Beyond ASCII – Parsing Programs with Graphical Presentations

Martijn M. Schrage
(Utrecht University, The Netherlands
martijn@cs.uu.nl)

S. Doaitse Swierstra
(Utrecht University, The Netherlands
doaitse@cs.uu.nl)

Abstract: Proxima is a generic structure editor suitable for a wide range of structured document types. It allows edit operations on the document structure as well as on its screen representation (i.e. free-text editing), without the need to switch between the two modes. The system maintains a bidirectional mapping between the document structure and its presentation. Besides obvious applications, such as word-processor and spread-sheet editors, Proxima is especially well-suited for defining source editors for programming languages.

Presentation-oriented edit operations require that an edited presentation can be parsed to yield an updated document structure. However, conventional parsing techniques cannot readily be applied, since presentations in Proxima are not restricted to text but may also contain graphical elements. For example, an exponential may be presented as 3^2. Although this graphical presentation may not be directly edited at the presentation level, its components may. Hence, instead of simply parsing the changed representation, we have to take into account the existing structure.

This paper explains the scanning and parsing process for presentations that are a possibly nested combination of text and graphical elements. For textual parts of the presentation a Haskell combinator parser needs to be provided. The parser for graphical parts, on the other hand, is constructed by Proxima, based on information in the presentation. White space in the presentation can be handled automatically, if desired.

Key Words: presentation-oriented editing, parsing, Haskell
Category: D.1.1., D.2.6.

1 Introduction

The generic structure editor Proxima [Schrage 2004] is suitable for a wide range of structured document types. Its key feature is the modeless combination of structural editing and presentation editing. Figure 1 is a screenshot of Proxima at work, showing an editor for the functional programming language Helium [Heeren et al. 2003]. The editor shows inferred type signatures and provides a list with type information for the identifiers in scope. Note that various graphical presentations of code fragments are supported, as shown in the declaration.

When editing program code, a structure-based view of the text often comes in handy. One might want to select a complete function or a subexpression by

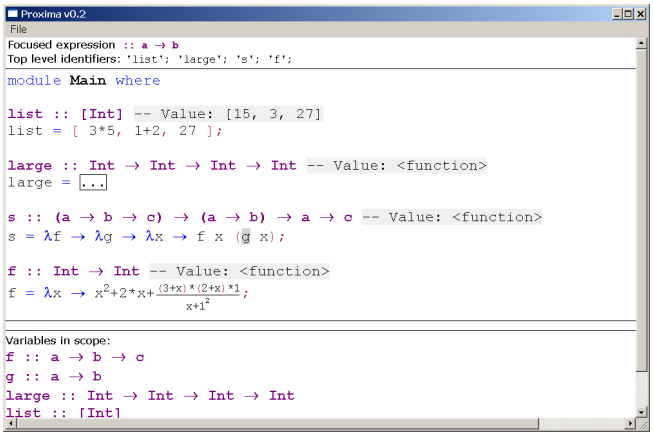

Figure 1: An editor for Helium.

a simple gesture and be assisted by an editor that knows the structure of the document. We refer to such edit operations as *structural* or *document-oriented*, edit operations.

On the other hand, we also have *presentation-oriented* edit operations that do not necessarily correspond to meaningful operations on the document. If, for example, the middle part of the expression $(1 + \boxed{2) \times (}\, 3 + 4)$ is deleted, we again get a correct expression $(1 + 3 + 4)$. This edit operation does not directly correspond to a structural edit operation.

In order to support editing on both the document structure and the presentation, Proxima maintains a bidirectional mapping between two data structures: the structural description of the document, and its actual presentation on the screen. This mapping is described as a composition of a number of smaller mappings, several of which are parameterized by so called *sheets*. Together with a document type definition, these sheets form the instantiation of an editor. By supplying a document type, a presentation sheet, and a scanner and parser, a syntax-aware editor may be constructed with little effort.

To illustrate the scanning and parsing process, we focus on a small example. Figure 2 shows a declaration that has a mixed textual and graphical presentation. The fraction and the exponentiation both have a graphical presentation, which we will refer to as a *structural* presentation. A textual presentation, on the other hand, is denoted by the term *sequential* presentation.

The figure shows that a sequential presentation may contain structural presentations (e.g. the sequential presentation of the declaration contains a structural fraction), and vice versa. The fraction, for example, has sequential presentations for the numerator and the denominator (the latter containing a structural presentation again for 3^2).

