POLITECNICO DI TORINO

Facoltà di Ingegneria II
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Implementazione di un motore per la colorazione della sintassi in GtkSourceView

Relatore:
Riccardo Sisto

Candidati:
Emanuele Aina
Marco Barisione

Maggio 2005

Sommario

GtkSourceView è un widget che estende il widget standard di testo di GTK+ 2, con l'aggiunta di colorazione della sintassi e di altre caratteristiche tipiche di un editor rivolto alla programmazione.

Nonostante GtkSourceView abbia dimostrato di essere una buona soluzione per una semplice colorazione della sintassi, non si è dimostrato all'altezza di altri widget analoghi, quali Scintilla o Kate. Il motore attuale, infatti, presenta numerosi problemi a causa di alcune limitazioni che gli consentono di riconoscere solo semplici espressioni, spesso ignorando la reale struttura del documento.

Per superare queste limitazioni si è sviluppato un nuovo motore di colorazione della sintassi basato su una macchina a stati e in grado di suddividere i documenti in contesti annidati, i quali vengono descritti in un file XML per ogni linguaggio. Per rendere questo possibile è stato necessario introdurre un nuovo formato per la descrizione dei linguaggi, dotandolo di una maggiore capacità espressiva rispetto a quello attuale, pur mantenendo una sintassi sufficientemente semplice.

Indice

Capitolo 1

Obiettivi

1.1 Introduzione

Ogni linguaggio di programmazione possiede una struttura rigida e ben definita, che consente all'utente di impartire comandi di complessità variabile all'elaboratore. Questa struttura rigida consente all'elaboratore di comprendere le direttive assegnategli ma, al crescere della complessità, diviene un ostacolo via via maggiore per l'immediata comprensione del codice da parte del programmatore. Ne consegue la necessità che l'ambiente di sviluppo del programmatore, in particolare l'editor, sia in grado di facilitarne il compito mostrando il maggior numero di indicazioni utili possibile, ricercando il giusto equilibrio al fine di non distogliere od ostacolare l'attenzione dal codice stesso.

Un utile strumento, diffuso e apprezzato per la sua immediatezza e potenza, è la colorazione della sintassi, la quale consente all'utente di individuare senza sforzo le parti che costituiscono la struttura del codice visualizzato, facilitandone la comprensione.

1.2 Cos'è la sintassi

La sintassi di un linguaggio non è altro che la forma con cui i concetti debbono essere espressi. Così come in un linguaggio naturale la sintassi (o grammatica) regola come le parole debbano essere disposte all'interno di frasi e come queste vadano collocate in un discorso, nei linguaggi di programmazione esistono regole, solitamente ben più rigide di quelle di un linguaggio naturale, che indicano come le istruzioni vadano formulate, affinché l'elaboratore sia in grado di interpretarle correttamente.

1.3 La colorazione della sintassi

Per consentire all'utente una lettura più agevole del codice, è possibile colorare con stili e colori differenti le diverse parti che lo compongono, consentendo, per esempio, di distinguere con una semplice occhiata un blocco di codice da un commento o di

individuare con precisione una costante o una stringa. La lettura del codice viene facilitata da suggerimenti visivi che permettono all'utente di concentrarsi sul significato di ciò che sta leggendo, alleggerendolo in buona parte dell'analisi sintattica.

1.4 GtkSourceView

La libreria GTK[1] consiste in un insieme di widget per la creazione di interfacce grafiche in ambienti diversi, tra cui il sistema X Window[2] (a sua volta disponibile per Linux, *BSD, Sun Solaris, IBM AIX e altre piattaforme), Mac OS, Microsoft Windows e il framebuffer di Linux. La libreria è scritta interamente in C ma dispone di binding per numerosi altri linguaggi, tra cui C++, Python, Perl, nonché per la piattaforma .NET con il progetto GTK#[3].

Tra i widget contenuti, GtkTextView è un widget di testo il cui funzionamento segue il paradigma MVC (Modello-Vista-Controllore). In particolare GtkTextView costituisce i componenti Vista e Controllore, mentre GtkTextBuffer contiene il testo vero e proprio, svolgendo la parte del Modello.

GtkSourceView è un widget di testo che estende GtkTextView, ampliandone le capacità al fine di essere impiegato come widget standard per editor avanzati, implementando la possibilità di visualizzare numeri di riga, marcatori a lato del testo, nonché la ripiegatura del testo e la colorazione della sintassi.

Attualmente GtkSourceView è principalmente utilizzato da gedit[4], l'editor che fa parte dell'ambiente desktop GNOME[5], e da MonoDevelop[6], un ambiente di sviluppo integrato per .NET basato sulla piattaforma Mono[7] e su GTK#.

1.5 Obiettivi del nuovo motore

L'obiettivo principale del progetto è lo sviluppo di un motore che garantisca una maggiore flessibilità nell'identificare i costrutti dei linguaggi di programmazione. Questo perché il motore attuale non è in grado di riconoscere alcuni costrutti presenti in alcuni linguaggi molto diffusi a causa della sua incapacità di rappresentarli internamente. Come nel motore attuale si è scelto di mantenere le descrizioni dei linguaggi separati dal codice del motore, in modo da consentirne uno sviluppo indipendente: in questo modo ogni utente può descrivere un nuovo linguaggio o migliorare la descrizione di uno già esistente senza dover addentrarsi nel codice del motore, soprattutto sollevando l'autore della libreria dalla manutenzione delle varie sintassi, in modo tale che si la possa affidare alla comunità di utenti interessati. Ciò consente, inoltre, una

[1] http://www.gtk.org/
[2] http://www.x.org/
[3] http://gtk-sharp.sourceforge.net/
[4] http://www.gnome.org/projects/gedit/
[5] http://www.gnome.org/
[6] http://www.monodevelop.com/
[7] http://www.mono-project.com/

migliore separazione tra il motore vero e proprio e la porzione adibita al riconoscimento della sintassi, impedendo di fare assunzioni errate nel codice del motore, vincolandolo beneficamente a una effettiva genericità. Per consentire agli utenti di modificare con facilità le descrizioni si è scelto di utilizzare anche nel nuovo motore un dialetto XML (eXtensible Markup Language), semplificando in questo modo anche la fase di parsing e di gestione degli errori sintattici grazie alla disponibilità di apposite librerie (libxml2[8]).

[8] http://www.xmlsoft.org/

Capitolo 2

Valutazione delle tecnologie esistenti

Prima di implementare un nuovo motore di colorazione della sintassi si sono valutati diversi motori attualmente disponibili sotto licenze compatibili con quella di GtkSourceView, il quale è posto sotto la tutela della GPL[1] (General Public License). Sfruttare un'implementazione esistente porterebbe con sé numerosi vantaggi, oltre al minor tempo di sviluppo richiesto; per esempio, il codice già scritto solitamente contiene meno errori, essendo già passato attraverso una fase di testing, più o meno accurata a seconda della diffusione del codice stesso. Inoltre, impiegando un motore già esistente, non vi sarebbe la necessità di riscrivere le descrizioni per ogni linguaggio di programmazione, fatto molto importante a causa dell'elevato numero di linguaggi esistenti e delle grandi diversità riscontrabili nelle differenti sintassi.

2.1 GtkSourceView

Il motore attuale di GtkSourceView presenta notevoli limitazioni, essendo basato su una struttura simile a quella di una macchina a stati dotata di soli due stati: lo stato predefinito, per il quale non viene effettuato nessuna colorazione, e lo stato che indica il riconoscimento di un costrutto, il quale viene colorato a seconda della configurazione. Ciò significa che ogni costrutto viene identificato indipendentemente dal contesto circostante, rendendo impossibile visualizzare correttamente linguaggi che prevedano l'uso delle cosiddette "isole": un esempio diffuso è l'uso di HTML (HyperText Markup Language) contenente regole CSS (Cascading Style Sheet) o codice JavaScript. In questi casi i tag HTML verrebbero erroneamente riconosciuti all'interno del codice JavaScript o nell'isola CSS e, viceversa, le istruzioni dei due linguaggi verrebbero colorate all'interno del documento HTML in modo sbagliato. Un altro problema riscontrabile, ad esempio, si manifesta con le direttive `#if 0` del preprocessore C. Queste vanno trattate come commenti, fatta eccezione del caso in cui contengano `#if` annidati, per

[1] http://www.gnu.org/copyleft/gpl.html

cui le direttive `#endif` associate a questi ultimi debbano essere ignorate. Il motore attuale, invece, non è in grado di comportarsi correttamente in questi casi e termina il commento `#if 0` alla prima direttiva `#endif` incontrata, anche se si riferisce a un `#if` annidato. Un'ulteriore limitazione dell'attuale motore consiste nella mancanza di una politica di priorità per la selezione dei contesti in caso di corrispondenze multiple.

2.2 GtkSourceStackEngine

Nel CVS (Concurrent Versioning System) ufficiale di GtkSourceView è disponibile, in un branch separato denominato "engine-split", una versione sperimentale del widget. In tale versione è stata operata una rifattorizzazione di alcune funzioni, al fine di aumentare la modularità della porzione adibita alla colorazione della sintassi e di consentire quindi l'implementazione e uso contemporaneo di differenti motori di colorazione. In questo ramo sperimentale è inclusa una bozza di un motore avanzato basato su uno stack che contiene gli stati della macchina a stati usata per il riconoscimento della sintassi. Come il motore originale di GtkSourceView, anche questo è configurabile mediante documenti XML in cui vengono definite le transizioni tra stati, utilizzando le istruzioni di push e pop che operano sullo stack impiegato internamente dal motore. L'esposizione dell'implementazione interna del motore ne semplifica l'implementazione, provocando però un notevole incremento della complessità delle descrizioni dei linguaggi nei file XML i quali devono utilizzare istruzioni di basso livello per esprimere la struttura del testo da riconoscere. Dal momento che le descrizioni dei linguaggi vengono create e mantenute dagli utenti stessi, è fondamentale che queste siano di immediata comprensione, in modo che la curva di apprendimento sia la più bassa possibile. In questo modo è possibile coinvolgere un numero sufficiente di utenti per garantire un supporto completo ai numerosi linguaggi esistenti, facendo in modo che le loro descrizioni siano corrette e facilmente aggiornabili. Purtroppo la complessità esposta da questo motore di colorazione e il fatto che la sua implementazione fosse solo accennata, ha portato alla conclusione di procedere con l'implementazione di un paradigma più naturale dal punto di vista dell'utente, anche se ciò ha causato un marginale incremento della complessità nel motore stesso.

2.3 Colorer

Il progetto Colorer[2] fornisce una libreria che fornisce servizi di colorazione della sintassi in tempo reale per editor esterni, rilasciata con licenza MPL[3] (Mozilla Public License) o, alternativamente, con una tra le licenze GPL e LGPL[4] (Lesser General Public License). Oltre a funzionare correttamente sia su piattaforme unix che Windows e Macintosh, la libreria è dotata di una API Java grazie alla quale è disponibile un plugin

[2] http://colorer.sourceforge.net/
[3] http://www.mozilla.org/MPL/MPL-1.1.html
[4] http://www.gnu.org/copyleft/lesser.html

che le consente di essere utilizzata nell'IDE (Integrated Developement Environment) multipiattaforma Eclipse[5].

Analogamente a GtkSourceView, Colorer utilizza un dialetto XML per le descrizioni dei linguaggi, detto HRC (Highlighting Resource Codes). Insieme a questo, Colorer affianca un ulteriore dialetto XML definito HRD (Highlighting Resource Descriptions) che contiene le informazioni riguardo a stile e colorazione da impiegare per i costrutti riconosciuti.

La sintassi dei documenti che descrivono i linguaggi, però, risulta essere alquanto complessa e di difficile comprensione, rendendone difficile l'adozione da parte degli utenti.

Internamente, la libreria è basata su una gerarchia di contesti, fatto che le garantisce una notevole flessibilità espressiva. Purtroppo, essendo realizzata in C++, si renderebbe necessaria una conversione in C, in quanto quest'ultimo risulta essere un requisito inderogabile per l'inclusione in GtkSourceView. Nonostante questo non sia, in linea di principio, un ostacolo insormontabile, la struttura interna di Colorer è oltremodo complessa, rendendo difficile la conversione agli standard usati in GtkSourceView. Oltre alla complessa gerarchia di oggetti impiegata da Colorer, nella libreria vengono reimplementati internamente un sistema per la gestione delle codifiche Unicode e UTF-8, un motore per le espressioni regolari, un client HTTP e un parser XML, invece che appoggiarsi a librerie esterne. Per fornire un confronto quantitativo rispetto all'implementazione attuale di GtkSourceView, l'intero albero dei sorgenti della versione take5 di Colorer ammonta a 18 Megabyte, mentre quello di GtkSourceView alla versione 1.1.1 si limita a 3,7 Megabyte. A fronte di una implementazione così complessa si è ritenuto che il lavoro necessario per riuscire a impiegare la libreria in GtkSourceView fosse paragonabile a una riscrittura completa, vanificando i vantaggi del riutilizzo del codice.

2.4 Kate

Il progetto KDE[6] (K Desktop Environment) è uno dei due ambienti desktop principali per sistemi unix basati su X Window. Tra i programmi facenti parte di questo ambiente è incluso editor di testo Kate, dotato di capacità di colorazione della sintassi analoghe a quelle prese in considerazione per GtkSourceView. Infatti, al pari del widget usato in gedit, Kate usa descrizioni dei linguaggi realizzate mediante documenti XML. Facendo parte di KDE, Kate è scritto in C++ e fa uso delle librerie Qt: pertanto, al fine di riutilizzarne la porzione di colorazione della sintassi si renderebbe necessaria anche in questo caso una conversione in C, rimuovendo anche la dipendenza da Qt. Seppure l'architettura interna del sistema sia notevolmente più semplice e lineare di quella di Colorer, il formato delle descrizioni dei linguaggi sembra essere solo marginalmente migliore di quello attualmente usato da GtkSourceView, non giustificando quindi il lavoro necessario per effettuare l'integrazione necessaria.

[5] http://www.eclipse.org/
[6] http://www.kde.org/

2.5 Scintilla

Il progetto Scintilla[7] consiste in un componente riutilizzabile pensato specificatamente per l'editing di codice sorgente. Include funzioni avanzate sia per la scrittura che per il debug dei programmi, come la colorazione della sintassi, indicatori di errore, completamento automatico del codice e suggerimenti interattivi. Attualmente supporta nativamente la piattaforma win32, ma sono presenti anche versioni per GTK 1.2, GTK 2.0 ed è disponibile una versione che si integra con le librerie Qt. Scintilla è un componente avanzato dotato di caratteristiche di estrema potenza, ottenute però con un incremento della complessità e una rigidità notevole. La sua archiettura, infatti, si basa su analizzatori lessicali scritti ad-hoc per ogni linguaggio. Nonostante questo conceda la maggiore libertà ed espressività al programmatore, richiede che l'utente interessato a supportare un linguaggio debba scrivere un analizzatore lessicale apposito in C. Data la complessità del compito, questo costituisce una barriera difficilmente superabile al fine di supportare il maggior numero di linguaggi, ponendo inoltre un grave peso sull'autore della libreria, il quale deve occuparsi della manutenzione dei diversi analizzatori lessicali. Nonostante esista una versione per GTK 2.0, questa è poco integrata, mostrando una evidente mancanza di coerenza con lo stile di programmazione tipico di tale libreria dovuto all'origine avvenuta impiegando la piattaforma win32, di cui l'influenza è profonda. Una maggiore integrazione richiederebbe una ristrutturazione del codice con ampie riscritture, tanto che neppure gli autori stessi hanno finora ritenuto opportuno intraprendere a tale strada.

2.6 VIM

VIM[8] è un editor avanzato basato sul classico editor UNIX VI, al quale sono state aggiunte potenti funzionalità rivolte in particolare ai programmatori. Oltre a possedere un linguaggio di scripting interno, VIM è in grado di evidenziare la sintassi di una grande quantità di linguaggi. Le descrizioni di tali sintassi sono però basate su script realizzati nel linguaggio di scripting interno, i quali vengono eseguiti al momento del caricamento. Ciò consente a VIM una buona flessibilità, ma rende pressoché impossibile la separazione del motore dal resto dell'applicazione.

Inoltre è stato considerato preferibile basare le descrizioni su un linguaggio dichiarativo, anziché imperativo come il linguaggio interno di VIM. Oltre a ciò, il motore di VIM presenta notevoli problemi prestazionali, i quali nell'editor sono stati risolti limitando il motore ad analizzare solo un numero prefissato di righe precedenti la porzione di testo visualizzata. Purtroppo un comportamento simile produce problemi di correttezza, i quali possono essere visibili, ad esempio, in caso di commenti lunghi che superano i limiti artificiali imposti al motore.

