A Parser Generator System for Level-based Programming Languages [1]

Michael Hanus[2] and Jan Rasmus Tikovsky[3]

Abstract: A programming language is usually taught by starting with a small kernel that is continuously extended to the full set of language features. Unfortunately, the existence of advanced language features might confuse students if they accidentally use them and get incomprehensible error messages. To avoid these problems, one should group the language features into different levels so that beginners start with a simple level and advance to higher levels with more features if they become more experienced. In order to support such a concept for arbitrary programming languages, we present a parser generator system, called Levels, for level-based programming languages. With a level-based language, one can stepwise increase the language level in order to match the experience of the students. Furthermore, one can implement level-specific semantic analyses in order to provide comprehensible error messages. Our Levels system generates level-specific parsers from a unified syntax description and provides an infrastructure to implement level-specific semantic analyses as well as program editors to develop level-specific programs.

1 Introduction

When teaching a programming language to programming novices, one usually begins with rather small programs introducing the syntax as well as simple constructs of the language. For instance, a Java beginners' course might start off with the introduction of simple expressions and statements, proceed with the explanation of abstractions like the definition of methods and classes and conclude with advanced concepts, like inheritance or generics.

Hence, one can split up the teaching and learning contents of a programming course into levels of growing complexity building on one another. It would simplify the teaching and learning of programming languages considerably, if we could somehow represent these levels of programming experience in the implementation of a language environment.

With a level-based programming language, we could hide advanced language components on lower levels and provide students with specific warnings or compiler errors if they accidentally use them. By performing level-specific semantic analyses, one could provide students with novice-friendlier warnings and error messages on beginner levels. Furthermore, one could also consider the definition of so-called white and black lists to permit and prohibit the usage of certain language features, respectively. As the course proceeds, more complex language features can be made available by raising the language level.

[1] Copyright © 2016 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.
[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany, mh@informatik.uni-kiel.de
[3] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany, jrt@informatik.uni-kiel.de

This approach to teaching programming has been pioneered for the language Scheme with the DrScheme environment [FCF$^+$02] and the corresponding textbook [FFFK01]. In order to support the development of arbitrary level-based programming languages, we need a formal description of the syntax components that are available on each level. From such a description, one could generate a corresponding parser for each level. Moreover, to facilitate the implementation of semantic analyses performed on an abstract syntax tree (AST), it would be desirable that each parser produces the same type of AST. Furthermore, the implementation of a level-based language should be distributed in form of an editor plugin providing students with an integrated development environment (IDE) to write level-based programs.

Unfortunately, existing parser generator tools like ANTLR[4] [Par07] or Happy[5], a parser generator system for Haskell, do not have features to directly model the syntax of a level-based programming language. Such parser generators produce one parser which, possibly, generates a corresponding AST from a given program. In order to define a language with levels—each with a corresponding parser—using one of these tools, we would have to specify a grammar and generate a parser for each level. Such an approach is not only cumbersome but also results in the generation of a separate AST for each level. This makes the implementation of semantic analyses for level-based languages more elaborate.

As mentioned above, there already exist approaches to use level-based programming languages to simplify the learning of a language. For instance, in the context of functional programming, there is the DrRacket system (formerly called DrScheme [FCF$^+$02]) for the Scheme dialect Racket. DrRacket is a programming IDE with levels partitioning the features of Racket into several logical units. Advanced language concepts, like operations with side effects, are not unlocked until one selects a higher language level. DrJava [ACS02] is a similar system for the object-oriented language Java. It is a lightweight IDE enabling users to interactively evaluate Java code. Simple expressions can be directly interpreted and evaluated without the definition of classes or a main method.

In this work we present a parser generator system, called Levels, for level-based programming languages. Levels is provided as a plugin for the Eclipse IDE[6]. The EBNF-like syntax description language of our system supports level annotations enabling language developers to restrict language features to certain levels. In contrast to existing parser generators, our system produces one parser for each level and a common AST structure for all levels (represented as a Haskell data type). Furthermore, the Levels system supports the combination of Levels-generated components with existing compilers and interpreters for a given programming language in a seamless way. This combination enables language developers to add levels to existing programming languages and implement (level-)specific semantic analyses for them. Finally, we show that IDEs for level-based languages developed with our system can be implemented in the form of editor plugins.

[4] `http://www.antlr.org/`
[5] `http://www.haskell.org/happy/`
[6] `http://www.eclipse.org/`

The rest of the paper is structured as follows. In Sect. 2 we present the syntax description language of our parser generator system and use it to develop a level-based language for a simple desk calculator. Section 3 shows further annotations provided by Levels in order to adapt the structure of the generated AST to specific demands. The usage of the parser generator and the generated components are described in Sect. 4. Section 5 provides a brief overview of the implementation of our tool. The application of the parser generator system to develop a level-based version of Ruby is presented in Sect. 6 before we conclude in Sect. 7.

2 Syntax Description of Level-Based Programming Languages

The syntax of a language is usually specified by regular expressions for the lexical units and context-free grammars for the language structure. The (extended) Backus-Naur Form (EBNF [Wir77]) has become the main notation to specify grammar rules in an easily comprehensible way.

In this section we present an EBNF-style grammar extended with level annotations serving as the input language for our parser generator system. Following up, we will show the syntactic components of Levels by developing the language for a simple desk calculator.

