Language Design with the Spoofax Language Workbench

Guido H. Wachsmuth, Gabriël D.P. Konat, and Eelco Visser,
Delft University of Technology

// State-of-the-art IDE support is essential for programming languages to be successful. The Spoofax language workbench is a comprehensive environment for developing languages with this support. //

LANGUAGE WORKBENCHES provide high-level mechanisms for implementing programming languages and make the development of new languages affordable. The Spoofax language workbench is a platform for developing textual programming languages. It supports exploratory language design by allowing incremental, iterative language development and by running generated editors in the same Eclipse instance in which the language is designed. Spoofax offers a comprehensive environment integrating syntax definition, name binding, type analysis, program transformation, and code generation. For each of these aspects, it provides highly declarative metalanguages that abstract over the implementation of language processors, letting language engineers focus on design.

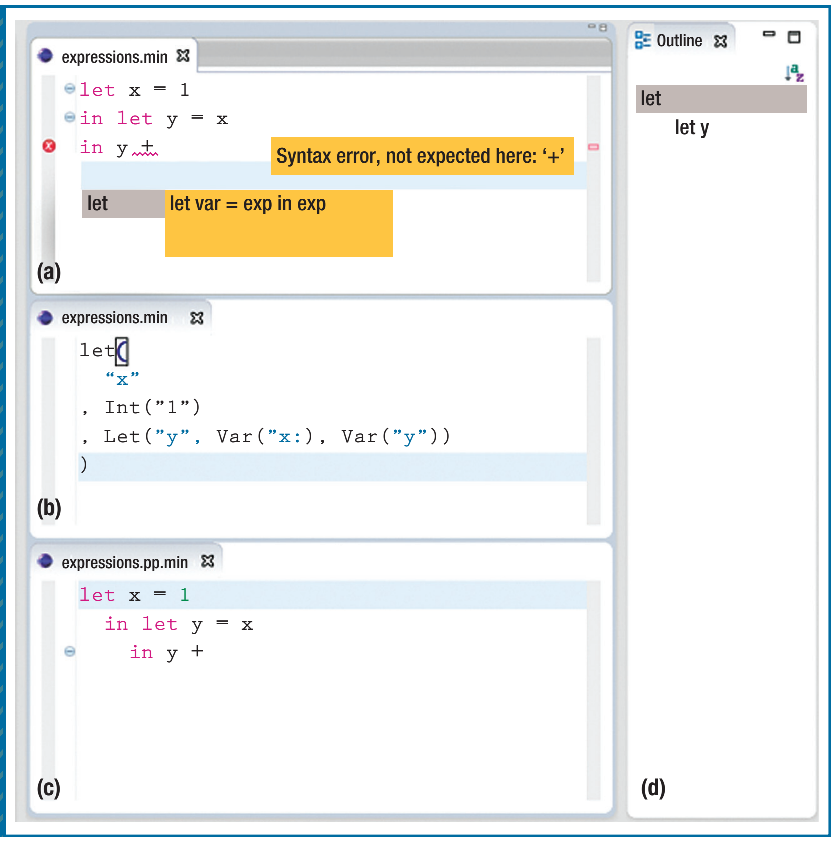
Implementing IDEs

IDEs have become fundamental to software engineering. Modern IDEs such as IntelliJ IDEA, Eclipse, and Visual Studio parse files as they're typed and perform name and type analyses. They also provide code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as they're typed, and the ability to refactor programs.

Although IDE features for mainstream programming languages are typically implemented manually, this often isn't feasible for new programming languages that must be developed with significantly fewer resources. Furthermore, manual implementation inhibits the exploratory design of new languages. Many IDE features concern the same language aspect. When a language aspect evolves, all IDE features must reflect this change and consistently co-evolve. When a language's syntax should change, implementations must be adopted for parsing, formatting, and completion templates. Designers must also ensure that the parser can handle code emitted by the formatter and completion templates. When a language's name-binding and scope rules should change, implementations must be adopted for name resolution, constraint checks, and name-based completions. This will ensure that completion proposals will be resolved to the proposed definition and won't violate constraints.

The interactive nature of IDEs gives rise to additional, typically cross-cutting concerns. An IDE must be able to provide services such as code navigation or content completion, even in erroneous states. This requires the parser to recover from parse errors. Name and type analyses must be able to work on multiple files in a project, propagating changes in one file to other affected files. Incremental analysis techniques are necessary to provide instant feedback while the program is being edited.