[7] http://www.scintilla.org/
[8] http://www.vim.org/

2.7 Conclusioni

In seguito alla valutazione delle librerie esistenti si è considerato opportuno costruire una nuova implementazione, facendo tesoro dei dati raccolti durante l'analisi effettuata. Come obiettivi per la nuova implementazione si sono scelte quelle che si sono ritenute le maggiori mancanze delle implementazioni esistenti: una sintassi delle descrizioni semplice e quanto più possibile immediata e, in secondo luogo, una implementazione interna snella. Analogamente sia a Colorer che a Kate, si è scelto di basare la nuova implementazione su contesti annidati, fornendo all'utente una visione sufficientemente immediata della struttura delle descrizioni dei linguaggi. Per quanto riguarda l'implementazione vera e propria, invece, si è scelta come base il branch "engine-split" di GtkSourceView, sfuttando l'infrastruttura creata per la costruzione del GtkSourceStackEngine, senza però utilizzarne il motore.

Come tutti i motori presi in considerazione, si è deciso di utilizzare un apposito dialetto XML per le descrizioni dei linguaggi, impiegando inoltre delle espressioni regolari per esprimere i costrutti testuali che compongono la sintassi descritta.

Capitolo 3

Espressioni regolari

3.1 Introduzione alle espressioni regolari

Le espressioni regolari si basano, formalmente, su grammatiche di tipo 3, dette appunto "grammatiche regolari". Queste sono, nell'ambito della teoria dei linguaggi, grammatiche in grado di esprimere concatenazioni e ripetizioni, nonché alternative, ma non capaci di esprimere annidamenti. Essendo le grammatiche più semplici che possano essere impiegate per compiti complessi, sono probabilmente la parte meglio studiata e compresa in ambito dei linguaggi formali e sono utilizzate diffusamente in tutto il settore informatico, in particolare per descrivere il contenuto di una stringa di testo in modo da poter essere compreso da un computer, facendo in modo che questo possa riconoscere corrispondenze e, in applicazioni più avanzate, suddividere parti del testo corrispondente.

Ogni espressione regolare è composta da quattro componenti fondamentali, ovvero modelli, asserzioni, quantificatori e riferimenti all'indietro.

3.2 Modelli

Ogni modello consiste in stringhe letterali e classi di caratteri. Un modello può contenere sotto-modelli, i quali non sono altro che modelli racchiusi in parentesi tonde.

3.3 Caratteri di escape

Alcuni caratteri, sia nei modelli che nelle classi di caratteri, hanno un significato particolare. Pertanto, per trovare corrispondenze letterali di tali caratteri è necessario contrassegnarli opportunamente, in modo che il motore di riconoscimento li interpreti con il loro significato letterale. Tale compito, detto "escape", è assegnato al carattere \ (barra rovesciata). Nel caso questo carattere fosse impiegato per effettuare l'escape di caratteri privi di significato speciale, il motore lo ignora, rendendo possibile contrassegnare qualsiasi carattere, in caso di dubbio sul suo significato. Al fine di ottenere una

barra rovesciata letterale, il carattere di escape va contrassegnato esso stesso usando due barre \\.

3.4 Classi di caratteri

Le classi di caratteri, invece, sono espressioni corrispondenti a uno o più caratteri, definiti ponendo i caratteri desiderati tra parentesi quadre `[]` oppure usando apposite classi predefinite.

Le classi di caratteri più semplici contengono solamente uno o più caratteri letterali, ad esempio `[abc]` corrisponde a una tra le lettere `a`, `b` e `c` oppure `[0123456789]` corrisponde a qualsiasi cifra.

Sfruttando l'ordinamento intrinseco di lettere e cifre è possibile utilizzare intervalli come `[a-c]` al posto di `[abc]` e `[0-9]` per ottenere una cifra qualsiasi. È anche possibile combinare entrambi i costrutti, usando `[a-ehjk1-7]` per indicare i caratteri da `a` fino a `e`, i caratteri `h`, `j`, `k` e le cifre da `1` a `7`. Normalmente le espressioni regolari distinguono tra caratteri maiuscoli e minuscoli, considerandoli due entità differenti.

Se il primo carattere tra le parentesi quadre è un accento circonflesso `^`, la classe di caratteri viene considerata `negativa` e corrisponde a tutti i caratteri eccetto quelli indicati al suo interno: `[^abc]` corrisponde a tutti i caratteri tranne `a`, `b` e `c`.

Alcuni caratteri particolari sono:

\t il carattere ASCII di tabulazione orizzontale;

\xHHHH il carattere Unicode specificato dal numero esadecimale `HHHH` (compreso tra 0x0000 e 0xFFFF);

. un carattere qualsiasi, compreso il ritorno a capo;

\d una cifra (uguale a [0-9]);

\D l'inverso di \d (uguale a `[^0-9]`);

\s un carattere di spaziatura tra spazio, tab e ritorno a capo (uguale a [\t\n\r]);

\S l'inverso di \s;

\w un carattere tra quelli solitamente ammessi per gli identificatori delle variabili.

All'interno delle classi alcuni caratteri hanno un significato particolare e deve essere effettuato un escape per poterli includere letteralmente:

^ indica una classe negativa se è il primo carattere;

- indica un intervallo;

**** il carattere di escape.

3.5 Sotto-modelli

Racchiudendo delle porzioni di espressione regolare tra parentesi tonde `()` si definiscono dei sotto-modelli che possono essere usati in numerosi modi.

Dal momento che, per indicare delle alternative è possibile usare il carattere | come separatore, spesso si racchiude la porzione con alternative tra parentesi. Ad esempio, per indicare tutte le occorrenze di `int` o `float` seguiti da spazi e poi un identificatore, è possibile usare `(int|float)\s+\w+`.

Utilizzando i riferimenti all'indietro, invece, è possibile riferirsi al contenuto di sotto-pattern già identificati. Nel caso in cui si volessero trovare le occorrenze di due parole uguali separate da spazi sarebbe possibile utilizzare l'espressione regolare `(\w+)\s+\1`, nella quale `\1` indica il contenuto della corrispondenza identificata dal primo sotto-modello racchiuso tra parentesi.

Se il sotto-modello inizia con `?=` o `?!` si tratta di una asserzione non catturante, in quanto controllano che il testo analizzato corrisponda a quanto espresso, senza che questo entri però a far parte della regione considerata come corrispondente.

All'interno dei sotto-modelli alcuni caratteri hanno significati particolari:

\ il carattere di escape;

^ asserzione che indica l'inizio di una riga;

$ asserzione che indica la fine della riga;

| effettua l'OR logico tra le alternative proposte;

\b asserzione che indica i delimitatori di parola, segnalandone inizio e fine.

3.6 Quantificatori

I quantificatori consentono a caratteri, classi di caratteri e sotto-modelli in una espressione regolare di corrispondere un numero definito di volte. Essi vengono definiti con l'uso di parentesi graffe {} e sono nella forma {`[minimo][,[massimo]]`}. Ad esempio, {`1`} definisce una occorrenza, {`0,1`} zero oppure una, così come {`,1`}, {`5,10`} definisce almeno cinque e al massimo dicei occorrenze, mentre {`5,`} ne indica almeno cinque, senza specificare alcun massimo.

Esistono alcune abbreviazioni per i quantificatori più usati:

***** analogo a {`0,`}, indica qualsiasi numero di occorrenze;

\+ analogo a {`1,`}, indica almeno una occorrenza;

? analogo a {`0,1`}, indica zero oppure una occorrenza.

Nei casi in cui si omette il massimo numero di occorrenze, le espressioni regolari si comportano in modo "greedy", cercando il maggior numero possbile di corrispondenze. Un `?` dopo il quantificatore rende l'espressione regolare non greedy, cioè viene cercata la stringa con il minor numero di corrispondenze possibile.

3.7 Implementazione impiegata

Al fine di utilizzare le espressioni regolari è necessario appoggiarsi a una libreria che ne implementi le funzionalità, talvolta estendendole al fine di semplificarne l'uso allo sviluppatore. Attualmente GtkSourceView include una copia del codice relativo alle espressioni regolari tratto dalla libreria di sistema GNU GLibC, in quanto su sistemi non basati su questa libreria il supporto alle espressioni regolari in stile Posix potrebbe essere assente. Questa tipologia di espressioni regolari, infatti, fornisce alcune estensioni rispetto alle espressioni regolari base molto importanti quale, ad esempio, il separatore di parola `\b`. Internamente le funzioni fornite dal codice GNU vengono accedute tramite alcune funzioni wrapper fornite da GtkSourceRegex.

Nel nuovo motore, invece, si è scelto di basarsi su EggRegex, un wrapper a oggetti sperimentale contenuto in LibEgg. Quest'ultima è una libreria che contiene molte delle tecnologie che sono in fase di sperimentazione e test prima di essere ufficialmente incluse in GLib o GTK. EggRegex riveste la libreria PCRE[1], Perl Compatible Regular Expression, come motore per le espressioni regolari; questa libreria, usata in progetti come Apache, PHP, Python e nmap, supporta espressioni regolari in stile Perl, le quali sono più potenti delle espressioni regolari Posix, ma risultano essere talvolta incompatibili con esse. Per questo motivo è risultato impossibile sia impiegare il nuovo motore per leggere le vecchie definizioni dei linguaggi, sia convertire queste ultime nella nuova sintassi in modo automatico.

[1] http://www.pcre.org/

Capitolo 4

Tecnologie impiegate basate su XML

4.1 eXtensible Markup Language

XML (eXtensible Markup Language) è uno standard promosso dal W3C[1] (World Wide Web Consortium) e consiste in un semplice e flessibile formato testuale derivato da uno standard ISO precedente, SGML (ISO 8879). Simile a HTML, il quale infatti è una applicazione SGML, XML viene definito estensibile in quanto non è un formato fisso, ovvero non è dotato di elementi predefiniti. Ciò significa che XML in realtà è un metalinguaggio, ovvero un insieme di regole sintattiche volte a semplificare la definizione di linguaggi basati su di esso. In questo modo è possibile creare applicazioni XML apposite per ogni scopo, garantendo al contempo la flessibilità di poter creare linguaggi personalizzati e l'interoperabilità dovuta al fatto di poter usare strumenti comuni.

La filosofia seguita per il progetto di XML è quella di creare un generico contenitore di dati, senza assegnare alcuna semantica ai componenti del contenitore, lasciando il compito di definire il significato all'utente.

In particolare, XML è definito come una sequenza di elementi detti tag, delimitati da parentesi angolari, `<` e `>`, i quali possono essere dotati di attributi e possono contenere testo e altri elementi. Mentre HTML prevede un preciso insieme di tag validi (come `<body>`, `<a>`, `<table>`), XML non ne specifica alcuno, in quento tale compito è assegnato a ciascun linguaggio che si basa su XML. A differenza di HTML, in cui l'elemento `<a>` identifica sempre un collegamento, in XML questo non è sempre vero, essendo possibile attribuirgli un significato arbitrario, come riportato nei due esempi seguenti.

Listato 4.1. Esempio di XHTML

```
<html>
    <body>
```

[1] http://w3.org/

```
        <a href="http://www.polito.it">Politecnico di Torino</a>
    </body>
</html>
```

Listato 4.2. Esempio di linguaggio arbitrario

```
<spostamento>
    <da>Vercelli</da>
    <a>Torino</a>
</spostamento>
```

Nel *listato 4.1* l'elemento `<a>` è analogo a quello presente in HTML: infatti XHTML è una riprogettazione di HTML come applicazione XML. Nel *listato 4.2*, invece, è chiaro che il significato sia differente e che rappresenti una destinazione. Entrambi i casi sono esempi di XML valido, seppure i loro significati e ambiti di utilizzo siano decisamente differenti.

Il principale vantaggio di usare XML consiste nel disporre di un formato ben definito e diffuso, caratterizzato da una buona leggibilità e facilità d'uso, potendo inoltre impiegare parser disponibili e già collaudati, affidandosi a pratiche già esistenti per la soluzioni di problemi come la gestione delle codifiche, l'internazionalizzazione e la validazione dei documenti.

4.2 Requisiti di una sintassi XML

Dal momento che XML si limita a definire la forma grammaticale dei documenti, si rende necessario progettarne la forma sintattica, definendo gli elementi da usare e assegnando loro una semantica ben precisa.

Il principio fondamentale in tale progettazione è la chiarezza: il formato delle descrizioni deve risultare leggibile e comprensibile con il minimo sforzo, in modo da consentirne facilmente l'adozione da parte degli utenti. Per raggiungere tale scopo è necessario che i nomi degli elementi siano brevi e comprensibili, minimizzando ambiguità e dubbi senza inficiare la leggibilità, riducendo anche la quantità di testo da scrivere da parte dell'autore. Tali nomi, inoltre, devono riflettere fedelmente i concetti impiegati per la descrizione dei linguaggi, in modo che la loro sematica sia chiara e interpretabile univocamente. Oltre che alla descrizione vera e propria è necessario prevedere la possibilità di aggiungere metadati a ogni documento, in modo che sia possibile raccogliere il maggior numero di informazioni su ciascuno di essi. Per migliorare la leggibilità questi metadati devono essere ben separati dai dati veri, collocando ciascuno in appositi elementi contenitori distinti, di cui quello contenente i metadati posizionato prima dell'altro, cosicché i parser possano accedervi più facilmente.

Ogni sintassi XML, inoltre, deve contenere informazioni sulla propria versione, in modo che istanze di versioni differenti possano coesistere facilmente tra loro.

4.3 Implementazione impiegata

Per la lettura e l'analisi del linguaggio basato su XML si è usata la libreria Libxml2, che consiste in parser XML scritto in C e accessibile con numerose API differenti, nonché disponibile per un vasto insieme di linguaggi di programmazione, oltre al C. Libxml2 è ampiamente collaudata, essendo usata diffusamente nell'ambito del progetto GNOME e in numerose altre applicazioni, essendo conosciuta per la sua velocità e affidabilità, oltre che per il vasto numero di standard supportati. Infatti libxml2 supporta i namespace in XML, XPath, XPointer, XInclude, Canonical XML, XML Schema, Relax NG e possiede API pressoché compatibili con DOM (Document Object Model), SAX1 e SAX2. Insieme a Libxml2 è anche possibile usare Libxslt al fine di poter impiegare fogli di stile XSLT. Oltre alle interfacce di programmazione elencate precedentemente, Libxml2 fornisce anche una interfaccia XMLReader inspirata a quella disponibile per C#. La caratteristica di questa interfaccia è che non richiede che il documento sia interamente in memoria, requisito necessario invece per DOM, minimizzando quindi le risorse richieste, pur mantenendosi a un livello di astrazione maggiore rispetto all'interfaccia SAX, riducendo la complessità del codice necessario per l'analisi dei documenti.

4.4 Validazione dei documenti XML

Al fine di verificare la correttezza di un documento XML è possibile sottoporlo a validazione. Innanzitutto il documento deve essere ben formato, ovvero deve rispettare le regole sintattiche di base di XML (tag di apertura e chiusura corrispondenti, escape di caratteri speciali, ecc.). In secondo luogo è possibile assicurarsi che la sintassi del file analizzato corrisponda a quella del linguaggio precedentemente definito: per fare ciò sono disponibili numerosi linguaggi che consentono di descrivere la sintassi voluta, specificando formalmente le relazioni tra elementi, attributi e contenuti che il documento deve rispettare. In tal modo si delega il compito di gestire documenti non corretti al parser XML, minimizzando la porzione di gestione degli errori all'interno dell'applicazione.

4.4.1 Document Type Definition

La Document Type Definition, abbreviato in DTD, consiste in un un documento SGML che descrive un insieme di regole che delineano la struttura dei documenti SGML conformi. Dal momento che XML è una applicazione SGML, ovvero è un documento SGML valido a cui sono imposte alcune restrizioni, è possibile impiegare le DTD per la validazione di documenti XML. Essendo però SMGL più flessibile di XML, ne risulta anche che sia notevolmente più complesso e che questa complessità si ripercuota sulle DTD. Talvolta, inoltre, può risultare difficile o impossibile specificare con esattezza alcuni vincoli, in particolare sul contenuto degli elementi, mentre è pressoché assente il supporto ai namespace, concetto molto usato in XML.

4.4.2 XML Schema

A differenza delle DTD, XML Schema è stato progettato dal W3C appositamente per la validazione di XML. XML Schema è suddiviso in due parti: una definisce la porzione incaricata di descrivere la struttura dei documenti XML, mentre la seconda stabilisce una gerarchia di tipi di dati, usata per garantire un controllo più preciso sui contenuti degli elementi rispetto a quanto possibile con DTD. A differenza di queste, XML Schema è definito come una applicazione XML, rendendo possibile impiegare un solo parser per lo schema e per il documento. Inoltre, XML Schema supporta pienamente i namespace. Purtroppo, XML Schema è estremamente complesso, tanto che libxml2 attualmente ne implementa solamente alcune porzioni, in particolare la parte dello standard dedicata ai tipi di dati.