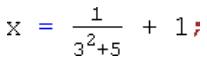

Figure 2: A graphically presented declaration.

Because of the complexity of parsing visual languages in general, Proxima disallows presentation editing on structural presentations. Any sequential subpresentation (such as the numerator and denominator), however, is editable again. Hence, in Figure 2 we can type "`+1`" next to the `1` in the numerator, but we cannot delete the horizontal line. We can delete the entire fraction though, by putting the caret after it and pressing backspace. A fraction may be introduced using a menu, or by typing a '/' operator, which is replaced by a graphical fraction after the next successful parse. The caret remains at the right place after such a transformation.

Summarizing, we have two kinds of presentations:

- **Sequential presentation** A presentation that consists of a sequence of items, each of which is either text or a structural presentation. It supports structure editing as well as presentation editing.
- **Structural presentation** A possibly graphical presentation that does not support presentation editing. It may contain either nested structural or sequential presentations, with the latter supporting presentation editing again.

After the presentation of a document has been edited, the modified presentation needs to be mapped back onto the document. However, due to the mix of structural and sequential presentations, conventional parsing methods cannot readily be used. In this paper, we show how the Proxima scanner and parser co-operate in this reverse mapping.

Figure 3 shows the result of the Proxima scanner when it is applied to the presentation in Figure 2. It consists of a nested structure of `SequentialTk` tokens and `StructuralTk` tokens that matches the structure of the presentation. Textual tokens are represented by `UserTk` tokens. Each token has a unique number, which is denoted by a subscript, and is referred to as its *presentation identity*. Presentation identities have type `IDP` and are used to associate a token with its white space. In addition to a sequence of tokens that represent its children, a structural token also contains a reference to the structured-document fragment it represents (denoted by the parameter `loc`). The document fragment is necessary, because a structural presentation does not always contain enough information to reconstruct the document tree. In the `scannedDecl` value, we represent the location reference by the document subtree. (A `SequentialTk` token also contains a `loc` parameter, as well as a parser, but for brevity, these are not shown in the value.) Together with the tokens, the scanner produces a *white-space map*: a mapping between a token's presentation identity and its trailing white space.

```
data Token loc userToken
 = SequentialTk IDP loc (Parser userToken) [Token node userToken]
 | StructuralTk IDP loc                    [Token node userToken]
 | UserTk IDP userToken String
 | ErrorTk IDP String

scannedDecl :: Token loc UserToken
scannedDecl =
 SequentialTk0
  [ UserTk1 (IdentToken "x")
  , UserTk2 (OpToken "=")
  , StructuralTk3 (DivExp (IntExp 1) (PlusExp ...))
    [ SequentialTk4 [ UserTk5 (IntToken 1) ]
    , SequentialTk6 [ StructuralTk7 (PowerExp (IntExp 3) (IntExp 2))
                      [ SequentialTk8 [ UserTk9 (IntToken 3) ]
                      , SequentialTk10 [ UserTk11 (IntToken 2) ] ]
                    , UserTk12 (OpToken "+")
                    , UserTk13 (IntToken 5) ] ]
  , UserTk14 (OpToken "+")
  , UserTk15 (IntToken 1)
  , UserTk16 (SymToken ";") ]

whitespaceMap :: IntMap IDP (NrOfBreaks, NrOfSpaces)
whitespaceMap = [1 ↦ (0,1), 2 ↦ (0,1), 3 ↦ (0,1), 14 ↦ (0,1), 16 ↦ (1,0)]
```

Figure 3: A scanned declaration.

Parsing the token structure in Figure 3 is relatively straightforward. For the sequential parts, we use a combinator parser that has a special primitive for structural tokens. The presentation sheet specifies the parser to be used. The structural parts, on the other hand contain enough information to be parsed without the need to specify a parser.

The paper is organized as follows. We start by providing a brief overview of Proxima's architecture (Section 2) and the kind of document types that can be defined (Section 3). Then, we explain the components and data types involved in computing the presentation of a document in Section 4. The Proxima scanning and parsing algorithms, explained in sections 5 and 6, form the core technical content of this paper. Section 7 describes related work, and Section 8 concludes.