2.1 Lexer Rules

A Levels grammar is divided into three sections:

1. Token classes
2. Lexer rules
3. Parser rules

The token class section of Levels is optional and enables the declaration of names for token classes. These names can be used by tokens defined in the lexer section. The primary purpose of a token class is to categorize tokens. For instance, the following Levels declaration defines token classes for keywords and comments in order to denote tokens accordingly, enabling a common syntax highlighting for all tokens of a class:

```
TokenClasses {
  Keyword, Comment;
}
```

The lexer section of Levels allows the declaration of tokens by specifying lexer rules of the following form:

LexerRule	::=	*TokName* '=' *Regex* [*Flags*] [':: ' *Type*] [*Str2Type*] [*Type2Str*] [*TokClass*] ';'
Flags	::=	`Macro` | `NoAction`
Type	::=	`String` | `Int` | `Bool` | `Float` | ...
Str2Type	::=	`Read` | '»' *HaskellLambdaExp* '»'
Type2Str	::=	`Show` | '«' *HaskellLambdaExp* '«'
TokClass	::=	'[' *DeclaredTokenClass* ']'

For each token we must at least specify a name and a regular expression[7] determining the form of accepted strings. The token name (*TokName*) must start with a lowercase letter and each lexer rule is terminated with a semicolon. Besides the name and the regular expression, a lexer rule has a number of optional arguments. For instance, there are two flags which can be specified for a lexer rule: the `Macro` and the `NoAction` flag. With the former, a lexer rule is identified as a macro which can only be used in the definition of other lexer rules and is not considered during the actual lexing process. The latter denotes strings which shall be skipped by the lexer, e.g., comments. Each token defined in the lexer section can optionally be associated with a previously declared token class. Listing 3 shows a small example of a Levels' lexer section introducing tokens for numbers, variables beginning with a lowercase letter or an underscore, and Haskell-like inline comments.

For reasons explained in Sect. 3, we want to generate an abstract syntax tree (AST) represented by data structures implemented in the functional programming language Haskell [PJ03]. Therefore, we allow the optional specification of Haskell types and type conversion functions for each lexer rule. In the default case, each lexed value is represented as a Haskell `String`. However, if we want to define a token for integer numbers and represent the corresponding lexed values in the generated AST using Haskell's `Int` type, we have to add this type to the rule (see Listing 3 for an example). Moreover, in order to convert between the string representation read by the lexer and the representation of the specified type and vice versa, we have to declare proper Haskell conversion functions (*Str2Type* and *Type2Str*). Since many Haskell types already provide such conversion functions by implementing the type classes `Read` and `Show`, respectively, Levels is able to directly reuse these implementations if the keywords `Read` and `Show` are stated. Apart from this standard behavior, a developer can also use conversion functions of his own by specifying appropriate functions as Haskell lambda expressions encapsulated in French quotes.

2.2 Parser Rules

The syntax of parser rules in a Levels grammar is similar to that of EBNF grammar rules. Each rule consists of a name and a definition and must be terminated with a semicolon. The name of a rule starts with an uppercase letter. Parser rule definitions can be constructed using a sequence of simple elements, like terminal symbols and non-terminal symbols (which refer to other parser rules). Terminal symbols can be specified directly by encapsulating

[7] We do not specify the detailed structure of the non-terminal *Regex* since it follows the usual syntax of regular expressions.

the corresponding symbol in single or double quotes or indirectly by referencing a token declared in the lexer section of the grammar. In the former case, an appropriate regular expression for the terminal symbol is automatically created during the generation of the lexer.

Similar to EBNF grammars, Levels' grammar elements can be grouped by encapsulating them in parentheses. Moreover, elements can be denoted as optional and repetitive using square brackets and curly braces, respectively. Finally, alternative definitions can be specified by separating them with vertical bars.

Listing 1 shows most of these features of Levels by modeling the grammar for statements of a very simple imperative language. Programs written in this language consist of variable declarations, assignments, and `if` statements with optional `else` branches.

```
Parser {
  Stms = Stm ';' {Stm ';'};

  Stm  = 'declare' var
       | var '=' Expr
       | 'if' Expr 'then' Stms ['else' Stms] 'end';

  Expr = number | ...
}
```

List. 1: An example of a Levels parser section

In addition to the well-known EBNF-like grammar components presented above, Levels supports the declaration of *level annotations*. For this purpose, all grammar elements, (non-)terminals as well as complex grammar expressions, can be annotated with information denoting their (non-)availability on certain levels. Grammar elements without level annotations are available on all levels of the language. A level annotation is introduced by the symbol `@` followed by a level declaration in parentheses. Level declarations are lists of level specifications with at least one level specification. Each level specification consists of an optional flag and a level range. Valid ranges are positive integer numbers including 0 and intervals of numbers, where the upper boundary is optional. The following grammar summarizes the syntax of level annotations:

LvlAnnotation	::=	'`@`' '`(`' *LvlDeclaration* '`)`'
LvlDeclaration	::=	*LvlSpecification* {'`,`' *LvlSpecification*}
LvlSpecification	::=	[*LvlFlag*] *LvlRange*
LvlFlag	::=	'`?`' | '`!`'
LvlRange	::=	*Number* ['`-`' [*Number*]]

If n is the maximal level annotation specified in a Levels grammar, our system generates $n+1$ parsers, i.e., one parser for each level starting from the base level 0. Hence, if level n is the only level annotation used in a grammar, our system produces $n-1$ identical parsers and one parser which additionally takes the level-annotated grammar element(s) into account.

As an alternative approach, one could generate parsers only for levels explicitly specified in the grammar, thus, allowing language developers to provide for intermediate levels in case a grammar needs to be modified or extended by additional levels at a later time.

Listing 2 shows a level-annotated version of our simple statement language. According to the level annotations, compound statements are only available from level 2 on and `else` branches of `if` statements can only be used on level 3. For this level-annotated grammar, our tool would generate four parsers with identical parsers for the levels 0 and 1.

```
Parser {
  Stms = Stm ';' {Stm ';'}@(2-);

  Stm  = 'declare' var
       | var '=' Expr
       | 'if' Expr 'then' Stms ['else' Stms]@(3) 'end'

  Expr = number | ...
}
```

List. 2: Parser section with level annotations

The level flags can be used to switch between optional and mandatory elements on certain levels. For instance, the flag “`!`” denotes an optional grammar element as mandatory on the specified levels, and the flag “`?`” transforms a mandatory element into an optional one on the specified levels. For instance, consider the following excerpt of a Levels grammar:

```
Parser {
  ...
  Method = ('public' | 'private' | 'protected')@(?0,1-) MethodId Stms;
  Stms   = Stm [';']@(!0,1-) {Stm [';']@(!0,1-)};
  ...
}
```

In this level-based language, the access modifiers of methods are optional on level 0 and mandatory from level 1 onwards. Statements have to be terminated with a semicolon on level 0. Starting from level 1, ending a statement explicitly with a semicolon becomes optional. Note that the “`!`” flag can only be used in level annotations for grammar elements which were declared as optional, whereas the “`?`” flag can only be specified for non-optional grammar elements.

