MixedCC and SPindent

Mixed-Layer Template Compiler Compiler and Server Page Indentator

Peter Kehl

MixedCC and SPindent: Mixed-Layer Template Compiler Compiler and Server Page Indentator

Peter Kehl
Copyright © 2004 Peter Kehl

Table of Contents

Chapter 1. Introduction

Mission	Syntax and structural validation, formatting, indentation, transformation of Mixed Layer templates. Realistic way - no semantic checks implemented.
Vision	Provide a generic and extensible framework for Mixed Layer definition, parsing and manipulation.
Values	Reusability, assurance, good documentation, open architecture.
Challenge	To analyze whether to move layer switch control and underling layer manipulation from SPindent to MixedCC.

Why "Mixed"?

Multiple layer	Templates have the layers separated in either unbroken blocks or different files. The layers refer to each other by links/anchors, identifiers or some sort of hooks/events.
Mixed layer	Templates have their layers intermixed, interlaced and their relations are implicitly given by position of elements to others.

This document doesn’t comment on what approach is better or more appropriate, either generally or for a specific purpose. The goal is to provide framework for Mixed layer templates.

Chapter 2. Mixed Parsing Principles

Step-by-step outside-driven parsing. Parallely run virtual state-less parsers that store the parser status and BNF call stack of their respective layer.

Divide et Impera. While mixed-layer language can be described by one grammar, it would be complicated to operate that way. We rather specify grammar of each layer. Tokens are then directed to respective RuleEngines. Insignificant Tokens are even filtered out and don't get to RuleEngine at all.

SPindent uses one JavaCC lexer for all layer's Tokens. Those are detected in different lexer States which reflect respective layers. Token stream is then directed to corresponding RuleEngine.

LookAheads aren't used. Postponed LookBehind (Flags) is used in validation/ implementation actions, rather than parsing control.

Benefits	Fast parsing, simple engine.
Drawback	BNF rules must be LL(1) transformed. Grammars are more flat and less readable. This might not be a big issue as SPindent doesn't need detailed grammars - it cares about structural statements only. If JavaCC-generated lexer is used then lexer clarity and BNF simplicity prevails over JavaCC tips on lexer speed enhancements which are not of concern there.

Standard way. To process outermost layer, generate its outputs - documents consisting of unprocessed inner layers - for each outer layer branch. Then to iterate the process for immediate inner layer. That allows for common parsers which are pipeline-connected. However that requires to process and generate full documents on each run.

That doesn't answer all questions either as layers might contain loop structures that depend on runtime parameters. Those would require assumptions of runtime situations by means of contract. It must be specified in some generic way so it would span accross the layers.

Such a technigue would require a sophisticated inference mechanism over symbolic statements. It might make burden for users by forcing them to specify extra information required by validation. And it might create a false notion of assurance (like compile type checking).

Languages without compile-time type checking such as PHP, Javascript and TCL would hardly conform that methodology.

MixedCC - its core is general RuleEngine. No reliant on lexer implementation which feeds it by Tokens, neither on specific grammars nor mixed layer hierarchy. SPindent - full-featured "implementation" that uses MixedCC and JavaCC-generated lexer to validate/indent JSP/PHP/HTML/XML with inner Javascript. Here SPindent represents any customized client of MixedCC.

Chapter 4. Stored Parser’s status and Grammar

Parser status is stack of "called" Rule.State objects represents a parser’s session. However, we use term State for SPindent lexer layer status which is stored in Tokens. Flags store additional information which is usually not essential for parsing and is used by Rule.Executives.

Each layer has its own grammar represented by BNF Rule.State object net. It is constructed using API, or by Ruler-generated class.

Rule	An envelope of Rule.State one of its nodes is a starting point.
Rule.State	Basic nodes of Parser’s stack machine. They can’t link accross different Rules, but can call either other or current Rule.
Rule.State	
Rule.Consumer	
Rule.Conditional	
Rule.Executive	
Rule.Proxy	

Chapter 5. RuleEngine

RuleEngine is BNF stacked Rule.State transition machine.

Accept/process. 2 phase mechanism first acceptable Rule.State is chosen (checked by RuleChecker). RuleStateChain contains Rule.Executive and Rule.Consumer items only. Rule.Proxys and Rule.Conditional are not stored to RuleStateChain because they bear no Parser action. Rule.Conditionals and final Rule.Consumer are evaluated in first stage. If all Rule.Conditionals and final Rule.Consumer of a given grammar branch accept current Token then the branch is *accepted.*

Consequence. Rule.State can’t access Flags created by prior Rule.States (ie Rule.Executives) before those are processed. That shouldn’t make sense anyway.

Due to accept/process technique the top of Rule.State stack is thrown away when a Rule returns. That implies two facts (not really restrictions): - Once a Rule returns, it must have been accepted. - Every Rule must accept at least one Token, either by itself or any Rule which it calls (possibly deeply), for any branch combination.

Mixed layers picture Level/Slice, ownership, copy-on-write Rule and Rule.State can’t contain any parser-session specific information. Rule-s don’t have any arguments. Those two roles are performed by Flags. Flags - BNF directives, act as BNF arguments.

Ruler transforms BNF JavaCC-like grammars to net of Rule objects. It suppports easy-syntax use of JavaCC-generated Tokens. Ruler itself is written in JavaCC. Ruler eliminates need of manipulation with Rule.State and their synthesis. Constructed Rule.State object net fulfils some of the correctness requirements, and the rest is verified by RuleChecker. Ruler partially prevents from mistaken use of a Rule.State in a different Rule and from empty Rules. That is done in build-time (ANT) and it makes development cycle faster. Those qualities along with clarity and intuitiveness surpass the overhead of additional Rule.Proxy objects that function as skeleton nodes, possibly optimized in future. RuleChecker ambiguity, reachability, left recursion