2 The architecture of Proxima

The core architecture of Proxima consists of a number of layers, each communicating with its direct neighbors. The layered structure is based on the staged nature of the *presentation* process and its inverse, the *interpretation* process. The positions at which the document, the rendering, and the intermediate data structures reside are called *levels*. Between each pair of levels we have a *layer* that maintains the mappings between its adjacent levels. Each layer consists of a presentation component and an interpretation component and may be pa-

rameterized by a *sheet*. Figure 4 schematically shows the levels and layers of Proxima. From a document type definition, a code generator generates a number of Haskell modules, which are compiled together with the sheets and the Proxima base modules to yield an editor.

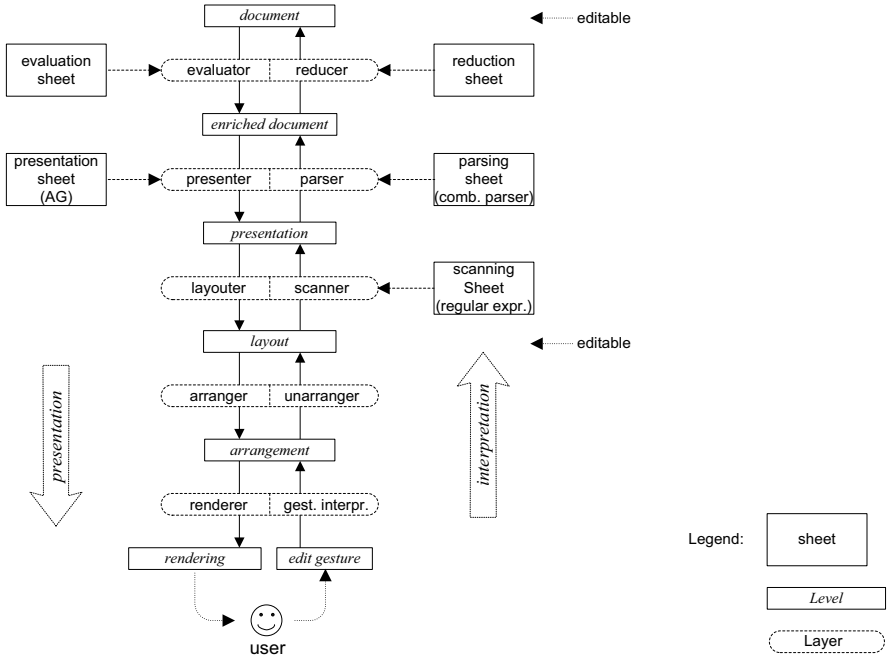

Figure 4: The levels and Layers of Proxima.

A data level in Proxima is not simply an intermediate value in the presentation computation. It is an entity in its own right and maintains part of the state of the editor. The six levels of Proxima are:

- **Document:** The document structure.
- **Enriched Document:** The document attributed with derived values and structures, such as the type of a function or a table of contents, typically computed by an attribute grammar [Reps and Teitelbaum 1984].
- **Presentation:** A logical description of the presentation of the document, consisting of rows and columns of presentation elements with attributes. The presentation also supports formatting based on available space (e.g. line breaking).

```
data Decl = Decl ident:Identifier exp:Exp {idP1,idP2}

data Identifier = Ident str:String        {idP1}

data Exp = PlusExp  exp1:Exp exp2:Exp     {idP1}
         | DivExp   exp1:Exp exp2:Exp     {idP1}
         | PowerExp exp1:Exp exp2:Exp     {idP1}
         | IntExp   val:Int               {idP1}
```

Figure 5: A document type for simple declarations.

- **Layout:** A presentation with explicit white space, which does not contain tokens.
- **Arrangement:** A formatted layout with absolute size and position information.
- **Rendering:** A bitmap of the arrangement.

Presentation-oriented editing actually takes place at the layout level rather than the presentation level, thus allowing free-text editing also on white space (which is absent on the presentation level). Hence the two levels that are directly editable are the document level and the layout level. After an edit operation on the document, all levels from document to rendering are updated to reflect the update. After an edit operation on the layout level, the modified layout is scanned, parsed and reduced, to obtain the corresponding updated document, from which an updated rendering is computed. Scanning and parsing does not occur after every presentation edit operation. Depending on the editor, it may occur either on a navigation operation, after a certain time interval, or at an explicit request by the user.

In this paper, we will focus mainly on the presentation layer and layout layer, and, more specifically, on the scanner and parser in these layers. Because we do not refer to the evaluation layer, we will use the term document also to refer to the enriched document.