We conclude this section by presenting a complete Levels grammar for a simple level-based expression calculator, as shown in Listing 3. In this example, we use a further feature of Levels. Similarly to parser generators like Yacc or Happy, the parser section of a Levels grammar can also contain declarations of operator precedences and associativities in order to solve conflicts that might occur during the generation of the parsing tables. The operators can be specified directly as a string or by reference to a lexer rule defined in the lexer section. Each operator declaration states an associativity, i.e., whether the operator is left-, right-, or non-associative. The order in which operators are declared determines their precedences. If two operators share the same precedence and associativity, they can be declared in one line. For instance, consider the upper part of the Levels parser section of Listing 3 defining common arithmetic operators and their precedences for the input language of the calculator. Here, the right-associative power operator has the highest precedence. All other operators are left-associative with "*" binding more tightly than "+" and "-".

The level annotations in the calculator grammar are used to introduce expressions of increasing complexity in the various levels. On the first level, only simple arithmetic expressions like addition, subtraction, and multiplication of numbers can be handled by our calculator. From level 1 onwards, the computation of the *n*-th power is also permitted. Finally, level 2 enables the binding and usage of local variables introduced by Haskell-like `let` expressions to the level-based language of the calculator.

```
TokenClasses {
  Keyword, Comment;
}

Lexer {
  digit   = /[0-9]/ Macro;
  number  = /@digit+/ :: Int Read Show;
  var     = /[a-z_]@digit*/ :: String;
  comment = /\-\-.*/ NoAction [Comment];
  let     = /"let"/ :: String [Keyword];
  in      = /"in"/ :: String [Keyword];
}

Parser {
  %left '+' '-';
  %left '*';
  %right '**';

  TopExpr = (let var '=' Expr in Expr)@(2-) | Expr;

  Expr = number | Expr '+' Expr | Expr '-' Expr | Expr '*' Expr
       | (Expr '**' Expr)@(1-)
       | '(' TopExpr ')'
       | var@(2-);
}
```

List. 3: Levels grammar for a level-based calculator

3 Abstract Syntax Tree Generation

In this section we will discuss the transformation of a grammar into a corresponding type describing an abstract syntax tree (AST) to represent programs, as supported by the Levels system. First, we will take a look at the generation of an AST type for simple grammars without level annotations. Then we will show further annotations provided by Levels in order to specify more descriptive names for components of the AST structure and, thus, improving its comprehensibility considerably. Finally, we will explain how level-annotated grammars can be transformed to a single AST type, which provides the means to include the necessary information for each level.

An important objective of our approach is an easy integration of user-defined semantic analyses into the compiler components automatically generated by our tool. Typically, a semantic analysis traverses an AST, collects semantic information and annotates the AST with this information. Hence, the implementation of a semantic analysis involves the traversal and transformation of structured data terms. Since these kinds of tasks can be easily implemented with functional programming languages by exploiting algebraic data types, pattern matching, and function composition [Hug89], we chose the functional language Haskell to represent ASTs and implement semantic analyses. In principle, we could also generate AST representations in other languages by modifying the semantic actions of the parsers generated by the Levels system.

In the following, we assume familiarity with the basic features of Haskell [PJ03]. In order to demonstrate the representation of grammar elements and level annotations as Haskell data types, we will look at excerpts of Levels grammars and show the corresponding AST structures generated by our tool. We start by considering the Levels grammar for the simple imperative language already shown in Listing 1. Our system generates the following AST structure for this grammar:

```
data Stms = Stms Pos Stm [Repetition]

data Stm = Stm   Pos (Pos, String)
         | Stm_0 Pos (Pos, String)  Expr
         | Stm_1 Pos  Expr  Stms (Maybe Option)

data Expr = Expr Pos (Pos, Int)
          | ...

data Repetition = Repetition Pos Stm

data Option = Option Pos Stms
```

List. 4: AST structure generated from Listing 1

Thus, non-terminal symbols, like `Stms`, `Stm` and `Expr`, are mapped to Haskell data types with the same name. For each alternative definition of a non-terminal, a constructor is added to the corresponding type declaration, e.g., `Stm`, `Stm_0`, or `Stm_1`.

Terminal symbols that are explicitly declared in the lexer section of the grammar are represented by the Haskell type which was specified in their declaration. If a specific Haskell type was omitted, a terminal symbol is per default represented as a `String`. Terminal symbols without an explicit declaration in the lexer section are not part of the abstract syntax and, thus, our tool ignores them when generating the AST type. In the calculator example (see Listing 3), variables (`var`) and numbers (`number`) were explicitly declared as terminal symbols. Hence, they are represented in the AST structure by the `String` and `Int` type, respectively. However, the equal sign, the semicolon, and keywords like "`declare`" or "`if`" are not important for the abstract representation of a program and, thus, they are omitted in the AST type.

The AST structure also contains position information collected during the lexical analysis of the program. Position information is represented by the type

```
type Pos = (Int, Int, Int)
```

The first component of this triple represents the absolute position, the second and third refer to the line and the column number in the considered input string, respectively.

Hence, terminal symbols are represented as pairs of the position information and their Haskell type, and each AST constructor contains the information about its starting position as the first argument. The position information is useful to generate meaningful warnings and error messages when performing a semantic analysis on the AST.

Compound grammar elements, like groups, repetitions and optionals, are represented in the AST structure by introducing a new data type for each compound element, like `Repetition` or `Option` in Listing 4 (if there are several of these elements, the data type names are suffixed with some index in order to avoid name conflicts). These types are then used at the position of the original compound elements in the AST type, where repetitions are mapped to lists, and optionals are mapped to Haskell's `Maybe` type. For instance, our simple imperative language allows the usage of an arbitrary number of compound statements. This repetition is represented as the list argument `[Repetition]` of the AST constructor `Stms`. The elements of the repetition are defined by the type `Repetition` which basically corresponds to the type of `Stm`. Similarly, the optional `else` branch in an `if` statement is represented by the type `(Maybe Option)`.