4.4.3 Relax NG

Relax NG è un linguaggio creato da una commissione tecnica interna a OASIS[2] e in corso di adozione come standard ISO/IEC 19757-2. Questo linguaggio basa la sua potenza sulla estrema semplicità e su solida fondamenta matematiche, poggiandosi su concetti di teoria degli automi. Al fine di rendere opzionale la prolissità intrinseca di XML, Relax NG è disponibile sia come linguaggio XML che con una sintassi più compatta che ricalca in parte quella usata per le espressioni regolari (quest'ultima non è supportata da libxml2). Oltre a supportare con semplicità i namespace, Relax NG impiega al suo interno la gerarchia di tipi di dati di XML Schema, cercando però di superare i limiti di quest'ultimo rendendo più semplice l'impiego di contenuti misti (elementi e testo) o privi di ordinamento, minimizzando, inoltre, le differenze tra attributi ed elementi. Visti i vantaggi presentati da Relax NG, quali la semplicità, la flessibilità e l'ottimo livello di supporto da parte di Libxml2, si è scelto di usare quest'ultimo per validare le descrizioni delle sintassi.

[2] http://www.oasis-open.org/

Capitolo 5

Descrizioni dei linguaggi

5.1 Sintassi delle descrizioni dei linguaggi

Scegliendo di implementare un motore completamente agnostico per ciò che concerne i linguaggi da colorare, è possibile fare in modo che le loro descrizioni vengano prodotte direttamente dagli utenti, invece di assegnare questo ulteriore compito al programmatore. Così facendo, oltre a ridurre il costo di manutenzione del pacchetto software alleggerendo il lavoro del programmatore, si diviene in grado di garantire un supporto a una gamma di linguaggi notevolmente più ampia rispetto a quella disponibile facendo diversamente. Infatti, se fosse necessario implementare un analizzatore apposito per ogni linguaggio, pochi sarebbero in grado di realizzarne uno, ed è impensabile che coloro che ne fossero capaci conoscano tutti i linguaggi usati oggigiorno. La gravosità del compito, inoltre, renderebbe discutibile il supporto a linguaggi poco diffusi o di nicchia. Al fine di coinvolgere il maggior numero di utenti, quindi, è necessario rendere il più semplice possibile la scrittura di una descrizione di sintassi, utilizzando strumenti conosciuti e diffusi. Per questo motivo si è scelto di impiegare un linguaggio basato su XML, trattandosi di uno strumento conosciuto e usato in tutti i settori informatici, e costruendo su di esso una sintassi semplice e chiara.

5.2 Language Definition 2.0

Grazie alle informazioni di versione contenute nel formato del motore attuale di GtkSourceView è possibile impiegare sia le descrizioni esistenti che descrizioni realizzate nel nuovo formato. Dal momento che la versione viene determinata dall'attributo `version` nell'elemento radice `<language>`, questo è stato mantenuto invariato, con tutti gli attributi definiti identicamente alla versione 1.0.

Come requisito di progettazione si è scelto di evitare la stesura di una sintassi che riflettesse la struttura interna del programma, in quanto ciò, pur semplificando la scrittura iniziale del codice, rende notevolmente difficoltoso effettuarvi successive modifiche, accoppiando in modo eccessivamente stretto le descrizioni all'implementazione del motore. Per questo motivo, inizialmente si è modellata la sintassi delle descrizioni

senza tenere conto dell'implementazione interna del motore, il quale è stato realizzato in seguito.

Seguendo le pratiche più diffuse e consigliate è stato posto all'inizio del documento un elemento opzionale `<metadata>`, contenente informazioni sulla descrizione stessa quali, per esempio, una descrizione, una lista di autori oppure dei dati arbitrari codificati impiegando RDF (Resource Description Framework).

Al fine di separare quanto più nettamente la porzione di descrizione da quella di presentazione, si è scelto di impiegare un elemento `<styles>` separato, il quale contiene la definizione di ogni stile che dovrà essere usato per l'evidenziazione della sintassi. Tale elemento contiene un elemento `<style>` per ogni stile, definendone mediante l'attributo `id` l'identificatore che verrà usato internamente alla descrizione nel fare riferimento allo stile stesso, mentre l'attributo `name` contiene il nome che verrà presentato all'utente nel caso quest'ultimo desiderasse cambiare le impostazioni predefenite per lo stile in questione. Tali impostazioni predefinite vengono assegnate facendo riferimento agli stili contenuti in `def.lang` all'interno dell'attributo `map-to`.

La descrizione della sintassi vera e propria avviene all'interno dell'elemento `<definitions>`: qui, infatti vengono definiti i contesti utilizzando l'apposito elemento `<context>`, il quale costituisce il blocco principale con cui costruire la descrizione. L'associazione tra un particolare contesto e uno stile avviene grazie all'attributo `style-ref`, il quale contiene un riferimento a uno degli stili definiti all'interno dell'elemento `<styles>`.

Dopo il parsing, il motore utilizza come stato iniziale il contesto radice dotato di un attributo `id` corrispondente a quello dell'elemento radice `<language>`.

Esistono diversi tipi di elemento `<context>`: i contesti semplici contengono un elemento `<match>` nel quale è definita l'espressione regolare che indica a cosa corrisponde il contesto. Oltre a ciò può essere contenuto un elemento `<include>`, nel quale vi possono essere elementi `<context>` dotati di attributo `sub-pattern` che indica a quale sotto-corrispondenza dell'espressione regolare del contesto genitore si riferiscono. Questi sono utilizzati per colorare diversamente porzioni di una espressione regolare complessa suddivisa in gruppi.

I contesti contenitore, invece, sono dotati di una espressione regolare di apertura nell'elemento `<start>` e di una di chiusura nell'elemento `<end>`. All'interno del relativo `<include>` possono essere contenuti altri contesti, sia semplici che contenitori, nonché sub-pattern. In tal caso questi possono essere dotati di un attributo `where` per specificare se si riferiscano a un gruppo contenuto nell'espressione regolare di apertura o di chiusura. L'attributo booleano `extend-parent` indica se il contesto contenuto è in grado di estendere il genitore o meno, nel caso contengala stringa di terminazione di questo. Ciò è particolarmente utile, per esempio, per caratteri di escape contenuti in stringhe secondo lo stile usato in nel linguaggio C: il contesto contenitore ha come espressioni regolari di inizio e fine le virgolette doppie e contiene un contesto semplice per i caratteri di escape, che corrisponderà alla barra rovesciata \ seguita da un qualsiasi carattere. Se fosse presente una virgoletta preceduta da una barra rovesciata, verrebbe assegnata al contesto più interno, evitando quindi che il contenitore venga

terminato. Nel caso si desideri il comportamento opposto, invece, è possibile specificare l'attributo `extend-parent` assegnandogli il valore `false`, in modo che il contesto contenuto abbia una priorità minore della terminazione del contenitore.

Per favorire il riuso del codice è possibile dichiarare che un contesto contiene un altro contesto precedentemente definito facendovi riferimento inserendo un elemento `<context>` con l'attributo `ref` posto al valore dell'attributo `id` assegnato al contesto contenuto.

All'interno delle espressioni regolari è possibile impiegare porzioni riutilizzabili definite in precedenza all'interno di un elemento `<define-regex>`, utilizzando una sintassi apposita \%{nome}, dove `nome` è il contenuto dell'attributo `id` dell'elemento `<define-regex>` desiderato.

Al fine di semplificare la definizione delle sintassi, è possibile utilizzare elementi `<context>` contententi una lista di elementi `<keyword>` per identificare parole chiave, le quali verranno poi concatenate e implementate come contesti semplici internamente dal motore.

5.3 Guida di riferimento per Language Definition v2.0

Nelle seguenti sezioni è contenuta la specifica di ogni elemento del formato XML Language Definition v2.0. La definizione formale è contenuta nello schema RelaxNG riportato nell'*appendice C*, il quale viene installato nella directory `${PREFIX}/share/gtksourceview-1.0/schemas/` (dove `${PREFIX}` può essere `/usr/` o `/usr/local/` se si è eseguita una installazione a partire dai sorgenti).

Tag `<language>`

L'elemento radice per i file Language Definition.

Elementi contenuti: `<metadata>` (opzionale), `<styles>` (opzionale), `<default-regex-options>` (opzionale), `<keyword-char-class>` (opzionale), `<definitions>` (obbligatorio).

Attributi:

`id` (obbligatorio) – Identificatore della descrizione. Viene usato per riferimenti esterni e deve essere unico tra le descrizioni dei linguaggi. Può contenere una stringa di lettere, cifre, trattini alti (`-`) o bassi (_).

`name` (obbligatorio) – Il nome traducibile presentato all'utente per il linguaggio. Può venire segnato per la traduzione anteponendo un trattino basso davanti al nome dell'attributo.

`version` (obbligatorio) – La versione del formato XML (attualmente `2.0`).

`section` (opzionale) – Il nome traducibile della categoria in cui il linguaggio deve venir raggruppato quando viene mostrato all'utente. Le categorie attualmente

usate in GtkSourceView sono “Sources”, “Scripts”, “Markup” e “Others”, ma è possibile usare categorie arbitrarie, seppur questo sia sconsigliato.

mimetypes (opzionale) – La lista di tipi MIME associati al linguaggio. Ogni tipo MIME deve essere separato con un punto e virgola (`;`).

Tag `<metadata>`

Contiene meta-informazioni opzionali riguardanti la descrizione stessa del linguaggio.

Elementi contenuti: attualmente non stato è definito alcun elemento, ma è possibile utilizzare elementi arbitrari posti in namespace differenti (ad esempio elmenti con meta-informazioni Dublin Core, ecc.).

Tag `<styles>`

Contiene la definizione di ciascuno stile impiegato nel linguaggio corrente e le associazioni tra questi e gli stili predefiniti in GtkSourceView.

Elementi contenuti: `<style>` (uno o più).

Non possiede attributi.

Tag `<style>`

Definisce uno stile, associandovi un nome traducibile da mostrare all'utente e uno stile predefinito all'interno di GtkSourceView.

Elementi contenuti: nessuno.

Attributi:

id (obbligatorio) – Identificatore per lo stile. Viene usato nel linguaggio corrente per fare riferimento allo stile e deve essere unico all'interno del documento corrente. Può contenere una stringa di lettere, cifre e trattini alti (`-`) o bassi (`_`).

name (obbligatorio) – Il nome visibile all'utente per lo stile. Deve essere preceduto da un trattino basso per essere segnato per la traduzione.

map-to (opzionale) – Usato per associare lo stile con uno stile predefinito, al fine di utilizzare i colori e le proprietà dei caratteri già definite per questi ultimi. L'identificatore dello stile predefinito deve essere preceduto dall'identificatore del linguaggio in cui questo è definito, separandoli con il carattere `:`. Nel caso venga omesso, lo stile non viene considerato derivato da alcuno stile e, pertanto, non verrà colorato fintanto che l'utente non specifichi uno schema di colori apposito.

Tag `<keyword-char-class>`

Contiene una classe di caratteri come quelle usate nelle espressioni regolari, usandola per ridefinire i delimitatori di parola personalizzabili `\%[` and `\%]`. Questa classe

consiste nell'insieme di caratteri che possono costituire una parola chiave. Se l'elemento viene omesso i due delimitatori vengono considerati equivalenti a `\b`.

Elementi contenuti: nessuno.

Non possiede attributi.

Tag `<default-regex-options>`

Una stringa usata per impostare le opzioni predefinite per le espressioni regolari PCRE.

Le opzioni disponibili sono:

`i` non distingue maiuscole e minuscole;

`x` espressioni estese (gli spazi vengono ignorati ed è possibile introdurre commenti che iniziano con # e terminano alla fine della riga);

`s` al metacarattere `.` corrisponde anche il ritorno a capo `\n`.

Queste opzioni possono venire sovrascritte in ogni espressione regolare utilizzando la sintassi `/regex/opzioni`: per disabilitare un gruppo di opzioni già abilitate è sufficiente precederle con un trattino (ad esempio le corrispondenze di `/[A-Z][a-z]*/-i` terranno conto delle differenze tra maiuscole e minuscole anche se l'impostazione predefinita dovesse essere quella di ignorare tali differenze).

Elementi contenuti: nessuno.

Tag `<definitions>`

L'elemento che contiene la descrizione vera e propria della sintassi da colorare. Contiene uno o più elementi `<context>` e un numero arbitrario di elementi `<define-regex>`, mischiati liberamente tra di loro.

Ogni elemento contenuto deve contenere nel proprio attributo `id` un identificatore unico all'interno del documento. Uno e un solo elemento tra gli elementi `<context>` contenuti deve aver il proprio attributo `id` uguale all'attributo `id` dell'elemento radice `<language>`. Tale contesto rappresenta il contesto iniziale per la colorazione, quello in cui il motore entra all'inizio dell'analisi del file da colorare.

Elementi contenuti: `<context>` (uno o più), `<define-regex>` (zero o più).

Non possiede attributi.

Tag `<define-regex>`

Definisce una espressione regolare che può essere riutilizzata all'interno di altre espressioni regolari, in modo da evitare la duplicazione di porzioni comuni. Queste espressioni regolari sono espressioni regolari PCRE nella forma `/regex/opzioni`. Nel caso in cui non si intenda specificare alcuna opzione e non si desideri che gli spazi all'inizio e alla fine dell'espressione regolare vengano considerati nelle corrispondenze, è possibile omettere le barre, scrivendo solo `regex`. Le possibili

opzioni sono quelle specificate precedentemente nella descrizione dell'elemento `<default-regex-options>`. Per disabilitare un gruppo di opzioni, invece, è necessario precederlo con un trattino (`-`). In GtkSourceView sono anche disponibili alcune estensioni rispetto alle comuni espressioni regolari in stile Perl:

\%[e \%] sono delimitatori di parola personalizzabili, ridefinibili mediante l'elemento `<keyword-char-class>`, diversamente da \b;

\%{id} include l'espressione regolare definita in un altro elemento `<define-regex>` con l'`id` specificato.

Elementi contenuti: nessuno.
Attributi:

id (obbligatorio) – Identificatore per l'espressione regolare. Viene usato per includere l'espressione regolare definita e deve essere unico all'interno del documento corrente. Può contenere una stringa di lettere, cifre e trattini alti (`-`) o bassi (_).

Tag `<context>`

Questo è l'elemento più importante nel descrivere una sintassi: il file da colorare viene, infatti, partizionato in contesti che rappresentano le porzioni di testo da colorare in modo differente. I contesti possono, inoltre, contenere altri contesti al loro interno. Esistono diversi tipi di elementi contesto: contesti semplici, contesti contenitori, contesti sub-pattern, contesti riferimento e contesti keyword.

Contesti semplici

Contengono un elemento obbligatorio `<match>` e un elemento `<include>` opzionale. Il contesto si estende per tutta la lunghezza della stringa corrispondente all'espressione regolare contenuta nell'elemento `<match>`. Nell'elemento `<include>` possono venire posti solamente contesti sub-pattern.

Elementi contenuti: `<match>` (obbligatorio).
Attributi:

id (opzionale) – Identificatore per il contesto, viene usato nei riferimenti al contesto corrente e deve essere unico all'interno del documento. Può contenere una stringa di lettere, cifre e trattini alti (`-`) o bassi (_).

extend-parent (opzionale) – Un valore booleano che indica al motore se il contesto possiede una priorità maggiore della fine del contesto genitore. Se non specificato viene inteso come `true`.

Contesti contenitori

Contengono un elemento `<start>` e un elemento opzionale `<end>`, i quali contengono, rispettivamente, l'espressione regolare per la quale il motore entra nel contesto e quella per la quale il motore ne esce. Nell'elemento opzionale `<include>` è possibile elencare contesti di ogni tipo (semplici, contenitori, sub-pattern o riferimenti). Se l'elemento `<start>` viene omesso, allora l'attributo `id` e l'elemento `<include>` diventano obbligatori (in quanto il contesto può essere usato solo come contenitore per includerne tutti i contesti figli).

Elementi contenuti: `<start>` (opzionale), `<end>` (opzionale) e `<include>` (opzionale).

Attributi:

`id` (obbligatorio solo se `<start>` non è presente) – Identificatore per il contesto, viene usato nei riferimenti al contesto corrente e deve essere unico all'interno del documento. Può contenere una stringa di lettere, cifre e trattini alti (`-`) o bassi (`_`).

`extend-parent` (opzionale) – Un valore booleano che indica al motore se il contesto possiede una priorità maggiore della fine del contesto genitore. Se non specificato viene inteso come `true`.

`end-at-line-end` (opzionale) – Un valore booleano che indica al motore se il contesto deve essere terminato forzatamente alla fine della riga, eventualmente visualizzando un errore. Se non specificato viene inteso come `false`.

Contesti sub-pattern

Si riferiscono a un gruppo all'interno di una delle espressioni regolari del contesto genitore, rendendo possibile colorare diversamente solo una o più porzioni dell'espressione regolare trovata.

Elementi contenuti: nessuno.

Attributi:

`id` (opzionale) – Identificatore per il contesto che deve essere unico all'interno del documento. Può contenere una stringa di lettere, cifre e trattini alti (`-`) o bassi (`_`).

`sub-pattern` (obbligatorio) – Il sub-pattern a cui si fa riferimento. `0` indica l'intera espressione regolare, `1` il primo gruppo, `2` il secondo, ecc. Se vengono usati sub-pattern con nome è anche possibile farvi riferimento specificandone il nome.