3 The document structure

The document type in Proxima is a monomorphic (i.e. parameter free) Haskell data type with lists. Figure 5 shows the definition for a type `Decl` that represents declarations of simple expressions. This definition is the base from which a code generator creates a Haskell data type, as well as a number of utility functions.

In contrast to Haskell syntax, named fields are specified by putting an identifier and a colon in front of a child type. Furthermore, at the right of each constructor we have a number of identifiers that declare presentation identity fields, which refer to the tokens occurring in the presentation. For example,

```
Decl (Identifier "x")
     (PlusExp (DivExp (IntExp 1)
                      (PlusExp (PowerExp (IntExp 3) (IntExp 2))
                               (IntExp 5)))
              (IntExp 1))
```

Figure 6: A value of type `Decl`.

`Decl` has two fields, one for the equals sign and one for the semicolon, whereas `PlusExp` has only one for the plus operator. A future version of Proxima will allow presentation identities to be specified in the presentation sheet rather than the document type. In the Haskell data type that is generated from this type definition, the presentation identity fields are placed in front of the other fields. Furthermore, two special constructors are added to each type: a *hole* constructor for representing incomplete documents, and *parse error* constructor for representing parse errors.

If we leave away the presentation identities, the declaration in Figure 2 is represented by the value in Figure 6. (The full version is provided in Section 6.3.)

4 The presentation process

Before discussing the scanner and parser components, we briefly discuss their counterparts in the presentation direction: the presenter and layout components. The presenter component maps an (enriched) document onto the presentation level, according to rules in the presentation sheet, which is specified with an attribute grammar. The presentation itself is a value of type `Xprez`, computed in a compositional way, using the `Xprez` combinator language. After the presentation phase, the presentation level is mapped onto the layout level by the layout component. In the next three subsections, we introduce the language XPREZ, the attribute grammar formalism used in the presentation sheet, and the layout component.

4.1 The XPREZ presentation language

XPREZ [Schrage 2004] is a combinator library for specifying graphical presentations with support for alignment and stretching. The basic building blocks of XPREZ are text and graphical elements such as polygons, circles, and images. Combinators are used to combine presentations in rows and columns. The elements of a row or column are aligned along horizontal and vertical reference lines of their children, and do not overlap. Besides rows and columns, XPREZ also supports overlapping presentations and a flow layout for line breaking. We

```
frac :: Xprez -> Xprez -> Xprez
frac e1 e2 = let numerator   = hAlignCenter (pad (shrink e1) )
                 denominator = hAlignCenter (pad (shrink e2) )
             in  colR 2 [ numerator, vSpace 2, hLine
                        , vSpace 2, denominator ] `withHStretch` False

pad xp = row [ hSpace 2, xp, hSpace 2 ]

shrink e = e `withFontSize_` (\fs -> (70 `percent` fs) `max` 10)
```

Figure 7: The definition of `Frac`.

give an introduction to the language based on an example function `frac`, defined in Figure 7. The result of `frac (text "1") (text "1+x")` is:

$$\frac{1}{1+x}$$

The `colR` combinator takes an argument that denotes which of its children provides the vertical reference line (in this case, the horizontal line in the fraction). The `withHStretch` function prevents the fraction from being horizontally stretchable. Finally, to shrink presentations, we use the combinator `withFontSize_`, which takes a function argument that computes the new font size, given its previous value. Besides combinators that produce presentations, Xprez also has combinators for specifying edit operations in context menus, reactions to mouse clicks, and keeping track of document locations in the presentation.

4.2 Document presentation

For the presentation of the document, as well as for the computation of derived values and structures, Proxima uses the attribute grammar formalism. The presentation sheet is a file with an attribute grammar definition, which is compiled to Haskell by the Utrecht University AG compiler [Swierstra et al. 2008].

For each type of node in the document, the presentation sheet defines a synthesized attribute `pres` of type `Xprez`. The definition of `pres` may refer to presentations of children of the document node. Besides the presentation, any number of attributes can be defined on the document tree. In this way we can easily add static checks or compute all variables in scope at some document location. Moreover, functions from external Haskell modules may be called, allowing for more complex computations, such as type checking.