In principle, we could already use the AST structure in Listing 4 to implement a semantic analysis for our imperative language. However, in its current form the generated AST type does not really meet the quality of a manually written functional program. For instance, it is hard to identify the different kinds of statements provided by our example language in the AST with automatically generated names like `Stm_0`. Of course, we could simply perform some renamings and refactorings in the programs generated by our tool. However, this simple approach would be rather cumbersome and also error-prone, since we would have to perform these renamings and refactorings again whenever we modify the original Levels grammar.

Fortunately, Levels supports the definition of user-defined names for the components of the generated AST structure by adding further annotations to the parser rules of a grammar. For instance, we can specify descriptive names for alternative definitions of non-terminals by simply adding an appropriate name followed by “:” in front of the respective alternative. Furthermore, we can also define names for non-terminals and complex grammar elements occurring on the right-hand side of parser rules by denoting these names encapsulated in angle brackets directly after the particular grammar element.

```
Parser {
  Stms = Stm ';' {Stm ';'}<CompStmItem>;

  Stm  = DeclareStm: 'declare' var
       | AssignStm:  var '=' Expr
       | IfStm:      'if' Expr<Cond> 'then' Stms
                                ['else' Stms]<ElseBranch> 'end'

  Expr = NumExpr: number;
       | ...
}
```

List. 5: Levels Parser rules with naming annotations

Listing 5 shows such naming annotations, where these annotations are used to improve the readability of the generated AST code. For instance, the alternatives of the non-terminals `Stm` and `Expr` are annotated with descriptive names like `DeclareStm`, `AssignStm`, `IfStm`, and `NumExpr`. Moreover, compound statement items, the conditional expression, and the optional `else` branch of the `if` statement have also specific names.

```
data Stms = Stms Pos Stm [CompStmItem]

data Stm = DeclareStm Pos (Pos, String)
         | AssignStm  Pos (Pos, String)  Expr
         | IfStm      Pos  Cond  Stms (Maybe ElseBranch)

data Expr = NumExpr Pos (Pos, Int)
          | ...

data CompStmItem = CompStmItem Pos Stm

data ElseBranch  = ElseBranch  Pos Stms

type Cond = Expr
```

List. 6: AST structure with descriptive names, generated from Listing 5

The types generated to describe a corresponding AST for this annotated Levels grammar is shown in Listing 6. Now the constructors for alternative definitions of non-terminals have descriptive names like `AssignStm` instead of `Stm_0`. Defining names of non-terminals occurring on the right-hand side of a parser rule are represented by the introduction of type synonyms, like the type `Cond` for the conditional expression in an `if` statement of our example. The naming annotation of complex grammar elements, like repetitions and optionals, yields more descriptive names for the AST data types. For instance, by giving the optional `else` branch of the `if` statement a defining name, the `Option` type shown in Listing 4 is changed to `ElseBranch`. Note that arbitrary sequences of grammar elements occurring on the right-hand side of a parser rule can be annotated with a name by grouping them before adding an appropriate annotation.

If we do not specify a defining name in a naming annotation, but just annotate a grammar element with an opening and closing angle bracket, this grammar element is not represented in the generated AST structure. Applying naming annotations in that way can be useful to enforce the omission of certain grammar elements in the AST representation. For instance, if we want the keywords of our language to be equally highlighted in a text editor, we have to introduce a token class `keyword` and explicitly declare a lexer rule for each keyword referencing this token class as explained in Sect. 2. However, applying these explicitly defined tokens in the specification of parser rules results in their explicit representation in the AST. Since keywords are usually not part of the abstract representation of a program, we can enforce their omission in the AST representation by annotating them with <>.

In the remaining part of this section, we discuss the generation of AST types for grammars containing level annotations. As already mentioned, our system generates several parsers but a single AST structure for an input grammar with level annotations. The generation of only one AST type is motivated by the fact to reuse common parts of the semantic analyses at different levels. Since level annotations allow to hide the usage of language components on particular levels, we have to ensure that the generated AST is valid even if these components are hidden. Therefore, the corresponding parts of the AST have to be optional, which can be expressed in Haskell by the type constructor `Maybe`. Thus, our system transforms level-annotated grammar elements into `Maybe` types.

For instance, consider again our running example. Suppose we want to modify the grammar of Listing 5 in two ways: First, we want `if` statements to be only available from level 1 onwards (remember that the numbering of levels starts at 0). Second, we want to include type information in our language on higher levels. Hence, on level 2 the specification of a primitive type is required when declaring a variable. For the implementation of these extensions we modify the grammar as follows:

```
...
Stm  = DeclareStm: 'declare' var (':' Type)<TypeDecl>@(2-)
     | AssignStm:  var '=' Expr
     | IfStm: ('if' Expr<Cond> 'then' Stms
                              ['else' Stms]<ElseBranch> 'end')@(1-)

Type = Int: 'int' | Float: 'float' | Bool: 'bool'
...
```

In order to add a level annotation for the complete `if` statement, we group all its grammar elements using parentheses. Furthermore, we define a name for the newly introduced group to maintain the readability of the generated AST. In a similar way, we specify that an annotation of a primitive type, like `int`, is required on level 2 when declaring a variable.

Observe that a naming annotation always has to be specified before a level annotation when both of them are used for the same grammar element. The AST generated from this modified grammar is shown in Listing 7.

```
data Stms = Stms Pos Stm [CompStmItem]

data Stm = DeclareStm Pos (Pos, String) (Maybe TypeDecl)
         | AssignStm  Pos (Pos, String) Expr
         | IfStm      Pos Cond Stms (Maybe ElseBranch)

data TypeDecl = TypeDecl Pos Type

data Type = Int Pos | Float Pos | Bool Pos

data Expr = NumExpr Pos (Pos, Int)
          | ...

data CompStmItem = CompStmItem Pos Stm

data ElseBranch = ElseBranch Pos Stms

type Cond = Expr
```

List. 7: AST structure generated from a level-annotated grammar

If we compare Listing 6 and Listing 7, we see that the additional types `TypeDecl` and `Type` were generated to represent the newly introduced level-restricted group for optional type annotations. Moreover, with the generated types describing the AST we can represent programs including `if` statements and variable declarations with type annotations as well as programs in which those statements are not available. The fact that programs include additional type annotations in variable declarations on level 2 is represented in the AST structure by the type `(Maybe TypeDecl)` in the constructor `DeclareStm`. However, if an entire grammar rule alternative, like the `if` statement in our running example, is only available at specific levels, there is no need to use a `Maybe` type in the AST representation, because, depending on the chosen parser level, an AST will either include only complete `if` statements or no `if` statements at all.

