\\|symbol\\)-macro\\)\\)\\|"
;; Variable declarations.
"\\(const\\(ant\\)?\\|custom\\|varalias\\|face\\|parameter\\|var\\)\\|"
;; Structure declarations.
"\\(class\\|group\\|theme\\|package\\|struct\\|type\\)"
"\\)\\)\\>"
```

```
;; Any whitespace and defined object.
"[ \t'\(]*"
"\\(setf[ \t]+\\sw+)\\|\\sw+\\)?")
       (1 font-lock-keyword-face)
       (9 (cond ((match-beginning 3) font-lock-function-name-face)
((match-beginning 6) font-lock-variable-name-face)
(t font-lock-type-face))
  nil t))
      ;; Emacs Lisp autoload cookies.  Supports the slightly different
      ;; forms used by mh-e, calendar, etc.
      ("^;;;###\\([-a-z]*autoload\\)" 1 font-lock-warning-face prepend)
      ;; Regexp negated char group.
      ("\\[\\(\\^\\)" 1 font-lock-negation-char-face prepend)))
  "Subdued level highlighting for Lisp modes.")

(defconst lisp-font-lock-keywords-2
  (append lisp-font-lock-keywords-1
   (eval-when-compile
     `(;; Control structures.  Emacs Lisp forms.
       (,(concat
  "(" (regexp-opt
       '("cond" "if" "while" "while-no-input" "let" "let*"
 "prog" "progn" "progv" "prog1" "prog2" "prog*"
 "inline" "lambda" "save-restriction" "save-excursion"
```

```
"save-selected-window" "save-window-excursion"
"save-match-data" "save-current-buffer"
"unwind-protect" "condition-case" "track-mouse"
"eval-after-load" "eval-and-compile" "eval-when-compile"
"eval-when" "eval-at-startup" "eval-next-after-load"
"with-case-table" "with-category-table"
"with-current-buffer" "with-electric-help"
"with-local-quit" "with-no-warnings"
"with-output-to-string" "with-output-to-temp-buffer"
"with-selected-window" "with-selected-frame" "with-syntax-table"
"with-temp-buffer" "with-temp-file" "with-temp-message"
"with-timeout" "with-timeout-handler") t)
 "\\>")
 .  1)
      ;; Control structures.  Common Lisp forms.
      (,(concat
 "(" (regexp-opt
      '("when" "unless" "case" "ecase" "typecase" "etypecase"
"ccase" "ctypecase" "handler-case" "handler-bind"
"restart-bind" "restart-case" "in-package"
"break" "ignore-errors"
"loop" "do" "do*" "dotimes" "dolist" "the" "locally"
"proclaim" "declaim" "declare" "symbol-macrolet"
"lexical-let" "lexical-let*" "flet" "labels" "compiler-let"
```

```
 "destructuring-bind" "macrolet" "tagbody" "block" "go"
 "multiple-value-bind" "multiple-value-prog1"
 "return" "return-from"
 "with-accessors" "with-compilation-unit"
 "with-condition-restarts" "with-hash-table-iterator"
 "with-input-from-string" "with-open-file"
 "with-open-stream" "with-output-to-string"
 "with-package-iterator" "with-simple-restart"
 "with-slots" "with-standard-io-syntax") t)
  "\\>")
  . 1)
       ;; Exit/Feature symbols as constants.
       (,(concat "(\\(catch\\|throw\\|featurep\\|provide\\|require\\)\\>"
 "[ \t']*\\(\\sw+\\)?")
(1 font-lock-keyword-face)
(2 font-lock-constant-face nil t))
       ;; Erroneous structures.
       ("(\\(abort\\|assert\\|warn\\|check-type\\|cerror\\|error\\|signal\\)\\>" 1 font-lock-warning-face)
       ;; Words inside \\[] tend to be for ‘substitute-command-keys’.
       ("\\\\\\\\\\[\\(\\sw+\\)\\]" 1 font-lock-constant-face prepend)
       ;; Words inside ‘’ tend to be symbol names.
       ("‘\\(\\sw\\sw+\\)’" 1 font-lock-constant-face prepend)
       ;; Constant values.
       ("\\<:\\sw+\\>" 0 font-lock-builtin-face)
```

