

Layout Handling in BitC[†]

Jonathan Shapiro, Ph.D.
The EROS Group, LLC

August 25, 2010

Abstract

One of the mechanisms that BitC borrows from Haskell is the “layout” system. We’re trying to take it a good bit farther than Haskell does, so it seems appropriate to describe how we manage it — if only as a sanity check.

In particular, the implementation used in `hugs98`, which is a `yacc/bison` based parser, is built on a tricky collaboration between the parser and the tokenizer to implement the layout phase. This approach has limitations we identify here. The BitC parsing subsystem adopts a slightly different technique that offers a cleaner, separation of phases, and consequently supports a broader set of layout design options.

1 Introduction

The Haskell “Layout” rules simplify the user-visible syntax of the language while preserving an unambiguous parse. While there are corner cases that it does not handle, the Haskell layout insertion policy has worked well for over twelve years. BitC uses a similar layout processing strategy. Our use of the feature is more aggressive than the one used in Haskell, and in consequence, we wanted a somewhat cleaner implementation.

Layout processing adds a layer between the parser and the tokenizer in which curly braces and semicolons¹ are automatically inserted. In effect, it introduces a form of staged parsing that is accomplished with minimal context information. The layout processor applies a set of rules based on the current token type, the previous token type, the current indentation, and a stack of currently outstanding blocks and their indentations. Most layout rules can be applied without knowledge of the current parse state or lookahead table.

This approach enables a different set of syntax design options than is possible by making tokens optional in the parser. With care, the two approaches can be combined, but there is a catch. The catch is, in some sense, the subject of this design note. The note describes how we achieve a cleaner separation of logic in BitC. In addition to providing a cleaner implementation of layout, the approach described is relevant to hand-assisted recovery from parse errors.

2 The Problem

As with most such capsule descriptions, a wealth of complication hides behind the word “most.” Problems begin to arise when the layout engine requires cooperation from the parser, as is the case in Haskell and BitC. In Haskell, the problematic layout rule in the specification is the one that appeals to information about parse errors:

Whenever a parse error is encountered, if insertion of a close brace will clear the error condition, do so and continue. This rule is applied iteratively.

This rule exists to force a closing `’}’` to be inserted in cases like:

[†] Copyright © 2010, Jonathan S. Shapiro.

This is a provisional design note and disclosure whose implementation is not yet committed.

¹ Or other tokens, but like Haskell the BitC scheme is concerned with curly braces and semicolons.

```

-- User types:
let x = 5 in ...
-- Parser sees:
let { x = 5 . in ...
      ^ error here
      ^ inserted by layout processor
-- But parser error fixup produces:
let { x = 5 } in ...

```

Unfortunately, implementing this rule requires a tie-in between the parser and the tokenizer. Here is the relevant parser fragment from the `hugs98` implementation, which is a `yacc`-based LALR(1) parser:

```

end: '{\}'      {\} /* okay */ {\}
end: error     {\}
      errorok; /* cancel error */
      /* fix up lexer's context stack as if
      '{\}' had been seen and proceed. */ {\}

```

Note that *if* the `}` had actually been present in the input stream, the layout processing layer would have taken different actions. In this case, it would have popped a context entry from the indentation context stack maintained by the layout manager. The action associated with the error handler here is fixing things up by hand, which is sufficient for the implementation requirements of Haskell-98 and Haskell-2010. Unfortunately, this won't suffice for BitC.

3 The Problem in BitC

BitC is an eager evaluation language, and uses `'{'` and `'}'` to enclose semicolon-separated sequences of expressions. In consequence, we perform implicit `'{'` insertion after `then` and `else`, leading to code like:

```

-- User types:
let x = if true then 5 else 4 in ...
-- Parser sees:
let { x = if true then { 5 } else { 4 . in ...
      ^ error here
-- Parser fixup needs to produce:
let { x = if true then { 5 } else { 4 } } in ...

```

Where the Haskell rule is to insert one close brace and see if that fixes matters, the BitC rule is:

Before the token `in`, close all implicit open blocks back to the matching `let`, `letrec`, or `case`.

Note that this rule cannot be implemented blindly without support from the parser! If the closing braces are present in the input, then code like:

```

-- User types:
let x = let { y = 5 } . in y in ...
-- Parser sees:
let { x = let { y = 5 } in y . in ...
      ^ parse error here
      ^ Careful: inner let already closed!
-- Parser fixup needs to produce:
let { x = let { y = 5 } in y } in ...

```

The specification relies on the fact that *closing braces will only be inserted when a parse error makes them necessary*, but also relies on the fact that *multiple closing braces may need to be inserted*. The trigger for action must come from the parser, but the knowledge of what to do should reside in the layout processing layer.