Note that all level-annotated grammar elements are represented as optional using `Maybe` in the generated AST types. In fact, it does not make a difference, if we specify flags like `?` or `!` in the level annotation or if we annotate simple grammar elements like non-terminals or complex grouped elements. In all of these cases, a `Maybe` type will be used in the AST representation. However, there are two exceptions to this general rule. As mentioned before, if an entire grammar rule alternative is level-annotated, no optional type is required in the AST representation, because the corresponding syntactic alternative will be either completely included in a generated AST or not included at all. The second exception are terminal symbols which were not explicitly declared in Levels' lexer section. Since these terminal symbols are not part of the abstract syntax, they are not represented in the AST, regardless of the level annotations specified.

As mentioned above, the generation of a single AST structure for all language levels enables the reuse of semantic analyses on different language levels. For instance, consider we want to check that every identifier used in a statement was declared in some preceding statement. Listing 8 shows an excerpt of the implementation of such an analysis for our running example using the generated AST structure depicted in Listing 7.

```
type VarId    = String
type SymTable = [(VarId, Pos, Maybe TypeDecl)]

declared :: Stms → SymTable
declared (Stms _ stm csi) = foldr addDecl [] (stm : stms)
  where stms = map (\(CompStmItem _ s) → s) csi

addDecl :: Stm → SymTable → SymTable
addDecl (DeclareStm _ (p, i) mty) sym = (i, p, mty) : sym
  ...

-- performed for all programs
usedDeclCheck :: SymTable → Stms → Either LevelsError ()

-- only performed for level 2 programs
typeCheck :: SymTable → Stms → Either LevelsError ()
```

List. 8: Implementation of a used-declared analysis (excerpt)

In order to check that every identifier used in a program was properly declared, we traverse the AST structure and create a symbol table. Whenever we come across a declaration statement during this traversal, we add information like the position or the optional type of the corresponding variable to the table. Then we can implement the used-declared analysis using the information collected in the symbol table. Since our tool generates only a single AST type for all levels of a language, we can perform this analysis for all programs regardless of their level. Furthermore, we can also specify level-specific analyses. For example, we could implement a type checker for our running example using the type information collected in the symbol table for each identifier. As type declarations are only required on level 2, the type checker is an example for a level-specific semantic analysis.

4 Generated Components of a Levels Project

After having introduced the syntax of Levels grammars and the transformation of Levels grammars into corresponding abstract syntax trees, we discuss in this section the overall structure of a Levels project and the (remaining) components automatically generated from a Levels grammar by our system.

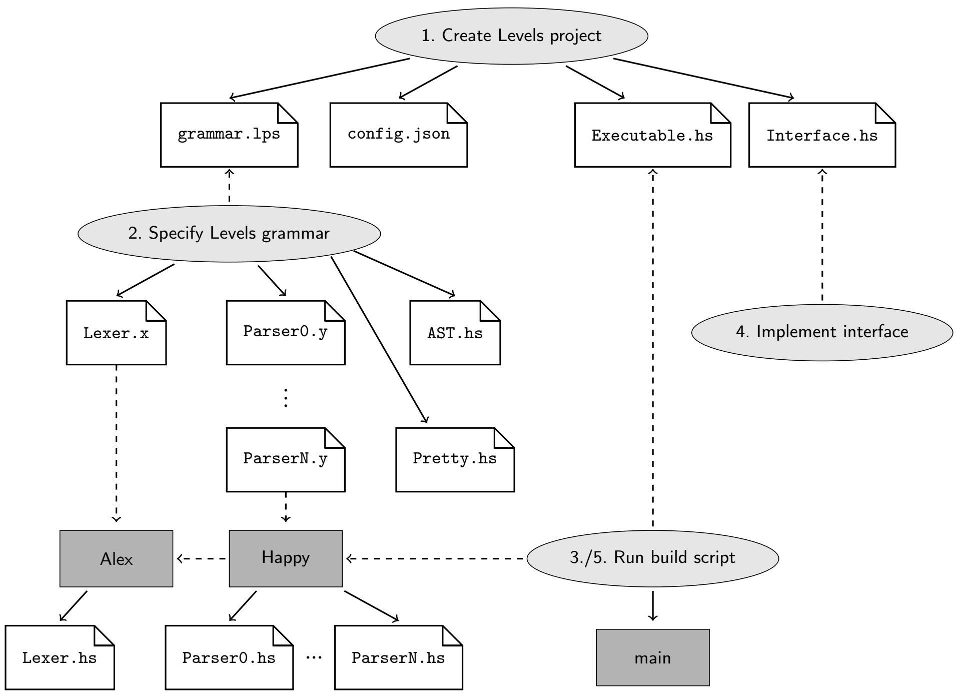


Fig. 1: Development of a level-based programming language with Levels

Basically, our system is provided as a plugin for the Eclipse IDE. Hence, if we want to begin with the development of a new level-based programming language, we run Eclipse Luna (or higher), install the plugin and select the option to generate a new Levels project. By generating a new project, an initial project framework is created in Eclipse's workspace. Among other things, this project template provides an empty Levels grammar file, a configuration file, Haskell source files for a simple command line executable and an Ant[8] build script. With a configuration file written in JSON syntax, we can configure a Levels project. The file contains

- meta information, like the name of the level-based language,
- names and descriptions for the supported levels, and
- information and commands for the execution of level-based programs.

The command line executable and similar back ends use this information to compile and run level-based programs. In Sect. 6 we will show more details about the configuration file and the functionality of the executable for a concrete application of Levels.