`where` (obbligatorio solo in contesti contenitori) – Può contenere `start` o `end` e deve essere usato solo in contesti contenitori per specificare se il sub-pattern a cui si fa riferimento è all'interno dell'espressione regolare contenuta nell'elemento `<start>` oppure nell'elemento `<end>`. In contesti semplici deve essere omesso.

Contesti riferimento

Vengono usati per includere contesti precedentemente definiti.
Elementi contenuti: nessuno.
Attributi:

ref (obbligatorio) – L'identificatore del contesto da includere. Mettendo la stringa `:*` alla fine dell'identificatore si indica al motore che il contesto genitore deve includere tutti i figli del contesto specificato, invece del contesto stesso. Facendo precedere l'identificatore del contesto dall'identificatore di un altro linguaggio, separati da due punti (`:`), è possibile fare riferimento a contesti definiti all'interno di tale linguaggio esterno.

Contesti keyword

Contengono un elenco di elementi `<keyword>` e corrispondono ad ogni parola chiave elencata. È possibile porre elementi opzionali `<prefix>` e `<suffix>` comuni ad ogni parola chiave.

Elementi contenuti: `<prefix>` (opzionale), `<suffix>` (opzionale) e `<keyword>` (uno o più).

Gli attributi disponibili sono gli stessi usati nei contesti semplici.

Tag `<include>`

Contiene l'elenco dei contesti contenuti nel contesto corrente.
Elementi contenuti: `<context>` (uno o più), `<define-regex>` (zero o più).

Tag `<match>`

Contiene l'espressione regolare per il contesto semplice corrente. L'espressione è nella forma usata negli elementi `<define-regex>`.
Elementi contenuti: nessuno.
Non possiede attributi.

Tag `<start>`

Contiene l'espressione regolare di ingresso per il contesto contenitore corrente. L'espressione è nella forma usata negli elementi `<define-regex>`.
Elementi contenuti: nessuno.
Non possiede attributi.

Tag `<end>`

Contiene l'espressione regolare di uscita per il contesto contenitore corrente. L'espressione è nella forma usata negli elementi `<define-regex>`, con una ulteriore

estensione: \%{sub-pattern@start} viene sostituito dal motore con la stringa corrispondente al sub-pattern specificato (sia mediante numero o mediante nome se vengono usati sub-pattern con nome) all'interno dell'elemento <start> precedente. Ad esempio si possono implementare i costrutti "here-document" delle shell Unix con il codice riportato nel *listato 5.1*.

Listato 5.1. Implementazione dei costrutti "here-document"

```
<context id="here-doc">
    <start>&lt;&lt;\s*(\S+)$</start>
    <end>\A\%{1@start}$</end>
</context>
```

Elementi contenuti: nessuno.
Non possiede attributi.

Tag <keyword>

Contiene una parola chiave che deve essere trovata dal contesto corrente. La parola chiave è una espressione regolare nella forma usata negli elementi <define-regex>.
Elementi contenuti: nessuno.
Non possiede attributi.

Tag <prefix>

Contiene un prefisso comune a ogni parola chiave seguente nel contesto corrente. Il prefisso è una espressione regolare nella forma usata negli elementi <define-regex>. Se non specificato viene utilizzato \%[come prefisso.
Elementi contenuti: nessuno.
Non possiede attributi.

Tag <suffix>

Contiene un suffisso comune a ogni parola chiave seguente nel contesto corrente. Il suffisso è una espressione regolare nella forma usata negli elementi <define-regex>. Se non specificato viene utilizzato \%] come suffisso.
Elementi contenuti: nessuno.
Non possiede attributi.

5.4 Confronto tra Language Description v1.0 e v2.0

Il formato delle descrizioni nella versione impiegata dal vecchio motore, oltre a non poter prevedere annidamenti tra contesti a causa delle limitazioni del motore stesso, prevede l'uso di numerosi elementi specifici e poco flessibili, la cui funzione è pesantemente influenzata dalla sintassi del linguaggio C.

Pertanto, tali elementi spesso risultano essere distinti in base al loro uso all'interno della descrizione del C, pur possedendo funzionalità del tutto identiche tra loro: ad esempio, la struttura degli elementi `<block-comment>`, `<string>` e `<syntax-item>` è esattamente la stessa, prevedendo una espressione regolare di inizio e una di fine.

Per evitare questo nella nuova sintassi si è passati ad un livello di astrazione superiore, impiegando il generico concetto di contesto per implementare ogni costrutto della sintassi. Agli elementi precedenti corrispondono i contesti contenitori, agli elementi `<pattern-item>` i contesti semplici, mentre agli elementi `<keyword-list>` i contesti keyword.

La gestione dei caratteri di escape, inoltre, era evidentemente basata sul comportamento previsto dal linguaggio C, risultando inutilizzabile per schemi differenti, prevedendo un elemento `<escape-char>` che il motore automaticamente inseriva all'interno di ogni contesto. Ciò provoca una colorazione errata anche nel linguaggio C stesso, in particolare inserendo erroneamente l'escape anche in commenti multi-riga (`/* */`), in quanto al loro interno tale carattere deve essere trattato come un carattere qualsiasi.

Nella nuova sintassi non vi è alcun trattamento speciale per i caratteri di escape, bensì si sfrutta la capacità del nuovo motore di gestire contesti annidati, includendo nei contesti per le stringhe un contesto di escape.

Un ulteriore scelta progettuale della nuova sintassi, infine, è stata quella di separare nettamente la descrizione della sintassi, contenuta nella sezione `<definitions>`, da quella presentazionale, contenuta nella sezione `<styles>`, dove vengono definiti chiaramente gli stili sulla base di ogni linguaggio, gestendone le associazioni agli stili predefiniti.

Capitolo 6

Analisi e colorazione della sintassi

6.1 Analisi sintattica

Prima di iniziare l'analisi sintattica viene caricato il file di descrizione del linguaggio appropriato e viene creata una tabella nella quale ad ogni identificatore di contesto è associata la lista delle possibili transizioni verso altri contesti.

Ogni identificatore di contesto viene preceduto dall'identificatore del linguaggio e da un carattere usato come separatore (`:`); in questo modo è possibile riferirsi a contesti definiti in altri file senza collisioni tra nomi, in modo simile a quanto avviene con i namespace in molti linguaggi di programmazione. Ad esempio, il contesto commento in HTML ha come identificatore "html:comment", mentre il commento Javascript è identificato come "js:comment".

L'analisi del testo contenuto nel GtkTextBuffer avviene creando in memoria un albero di contesti. Ogni contesto rappresenta lo stato dell'analizzatore sintattico in una specifica porzione di testo alla quale può essere associato uno stile che ne cambia il modo in cui questa venga mostrata all'utente, ad esempio l'albero associato all'esempio seguente (*listato 6.1*) è riportato in *figura 6.1*. Il file di descrizione del linguaggio utlizzato per questo e per gli altri esempi del capitolo è riportato in *appendice D*.

Listato 6.1. Esempio di codice C

```
#include <stdio.h>

#if 0
#  if A
#    if B
#    endif
#  endif
#endif
```

```
int funzione (int a, int b)
{
    // http://www.polito.it/
    printf ("A\nB\nC\n");
}
```

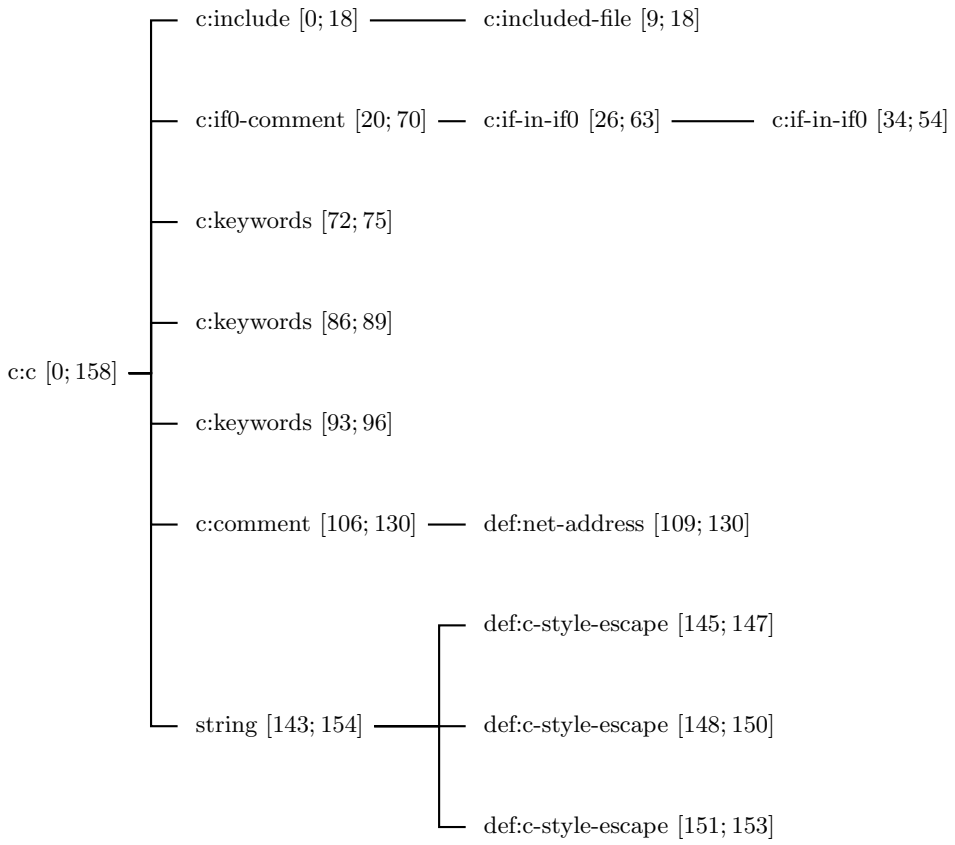

Figura 6.1. Albero sintattico per il *listato 6.1*

Inizialmente viene creata solo la radice dell'albero, la quale è costituita dal contesto il cui identificatore corrisponde a quello del linguaggio (per il linguaggio C, quindi, la radice avrà id "c:c"); questo contesto inizia alla posizione 0 e termina alla fine del buffer, per cui tutti gli altri contesti discendono dal contesto radice (ovvero sono contenuti in esso).

Terminata la fase di impostazione iniziale, il motore comincia ad effettuare l'analisi vera e propria, analizzando separatamente ogni riga (quindi una espressione regolare usata per definire i contesti non può estendersi su più righe) e verificando, carattere per carattere all'interno della riga stessa, se in tale posizione inizia un contesto figlio o se vi termina il contesto corrente. L'ordine delle transizioni elencate nel file di linguaggio è importante in quanto, nel caso in cui al carattere attualmente analizzato siano possibili

più transizioni, solo la prima è considerata. Il comportamento dell'analizzatore cambia in base al tipo di contesto, a seconda che questo sia un contesto contenitore, semplice o sub-pattern.

6.2 Contesti contenitori

I contesti contenitori sono i contesti che iniziano quando viene trovata una corrispondenza con l'espressione regolare di inizio (espressa, nel file di linguaggio, con il tag `<start>`) e terminano quando viene trovata una corrispondenza con l'espressione regolare di fine (tag `<end>`).

Per ogni carattere l'analizzatore deve quindi verificare se inizia un nuovo contesto raggiungibile da quello corrente (cioè per il quale esiste una transizione nella tabella creata inizialmente): se viene trovata una corrispondenza il contesto corrente viene aggiornato con quello appena individuato e si riprende l'analisi a partire dal termine del delimitatore trovato. Se, invece, al carattere specificato non inizia nessun nuovo contesto è necessario verificare se termina il contesto corrente, aggiornando posizione e contesto corrente di conseguenza, riportandosi nel contesto genitore. Solo se non vi è nessuna variazione di contesto si passa all'analisi del carattere successivo.

Listato 6.2. Contesti contenitori

```
/* Commento. */
#if 0
#  if A
#  endif
#endif
```

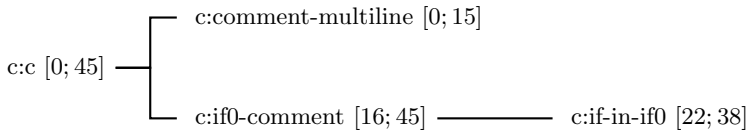

Figura 6.2. Albero sintattico per il *listato 6.2*

I passi necessari per analizzare il codice riportato nel *listato 6.2* (generando l'albero in *figura 6.2*) sono:

1. Inizialmente nell'albero dei contesti vi è solo il contesto radice "c:c" che inizia alla posizione 0, il contesto corrente è quindi "c:c".
2. Alla posizione 0 viene trovato `/*`, il nuovo contesto corrente è quindi "c:comment-multiline" e la posizione corrente è 2.
3. Alla posizione 13 non iniziano nuovi contesti ma vi è il terminatore di "c:comment-multiline", quindi il contesto corrente è nuovamente "c:c" e la nuova posizione è 15.

4. Alla posizione 16 inizia il contesto “c:if0-comment”.

5. Alla posizione 22 inizia il contesto “c:if-in-if0”.

6. Alla posizione 38 termina il contesto “c:if-in-if0” e il contesto corrente è nuovamente “c:if0-comment”.

7. Alla posizione 45 termina il contesto “c:if-0” e il contesto corrente è “c:c”.

6.3 Contesti semplici e keyword

I contesti semplici non sono caratterizzati da una espressione regolare di inizio e da una di fine ma da una singola espressione che corrisponde all’intero contesto (tag `<match>` nel file di linguaggio). Sono utili per esprimere contesti su una singola riga che possono contenere solo sotto-contesti di tipo sub-pattern.

Quando alla posizione corrente viene trovato un contesto di questo tipo, il contesto corrente non viene modificato e l’analisi riprende dopo la fine della corrispondenza, poiché non è necessario analizzare il testo per ricercare sotto-contesti.

Listato 6.3. Contesti semplici

```
/* FIXME: http://www.polito.it/ */
```

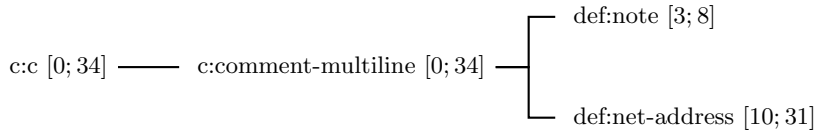

Figura 6.3. Albero sintattico per il *listato 6.3*

Nel *listato 6.3* si può vedere il codice contentente un contesto contenitore e un contesto semplice che porta alla creazione dell’albero in *figura 6.3* grazie ai seguenti passaggi:

1. Alla posizione 0 inizia un contesto contenitore “c:comment-multiline”, la nuova posizione corrente è 2.

2. Alla posizione 3 c’è un contesto semplice di tipo “def:note”, la posizione corrente viene spostata al carattere 8.

3. Alla posizione 10 c’è un contesto semplice di tipo “c:net-address”, la posizione corrente viene spostata al carattere 31.

4. Alla posizione 32 c’è il terminatore di “c:comment-multiline” che quindi termina alla posizione 34.

I contesti di tipo keyword internamente vengono convertiti, direttamente dal parser, in contesti semplici unendo le singole keyword in un'unica espressione regolare usando l'operatore |. Gli eventuali prefisso (tag `<prefix>`) e suffisso (`<suffix>`) vengono inseriti rispettivamente all'inizio e alla fine dell'espressione regolare. I valori di default per il prefisso e per il suffisso sono rispettivamente `\%[` e `\%]`, cioè i delimitatori di inizio e fine parola. Ad esempio la definizione di contesto nel *listato 6.4* viene trasformata in modo da essere equivalente alla definizione nel *listato 6.5*.

Listato 6.4. Definizione di contesti "keyword"

```
<context>
    <keyword>if</keyword>
    <keyword>for</keyword>
    <keyword>while</keyword>
    <keyword>do</keyword>
</context>
```

Listato 6.5. Equivalenza dei contesti "keyword"

```
<context>
    <match>\%[(if|for|while|do)\%]</match>
</context>
```

6.4 Contesti sub-pattern

Ogni coppia di parentesi all'interno delle espressioni regolari definisce un sub-pattern, essi sono numerati da sinistra a destra a partire da 1 mentre il sub-pattern 0 rappresenta l'intero testo corripondente all'espressione regolare. Ad esempio, usando l'espressione `#include (<.*>)` con il testo `#include <stdio.h>`, il sub-pattern 1 corrisponde al nome del file incluso, cioè `<stdio.h>`.

La libreria PCRE supporta anche i sub-pattern con nome in stile Python[1], nella forma `(?P<nome>...)`, dove `nome` può essere utilizzato al posto dell'identificatore numerico.

I contesti di tipo sub-pattern permettono, quindi, di colorare in modo diverso una parte del testo che corrisponde ad una espressione regolare specificata con `<match>`, `<start>` o `<end>`, usando i sub-pattern delle espressioni regolari.

Ad esempio il comando `#include` del C può essere definito con il codice riportato nel *listato 6.6*, colorando il nome del file incluso in modo differente.