Each presentation rule states whether the presentation is sequential or structural. A sequential presentation consists of a sequence of tokens, which may be strings or structural presentations. In the presentation sheet, the top-most element of the sequential presentation (the one that is an immediate child of a

```
SEM Decl
  | Decl    loc.pres = sequence parseDecl [ @ident.pres, key @idP1 "="
                                           , @exp.pres, symb @idP2 ";" ]
SEM Exp
  | PlusExp loc.pres = sequence parseExp [ @exp1.pres
                                         , operator @idP1 "+"
                                         , @exp2.pres ]
  | DivExp  loc.pres = sequence parseExp [ structuralToken @idP1 $
                                           frac @exp1.pres @exp2.pres ]

key      idp str = token idp str `withColor` blue
operator idp str = token idp str `withColor` green
sym      idp str = token idp str `withColor` orange
```

Figure 8: Presentation sheet fragment.

structural presentation) must specify a parser, which is the parser to be applied to the sequence after it has been edited.

Since a structural presentation may not be edited at the presentation level, it is straightforward to map it back onto the document level, even if it has a graphical presentation. Hence, no parser needs to be specified in the presentation sheet.

Figure 8 shows three presentation rules for the declaration document type from Section 3. For brevity, the rules for `Ident`, `PowerExp`, and `IntExp` have been omitted. The Haskell type system enforces a sequential presentation to consist only of tokens and nested sequential presentations. Two functions are available for creating tokens: `token` and `structuralToken`. The first parameter of both functions is a presentation identity, the value of which comes from one of the `idP` fields specified in the document type definition (Figure 5). Note that we not only specify a parser for `Decl`, since it is the top-level type, but also for `Exp`, since it may appear as a child of the structurally presented `DivExp` (or `PowerExp`). The definition of these parsers is provided in Section 6.

4.3 The layout component

The main function of the layout component is to restore the implicit white space that is kept separately in the white-space map. Besides the white space, also the presentation focus (i.e. the caret or the selection) is stored in the white-space map. This is necessary because parsing followed by presenting is not necessarily an identity mapping. After parsing, document structures represented by tokens may be presented graphically, and thus give rise to a restructured presentation. Recall that typing a '`/`' in the Helium editor causes the introduction of a graphical fraction after parsing and presenting. The focus restoration mechanism ensures that after parsing and presenting the presentation focus remains in the same token, provided it is still part of the presentation. If the presentation

focus cannot be restored from the tokens, the layout component will restore it by using its absolute coordinates in the presentation.

Apart from white space, the layout level has the same structure as the presentation level. Hence, analogously, we have *sequential* and *structural* layouts. The layout component is not parameterized by a sheet. The reason is that although we need a specification in order to create tokens from a string, the reverse process is straightforward, since each token contains its string representation.

5 Scanning

The Proxima scanner maps character sequences in sequential layouts onto tokens, as specified in the scanning sheet. If the sequential layout contains any structural layouts, these are recursively scanned and recorded by special tokens.

5.1 The `Token` type

Tokens are represented by the type `Token`, of which a simplified version is shown in Figure 3. A number of type parameters that are not important for this discussion are left out. Each constructor has an `IDP` field that denotes its presentation identity number. Both `SequentialTk` and `StructuralTk` have a `loc` field that refers to the node in the document tree from the presentation of which the token originated. (Because `ErrorTk` tokens originate from the scanner they do not have this field.) The `loc` field is used when parsing structural layouts (which is explained in Section 6.1.)

A `SequentialTk` represents a sequential layout. The `Parser userToken` field is the parser that is used to parse the list of child tokens. This list contains no further `SequentialTk` tokens; tokens from all sequential descendents are collected and placed in the same list. On the other hand, a `StructuralTk` represents a structural layout and has a list of tokens for each of its child layouts. Finally, a `UserTk` represents a string token, and an `ErrorTk` is used to represent lexical errors.

5.2 Scanning the layout tree

The scanner creates a tree of structural and sequential tokens that matches the structure of the layout level. Its behavior is determined by the kind of layout on which it is called.

Structural layout. A structural layout of a document node is an Xprez tree containing layouts stemming from the presentation of child nodes. The scanner traverses the layout tree and makes a recursive call on each child layout that is encountered. The list of child tokens is put in a `StructuralTk` and returned as the result of the scanner.

```
data
  UserToken =
    IdentToken String | OpToken String | SymToken Char | IntToken Int

$opChar = [\+ \- \=]
$symChar = [\;]
tokens :-
 $digit+                       { mkToken $ \s   -> IntToken (read s) }
 $opChar+                      { mkToken $ \s   -> OpToken s         }
 $symChar                      { mkToken $ \[c] -> SymToken c        }
 $lower [$alpha $digit \_ \']* { mkToken $ \s   -> IdentToken s      }
```

Figure 9: Example `UserToken` and scanning sheet.