[8] `http://ant.apache.org/`

The next step when developing a level-based language with our tool is to define its syntax by specifying a Levels grammar. As it is general practice for most Eclipse language plugins, our tool supports auto-completion for Levels as well as immediate detection and notification of syntax errors when the lexer and parser rules are written. As soon as we have specified the Levels grammar for a language and saved it, the Levels system automatically generates the following files:

1. A Haskell module (`AST.hs`) with the data structures representing the abstract syntax tree generated from the syntax description of the level-based language, as described in Sect. 3.
2. A Haskell module (`Pretty.hs`) with a pretty printer (based on standard Haskell modules for pretty printing [Wad03]) to transform an abstract representation of a program back into its concrete syntax. This pretty printer could be used to produce a modified source program that is fed to the actual language compiler or interpreter. For instance, in a level-based language for Java, where the basic level contains only procedures and statements but not classes, one can extend the generated pretty printer in order to wrap a program of this basic level with a class definition, add `public` modifiers to each procedure, and enclose the statements in a `main` method.
3. A scanner description file: Since we use Haskell for the representation of the AST, we decided to use scanner and parser tools available for Haskell, i.e., Alex and Happy, to implement the lexical and syntactic analysis. Hence, the Levels system produces an Alex specification (`Lexer.x`) from which a scanner can be generated. Since level annotations are only used in the parser rules of a Levels grammar, a single scanner is generated for all parser levels.
4. A parser description file: For each level *n*, starting from level 0 up to the maximum level specified in the given Levels grammar, a Happy parser specification (`Parser`*n*`.y`) is generated.

In the next step, we can run the Ant build script generated by Levels. By executing this script, Alex and Happy are called in order to produce Haskell source files for the lexer and the parsers—one for each level—from the previously generated `.x` and `.y` files.

Before we can put everything together in order to build an executable which can analyze and run level-based programs, we have to implement an `Interface.hs` module which is used in the implementation of the executable. The Levels system already provides a template version of this module when a new Levels project is started. Basically, one has to define the function

```
compileLevelCode :: Int → FilePath → IO FilePath
```

in this module. This function takes a level and a level-based program, parses it, possibly performs semantic analyses and yields a corresponding object code file of the compiled level-based program. We can implement this function by applying the operations of the previously generated modules together with a compiler or interpreter for the actual language.

In order to support the generation of warnings and error messages during the semantic analysis, the interface provides the following data types for the representation of errors:

```
data LevelsError = LevelsError
  { errorSource   :: Source
  , errorPosition :: Maybe Pos
  , errorMessage  :: String
  }

data Source = Lexer | Parser | Analyzer | Printer | ...
```

Thus, an error includes information about the component in which the error occured, an optional position, and an error message. In order to create a new error message when implementing a semantic analysis for a Levels project, one can use the function

```
analyzerError :: Maybe Pos → String → LevelsError
analyzerError = LevelsError Analyzer
```

For the implementation of warnings, similar types and functions are provided. The actual printing of the warnings and error messages collected during the semantic analysis must be explicitly performed in the implementation of the main function `compileLevelCode` introduced above.

The final step in the development of a level-based language is to once again run the provided build script. Now, everything will be put together by building the command line executable with the Haskell package system Cabal.[9]

Figure 1 summarizes the necessary steps to develop a level-based programming language with Levels. The picture also shows most of the generated components and tools involved during the development. Arrows with a solid line point to components that are automatically generated by performing the action or using the tool which is depicted at the origin of the arrow. Dashed arrows represent the invocation of an action to a certain component or the fact that one component is used as input for a certain tool. The result of the build process is a command-line executable (denoted with `main` in the picture) which—in combination with the information specified in the configuration file—allows the execution of level-based programs.

5 Implementation

This section will give a brief overview of the implementation of Levels. As explained in the previous section, Levels is a plugin for the Eclipse IDE.

The Levels syntax description language is implemented with Xtext[10], a framework to develop textual domain-specific languages. For this purpose, we define an Xtext grammar

[9] `https://www.haskell.org/cabal/`
[10] `http://www.eclipse.org/Xtext/`

which specifies the basic syntactic structure of Levels' grammar input language, i.e., the lexer and parser sections of a Levels grammar together with level and naming annotations as described in Sections 2 and 3. By the use of the Xtext framework, an ANTLR3 lexer and parser [Par07] is provided, which transform a given Levels grammar into an EBNF object.

In order to automatically generate input files for Alex and Happy as well as Haskell code for an AST and a pretty printer for a given Levels grammar, we implement conversion tools and generators using Xtend[11], a statically typed, functional, and object-oriented programming language for the JVM. Since Happy can only handle grammars specified in BNF syntax, we first apply a converter to transform the EBNF object of a given Levels grammar into a corresponding BNF object. Complex grammar components like optionals, repetitions and groups are removed by introducing fresh non-terminals and additional rules. For the remaining grammar components, i.e., terminals, non-terminals, level annotations, and lexer rules, Xtend classes are implemented. For instance, the class `BNFLevel` for representing level annotations provides the methods `getMaxLevel` and `isAvailableAtLevel`. These methods can be used to determine the maximum level occuring in an annotation and to check whether a given level is available in an annotation, respectively.

The BNF object created for an input grammar bundles all necessary information for the creation of the parser components. It includes a hash set of lexer rule objects, a hash map representing the parser rules, i.e., mapping non-terminal objects to lists of grammar objects occurring in the right-hand sides of rules, and the maximum level specified in the input grammar.

By using the information provided by the BNF object, we implement a set of generators to create input files for Alex and Happy as well as Haskell files for the representation of the AST and the pretty printer, as described in Sect. 4. For example, the AST generator introduces Haskell data type definitions for all non-terminals occuring in the BNF object. Each constructor of such a data type definition matches an alternative of the right-hand side of a parser rule in the BNF object.

Figure 2 gives an overview of the implementation of Levels.

6 Application

In this section we will briefly outline a practical application of the Levels system to develop a level-based version of Ruby [FM08], a dynamically typed object-oriented language. At Kiel university, our team offers a programming course for students who chose computer science as minor subject. In this course we teach programming concepts with Ruby. In order to simplify the learning and enforce some programming discipline, we want to restrict the usage of Ruby for the beginners with different language levels.

The levels we introduce comply with certain stages of the lecture. For instance, there is a beginners level in which all statements have to be explicitly terminated with a semicolon.