Listato 6.6. Definizione del comando `#include`

```
<context id="include" style-ref="preprocessor">
    <match>
        \A#include (".*"|&lt;.*&gt;)
    </match>
```

[1] http://www.python.org/

```
    <include>
        <context sub-pattern="1" style-ref="included-file"/>
    </include>
</context>
```

6.5 Accelerazione della ricerca delle espressioni regolari

Utilizzando le espressioni regolari si possono eseguire ricerche ancorate o meno; una ricerca è ancorata se la corrispondenza può avvenire solo alla posizione specificata, non lo è se la corrispondenza può avvenire anche ad una posizione successiva a quella specificata.

L'analisi sintattica richiede che, per ogni carattere, avvenga una ricerca ancorata per ogni possibile transizione di contesto. Questo significa che, ad esempio, per una riga di dimensione media di 50 caratteri e con 10 espressioni regolari che potrebbero determinare una transizione, sarebbero necessarie 500 ricerche per riga, indipendentemente dal numero di contesti presenti nella riga. È chiaro che un approccio del genere sarebbe notevolmente lento.

Il nuovo motore, quindi, si comporta in modo diverso per aumentare le prestazioni:

1. Crea un'espressione regolare unendo con l'operatore | tutte le espressioni regolari che potrebbero determinare una transizione.

2. Esegue una ricerca non ancorata a partire dalla posizione corrente dell'espressione regolare creata al passo precedente.

3. Se non vi sono corrispondenze significa che nella riga non vi sono altre transizioni.

4. Altrimenti al carattere dove è situata la prima occorrenza esegue la normale procedura per determinare quale transizione deve avvenire.

L'algoritmo così modificato esegue una ricerca non ancorata e alcune ancorate invece di numerose ricerche ancorato con un notevole aumento di prestazioni (circa un ordine di grandezza).

6.6 Estensione e terminazione dei contesti antenati

Normalmente l'analizzatore sintattico ignora i delimitatori di fine di un contesto antenato all'interno del contesto corrente. Ad esempio nel *listato 6.2* il primo `#endif` determina la fine del contesto corrente ma non del contesto "c:if0", cioè il contesto "c:if-in-if0" estende il genitore "c:if0". Un altro esempio è il contesto "def:c-style-escape" che estende il genitore "c:string".

In altri casi, però, è necessario che il contesto possa essere terminato dal delimitatore genitore, ad esempio nel *listato 6.7* l'espressione regolare che definisce il contesto "def:net-address" può anche comprendere il terminatore `*/`, portando tutto il codice seguente ad essere considerato come un commento.

Listato 6.7. Terminazione errata di un contesto in C

```
/*http://www.polito.it/*/
int funzione (int a, int b);
```

Un altro problema simile si riscontra nel *listato 6.8*, qui il codice Javascript deve terminare con il tag `</script>` anche se questo si trova dentro a un commento Javascript.

Listato 6.8. Terminazione errata di un contesto in HTML

```
<script type="text/javascript">
function f ()
{
    return 42;
}
//</script>
```

Il problema viene risolto con l'attributo `extend-parent` del tag `<context>`, se questo è impostato a `true` (valore di default) allora il contesto può estendere il genitore, altrimenti no.

L'algoritmo deve, quindi, essere modificato in modo che controlli non solo la fine del contesto corrente, ma anche la fine di tutti gli antenati non estesi dal figlio.

6.7 Terminazione a fine riga

Alcuni contesti, come le stringhe in C, devono estendersi su una singola riga di codice, la mancanza del delimitatore di fine del contesto sulla stessa riga è da considerarsi un errore; questo caso è gestito dal motore grazie all'attributo `end-at-line-end` del tag `<context>` nel file di definizione di linguaggio. Se alla fine dell'analisi di una riga di testo un contesto con l'attributo `end-at-line-end` non esteso da un figlio è ancora aperto, allora è terminato insieme a tutti i suoi discendenti aperti.

L'attributo `end-at-line-end` non deve però essere utilizzato per la normale terminazione di contesti contenuti su una singola riga, come il commento C, ma solo quando la mancanza del delimitatore di chiusura deve essere considerata un errore. Per terminare un contesto come il commento C al termine della riga è sufficiente usare il simbolo di fine riga (`$`) nel tag `<end>`.

L'uso dell'attributo `end-at-line-end` ha effetti positivi sulle prestazioni durante la digitazione del testo, infatti l'apertura di un contesto di questo tipo può influenzare solo una riga e non tutto il contenuto del buffer dalla posizione della modifica in poi.

6.8 Pseudo-codice dell'algoritmo

Nel *listato 6.9* è riportato lo pseudo codice che descrive in modo semplificato l'algoritmo per la colorazione della sintassi.

Listato 6.9. Pseudo codice completo dell'algoritmo di analisi sintattica

```
/* Tipo di contesto. */
enum TIPO
    /* Definito con il tag <match>. */
    SEMPLICE
    /* Definito con i tag <start> e <end>. */
    CONTENITORE
end

/* Definizione di un contesto. */
class Definizione
    tipo         : TIPO
    /* Altre definizioni raggiungibili da quaesta
     * definizione. */
    figli        : list of Definizione
    /* Valore dell'attributo extend-parent. */
    estende      : boolean
    /* Valore dell'attributo end-at-line-end. */
    singola_riga : boolean
    /* Contenuto del tag <match>. */
    re_match     : Regex
    /* Contenuto del tag <start>. */
    re_start     : Regex
    /* Contenuto del tag <end>. */
    re_end       : Regex
end

/* Nodo dell'albero dei contesti. */
class Contesto
    /* Definizione per il contesto, ovviamente più contesti
     * possono condividere la stessa definizione. */
    definizione  : Definizione
    /* Contesto che contiene il contesto. */
    genitore     : Contesto
    /* Espressione regolare di tutte le possibili
     * transizioni. */
    completa     : Regex
    /* Posizione di inizio del contesto. */
    inizio       : integer
    /* Posizione di fine del contesto. */
    fine         : integer
end

/* Cerca l'espressione regolare "regex" in "testo" a partire
 * da "indice_inizio". Restituisce la posizione alla quale è
 * stata trovata la prima corrispondenza o -1 se la stringa
 * non corrisponde. */
function cerca_regex (regex, testo, indice_inizio) : integer
    [...]
```

```
end

/* Cerca l'espressione regolare "regex" alla posizione
 * "indice_inizio" in "testo". Restituisce "true" se
 * l'espressione regolare corrisponde. */
function match_regex (regex, testo, indice_inizio) : boolean
    [...]
end

/* Legge le definizioni dal file di linguaggio. */
function leggi_definizioni_da_file () : hash_table
    [...]
end

/* Applica i sotto contesti di tipo "tipo" a "contesto". */
procedure appl_sub_pattern (contesto, tipo)
    [...]
end

/* Crea l'espressione regolare di tutte le possibili transizioni
 * da "contesto". */
function crea_regex_transizioni (contesto) : Regex
    transizioni ← []

    /* Aggiunge le transizioni dovute ai figli. */
    for each figlio in def.figli
        if figlio.tipo = SEMPLICE
            transizioni.append (figlio.re_match)
        else if figlio.tipo = CONTENITORE
            transizioni.append (figlio.re_start)
    end

    /* Aggiunge la transizione dovuta alla fine del contesto
     * corrente. */
    transizioni.append (contesto.definizione.re_end)

    /* Aggiunge le transizioni dovute ai contesti antenati. */
    tmp ← contesto
    while tmp ≠ nil
        if tmp.definizione.estende = false
            transizioni.append (tmp.genitore.definizione.re_end)
        end
        tmp ← tmp.genitore
    end

    /* Concatena gli elementi della lista usando "|" come
     * separatore. */
    contesto.completa ← join_lista (transizioni, "|")
end
```

```
/* Verifica se un contesto antenato termina alla posizione
 * corrente. In caso affermativo restituisce il nuovo contesto
 * corrente, altrimenti restituisce "nil". */
function verifica_antenati (contesto, riga, pos_testo
                            indice) : Contesto
    /* Percorre tutti i contesti fino alla radice e inserisce
     * nella lista "da_verificare" tutti i contesti che
     * potrebbero interrompere il contesto figlio. */
    da_verificare ← []
    tmp ← contesto
    while tmp ≠ nil
        if tmp.definizione.estende = false
            da_verificare.append (tmp.genitore)
        tmp ← tmp.genitore
    end

    /* Percorre la lista al contrario, se un contesto in
     * "da_verificare" termina alla posizione corrente, allora
     * anche tutti i discendenti devono essere interrotti. */
    da_terminare ← nil
    for each antenato in da_verificare.reverse()
        if (match_regex (antenato.definizione.re_end, riga,
                         indice)
            da_terminare ← antenato
            break
        end
    end

    if da_terminare ≠ nil
        /* Termina tutti i discendenti. */
        tmp ← contesto
        while tmp ≠ da_terminare
            tmp.fine ← pos_testo
            tmp ← tmp.genitore
        end
        /* Termina il contesto antenato e restituisce il nuovo
         * contesto corrente. */
        da_terminare.fine ← pos_testo +
            da_terminare.definizione.re_end.lunghezza
        appl_sub_pattern (da_terminare, "end")
        indice ← da_terminare.fine
        return da_terminare.genitore
    end

    /* Nessun antenato termina alla posizione corrente. */
    return nil
end
```

```
/* Analizza una singola riga di testo. "contesto" è il
 * contesto corrente. */
function analizza_riga (contesto, riga, inizio_riga) : Contesto
    indice ← 0

    while true
        indice ← cerca_regex (contesto.completa,
                              riga, indice)
        if indice = -1
            /* Analisi della riga terminata, non ci sono
             * altri contesti sulla riga. */
            return break
        end

        pos_testo ← inizio_riga + indice
        trovato_contesto ← false

        /* Percorre la lista di tutti i contesti figli per
         * verificare se uno di essi inizia alla posizione
         * corrente. */
        for each def in contesto.definizione.figli
            if def.tipo = SEMPLICE
                if match_regex (def.re_match, riga, indice)
                    /* Crea il nuovo contesto. */
                    nuovo_contesto ← new Contesto
                    nuovo_contesto.definizione ← def
                    nuovo_contesto.genitore ← contesto
                    nuovo_contesto.inizio ← pos_testo
                    nuovo_contesto.fine ← pos_testo +
                        def.re_match.lunghezza
                    appl_sub_pattern (nuovo_contesto, "match")
                    /* Muove l'indice alla fine del nuovo
                     * contesto e prosegue l'analisi, senza
                     * modificare il contesto corrente. */
                    indice ← nuovo_contesto.fine
                    trovato_contesto ← true
                end
            else if def.tipo = CONTENITORE
                if match_regex (def.re_start, riga, indice)
                    /* Entra nel nuovo contesto creato. */
                    nuovo_contesto ← new Contesto
                    nuovo_contesto.definizione ← def
                    nuovo_contesto.genitore ← contesto
                    nuovo_contesto.inizio ← pos_testo
                    nuovo_contesto.completa ←
                        crea_regex_transizioni (nuovo_contesto)
                    appl_sub_pattern (nuovo_contesto, "start")
                    contesto ← nuovo_contesto
                    indice ← def.re_start.lunghezza +
```

```
                            pos_testo
                    trovato_contesto ← true
                end
            end

            if trovato_contesto = true
                /* Se più contesti possono iniziare alla
                 * posizione corrente solo il primo viene
                 * considerato. Per questo motivo è rilevante
                 * l'ordine di definizione dei contesti. */
                break
            end
        end

        if trovato_contesto = false
            /* Non è stato trovato nessun contesto figlio,
             * quindi si verifica se termina un contesto
             * antenato. */
            nuovo_contesto ← verifica_antenati (contesto,
                                riga, pos_testo, indice)
            if nuovo_contesto ≠ nil
                /* Aggiorna il contesto corrente. */
                contesto ← nuovo_contesto
            else
                /* Nessun contesto antenato termina alla
                 * posizione corrente, quindi si verifica
                 * se termina il contesto corrente. */
                if match_regex (contesto.definizione.re_end,
                                riga, indice)
                    contesto.fine ← indice
                    appl_sub_pattern (contesto, "end")
                    contesto ← contesto.genitore
                    indice ← pos_testo +
                        contesto.definizione.re_end.lunghezza
                end
            end
        end
    end

    return contesto
end

/* Analizza "testo" usando la sintassi del linguaggio con
 * "id_linguaggio" come identificatore. */
procedure analizza (testo, id_linguaggio)
    tutti_i_contesti ← leggi_definizioni_da_file ()

    /* Crea il contesto radice. */
    contesto ← new Contesto
```

```
    nome_radice ← id_linguaggio + ":" + id_linguaggio
    contesto.definizione ← tutti_i_contesti [nome_radice]
    contesto.genitore ← nil
    contesto.inizio ← 0
    contesto.completa ← crea_regex_transizioni (contesto)

    for each riga in testo
        contesto ← analizza_riga (contesto, riga,
                                  posizione di riga in testo)

        /* Termina i contesti attivi (cioè il contesto corrente
         * o gli antenati) che devono terminare a fine riga. */
        da_terminare ← nil
        tmp ← contesto
        while tmp ≠ nil
            if tmp.singola_riga = true
                da_terminare ← tmp
            tmp ← tmp.genitore
        end
        if da_terminare ≠ nil
            tmp ← contesto
            while tmp ≠ da_terminare.genitore
                tmp.fine ← fine di riga
                tmp ← tmp.genitore
            contesto ← da_terminare.genitore
        end
    end
end
```

6.9 Modifica del testo

L'algoritmo fin qui descritto funziona nel caso si analizzi tutto il testo presente nel GtkSourceBuffer ma, nel caso di modifiche al testo, non è possibile ripetere tutto il processo per motivi prestazionali.

È quindi necessario riutilizzare il più possibile le informazioni già presenti nell'albero dei contesti in modo da minimizzare la durata dell'analisi dopo ogni modifica. Per fare ciò l'albero viene diviso in due alberi, il primo contiene i contesti che sono rimasti sicuramente validi, il secondo invece contiene i contesti che potrebbero essere ancora validi. Nel secondo albero dei contesti le posizioni di inizio e fine di ogni nodo sono aggiornate per riflettere le nuove posizioni dopo l'inserzione o la cancellazione di un determinato numero di caratteri. Lo scopo di questo procedimento è il riutilizzo di parte o di tutto l'albero rimosso, se possibile.

L'analisi riparte dall'inizio della riga, infatti, se ripartisse dalla posizione della modifica, alcuni contesti potrebbero essere persi. Questo è il motivo per cui le espressioni regolari usate nei file di definizione dei linguaggi non possono estendersi su più righe; infatti, se ciò fosse possibile, un'inserzione potrebbe modificare completamente l'albero

sintattico fin dal primo carattere, rendendo così necessario rianalizzare tutto il testo contenuto nel buffer.

L'analisi procede come nel caso normale ma, ogni volta che un delimitatore viene trovato ad una posizione successiva a quella della modifica, l'analizzatore esegue le seguenti operazioni:

1. Sono eliminati i contesti sicuramente non più validi contenuti nel secondo albero, cioè i contesti che terminerebbero prima della posizione corrente di analisi.

2. Si considerano:

   - l'insieme dei contesti che vanno dalla radice al contesto corrente;
   - l'insieme dei contesti che vanno dalla radice del secondo albero fino al contesto che sarebbe il contesto corrente se quest'albero fosse ancora valido.

   Se i due insiemi hanno la stessa lunghezza e ogni contesto contenuto nel primo insieme ha la stessa definizione e la stessa posizione di inizio del contesto corrispondente nel secondo, allora i due alberi vengono unificati e l'algoritmo termina.

3. Altrimenti si procede nella ricerca del prossimo delimitatore.

Listato 6.10. Codice prima della modifica

```
a = 42 / primo */ + 1 /* secondo */;
```

Listato 6.11. Codice dopo la modifica

```
a = 42 /* primo */ + 1 /* secondo */;
```

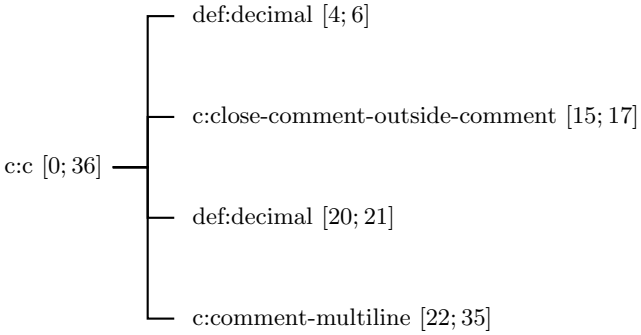

Figura 6.4. Albero sintattico per il *listato 6.10*

Come esempio si può considerare la modifica (l'inserzione di un `*` mancante) per passare dal codice riportato nel *listato 6.10* al codice nel *listato 6.11*:

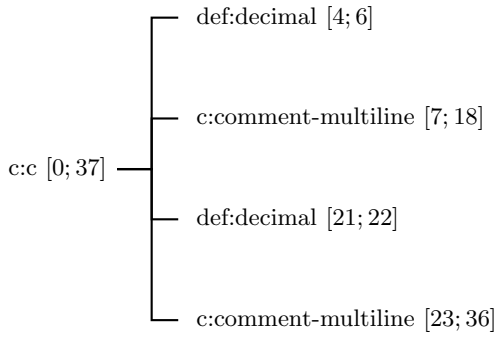

Figura 6.5. Albero sintattico per il *listato 6.11*

1. Poiché l'analisi deve riprendere da inizio riga in seguito all'inserzione del carattere, il motore rimuove dall'albero in *figura 6.4* tutti i contesti, fatta eccezione della radice "c:c", e inserisce quelli che iniziano in una posizione successiva a quella in cui è avvenuta la modifica in un albero temporaneo, dopo averne aggiornato le posizioni in base alla lunghezza del testo aggiunto (*figura 6.6*).