Sequential layout. A sequential layout consists of a column of rows, which contain either strings or structural layouts. Each structural layout is mapped onto a structural token by recursively scanning it. The sequences of strings between the structural tokens are first extended with new-line characters to mark the transitions between rows. The resulting lists of characters are mapped onto lists of `UserTk` tokens according to the scanning sheet. The final list of child tokens is the result of merging the structural tokens with the recognized user tokens.

5.3 The scanning sheet

The lexical analysis of textual tokens is based on the Haskell lexical analyzer generator Alex [Marlow 2007], which is comparable to the lex and flex tools for C and C++. An editor designer has to define the data type `UserToken` and provide an Alex specification for the tokens. Figure 9 shows an example `UserToken` and scanning sheet. The Alex specification consists of a a number of macro definitions followed by a set of rules, each defining a token. A rule is a regular expression together with an action that constructs the token.

5.4 Handling white space

In order to use the automatic white-space recognition, the following rule must be added to the scanning sheet:

```
[\n \ ]+        { collectWhitespace }
```

This rule causes the scanner to emit a special white-space token for each sequence of white space. A post-processing phase removes these white-space tokens, and records the trailing white space for each token in the white-space map. For the first token, also its leading white space is recorded. In case there are no tokens, any white space is associated with the presentation identity of the `SequentialTk` value that contains the list of tokens.

The white-space model is currently somewhat limited, since white space is assumed to be a number of line breaks followed by a number of spaces. However, the model is easily extended to handle arbitrary white space and also comments that need to be treated similar to white space.

6 Parsing

Unlike ordinary parsers, which take a list of tokens to produce a value, the Proxima parser is a function that takes only one token as input. This token can be either a structural token or a sequential token. In case of a structural token, the value is constructed automatically from the list of child tokens. If the token is a sequential token, its list of children is fed into the parser that was specified in the presentation sheet.

6.1 Structural presentations

A structural token corresponds to the presentation of a certain document node and contains a list of tokens that correspond to presentations of children of that document node. Each child may be presented multiple times, or even not at all. Furthermore, the order in which the child presentations appear may not correspond to the order of the children in the document node.

Nevertheless, we can parse a structural token automatically, since each child token contains a `loc` reference to the document node (and path) of which it is a presentation. Hence, for each token, we can determine the corresponding child of the document node. Because structural presentations are not editable at the presentation level, this information remains valid under presentation editing.

For child i of the document node, the parser takes the list of tokens corresponding to presentations of that child. If this list is empty, the presentation does not contain a presentation for the child, and we use its previous value, which is stored in the structural token. If the list is not empty, the token for the presentation that was edited is recursively parsed to yield a value for child i. In case no presentation was edited, then the child value is also reused from the structural token for efficiency reasons. There will be at most one edited presentation, since Proxima does not allow editing multiple presentations of a single value at the same time.

6.2 Sequential presentations

The parser for a sequential presentation cannot be constructed automatically. Instead, we use the parser that was specified in the presentation sheet, which is stored in the `SequentialTk` value. Such parsers are specified using the UU Parsing library [Hughes and Swierstra 2003, Swierstra 2008]. This is a library

```
parseDecl :: Parser Decl
parseDecl = (\id eqIdp exp semiIdp-> Decl eqIdp semiIdp id exp)
        <$> parseIdent <*> pToken (KeyToken "=")
        <*> parseExp   <*> pToken (SymToken ';')

parseExp :: Parser Exp
parseExp = pStructural Node_Div
       <|> pStructural Node_Power
       <|> (\plusIdp e1 e2 -> PlusExp plusIdp e1 e2)
           <$> pToken (OpToken "+") <*> parseExp <*> parseExp
```

Figure 10: Parsing-sheet fragment for `Decl` and `Exp`.

for creating fast error correcting parsers with support for user-friendly error messages. Because of the error recovery, parsing does not stop after the first error, and hence multiple parse errors in different parts of the presentation can be reported.

A Proxima parser is very similar to a regular parser specified with a combinator parser library. The only difference is that, in order to let the scanner handle white space and focus restoration, the presentation identities of the parsed tokens need to be stored in the appropriate fields of the document node.