Spoofax automatically derives efficient implementations for various IDE features from declarative language designs. It also supports the generation of a stand-alone Eclipse



```
let x = 1
in let y = x
in y +
```

Syntax error, not expected here: '+'

let | let var = exp in exp

```
let(
  "x"
, Int("1")
, Let("y", Var("x:), Var("y"))
)
```

```
let x = 1
  in let y = x
    in y +
```

```
let
    let y
```

FIGURE 1. Syntax-based IDE features for a basic expression language. (a) Syntax highlighting, error marking, and content completion. (b) Abstract syntax representation. (c) Formatting. (d) The outline view.

plugin for a language, which can be deployed without exposing Spoofax's language design facilities.

Modular Syntax Definition in SDF3

Spoofax provides the syntax definition formalism SDF3.[1] The declarative, highly modular syntax definitions in SDF3 combine lexical and context-free syntax into one formalism and can define concrete and abstract syntax together in templates. SDF3 is closed under composition, ensuring support for language extensions and embeddings. Spoofax derives implementations from a syntax definition in SDF3 for the following IDE features:

- a parser that turns concrete syntax into an abstract representation and recovers from (multiple) syntax errors,
- a formatter that turns an abstract representation into concrete syntax,
- syntax highlighting,
- code folding,
- an outline view,
- bracket highlighting and insertion, and
- syntactic code completion.

Figure 1 illustrates some of these syntax-based IDE features for a basic expression language.

To demonstrate how Spoofax works, we'll develop relevant parts of the basic expression language's definition. Arithmetic expressions are integer constants, additions, multiplications, or parenthesized expressions. The syntax definition for arithmetic expressions in SDF3 is

```
module Arithmetics imports Common

context-free syntax
 Exp.Int   = <<INT>>
 Exp.Add   = <<Exp> + <Exp>> {left}
 Exp.Mul   = <<Exp> * <Exp>> {left}
 Exp       = <( <Exp> )>    {bracket}

context-free priorities
 Exp.Mul > Exp.Add
```

A syntax definition in SDF3 consists of zero or more template productions. Each production takes the form **s.c = <t>**, where **s** is the syntactic sort being defined, **c** is the constructor label used in the abstract representation, and **t** is a template that might include concrete syntax, placeholders for other sorts, and layout. The first production defines a template for integer constants, consisting of an <Int> placeholder. The second and third productions specify templates for addition and multiplication consisting of an <Exp> placeholder, whitespace, the operator symbol, more whitespace, and another <Exp> placeholder. The parser derived from a template will accept layout around placeholders and concrete syntax elements. The derived formatter will use the whitespace from the template.

We can extend Syntax definitions in SDF3 with annotations and disambiguation rules to express language characteristics such as associativity and operator precedence. In arithmetic expressions, addition and multiplication are left-associative (the **left** annotation), and multiplication takes precedence over addition (the Exp.Mul > Exp.Add disambiguation rule).

The last template production covers parenthesized expressions. It doesn't specify any constructor but is annotated with **bracket**. The abstract representation of a parenthesized expression is the same as the representation of the inner expression. The derived parser will consider parentheses with respect to associativity and operator precedence. The derived formatter will add parentheses where needed to preserve associativity and operator precedence.

We can define a separate syntax module for Boolean expressions:

```
module Booleans imports Common

context-free syntax
 Exp.True      = <true>
 Exp.False     = <false>
 Exp.Or        = <<Exp> | <Exp>> {left}
 Exp.And       = <<Exp> & <Exp>> {left}
 Exp           = <( <Exp> )>    {bracket}

context-free priorities
 Exp.And > Exp.Or
```

Both modules can be used independently. To support arithmetic and Boolean expressions in the same language, we can import both into another module:

```
module Expressions

imports Arithmetics Booleans

context-free priorities
 Exp.Add > Exp.And
```

We need to add another disambiguation rule to specify the precedence between operators from both modules. This rule takes the imported precedence rules into account and is equivalent to Exp.Mul > Exp.Add > Exp.And > Exp.Or.

Iterative Language Design

Rather than designing a completely new programming language before its implementation, it's good to incrementally introduce features and abstractions through iterative design. This means that programs and the programming language evolve together. Spoofax enables quick turnaround for iterative language design by running generated editors for a language in the same Eclipse instance in which the language is designed.

Additionally, the Spoofax testing language provides a reusable, generic basis for declaratively specifying language design tests.[2] This lets us systematically test language features. Figure 2 specifies test cases for parsing **let** expressions. The expression **let** x = i **in** body consists of a variable x, an initialization expression i, and an expression body, in which x will be bound to the value of i.