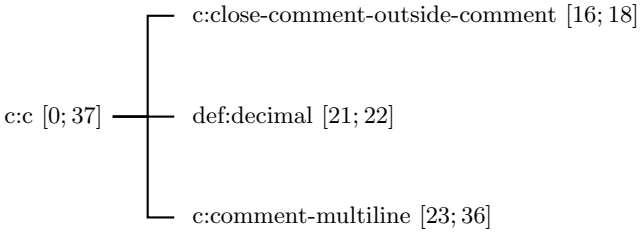

Figura 6.6. Albero dei contesti rimossi

2. L'analisi prosegue secondo il metodo normale fino alla posizione alla quale è avvenuta la modifica: in questo caso viene riconosicuta la stringa `42` come numero decimale e poi viene trovato l'inizio di un contesto "c:comment-multiline".

3. A questo punto è stata superata la posizione di inserzione e il motore tenta di riutilizzare i contesti precedentemente rimossi (il primo albero in *figura 6.7* è quello corrente, mentre il secondo contiene gli elementi rimossi).

4. Il primo tentativo di riutilizzo dell'albero sintattico fallisce in quanto i contesti da confrontare, sottolineati in *figura 6.7*, differiscono tra loro; l'analisi quindi procede e viene trovato il terminatore di "c:comment-multiline".

5. Dall'albero dei contesti rimossi viene eliminato il contesto "c:close-comment-outside-comment" in quanto si trova in una posizione già analizzata, quindi i due alberi in *figura 6.8* vengono confrontati; dal momento che alla posizione

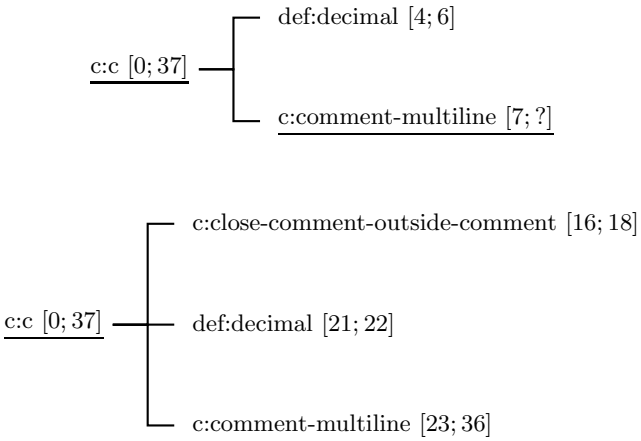

Figura 6.7. Alberi sintattici da confrontare (primo tentativo)

correntemente analizzata lo stato degli alberi corrisponde, questi possono venire riunificati, ottenendo l'albero in *figura 6.5*.

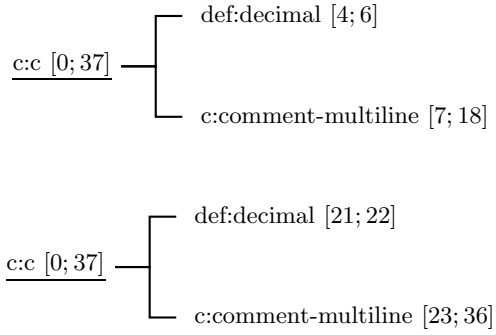

Figura 6.8. Alberi sintattici da confrontare (secondo tentativo)

6.10 Modifiche sincrone e asincrone

Le modifiche della colorazione del testo contenuto nel GtkSourceBuffer possono avvenire in modo sincrono, cioè riflettersi immediatamente dopo la modifica del testo, o asincrono, cioè essere applicare quando l'utente non sta digitando.

Il vantaggio dell'uso di un aggiornamento sincrono è l'immediatezza di riposta all'input dell'utente nella colorazione ma, nel caso il processo di aggiornamento sia lungo, può portare a rallentamenti nell'interfaccia grafica. Inoltre i ritardi dovuti alla scelta di un aggiornamento asincrono sono molto contenuti e poco visibili all'utente.

Nello sviluppo del motore di colorazione della sintassi si è scelta una soluzione

ibrida che utilizza sia modifiche sincrone sia asincrone. Nel caso più comune, cioè di inserzione o cancellazione di un singolo carattere, l'aggiornamento dell'albero dei contesti richiede generalmente un tempo molto breve, quindi l'operazione avviene in modo sincrono. Per testi più lunghi l'aggiornamento avviene asincronamente, in modo da non rallentare l'interfaccia grafica.

6.11 Divisione in blocchi per l'analisi

L'analisi del testo avviene riga per riga, ma il testo viene letto in blocchi di dimensione maggiore (qualche migliaio di byte) detti batch e poi diviso in memoria nelle singole righe; questo metodo richiede all'incirca un terzo del tempo richiesto per leggere singolarmente tutte le righe che compongono il batch.

Nel caso di modifica al testo, però, è probabile che le righe da analizzare, prima di riutilizzare il vecchio albero dei contesti, siano poche; la lettura di un intero batch, quindi, rallenta notevolmente l'algoritmo (dai test è risultato un rallentamento di circa il 500%). Per questi motivi, per massimizzare le prestazioni in caso di modifiche del testo, il motore legge le prime righe singolarmente, poi, se non è ancora stato possibile riutilizzare il vecchio albero dei contesti, passa al metodo di lettura basato su batch.

La dimensione dei batch non è fissa ma viene aggiornata dinamicamente facendo uso di un timer, in modo da adattarsi alle prestazioni del computer dell'utente.

Capitolo 7

Conclusioni

7.1 Prestazioni

Il motore sviluppato ha dimostrato buone prestazioni sia durante la fase di analisi dell'intero file, sia durante l'analisi in seguito a modifiche interattive. In media il vecchio motore risulta essere del 30% più veloce nel compiere l'analisi completa del file, rispetto al nuovo motore; questo è dovuto essenzialmente alla maggiore complessità delle operazioni svolte durante il processo. Questo risultato è perfettamente accettabile vista la maggiore correttezza fornita dal nuovo motore. Inoltre, l'operazione di colorazione vera e propria, compito svolto internamente dalla libreria GTK, risulta più dispendiosa in termini di tempo rispetto all'analisi, nonostante il nuovo motore presenti ancora possibilità di ulteriori ottmizzazioni.

Pur essendo più lento nell'analisi in seguito ad una modifica interattiva, il nuovo motore non presenta rallentamenti visibili all'utente, grazie alla capacità di riutilizzare i risultati della precedente analisi in modo più efficiente. Questo risulta essere particolarmente vero anche su macchine ormai considerate obsolete[1].

7.2 Semplicità delle descrizioni dei linguaggi

Nonostante il notevole aumento di complessità da parte del motore, le descrizioni dei linguaggi realizzate si sono mantenute sufficientemente chiare e semplici. La nuova sintassi, grazie alla sua flessibilità e genericità, ha consentito agevolmente il supporto di linguaggi molto differenti tra loro, quali C, XML e LaTeX. In alcuni casi è stato possibile ridurre notevolmente il codice richiesto per la descrizione di linguaggi derivati, come nel caso del C++, basato sul C al quale aggiunge solo alcune parole chiave.

[1] Il nuovo motore è stato sviluppato e provato su un portatile con processore PentiumIII con clock di 450 MHz.

7.3 Supporto del vecchio formato delle descrizioni

Al fine di garantire un'elevato grado di retrocompatibilità, è stata sfruttata la modularità del motore, affiancando il nuovo motore a quello già esistente. Questo porta ad una forte duplicazione del codice, con relativo aumento delle dimensioni della libreria.

Per contrastare questo problema è stato realizzato un foglio di stile XSLT, riportato in *Appendice E*, in grado di convertire le vecchie descrizioni nel nuovo formato in modo che possano essere utilizzate dal nuovo motore. Attualmente questa caratteristica non viene utilizzata automaticamente a causa di sottili incompatibilità tra le espressioni regolari in stile GNU, utilizzate dal vecchio motore, e quelle compatibili con Perl, utilizzate nella nuova implementazione.

Appendice A

Language Definition v2.0 reference

Overview

This is an overview of the Language Definition XML format, describing the meaning and usage of every element and attribute. The formal definition is stored in the RelaxNG schema file `language2.rng` which should be installed on your system in the directory `${PREFIX}/share/gtksourceview-1.0/schemas/` (where ${PREFIX} can be `/usr/` or `/usr/local/` if you have installed from source.

Tag `<language>`

The root element for Language Definition files.

Contained elements: `<metadata>` (optional), `<styles>` (optional), `<default-regex-options>` (optional), `<keyword-char-class>` (optional), `<definitions>` (mandatory).

Attributes:

`id` (mandatory) – Identifier for the description. This is used for external references and must be unique among language descriptions. It can contain a string of letters, digits, hyphens (`-`) and underscores (_).

`name` (mandatory) – The translatable name of the language presented to the user. It can be marked for translation putting an underscore before the attribute name (see the gettext documentation).

`version` (mandatory) – The version of the XML format (currently 2.0).

`section` (optional) – The translatable category in which the language has to be grouped when presented to the user. It can be marked for translation putting an

underscore before the attribute name. Currently used categories in GtkSourceView are Sources, Scripts, Markup and Others, but it is possible to use arbitrary categories (while usually discouraged).

mimetypes (optional) – The list of mimetypes associated to the language. Each mimetype has to be separated with a semicolon.

Tag <metadata>

Contains optional metadata about the language definition.

Contained elements: no element inside this one is currently defined, but it is possible to use arbitrary elements from different namespaces (e.g. Dublin Core metadata elements, etc.)

Tag <styles>

Contains the definitions of every style used in the current language and their association with predefined styles in GtkSourceView.

Contained elements: `<style>` (one or more).

Tag <style>

Defines a style, associating it's id with a user visible translatable name and a default style.

Contained elements: none.

Attributes:

id (mandatory) – Identifier for the style. This is used in the current language to refer to this style and must be unique for the current document. It can contain a string of letters, digits, hyphens (`-`) and underscores (_).

name (mandatory) – The user visible translatable name for the style. It has to be preceded with a underscore (_) to be marked for translation.

map-to (optional) – Used to map the style with a default style, to use colors and font properties defined for those default styles. The id of the default style has to be preceded with the id of the language where it is defined, separated with a semicolon :. When omitted the style is not considered derived from any style and will be not highlighted until the user specifies a color scheme for this style.

Tag <keyword-char-class>

Contains a regex character class used to redefine the customizable word boundary delimiters \%[and \%]. This class is the set of character that can be commonly found in a keyword. If the element is omitted the two delimiters default to \b.

Contained elements: none.

Tag <default-regex-options>

A string that will set the default regular expression options. Avaliable options are:

- `i`: case insensitive;
- `x`: extended (spaces are ignored and it is possible to put comments starting with # and ending at the end of the line);
- `s`: the metacharacter `.` matches the `\n`.

Those options can be overridden in every regular espression using the `/regex/options` syntax: to disable a group of enabled options, put an hyphen brefore them (e.g. `/[A-Z][a-z]*/-i` matches are case sensitive even if the default is to ignore case differencies).

Contained elements: none.

Tag <definitions>

The element containing the real description of the syntax to be highlighted. It contains one or more `<context>` element and an arbitrary number of `<define-regex>` elements, interleaved. It has no attributes. Every contained element must have its `id` attribute set to an identifier unique for the document. Exactly one of the contained `<context>` element must have the `id` attribute set to the `id` of the `<language>` root element, representing the initial context for the highlighting, the one the engine enters at the beginning of the highlighted file.

Contained elements: `<context>` (one or more), `<define-regex>` (zero or more).

Tag <define-regex>

Defines a regular expression that can be reused inside other regular expression, to avoid replicating common portions. Those regular expressions are PCRE regular expressions in the form `/regex/options` (see the documentation of PCRE for details). If there are no options to be specified and you don't need to match the spaces at the start and at the end of the regular expression, you can omit the slashes, putting here only `regex`. The possible options are those specified above in the description of

the <default-regex-options> element. To disable a group of options, instead, you have to prepend an hyphen - to them. In GtkSourceView are also available some extensions to the standard Perl style regular expressions:

- \%[and \%] are custom word boundaries, which can be redefined with the <keyword-char-class> (in contrast with \b);
- \%{id} will include the regular expression defined in another <define-regex> element with the specified id.

Contained elements: none.
Attributes:

id (mandatory) – Identifier for the regular expression. This is used for the inclusion of the defined regular expression and must be unique for the current document. It can contain a string of letters, digits, hyphens (-) and underscores (_).

Tag <context>

This is the most important element when describing the syntax: the file to be highlighted is partitioned in contexts representing the portions to be colored differently. Contexts can also contain other contexts. There are different kind of context elements: simple contexts, container contexts, sub-pattern contexts, reference contexts and keyword contexts.

Simple contexts

They contain a mandatory <match> element and an optional <include> element. The context will span over the strings matched by the regular expression contained in the <match> element. In the <include> element you can only put sub-pattern contexts.
Contained elements: <match> (mandatory).
Attributes:

id (optional) – A unique identifier for the context, used in references to the context. It can contain a string of letters, digits, hyphens (-) and underscores (_).

extend-parent (optional) – A boolean value telling the engine whether the context has an higher priority than the end of he parent. If not specified it defaults to true.

Container contexts

They contain a <start> element and an optional <end>. They respectively contain the regular expression that makes the engine enter in the context and the

terminating one. In the optional `<include>` element you can put contained contexts of every type (simple, container, sub-pattern or reference). If the `<start>` element is omitted, then the `id` attribute and the `<include>` become mandatory (the context can only be used as a container to include its children).

Contained elements: `<start>` (optional), `<end>` (optional), `<include>` (optional).

Attributes:

`id` (mandatory only if `<start>` not present) – A unique identifier for the context, used in references to the context. It can contain a string of letters, digits, hyphens (`-`) and underscores (_).

`extend-parent` (optional) – A boolean value telling the engine whether the context has an higher priority than the end of he parent. If not specified it defaults to `true`.

`end-at-line-end` (optional) – A boolean value telling the engine whether the context must be forced to end at the end of the line, displaying an error. If not specified it defaults to `false`.

Sub-pattern contexts

They refer to a group in a regular expression of the parent context, so it is possible to highlight differently only a portion of the matched regular expression.

Contained elements: none.

Attributes:

`id` (optional) – A unique identifier for the context. It can contain a string of letters, digits, hyphens (`-`) and underscores (_).

`sub-pattern` (mandatory) – The sub-pattern to which we refer. 0 means the whole expression, 1 the first group, 2 the second one, etc. If named sub-patterns are used you can also use the name.

`where` (mandatory only in container contexts) – Can be `start` or `end`. It has to be used only if the parent is a container context to specify whether the sub-pattern is in the regular expression of the `<start>` or the `<end>` element. In simple contexts it must be omitted.

Reference contexts

Used to include a previously defined context.

Contained elements: none.

Attributes:

ref (mandatory) – The id of the context to be included. A colon followed by an asterisk (`:*`) at the end of the id means that the parent should include every children of the specified context, instead of the context itself. Prepending the id of another language to the id of the context (separated with a semicolon :) is possible to include contexts defined inside such external language.

Keyword contexts

They contain a list of `<keyword>` and matches every keyword listed. You can also put a `<prefix>` and/or a `<suffix>` common to every keyword.

Contained elements: `<prefix>` (optional), `<suffix>` (optional), `<keyword>` (one or more).

The attributes are the same used in simple contexts.

Tag `<include>`

Contains the list of context contained in the current `<context>`.

Contained elements: `<context>` (one or more), `<define-regex>` (zero or more).

Tag `<match>`

Contains the regular expression for the current simple context. The expression is in the same form used in `<define-regex>` elements.

Contained elements: none.

Tag `<start>`

Contains the starting regular expression for the current container context. The expression is in the same form used in `<define-regex>` elements.

Contained elements: none.

Tag `<end>`

Contains the terminating regular expression for the current container context. The expression is in the same form used in `<define-regex>` elements, with an extension: `\%{sub-pattern@start}` will be substituted with the string matched in the corresponding sub-pattern (can be a number or a name if named sub-patterns are used) in the preceding `<start>` element. For instance you could implement shell-style here-documents with this code:

```
<context id="here-doc">
    <start>&lt;&lt;\s*(\S+)$</start>
```

```
    <end>^\%{1@start}$</end>
</context>
```

Contained elements: none.

Tag <keyword>

Contains a keyword to be matched by the current context. The keyword is a regular expression in the form used in `<define-regex>`.

Contained elements: none.

Tag <prefix>

Contains a prefix common to all of the following keywords in the current context. The prefix is a regular expression in the form used in `<define-regex>`. If not specified it defaults to `\%[`

Contained elements: none.