[11] `http://www.eclipse.org/xtend/`

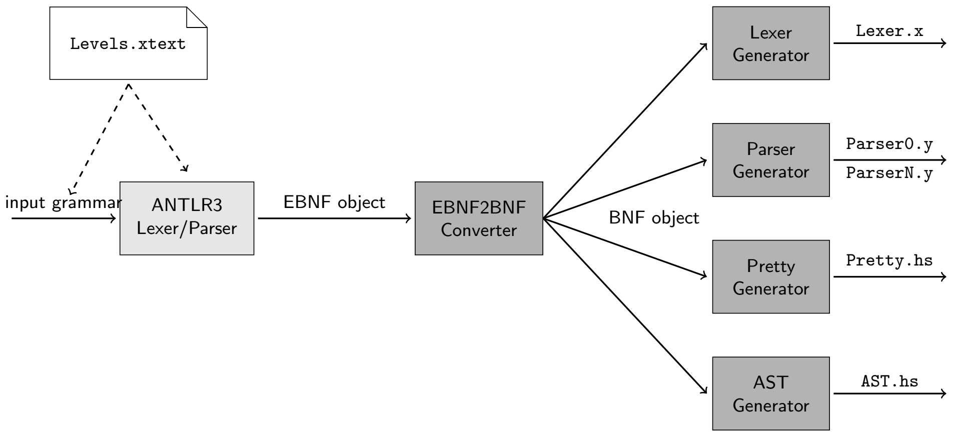


Fig. 2: Overview of the implementation of Levels

Moreover, on this level we distinguish functions and procedures regarding their usage: The former can only be used in expressions and the latter may only be applied in statements.

As the lecture proceeds, more advanced levels become available to the students. On higher levels, the termination of statements with semicolons is optional, and functions and procedures are not distinguished in their usage anymore. Additionally, concepts like block structures can only be used on advanced levels.

In order to extend Ruby with programming levels, we define a Levels grammar for Ruby with appropriate level annotations as explained in Sect. 2. Furthermore, we implement some simple semantic analyses based on the AST structure generated by our tool. For instance, there is a used-declared analysis very similar to the one presented in Section 3 which checks that every identifier used in a program is also defined. While the used-declared analysis is performed on every level, we also implement some level-specific analyses. For example, only on the beginners level, we check the correct usage of functions and procedures in expressions and statements as mentioned above and issue a warning in case of incorrect usage. Moreover, our level-based version of Ruby provides white and black lists for specific code fragments explicitly allowing and prohibiting their usage, respectively.

As mentioned in Sect. 4, for each Levels project a configuration file is provided in which meta information is specified in JSON syntax. Listing 9 shows an excerpt of the configuration file for the level-based Ruby language. Besides specifying names for the language and their levels, we can also define file extensions for the level-based language. Additionally, we can specify a command pattern for the execution of level-based programs. This information is used for the compilation and execution of level-based Ruby programs applying the command-line executable provided with each Levels project. Remember that this executable is a binary built from a Haskell program which combines all generated components, i.e., lexer, level-specific parsers, pretty printer, as well as all manually implemented components, like semantic analyses and program transformations.

```
"name": "LvlRuby",
"levels": [
  {
    "name": "Beginners Level",
    "description": "This level is for Ruby beginners."
  },
  ...
],
"levelCodeFileTypes": ["lrb","rbl"],
"objectCodeFileType": "rb",
"executionCommandPatterns": ["ruby <filePath>"]
```

List. 9: `config.json` for level-based Ruby

In case of our level-based Ruby system, the compilation and execution process performed by this tool is done in the following steps. In order to run a program with a particular level, the main executable is invoked with the chosen level, the path to the level-based program, and the path to the language configuration file. For instance, the command

```
> main 1 Example.lrb -l /path/to/config.json
```

compiles and executes the Ruby program `Example.lrb` on level 1. This initiates in the following actions:

1. The given input program is read by the scanner and parser for the corresponding level.
2. The semantic analysis is performed. This annotates the AST generated during parsing with additional information and might yield warnings or error messages.
3. The (annotated) AST is pretty printed by applying the generated pretty printer or a properly modified version in case the original AST structure has changed.
4. Finally, if no errors were detected, the resulting Ruby file is compiled and executed applying the command pattern provided in the configuration file.

By the configuration file shown in Listing 9, level-based Ruby programs are simply executed by invoking the standard Ruby compiler. Figure 3 shows this compilation process. The light grey components are automatically generated by Levels while the dark grey components[12] need to be implemented or extended by the user.

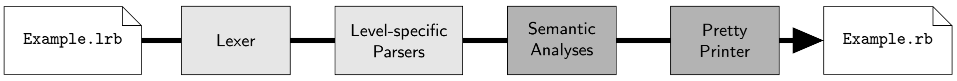
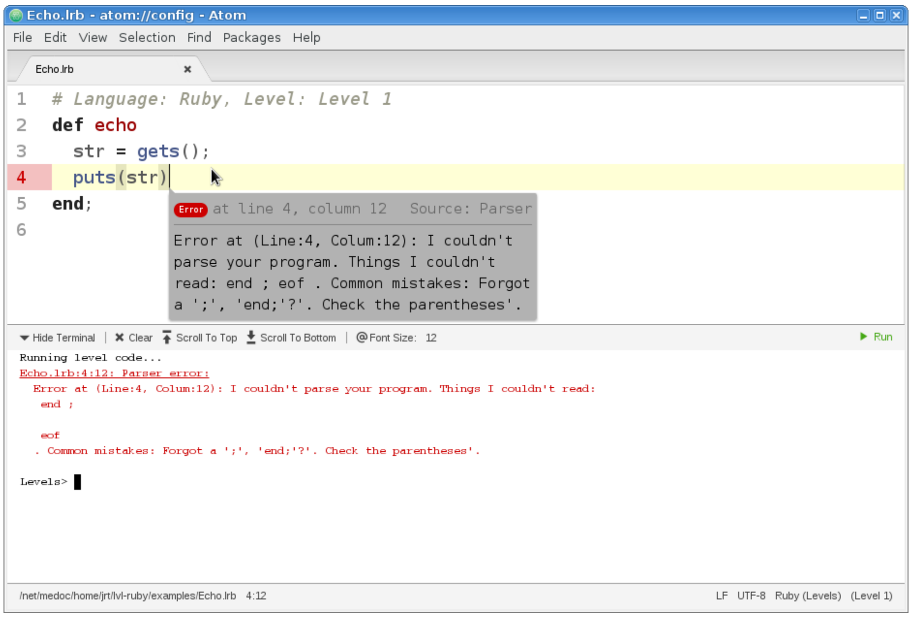


Fig. 3: Overview of the compilation process of a level-based Ruby program

[12] In fact, the pretty printer is also automatically created by Levels, but the implementation can be modified if necessary.