The practical problem here is that: (1) a lookahead token has already been read by the parser (2) that lookahead token is being used to determine that shifting the (distinguished) `error` token is appropriate, (3) if the `'}'` had actually been present in the input stream, the layout processing layer might have taken different actions, and (4) *those different actions might include injecting a different token ahead of the lookahead token that has already been delivered.*

The saving grace in this situation is in two parts:

1. Since we are prepared to fix this situation, the parser knows what the “correct” token at this point should have been. The fact that the `end` production has reduced by the wrong path isn't a problem, because we know what the proper path should have produced.
2. While the *next* token has already been fetched into the lookahead buffer, it will not be examined by the parser until after the reduce action has completed. With care, *we can reach into the parser state and replace it.*

4 The BitC Solution

The implementation approach in `hugs98` is clever, which in this context is a four letter word. One consequence is that the implementation of layout in `hugs98` is spread between the parser and the tokenizer. Another is that it abuses the error recovery mechanism in a way that makes error recovery in some circumstances challenging:

```
let x = { 5 . in ...
        ^ Fix is insert '}' because we see the "in".
        Otherwise we might insert ';' or '}'
```

The choice here matters! If the recovery decision is to proceed by inserting a `';`, the subsequent token stream returned by the layout engine will likely be altered. It is not enough to choose the right recovery token; we want to implement the choice in such a way that synchronization between the layout processing layer and the parse layer are maintained.

In a hand-written parser, we would probably solve this by implementing a bounded amount of token push-back and then proceeding as follows:

1. Push the current lookahead token currently held by the parser (if any) back onto the tokenizer push-back stack.
2. Push the missing `'}'` token back onto the tokenizer push-back stack.
3. Call `tokenizer.getToken()` to re-read the `'}'` token that was missing, for the sake of any processing side effects that this action will trigger in the layout layer.
4. Replace the lookahead state in the parser with the result of another call to `tokenizer.getToken()`.

Note that because the layout processing layer has now explicitly processed the missing `'}'` token, it is possible that the token returned by this call is something other than the originally held lookahead token.

Essentially the same implementation is possible in `yacc` or `bison`. The last step is implemented by advising the parser that the lookahead buffer is empty and must be re-filled.

5 Why This Is Better

To see why this approach is an improvement relative to the `hugs98` approach, it's important to remember that Haskell uses braces in fewer contexts than BitC, that only *one* of these contexts involves inserting an *open* brace, and that in that particular context it is easy to fix up the layout context stack. In BitC, for example, we want to automatically insert braces after the keyword `in` and around type declarations:

```

let x = 5 in . ...
           ^ brace here

struct S .
         ^ brace here
  i : int32

```

In contrast to close brace insertion, the layout layer has to process these inserted braces, because it must determine from the following token what the new prevailing indent level should be. In particular, perverse (but legal) input like:

```

// User types
let x = 5 in
...

// Layout insertions:
let { x = 5 } in {
           ^ inserted by error rule, then processed
};...
^ Inserted by matching start column rule
^ Inserted by layout by close-to-column rule

// Parser ultimately sees:
let { x = 5 } in {}; // typed as unit
...

```

can be processed correctly by all layers, and valid input like:

```

// User types
struct S .
  i : int32
  f : float
...

// Layout inserts:
struct S {
           ^ inserted by error rule, then processed
  i : int32
  ^ first token after '{', so establishes a new
  indent level
  ;f : float
  ^ Inserted by matching start column rule
};...
^ Inserted by matching start column rule
^ Inserted by layout by close-to-column rule

// Parser ultimately sees:
struct S {
  i : int32;
  f : float
};
...

```

can be processed consistently and sanely. From a practical perspective, this is why the BitC grammar can get away with requiring semicolons around type declarations without exposing them to users who strongly prefer an indentation-oriented syntax. The resulting surface syntax as seen by the user provides a clear improvement in readability.

Finally this approach places nicely with other techniques such as making tokens optional at the grammar level. Early (pre-layout) versions of BitC dealt with semicolons using a grammar rule:

```
SC: ';' {\} {\}  
SC: {\} {\}
```

and then used `SC` in the grammar body wherever `' ; '` had previously appeared. This technique remains available to us if it proves useful, and (if desired) can be implemented in a way that cooperates correctly with the layout management layer.

6 Conclusion

Layout processing is a useful technique that relies on error-driven feedback at the parser/tokenizer interface. This requirement interacts awkwardly with token lookahead. As a result, layout processing cannot *quite* be implemented as a staged preprocessor acting on the token stream. The approach described here shows how push-back techniques can be used to preserve synchronization across a staged parser, allowing a consistent view of the input stream to be maintained by all layers. This approach works even though some of the layers require lookahead.

Particular thanks is due to Mark Jones for help in understanding the “devious” implementation of layout (his term) used in `hugs98` and to various members of the BitC IRC community for prompting me to look for a cleaner approach.

References

- [1] **NOT USED** Unicode Consortium. The Unicode Standard, version 4.1.0, defined by *The Unicode Standard Version 4.0*, Addison Wesley, 2003, ISBN 0-321-18578-1, as amended by *Unicode 4.0.1* and by *Unicode 4.1.0*. <http://www.unicode.org>.