Tag <suffix>

Contains a suffix common to all of the following keywords in the current context. The suffix is a regular expression in the form used in `<define-regex>`. If not specified it defaults to `\%]`

Contained elements: none.

Appendice B

Language Definition v2.0 tutorial

A language definition for the C language

To describe the syntax of a language GtkSourceView uses an XML format which defines nested context to be highlighted. Each context roughly corresponds to a portion of the syntax which has to be highlighted (e.g. keywords, strings, comments), and can contain nested contexts (e.g. escaped characters.)

In this tutorial we will analyze a simple example to highlight a subset of C, based on the full C language definition.

Like every well formed XML document, the language description starts with a XML declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
```

After the usual preamble, the main tag is the `<language>` element:

```
<language id="c" _name="C" version="2.0" _section="Sources"
          mimetypes="text/x-c;text/x-chdr;text/x-csrc">
```

The attribute `id` is used in external references and defines a standard way to refer to this language definition, while the `name` attribute is the name presented to the user (it is translatable using gettext prepending a _.)

The attribute `section`, also translatable, tells the category where this language should be grouped when it is presented to the user. Currently available categories in GtkSourceView are Sources, Scripts, Markup and Others.

The attribute `mimetypes` contains a semi-colon separated list of mimetypes to be associated to the language.

The `<language>` element contains an optional `<metadata>`, which can contain arbitrary elements in different namespaces to specifiy author, creation time, etc., and `<styles>` element followed by a `<definitions>` element:

```
<styles>
```

This element contains every association between the styles used in the description and the defaults stored internally in GtkSourceView. For each style there is a `<style>` element:

```
<style id="comment" _name="Comment" map-to="def:comment"/>
```

This defines a `comment` style, which inherits the font properties from the defaults style `def:comment`. The `name` attribute contains the translatable name for this style, which is the name to show to the user.

For each style used in the language definition there is a corresponding `<style>` element; every style can be used in different contexts, so they will share the same appearance.

```
<style id="string"  _name="String"
                        map-to="def:string"/>
<style id="escape"  _name="Escape"
                        map-to="def:escape"/>
<style id="preprocessor" _name="Preprocessor"
                        map-to="def:preprocessor"/>
<style id="included-file" _name="Included File"
                        map-to="def:package"/>
<style id="char" _name="Character"
                        map-to="def:string"/>
<style id="keyword" _name="Keyword"
                        map-to="def:keyword"/>
<style id="data-type" _name="Data Type"
                        map-to="def:data-type"/>
</styles>
```

Following the `<styles>` element there is the `<definitions>` element, which contains the description proper of the syntax:

```
<definitions>
```

Here we should define a main context, the one we enter at the beginning of the file: to do so we use the `<context>` tag, with an `id` equal to the `id` of the `<language>` element:

```
<context id="c">
```

The element `<include>` contains the list of sub-contexts for the current context: as we are in the main context we should put here the top level contexts for the C language:

```
<include>
```

The first context defined is the one for single-line C style comments: they start with a double slash `//` and end at the end of the line:

```
<context id="comment" style-ref="comment">
    <start>\/\/</start>
    <end>$</end>
</context>
```

The `<start>` element contains the regular expression telling the highlighting engine to enter in the defined context, until the terminating regular expression contained in the `<end>` element is found.

Those regular expressions are PCRE regular expressions in the form `/regex/options` (see the documentation of PCRE for details.) If there are no options to be specified and you don't need to match the spaces at the start and at the end of the regular expression, you can omit the slashes, putting here only `regex`.

The possible options are:

- `i`: case insensitive;
- `x`: extended (spaces are ignored and it is possible to put comments starting with # and ending at the end of the line);
- `s`: the metacharacter `.` matches the `\n`.

You can set the default options using the `<default-regex-options` tag before the `<definitions>` element. To disable a group of options, instead, you have to preced them with a hyphen (`-`). [FIXME: add an example]

In GtkSourceView are available also some extensions to the standard perl style regular expressions:

- `\%[` and `\%]` are custom word boundaries, which can be redefined with the `<keyword-char-class>` tag (in contrast with `\b`);
- `\%{id}` will include the regular expression defined in the `<define-regex>` tag with the same id, useful if you have common portions of regular expressions used in different contexts;
- `\%{subpattern@start}` can be used only inside the `<end>` tag and will be substituted with the string matched in the corresponding sub-pattern (can be a number or a name if named sub-patterns are used) in the preceding `<start>` element. For an example see the implementation of here-documents in the `sh.lang` language description distribuited with GtkSourceView.

The next context is for C-style strings. They start and end with a double quote but they can contain escaped double quotes, so we sould make sure we don't end the string prematurely:

```
<context id="string" end-at-line-end="true"
        style-ref="string">
```

The `end-at-line-end` attribute tells the engine that the current context should be forced to terminate at the end of the line, even if the ending regular expression is not found, and that an error should be displayed.

```
<start>"</start>
<end>"</end>
<include>
```

To implement the escape handling we include a `escape` context:

```
    <context id="escape" style-ref="escape">
        <match>\\.</match>
    </context>
```

This is a simple context matching a single regular expression, contained in the `<match>` element. This context will extend its parent, causing the ending regular expression of the `string` context to not match the escaped double quote.

```
</include>
</context>
```

Multiline C-style comment can span over multiple lines and cannot be escaped, but to make things more interesting we want to highlight every internet address contained:

```
<context id="comment-multiline" style-ref="comment">
    <start>\/\*</start>
    <end>\*\/</end>
    <include>
        <context id="net-address" style-ref="net-address"
                 extend-parent="false">
```

In this case, the child should be terminated if the end of the parent is found, so we use `false` in the `extend-parent` attribute.

```
            <match>http:\/\/[^\s]*</match>
        </context>
    </include>
</context>
```

For instance in the following comment the string `http://www.gnome.org*/` matches the `net-address` context but it contains the end of the parent context (`*/`.) As `extend-parent` is false, only `http://www.gnome.org` is highlighted as an address and `*/` is correctly recognized as the end of the comment.

```
/* This is a comment http://www.gnome.org*/
```

Character constants in C are delimited by single quotes (`'`) and can contain escaped characters:

```
<context id="char" end-at-line-end="true" style-ref="string">
    <start>'</start>
    <end>'</end>
    <include>
        <context ref="escape"/>
```

The `ref` attribute is used when we want to reuse a previously defined context. Here we reuse the `escape` context defined in the `string` context, without repeating its definition.

```
    </include>
</context>
```

Using `ref` it is also possible to refer to contexts defined in other languages, preceding the id of the context with the id of the containing language, separating them with a colon:

```
<context ref="def:decimal"/>
<context ref="def:float"/>
```

The definitions for decimal and float constants are in a external file, with id `def`, which is not associated with any language but contains reusable contexts wich every language definition can import.

The `def` language file contains a `comment` context that can contain addresses and tags such as FIXME and TODO, so we can write a new version of our `comment-multiline` context that uses the definitions from `def.lang`.

```
<context id="comment-multiline" style-ref="comment">
    <start>\/\*</start>
    <end>\*\/</end>
    <include>
        <context ref="def:comment:*"/>
```

The `:*` after the id in the ref means the `comment-multiline` includes every children of the specified context, instead of the context itself.

```
    </include>
</context>
```

Keywords can be grouped in a context using a list of `<keyword>` elements:

```
<context id="keywords" style-ref="keyword">
    <keyword>if</keyword>
    <keyword>else</keyword>
    <keyword>for</keyword>
    <keyword>while</keyword>
    <keyword>return</keyword>
    <keyword>break</keyword>
```

```
    <keyword>switch</keyword>
    <keyword>case</keyword>
    <keyword>default</keyword>
    <keyword>do</keyword>
    <keyword>continue</keyword>
    <keyword>goto</keyword>
    <keyword>sizeof</keyword>
</context>
```

Keywords with different meaning can be grouped in different context, making possible to highlight them differently:

```
<context id="types" style-ref="data-type">
    <keyword>char</keyword>
    <keyword>const</keyword>
    <keyword>double</keyword>
    <keyword>enum</keyword>
    <keyword>float</keyword>
    <keyword>int</keyword>
    <keyword>long</keyword>
    <keyword>short</keyword>
    <keyword>signed</keyword>
    <keyword>static</keyword>
    <keyword>struct</keyword>
    <keyword>typedef</keyword>
    <keyword>union</keyword>
    <keyword>unsigned</keyword>
    <keyword>void</keyword>
</context>
```

You can also set a prefix (or a suffix) common to every keyword using the `<prefix>` and `<suffix>` tags:

```
<context id="preprocessor" style-ref="preprocessor">
    <prefix>^#</prefix>
```

If not specified, `<prefix>` and `<suffix>` are set to, respectively, `\%[` and `\%]`.

```
    <keyword>define</keyword>
    <keyword>undef</keyword>
```

Keep in mind that every keyword is a regular expression:

```
    <keyword>if(n?def)?</keyword>
    <keyword>else</keyword>
    <keyword>elif</keyword>
    <keyword>endif</keyword>
</context>
```

In C, there is a common pratice to use `#if 0` to express multi-line nesting comments. To make things easier to the user, we want to highlight these pseudo-comments as comments:

```
<context id="if0-comment" style-ref="comment">
    <start>^#if 0\b</start>
    <end>^#(endif|else|elif)\b</end>
    <include>
```

As `#if 0` comments are nesting, we should consider that inside a comment we can find other `#if`s with the corresponding `#endif`s, avoiding the termination of the comment on the wrong `#endif`. To do so we use a nested context, that will extend the parent on every nested `#if`/`#endif`:

```
    <context id="if-in-if0">
        <start>^#if(n?def)?\b</start>
        <end>^#endif\b</end>
        <include>
```

Nested contexts can be recursive:

```
            <context ref="if-in-if0"/>
        </include>
    </context>
    </include>
</context>
```

Because contexts defined before have higher priority, `if0-comment` will never be matched. To make things work we should move it before the `preprocessor` context, thus giving `if0-comment` a higher priority.

For the `#include` preprocessor directive it could be useful to highlight differently the included file:

```
<context id="include" style-ref="preprocessor">
    <match>^#include (".*"|<.*>)</match>
    <include>
```

To do this we use grouping sub-patterns in the regular expression, associating them with a context with the `sub-pattern` attribute:

```
        <context id="included-file" sub-pattern="1"
                 style-ref="package"/>
```

In the `sub-pattern` attribute we could use:

- 0: the whole regular expression;
- 1: the first sub-pattern (a sub-espression enclosed in parenthesis);

- 2: the second;
- ...
- `name`: a named sub-pattern with name `name` (see the PCRE documentation.)

We could also use a `where` attribute with value `start` or `end` to specify the regular expression the context is referring, when we have both the `<start>` and `<end>` element.

```
    </include>
</context>
```

Having defined a good subset of the C syntax we close every tag still open:

```
</include>
</context>
</definitions>
</language>
```

The full language definition

This is the full language definition for the subset of C taken in consideration for this tutorial:

```
<?xml version="1.0" encoding="UTF-8"?>
<language id="c" _name="C" version="2.0" _section="Sources"
        mimetypes="text/x-c;text/x-chdr;text/x-csrc">
    <styles>
        <style id="comment" _name="Comment"
                                map-to="def:comment"/>
        <style id="string" _name="String"
                                map-to="def:string"/>
        <style id="escape"  _name="Escape"
                                map-to="def:escape"/>
        <style id="preprocessor" _name="Preprocessor"
                                map-to="def:preprocessor"/>
        <style id="included-file" _name="Included File"
                                map-to="def:package"/>
        <style id="char" _name="Character"
                                map-to="def:string"/>
        <style id="keyword" _name="Keyword"
                                map-to="def:keyword"/>
        <style id="data-type" _name="Data Type"
                                map-to="def:data-type"/>
    </styles>
    <definitions>
        <context id="c">
```

```
<include>

    <context id="comment" style-ref="comment">
        <start>\/\/</start>
        <end>$</end>
    </context>

    <context id="string" end-at-line-end="true"
            style-ref="string">
        <start>"</start>
        <end>"</end>
        <include>
            <context id="escape"
                        style-ref="escape">
                <match>\\.</match>
            </context>
        </include>
    </context>

    <context id="comment-multiline"
                style-ref="comment">
        <start>\/\*</start>
        <end>\*\/</end>
        <include>
            <context ref="def:comment:*"/>
        </include>
    </context>

    <context id="char" end-at-line-end="true"
            style-ref="char">
        <start>'</start>
        <end>'</end>
        <include>
            <context ref="escape"/>
        </include>
    </context>

    <context ref="def:decimal"/>
    <context ref="def:float"/>

    <context id="keywords" style-ref="keyword">
        <keyword>if</keyword>
        <keyword>else</keyword>
        <keyword>for</keyword>
        <keyword>while</keyword>
        <keyword>return</keyword>
        <keyword>break</keyword>
        <keyword>switch</keyword>
        <keyword>case</keyword>
```

```
    <keyword>default</keyword>
    <keyword>do</keyword>
    <keyword>continue</keyword>
    <keyword>goto</keyword>
    <keyword>sizeof</keyword>
</context>

<context id="types" style-ref="data-type">
    <keyword>char</keyword>
    <keyword>const</keyword>
    <keyword>double</keyword>
    <keyword>enum</keyword>
    <keyword>float</keyword>
    <keyword>int</keyword>
    <keyword>long</keyword>
    <keyword>short</keyword>
    <keyword>signed</keyword>
    <keyword>static</keyword>
    <keyword>struct</keyword>
    <keyword>typedef</keyword>
    <keyword>union</keyword>
    <keyword>unsigned</keyword>
    <keyword>void</keyword>
</context>

<context id="preprocessor"
           style-ref="preprocessor">
    <prefix>^#</prefix>

    <keyword>define</keyword>
    <keyword>undef</keyword>
    <keyword>if(n?def)?</keyword>
    <keyword>else</keyword>
    <keyword>elif</keyword>
    <keyword>endif</keyword>
</context>

<context id="if0-comment"
           style-ref="comment">
    <start>^#if 0\b</start>
    <end>^#(endif|else|elif)\b</end>
    <include>
        <context id="if-in-if0">
            <start>^#if(n?def)?\b</start>
            <end>^#endif\b</end>
            <include>
                <context ref="if-in-if0"/>
            </include>
        </context>
```

```
                </include>
            </context>

            <context id="include"
                        style-ref="preprocessor">
                <match>^#include (".*"|<.*>)</match>
                <include>
                    <context id="included-file"
                            sub-pattern="1"
                            style-ref="inlcuded-file"/>
                </include>
            </context>