Furthermore, the executable and the configuration file can be used to implement additional tools, like graphical user interfaces (GUI) for the development of level-based programs. In case of our level-based Ruby system, we developed plugins for the Komodo Edit[13] and Atom[14] text editors. Using these plugins, a programmer can write a level-based program, select one of the levels provided by the language and directly execute the program within the editor. Besides level-specific syntax highlighting, the plugins also support graphical notifications of errors. A screenshot of an editor session with the Atom plugin of our level-based Ruby language is shown in Fig. 4.


Echo.lrb - atom://config - Atom
File Edit View Selection Find Packages Help
Echo.lrb
Hide Terminal
Clear
Scroll To Top
Scroll To Bottom
Font Size: 12
Run
/net/medoc/home/jrt/lvl-ruby/examples/Echo.lrb 4:12
LF UTF-8 Ruby (Levels) (Level 1)

```
1  # Language: Ruby, Level: Level 1
2  def echo
3    str = gets();
4    puts(str)
5  end;
6
```

```
Error at line 4, column 12   Source: Parser

Error at (Line:4, Colum:12): I couldn't
parse your program. Things I couldn't
read: end ; eof . Common mistakes: Forgot
a ';', 'end;'?'. Check the parentheses'.
```

```
Running level code...
Echo.lrb:4:12: Parser error:
  Error at (Line:4, Colum:12): I couldn't parse your program. Things I couldn't read:
   end ;

   eof
  . Common mistakes: Forgot a ';', 'end;'?'. Check the parentheses'.

Levels>
```

Fig. 4: Developing a level-based Ruby program using the Atom plugin

7 Conclusions and Future Work

In this work we presented Levels, a parser generator system for the development of level-based programming languages. A level-based programming language consists of various language levels where some syntactic elements can be omitted or enforced on a particular language level. In this way, we can group a language into levels of different complexity hiding advanced features and providing specific warnings and error messages on lower levels.

In order to support the implementation of level-based languages, we developed a parser generator system for such languages, called Levels. Levels provides a syntax description language with an EBNF-like syntax. Besides defining lexer and parser rules to describe

[13] `http://komodoide.com/komodo-edit/`
[14] `https://atom.io/`

the syntax of a language, a Levels grammar can also include specific level annotations for each grammar element. Moreover, the Levels system supports also the generation of appropriate abstract syntax trees from a given program. In order to reuse semantic analyses for different levels, we decided to use a single abstract syntax tree for all levels. The main task of implementing a level-based language with Levels is, beyond the grammar definition, the implementation of specific semantic analyses to provide warnings and error messages appropriate for the different levels. Since this task usually consists of traversing and transforming or annotating the AST, we chose Haskell with its support of algebraic data types and pattern matching to implement these components. While EBNF-like grammar components like repetitions and optionals can be represented in the AST structure using Haskell's list and `Maybe` types, all level-annotated grammar elements are treated like optionals. In addition to level annotations, Levels input language also supports naming annotations. With these annotations, a user can specify more descriptive names for grammar components like alternatives, groups, etc. This improves the comprehensibility of the generated types representing the AST considerably.

Besides a corresponding AST structure, Levels automatically generates a lexer, a parser for each specified level, and a pretty printer, as soon as a syntactically correct Levels grammar is defined. The actual lexer and parsers are created using the Haskell tools Alex and Happy. Furthermore, a Levels project enables the easy integration of semantic analyses implemented in Haskell with the Levels-generated components by providing an appropriate infrastructure to create a single executable.

Levels is provided as a plugin for the Eclipse IDE supporting standard features, like auto-completion and error notifications, when defining a level-based grammar. We implemented Levels' input language using Xtext. The EBNF object, which is automatically created by ANTL3 when specifying a Levels grammar, is then transformed into BNF syntax. Finally, we implemented various generators with Xtend to build the different Haskell components from the given BNF object.

As a proof of concept, Levels was applied to develop a level-based version of Ruby with level-specific semantic analyses. The language levels added to Ruby comply with certain stages of a programming lecture taught at Kiel university. Hence, they can be increased as the students become more experienced during the term. Moreover, building upon the executable provided by Levels, we developed two integrated development environments for level-based Ruby programs in the form of plugins for the Komodo Edit and Atom text editors. These editor plugins are currently successfully used in the practical exercises of the lecture "Computer Science as a Minor Subject".

For future work it might be interesting to apply Levels to the development of further level-based languages. For instance, one could develop a level-based version of Java for a beginner's course in (object-oriented) programming. By introducing levels to Java, one could hide complex features like classes and methods and start off with a small imperative kernel of the language including only expressions and simple statements. When compiling a level-based Java program, these simple statements are automatically inserted into the body of the main method of a class. Other languages like Python could be extended with levels in a similar way. Furthermore, one could provide additional integrated development environments for level-based programs by implementing plugins for other text editors or IDEs like Eclipse.

Acknowledgements

The authors are grateful to the development team of the Levels system (Thorben Droste, Malte Hecht, Julia Kunst, Lasse Kristopher Meyer, Jonas Oberschweiber, Thomas Rossow, Jasper Paul Sikorra, Mike Tallarek, and Thomas Ulrich) and Frank Huch for his suggestions that led to this development.

References

[ACS02] E.E. Allen, R. Cartwright, and B. Stoler. DrJava: a lightweight pedagogic environment for Java. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 137–141. ACM, 2002.

[FCF+02] R.B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[FFFK01] M. Felleisen, R.B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[FM08] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Par07] T. Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[Wad03] P. Wadler. A prettier printer. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 223–243. Palgrave Macmillan, 2003.

[Wir77] N. Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, 20(11):822–823, 1977.