The list of tokens to which the parser is applied consists only of `UserTk`, `StructuralTk`, and `ErrorTk` tokens, since nested `SequentialTk` tokens are not created by the scanner. Primitive parsers are available for user tokens and structural tokens. No primitive parser is offered for error tokens, since these represent a lexical error by the scanner, which should always lead to a parse error.

Figure 10 shows the parsing sheet for the declaration type from the previous examples. The parser does not take into account priorities. The `pToken` parser is a primitive parser that succeeds on its argument `UserToken` value and returns the presentation identity of the token. The `pStructural` parser succeeds on a structural token for the constructor that is denoted by its argument. The argument has type `Node`, which is a generated type that represents the union of all constructors in the document type.

Note that we do not restore the presentation identity of the `SequentialTk` value itself in the `Exp` parser. The reason is that this is only necessary if the parser accepts an empty list of tokens. This is not true for the expression parser, so when only white space is present, a parse error is returned, which takes care of handling the white space. For a top-level parser that does accept an empty token list (e.g. a parser for a list of declarations), a special combinator is available that provides the parser with the presentation identity of the `SequentialTk` value.

Although Proxima currently uses the UU Parsing library, this connection is not fixed. In fact, any parser library that is parameterized with its input token type is suitable. Hence, a binding with another library, such as, for example,

```
Decl2,16 (Identifier1 "x")
         (PlusExp14 (DivExp3 (IntExp5 1)
                             (PlusExp12 (PowerExp7 (IntExp9 3) (IntExp11 2))
                                        (IntExp13 5)))
                    (IntExp15 1))
```

Figure 11: A parsed declaration.

Parsec [Leijen and Meijer 2001], is also possible. The only thing that needs to be done in order to use a different library is to define a primitive parser `pStructural` and a combinator that applies a top-level parser and returns an error value in case of a parse error.

6.3 Example

As an example for the parser, we show the result of parsing the tokens in Figure 3 that were obtained from scanning `x = ` $\frac{1}{3^2+5}$ `+ 1;`

Figure 11 contains the resulting document structure. The value is in fact the example from Section 3 with the presentation identities shown as subscripts. The presentation identities correspond to the identities of the tokens in Figure 3. The `Decl` node has two presentation identities: 2 for the equals sign and 16 for the semicolon. All other nodes have only one.

6.4 Parse errors

On a parse error, a parser does not return a document tree, which means there will not be a document to present. To account for this, the parser returns the special value for which a constructor was added to each type in the document. For a document type *Type*, this constructor reads:

```
data Type =
          ...
          | ParseErr_Type IDP [ErrorMessage] [Token node userToken]
```

When a parse error is encountered, the parser constructs a `ParseErr` value and supplies it with a list of error messages and the list of tokens it tried to parse. The presentation identity comes from the parsed `SequentialTk` value and is used to handle white space in absence of tokens. The presenter uses the list of tokens when the parse error node is to be presented. In addition, squiggly lines are placed at the presentation of each token that is referred to by the error-message list. The white space for the tokens in the parse error node was already handled by scanner, and is restored by the layout component in the same way as it is for tokens stemming from ordinary presentations.

Each type of node in the document has a (generated) synthesized attribute `parseErrors`, which is the collection of all parse errors in the nonterminal or its descendents. This attribute can be used to show a list of parse errors in the presentation.

Lexical errors require a special treatment. When Alex encounters a lexical error, it stops at the offending character. The offending character and the remainder of the input are put in an `ErrorTk` token, which will always cause a parse error since no primitive parser is offered that accepts it. Since scanning the string stops at the offending character, all following white space will be recorded in the string, rather than stored in the white-space map. Therefore, the layout component treats error tokens specially by expanding any white space encoded in the string.

7 Related work

We can distinguish two main classes of structure editors: *syntax-directed* editors (which derive a presentation from the document structure), and *syntax-recognizing* editors (which derive the structure from the presentation). Syntax directed editors, such as the Synthesizer Generator [Reps and Teitelbaum 1984], LRC [Saraiva et al. 2000], SbyS [Magnusson et al. 1990], and the Redwood environment [Westphal et al. 2004] allow graphical presentations, but do not have a parser for editing mixed presentations. On the other hand, syntax recognizing editors, such as Pan [Ballance et al. 1992] and Harmonia [Boshernitsan 2001] support presentation-oriented editing, but do not allow mixed graphical presentations.