```
;; ELisp and CLisp ‘&’ keywords as types.
("\\<\\&\\sw+\\>" . font-lock-type-face)
;; ELisp regexp grouping constructs
((lambda (bound)
   (catch 'found
     ;; The following loop is needed to continue searching after matches
     ;; that do not occur in strings.  The associated regexp matches one
     ;; of ‘\\\\’ ‘\\(’ ‘\\(?:’ ‘\\|’ ‘\\)’.  ‘\\\\’ has been included to
     ;; avoid highlighting, for example, ‘\\(’ in ‘\\\\(’.
     (while (re-search-forward
             "\\(\\\\\\\\\\)\\(?:\\(\\\\\\\\\\)\\|\\((\\(?:\\?[0-9]*:\\)?\\|[|)]\\)\\)"
             bound t)
       (unless (match-beginning 2)
         (let ((face (get-text-property (1- (point)) 'face)))
           (when (or (and (listp face)
                          (memq 'font-lock-string-face face))
                     (eq 'font-lock-string-face face))
             (throw 'found t)))))))
 (1 'font-lock-regexp-grouping-backslash prepend)
 (3 'font-lock-regexp-grouping-construct prepend))
;;;  This is too general -- rms.
;;;  A user complained that he has functions whose names start with ‘do’
;;;  and that they get the wrong color.
;;;     ;; CL ‘with-’ and ‘do-’ constructs
;;;      ("(\\(\\(do-\\|with-\\)\\(\\s_\\|\\w\\)*\\)" 1 font-lock-keyword-face)
      )))
  "Gaudy level highlighting for Lisp modes.")

(defvar lisp-font-lock-keywords lisp-font-lock-keywords-1
  "Default expressions to highlight in Lisp modes.")

;;; Font-lock stuff

(require 'syntax)

; Function that is needed in order to run font-lock base of slime-dfl-fontify-regon
(eval-when-compile
  ;;
  ;; We don't do this at the top-level as we only use non-autoloaded macros.
  (require 'cl)
  ;;
  ;; Borrowed from lazy-lock.el.
  ;; We use this to preserve or protect things when modifying text properties.
  (defmacro save-buffer-state (varlist &rest body)
    "Bind variables according to VARLIST and eval BODY restoring buffer state."
    (declare (indent 1) (debug let))
    (let ((modified (make-symbol "modified")))
      `(let* ,(append varlist
      ‘((,modified (buffer-modified-p))
(buffer-undo-list t)
(inhibit-read-only t)
(inhibit-point-motion-hooks t)
(inhibit-modification-hooks t)
deactivate-mark
buffer-file-name
buffer-file-truename))
 (unwind-protect
     (progn
       ,@body)
   (unless ,modified
     (restore-buffer-modified-p nil)))))))

; This is copy paste from normal font-lock fontify-keyword + our stuff
(defun slime-dfl-fontify-region (beg end loudly)
  "Modification of font-lock-default-fontify-region function. Checks if SLIME
is connected and then performs smart highlighting."
  (save-buffer-state
      ((parse-sexp-lookup-properties
        (or parse-sexp-lookup-properties font-lock-syntactic-keywords))
       (old-syntax-table (syntax-table)))
    (unwind-protect
(save-restriction
(unless font-lock-dont-widen (widen))
;; Use the fontification syntax table, if any.
(when font-lock-syntax-table
  (set-syntax-table font-lock-syntax-table))
        ;; Extend the region to fontify so that it starts and ends at
        ;; safe places.
        (let ((funs font-lock-extend-region-functions)
              (font-lock-beg beg)
              (font-lock-end end))
          (while funs
            (setq funs (if (or (not (funcall (car funs)))
                               (eq funs font-lock-extend-region-functions))
                           (cdr funs)
                         ;; If there's been a change, we should go through
                         ;; the list again since this new position may
                         ;; warrant a different answer from one of the fun
                         ;; we've already seen.
                         font-lock-extend-region-functions)))
          (setq beg font-lock-beg end font-lock-end))
;; Now do the fontification.
(font-lock-unfontify-region beg end)
(when font-lock-syntactic-keywords
  (font-lock-fontify-syntactic-keywords-region beg end))
(unless font-lock-keywords-only
    (font-lock-fontify-syntactically-region beg end loudly))
          (if (and slime-dfl-colorize-dynamically (slime-connected-p))
              (slime-dfl-scan-region beg end)
            (font-lock-fontify-keywords-region beg end loudly))))
      ;; Clean up.
      (set-syntax-table old-syntax-table)))

;;; SLIME integration

(defun slime-dfl-set-font-lock ()
  "Sets font-lock defaults to include new fontify-region function, thus hooking the font-lock to the system."
  (set (make-local-variable 'font-lock-defaults)
       '((lisp-font-lock-keywords lisp-font-lock-keywords-1 lisp-font-lock-keywords-2)
         nil t (("+-*/.<>=!?$%_&~^:@" . "w")) nil
         (font-lock-mark-block-function . mark-defun)
         (font-lock-syntactic-face-function . lisp-font-lock-syntactic-face-function)
         (font-lock-fontify-region-function . slime-dfl-fontify-region))))

(defun slime-dfl-init ()
  "Initializing function."
  (slime-require :swank-dfl)
  (setf slime-dfl-used-faces
        (if slime-dfl-use-advanced-faces slime-dfl-advanced-faces slime-dfl-faces))
  (add-hook 'lisp-mode-hook 'slime-dfl-set-font-lock)
  (add-hook 'slime-connected-hook 'font-lock-fontify-buffer))

(provide 'slime-dfl)
------------------------- EOF --------------------------
```