        </include>
    </context>
</definitions>
</language>
```

Appendice C

Schema Relax NG per Language Definition v2.0

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes
            ">
<start>
    <element name="language">
        <choice>
            <attribute name="name"/>
            <attribute name="_name"/>
        </choice>
        <attribute name="id">
            <data type="string">
                <param name="pattern">([a-zA-Z0-9_]|-)+</param>
            </data>
        </attribute>
        <attribute name="version"/>
        <choice>
            <attribute name="section"/>
            <attribute name="_section"/>
        </choice>
        <attribute name="mimetypes"/>

        <optional>
            <ref name="metadata"/>
        </optional>

        <optional>
            <ref name="styles" />
        </optional>

        <optional>
```

```
            <element name="default-regex-options">
                <text/>
            </element>
        </optional>

        <optional>
            <element name="keyword-char-class">
                <text/>
            </element>
        </optional>

        <ref name="definitions" />
    </element>
</start>



<define name="id-type">
    <data type="string">
        <!-- FIXME: Why it doesn't work?
             It seems that [a-z-] is unsupported -->
        <!--
        <param name="pattern">([a-zA-Z0-9_-]+:)?[a-zA-Z0-9_-]+</
           param>
        -->
        <param name="pattern">(([a-zA-Z0-9_]|-)+:)?([a-zA-Z0-9_]|-)
           +</param>
    </data>
</define>

<define name="ref-type">
    <data type="string">
        <!-- FIXME: Why it doesn't work?
             It seems that [a-z-] is unsupported -->
        <!--
        <param name="pattern">([a-zA-Z0-9_-]+:)?[a-zA-Z0-9_-]+(:\*)
           ?</param>
        -->
        <param name="pattern">(([a-zA-Z0-9_]|-)+:)?([a-zA-Z0-9_]|-)
           +(:\*)?</param>
    </data>
</define>

<define name="boolean-value">
    <choice>
        <value>true</value>
        <value>false</value>
    </choice>
</define>
```

```
<define name="itemInAnotherNamespace">
    <element>
        <anyName>
            <except>
                <nsName/>
                <nsName ns=""/>
            </except>
        </anyName>

        <zeroOrMore>
            <ref name="itemInAnotherNamespace"/>
            <text/>
        </zeroOrMore>
    </element>
</define>



<define name="metadata">
    <element name="metadata">
        <zeroOrMore>
            <ref name="itemInAnotherNamespace"/>
        </zeroOrMore>
    </element>
</define>

<define name="styles">
    <element name="styles">
        <oneOrMore>
            <element name="style">
                <attribute name="id">
                    <data type="string">
                        <param name="pattern">([a-zA-Z0-9_]|-)+</
                           param>
                    </data>
                </attribute>
                <choice>
                    <attribute name="name"/>
                    <attribute name="_name"/>
                </choice>
                <optional>
                    <attribute name="map-to">
                        <ref name="id-type"/>
                    </attribute>
                </optional>
            </element>
```

```
        </oneOrMore>
    </element>
</define>

<define name="definitions">
    <element name="definitions">
        <interleave>
            <oneOrMore>
                <choice>
                    <ref name="context-to-be-included"/>
                    <ref name="context-container"/>
                    <ref name="context-simple"/>
                    <ref name="context-reference"/>
                </choice>
            </oneOrMore>
            <zeroOrMore>
                <ref name="define-regex"/>
            </zeroOrMore>
        </interleave>
    </element>
</define>


<define name="context-simple">
    <element name="context">
        <optional>
            <attribute name="id">
                <ref name="id-type"/>
            </attribute>
        </optional>
        <optional>
            <attribute name="style-ref">
                <ref name="id-type"/>
            </attribute>
        </optional>
        <optional>
            <attribute name="extend-parent">
                <ref name="boolean-value"/>
            </attribute>
        </optional>

        <choice>
            <element name="match"><text/></element>

            <group>
                <optional>
                    <element name="prefix"><text/></element>
                </optional>
                <optional>
```

```
                    <element name="suffix"><text/></element>
                </optional>
                <oneOrMore>
                    <element name="keyword"><text/></element>
                </oneOrMore>
            </group>
        </choice>

        <optional>
            <element name="include">
                <oneOrMore>
                    <ref name="context-subpattern-simple"/>
                </oneOrMore>
            </element>
        </optional>
    </element>
</define>

<define name="context-container">
    <element name="context">
        <optional>
            <attribute name="id">
                <ref name="id-type"/>
            </attribute>
        </optional>
        <optional>
            <attribute name="style-ref"/>
        </optional>
        <optional>
            <attribute name="extend-parent">
                <ref name="boolean-value"/>
            </attribute>
        </optional>
        <optional>
            <attribute name="end-at-line-end">
                <ref name="boolean-value"/>
            </attribute>
        </optional>

        <element name="start"><text/></element>
        <optional>
            <element name="end"><text/></element>
        </optional>

        <optional>
            <element name="include">
                <interleave>
                    <oneOrMore>
                        <choice>
```

```
                            <ref name="context-container"/>
                            <ref name="context-simple"/>
                            <ref name="context-to-be-included"/>
                            <ref name="context-subpattern-container
                                "/>
                            <ref name="context-reference"/>
                        </choice>
                    </oneOrMore>
                    <zeroOrMore>
                        <ref name="define-regex"/>
                    </zeroOrMore>
                </interleave>
            </element>
        </optional>
    </element>
</define>

<define name="context-to-be-included">
    <element name="context">
        <attribute name="id">
            <ref name="id-type"/>
        </attribute>

        <element name="include">
            <interleave>
                <oneOrMore>
                    <choice>
                        <ref name="context-container"/>
                        <ref name="context-simple"/>
                        <ref name="context-to-be-included"/>
                        <ref name="context-reference"/>
                    </choice>
                </oneOrMore>
                <zeroOrMore>
                    <ref name="define-regex"/>
                </zeroOrMore>
            </interleave>
        </element>
    </element>
</define>

<define name="context-subpattern-simple">
    <element name="context">
        <optional>
            <attribute name="id">
                <ref name="id-type"/>
            </attribute>
        </optional>
        <optional>
```

```
            <attribute name="style-ref"/>
        </optional>

        <attribute name="sub-pattern"/>
    </element>
</define>

<define name="context-subpattern-container">
    <element name="context">
        <optional>
            <attribute name="id">
                <ref name="id-type"/>
            </attribute>
        </optional>
        <optional>
            <attribute name="style-ref"/>
        </optional>

        <attribute name="sub-pattern"/>

        <attribute name="where">
            <choice>
                <value>start</value>
                <value>end</value>
            </choice>
        </attribute>
    </element>
</define>

<define name="context-reference">
    <element name="context">
        <attribute name="ref">
            <ref name="ref-type"/>
        </attribute>
    </element>
</define>



<define name="define-regex">
    <element name="define-regex">
        <attribute name="id">
            <ref name="id-type"/>
        </attribute>
        <text/>
    </element>
</define>

</grammar>
```

Appendice D

Descrizione della sintassi per il linguaggio C

```
<?xml version="1.0" encoding="UTF-8"?>
<language id="c" _name="C" version="2.0" _section="Sources"
   mimetypes="text/x-c;text/x-chdr;text/x-csrc">
    <styles>
        <style id="comment"           _name="Comment"
                                      map-to="def:comment"/>
        <style id="comment-multiline" _name="Multiline Comment"
                                      map-to="def:comment"/>
        <style id="error"             _name="Error"
                                      map-to="def:error"/>
        <style id="string"            _name="String"
                                      map-to="def:string"/>
        <style id="escape"            _name="Escape"
                                      map-to="def:escape"/>
        <style id="preprocessor"      _name="Preprocessor"
                                      map-to="def:preprocessor"/>
        <style id="included-file"     _name="Included File"
                                      map-to="def:package"/>
        <style id="char"              _name="Character"
                                      map-to="def:string"/>
        <style id="keyword"           _name="Keyword"
                                      map-to="def:keyword"/>
        <style id="data-type"         _name="Data Type"
                                      map-to="def:data-type"/>
    </styles>

    <definitions>
        <context id="c">
            <include>

                <!-- Comments -->
```

```
<context id="comment" style-ref="comment">
    <start>\/\/</start>
    <end>$</end>
    <include>
        <context id="comment-continue">
            <match>\\\n</match>
        </context>
        <context ref="def:comment:*"/>
    </include>
</context>

<context id="comment-multiline" style-ref="comment-
   multiline">
    <start>\/\*</start>
    <end>\*\/</end>
    <include>
        <context id="open-comment-in-comment" extend
           -parent="false" style-ref="error">
            <match>\/\*</match>
        </context>
        <context ref="def:comment:*"/>
    </include>
</context>

<context id="close-comment-outside-comment" style-
   ref="error">
    <match>\*\/(?!\*)</match>
</context>

<!-- Preprocessor -->
<define-regex id="preproc-start">
    \A\s*#\s*
</define-regex>

<context id="if0-comment" style-ref="comment-
   multiline">
    <start>\%{preproc-start}if\s*0\b</start>
    <end>\%{preproc-start}(endif|else|elif)\b</end>
    <include>
        <context id="if-in-if0">
            <start>\%{preproc-start}if(n?def)?\b</
               start>
            <end>\%{preproc-start}endif\b</end>
            <include>
                <context ref="if-in-if0"/>
            </include>
        </context>
    </include>
</context>
```

```
<context id="include" style-ref="preprocessor">
    <match>
        /
            \%{preproc-start}
            include\s*
            (".*?"|&lt;.*&gt;)
        /x
    </match>
    <include>
        <context id="included-file" sub-pattern="1"
           style-ref="included-file"/>
    </include>
</context>

<context id="preprocessor" style-ref="preprocessor">
    <match>
        /
            \%{preproc-start}
            (define|undef|error|pragma|if(n?def)?|
               else|elif|endif|line)
            \b
        /x
    </match>
</context>

<!-- Strings -->
<context id="string" end-at-line-end="true" style-
   ref="string">
    <start>"</start>
    <end>"</end>
    <include>
        <context id="string-continue" style-ref="
           escape">
            <match>\\\n</match>
        </context>
        <context ref="def:c-style-escape"/>
    </include>
</context>

<context id="char" end-at-line-end="true" style-ref
   ="char">
    <start>'</start>
    <end>'</end>
    <include>
        <context ref="def:c-style-escape"/>
    </include>
</context>
```

```
<!-- Numbers -->
<context ref="def:decimal"/>
<context ref="def:octal"/>
<context ref="def:hexadecimal"/>
<context ref="def:float"/>

<!-- Keywords -->
<context id="keywords" style-ref="keyword">
    <keyword>asm</keyword>
    <keyword>break</keyword>
    <keyword>case</keyword>
    <keyword>continue</keyword>
    <keyword>default</keyword>
    <keyword>do</keyword>
    <keyword>else</keyword>
    <keyword>for</keyword>
    <keyword>fortran</keyword>
    <keyword>goto</keyword>
    <keyword>if</keyword>
    <keyword>return</keyword>
    <keyword>sizeof</keyword>
    <keyword>switch</keyword>
    <keyword>while</keyword>
</context>

<context id="types" style-ref="data-type">
    <keyword>_Bool</keyword>
    <keyword>_Complex</keyword>
    <keyword>_Imaginary</keyword>
    <keyword>auto</keyword>
    <keyword>char</keyword>
    <keyword>const</keyword>
    <keyword>double</keyword>
    <keyword>enum</keyword>
    <keyword>extern</keyword>
    <keyword>float</keyword>
    <keyword>int</keyword>
    <keyword>inline</keyword>
    <keyword>long</keyword>
    <keyword>register</keyword>
    <keyword>restrict</keyword>
    <keyword>short</keyword>
    <keyword>signed</keyword>
    <keyword>static</keyword>
    <keyword>struct</keyword>
    <keyword>typedef</keyword>
    <keyword>union</keyword>
    <keyword>unsigned</keyword>
    <keyword>void</keyword>
```

```
                    <keyword>volatile</keyword>
                </context>

                <context id="common-defines" style-ref="preprocessor
                    ">
                    <keyword>NULL</keyword>
                    <keyword>TRUE</keyword>
                    <keyword>FALSE</keyword>
                    <keyword>MAX</keyword>
                    <keyword>MIN</keyword>
                    <keyword>__LINE__</keyword>
                    <keyword>__DATA__</keyword>
                    <keyword>__FILE__</keyword>
                    <keyword>__func__</keyword>
                    <keyword>__TIME__</keyword>
                    <keyword>__STDC__</keyword>
                </context>

            </include>
        </context>
    </definitions>
</language>
```

Appendice E

XSLT per la conversione da versione 1.0 a 2.0

Viene riportato il prototipo di foglio di stile XSLT per la conversione delle descrizioni della sintassi versione 1.0 verso la versione 2.0.

I file generati con questo foglio di stile in alcuni rari casi non possono essere impiegati immediatamente in quanto la conversione automatica non è in grado di gestire le incompatibilità tra le espressioni regolari della libreria GNU, usate nelle descrizioni in versione 1.0, e le espressioni regolari in stile Perl, utilizzate dal nuovo motore.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0" >

<xsl:output method="xml" indent="yes"/>
<!--<xsl:strip-space elements="*"/>-->


<xsl:template match="/">
<xsl:comment> Automatically converted from language spec v1.0 with
     lang_v1_to_v2.xslt, written by Emanuele Aina &lt;emanuele.
        aina@tiscali.it&gt;
</xsl:comment>
<xsl:apply-templates/>
</xsl:template>

<xsl:template match="/language">
<language>
    <xsl:copy-of select="@*"/>
    <xsl:attribute name="version">2.0</xsl:attribute>
    <xsl:if test="@name">
        <xsl:attribute name="id">
            <xsl:value-of
```

```
                select="translate (@name, ' \/
                    ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                    abcdefghijklmnopqrstuvwxyz')"/>
    </xsl:attribute>
</xsl:if>
<xsl:if test="@_name">
    <xsl:attribute name="id">
        <xsl:value-of
              select="translate (@_name, ' \/
                 ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                 abcdefghijklmnopqrstuvwxyz')"/>
     </xsl:attribute>
</xsl:if>

<styles>
    <xsl:for-each select="line-comment|block-comment|string|
       syntax-item|pattern-item|keyword-list">
    <xsl:if test="@style">
        <style>
            <xsl:copy-of select="@_name"/>
            <xsl:attribute name="id">
                <xsl:value-of
                  select="translate (@_name, ' \/
                     ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                     abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
            <xsl:attribute name="map-to">
                <xsl:text>def:</xsl:text>
                <xsl:value-of
                  select="translate (@style, ' \/
                     ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                     abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
        </style>
    </xsl:if>
    </xsl:for-each>
</styles>
<definitions>
    <xsl:if test="/language/escape-char">
    <context id="escape">
        <match><xsl:value-of select="//escape-char"/>.</match>
    </context>
    </xsl:if>

    <context>
        <xsl:attribute name="id">
            <xsl:value-of
```

```
                    select="translate (/language/@_name, ' \/
                       ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                       abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
            <include>
            <xsl:apply-templates />
            </include>
        </context>
    </definitions>
</language>
</xsl:template>


<xsl:template match="escape-char">
    <!-- suppressed -->
</xsl:template>

<xsl:template match="line-comment">
    <context>
        <xsl:if test="@style">
            <xsl:attribute name="style-ref">
                <xsl:value-of
                  select="translate (@_name, ' \/
                     ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                     abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
        </xsl:if>
        <xsl:comment> Name: <xsl:value-of select="@_name" /> </
           xsl:comment>
        <match>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="start-regex" />
            <xsl:text>.*$/</xsl:text>
        </match>
    </context>
</xsl:template>

<xsl:template match="block-comment">
    <context>
        <xsl:if test="@style">
            <xsl:attribute name="style-ref">
                <xsl:value-of
                  select="translate (@_name, ' \/
                     ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                     abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
        </xsl:if>
        <xsl:comment> Name: <xsl:value-of select="@_name" /> </
           xsl:comment>
```

```
        <start>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="start-regex" />
            <xsl:text>/</xsl:text>
        </start>
        <end>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="end-regex" />
            <xsl:text>/</xsl:text>
        </end>
    </context>
</xsl:template>


<xsl:template match="string">
    <context>
        <xsl:if test="@style">
            <xsl:attribute name="style-ref">
                <xsl:value-of
                  select="translate (@_name, ' \/
                     ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                     abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
        </xsl:if>
        <xsl:comment> Name: <xsl:value-of select="@_name" /> </
           xsl:comment>
        <start>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="start-regex" />
            <xsl:text>/</xsl:text>
        </start>
        <end>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="end-regex" />
            <xsl:text>/</xsl:text>
        </end>
        <include>
            <context ref="escape" />
        </include>
    </context>
</xsl:template>


<xsl:template match="syntax-item">
    <context>
        <xsl:if test="@style">
            <xsl:attribute name="style-ref">
                <xsl:value-of
```

```
                select="translate (@_name, ' \/
                    ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                    abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
        </xsl:if>
        <xsl:comment> Name: <xsl:value-of select="@_name" /> </
            xsl:comment>
        <start>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="start-regex" />
            <xsl:text>/</xsl:text>
        </start>
        <end>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="end-regex" />
            <xsl:text>/</xsl:text>
        </end>
    </context>
</xsl:template>


<xsl:template match="keyword-list">
    <context>
        <xsl:if test="@style">
            <xsl:attribute name="style-ref">
                <xsl:value-of
                  select="translate (@_name, ' \/
                    ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                    abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
        </xsl:if>

        <xsl:comment> Name: <xsl:value-of select="@_name" /> </
            xsl:comment>

        <xsl:variable name="begin-empty" select="not(
            @match-empty-string-at-beginning='FALSE')"/>
        <xsl:variable name="end-empty" select="not(
            @match-empty-string-at-end='FALSE')"/>

        <xsl:choose>
            <xsl:when test="@beginning-regex">
                <prefix>
                    <xsl:if test="$begin-empty">\b</xsl:if>
                    <xsl:value-of select="@beginning-regex"/>
                </prefix>
            </xsl:when>
            <xsl:otherwise>
                <xsl:if test="not($begin-empty)">
```

```
                    <prefix />
                </xsl:if>
            </xsl:otherwise>
        </xsl:choose>
        <xsl:choose>
            <xsl:when test="@end-regex">
                <suffix>
                    <xsl:if test="$end-empty">\b</xsl:if>
                    <xsl:value-of select="@end-regex"/>
                </suffix>
            </xsl:when>
            <xsl:otherwise>
                <xsl:if test="not($end-empty)">
                    <suffix />
                </xsl:if>
            </xsl:otherwise>
        </xsl:choose>

        <xsl:for-each select="keyword">
            <keyword>
                <xsl:value-of select="."/>
            </keyword>
        </xsl:for-each>
    </context>
</xsl:template>


<xsl:template match="pattern-item">
    <context>
        <xsl:if test="@style">
            <xsl:attribute name="style-ref">
                <xsl:value-of
                  select="translate (@_name, ' \/
                    ABCDEFGHIJKLMNOPQRSTUVWXYZ', '---
                    abcdefghijklmnopqrstuvwxyz')"/>
            </xsl:attribute>
        </xsl:if>

        <xsl:comment> Name: <xsl:value-of select="@_name" /> </
           xsl:comment>

        <match>
            <xsl:text>/</xsl:text>
            <xsl:value-of select="regex" />
            <xsl:text>/</xsl:text>
        </match>
    </context>
</xsl:template>
```

```
<xsl:template match="comment()">
  <xsl:copy />
</xsl:template>

</xsl:stylesheet>
```