Examples of systems that do support editing (and parsing) of mixed presentations are editors for mathematical formulas, such as MathSPad [Verhoeven 2000], Amaya [World Wide Web Consortium 2008], and the commercial system Mathematica. However, these systems work for a fixed document type, and are not easily extensible. Somewhat more general, Eisenberg and Kiczales describe a presentation extension formalism [Eisenberg and Kiczales 2007] for Eclipse, but their solution is targeted at extending Java only.

The Barista framework [Ko and Myers 2006] has similarities to Proxima, although it is targeted mainly at code editors. The system allows graphical presentations for program structures, but these are expanded to the underlying textual presentation when edited. Although this is good to have as optional behavior, it is in many cases unnecessary.

8 Conclusion

Editors for programming languages can benefit much from editable graphical presentations of programs. Such graphical presentations, however, cannot be parsed

with conventional parsing techniques. In this paper, we introduced a method for scanning and parsing mixed textual and graphical presentations. A combinator parser is used for textual parts of the presentations, whereas the graphical parts are recognized automatically, based on information in the presentation itself. Performance is adequate for regular-sized documents, whereas for larger documents (i.e. with a presentation of multiple thousands of lines), several hooks exist for optimizing the algorithms and adding incremental behavior. The scanner and parser are part of the Proxima generic editor, and have been used to implement a number of prototype editors.

References

[Ballance et al. 1992] Ballance, R. A., Graham, S. L., and Van De Vanter, M. L. (1992). The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):95–127.

[Boshernitsan 2001] Boshernitsan, M. (2001). Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report CSD-01-1149, University of California, Berkeley.

[Eisenberg and Kiczales 2007] Eisenberg, A. D. and Kiczales, G. (2007). Expressive programs through presentation extension. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 73–84, New York, NY, USA. ACM.

[Heeren et al. 2003] Heeren, B., Leijen, D., and van IJzendoorn, A. (2003). Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York. ACM Press.

[Hughes and Swierstra 2003] Hughes, R. J. M. and Swierstra, S. D. (2003). Polish parsers, step by step. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 239–248, New York, NY, USA. ACM.

[Ko and Myers 2006] Ko, A. J. and Myers, B. A. (2006). Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 387–396, New York, NY, USA. ACM.

[Leijen and Meijer 2001] Leijen, D. and Meijer, E. (2001). Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht.

[Magnusson et al. 1990] Magnusson, B., Bengtsson, M., Dahlin, L., Fries, G., Gustavsson, A., Hedin, G., Minor, S., Oscarsson, D., and Taube, M. (1990). An overview of the Mjølner/ORM environment: Incremental language and software development. In *TOOLS'90, Paris, France*, pages 635–646.

[Marlow 2007] Marlow, S. (2007). *Alex: A lexical analyser generator for Haskell.* `http://www.haskell.org/alex/`.

[Reps and Teitelbaum 1984] Reps, T. and Teitelbaum, T. (1984). The Synthesizer Generator. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 42–48. ACM Press.

[Saraiva et al. 2000] Saraiva, J., Swierstra, S. D., and Kuiper, M. (2000). Functional Incremental Attribute Evaluation. In David Watt, editor, *9th International Conference on Compiler Construction, CC/ETAPS2000*, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag.

[Schrage 2004] Schrage, M. M. (2004). *Proxima – a presentation-oriented editor for structured documents.* PhD thesis, Utrecht University, The Netherlands.

[Swierstra 2008] Swierstra, S. D. (2008). *Parser combinators*. Utrecht University, The Netherlands, `http://www.cs.uu.nl/wiki/HUT/ParserCombinators`.

[Swierstra et al. 2008] Swierstra, S. D., Middelkoop, A., and Baars, A. (2008). *Attribute Grammar System*. Utrecht University, The Netherlands, `http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem`.

[Verhoeven 2000] Verhoeven, R. (2000). *The Design of the MathSpad Editor*. PhD thesis, Eindhoven University, The Netherlands.

[Westphal et al. 2004] Westphal, B. T., Frederick C. Harris, J., and Dascalu, S. M. (2004). Snippets: Support for drag-and-drop programming in the redwood environment. *Journal of Universal Computer Science*, 10(7):859–871.

[World Wide Web Consortium 2008] World Wide Web Consortium (2008). Amaya web editor/browser. `http://www.w3.org/Amaya`.