With the current syntax definition, all positive test cases (parse succeeds) fail. Only negative test cases (parse fails) succeed. We can start fixing the syntax definition by adding a new template production:

```
Exp.Let = <let x = 1 in 2>
```

Now, the first test case succeeds. The second test case indicates we must abstract over different variable names:

```
Exp.Let = <let <ID> = 1 in 2>
```

This causes the second test case to succeed but both negative test cases to fail. We can fix this by adding lexical syntax rules that reject reserved keywords:

```
lexical syntax
 ID = "let" {reject}
 ID = "in"  {reject}
```

Now, the negative test cases succeed again. The third test case indicates we must abstract over subexpressions in the let expression:

```
Exp.Let = <let <ID> = <Exp> in <Exp>>
```

```
module syntax/let
language Expressions

test simple let x
 [[let x = 1 in 2]] parse succeeds
test simple let y
 [[let y = 1 in 2]] parse succeeds
test let w/ subexpressions
 [[let x = 1 + 2 in 3 + 4]] parse succeeds
test let w/ variable use
 [[let x = 1 + 2 in x]] parse succeeds
test nested let
 [[let x = 1 in
   let y = x in y]] parse succeeds
test let reserved
 [[let let = 1 in 2]] parse fails
test in reserved
 [[let in = 1 in 2]] parse fails
```

FIGURE 2. Test-driven language design in Spoofax. The module defines test cases for the syntax of **let** expressions in the Spoofax testing language.

This leaves us with two failing test cases that use variable references in expressions. We can add another template production for such expressions:

```
Exp.Var = <<ID>>
```

Now, all test cases succeed. However, when we use the generated editor and apply the formatter on a nested let expression, the expression will print on a single line. We can improve the formatter by adding line breaks and indentations to the template:

```
templates
 Exp.Let = <
  let
    <ID> = <Exp>
  in
    <Exp>>
 Exp.Var = <<ID>>

lexical syntax
 ID = "let" {reject}
 ID = "in"  {reject}
```

```
module bindings/let
language Expressions

test single let
 [[let [[x]] = 1 in [[x]]]]
 resolve #2 to #1
test nested let init
 [[let [[x]] = 1 in
   let y = [[x]] in y]]
 resolve #2 to #1
test nested let body
 [[let [[x]] = 1 in
   let y = 2 in [[x]]]]
 resolve #2 to #1
test hiding let init
 [[let [[x]] = 1 in
   let x = [[x]] in x]]
 resolve #2 to #1
test warn on hiding let
 [[let x = 1 in
   let [[x]] = 2 in x]] 1 warning
```

FIGURE 3. Test-driven language design in Spoofax. The module defines test cases for reference resolution and hiding constraints in let expressions in the Spoofax testing language.

Declarative Name Binding and Scope Rules in NaBL

Spoofax provides the name-binding language NaBL (pronounced *enable*) to declaratively specify name binding and scope rules of a language.[3] From NaBL rules, Spoofax derives implementations for the following IDE features:

- multifile name analysis that incrementally uses previous analysis results,[4]
- constraint checks for duplicate and hiding definitions,
- reference resolution, and
- name-based content completion.

NaBL provides four basic concepts. *Namespaces* distinguish different kinds of names so that an occurrence of a name in one namespace isn't related to an occurrence of the same name in another namespace. *Definitions* bind names. *References* use names (name binding connects references to definitions). *Scopes* restrict definitions' visibility.

Definitions, references, and scopes are defined in *binding rules*. A binding rule takes the form `pattern: clause`, where `pattern` is a term pattern and `clause` is a name-binding declaration about the language construct that matches `pattern`. Spoofax uses terms for the abstract representation of language constructs. A pattern is a term with variables. A term matches a pattern if its variables can be bound to subterms in the actual term. The name-binding rules for `let` expressions are

```
module names imports Expressions
namespaces Variable
binding rules
  Let(x, i, body):  defines Variable x
  Var(x):           refers to Variable x
```

There is only a single namespace *Variable* for variables. The first binding rule handles variable definitions. Its pattern matches terms representing a `let` expression and binds `x` to the name of the declared variable, `i` to the initialization expression, and `body` to the expression in the body. The **defines** clause specifies that terms matching the rule's pattern define the name `x` in *Variable*. The second binding rule handles variable references; it matches terms representing those references and binds `x` to the referred variable's name. The **refers to** clause specifies that terms matching the rule's pattern refer to a definition of `x` in the *Variable* namespace.

Figure 3 shows five test cases for name binding. The first four mark different occurrences of names with [[…]] and specify which occurrence should resolve to which other occurrence. The fifth test case specifies an expected warning on an inner `let` expression, which hides a variable declared in an outer `let` expression.

These test cases indicate a mistake in the binding rules we've specified so far. Currently, variable definitions aren't scoped. They're visible in all expressions, even if they're in different files. We can correct the corresponding binding rule and restrict a variable's scope to the body of the declaring `let` expression:

```
Let(x, i, body): defines Variable x in body
```

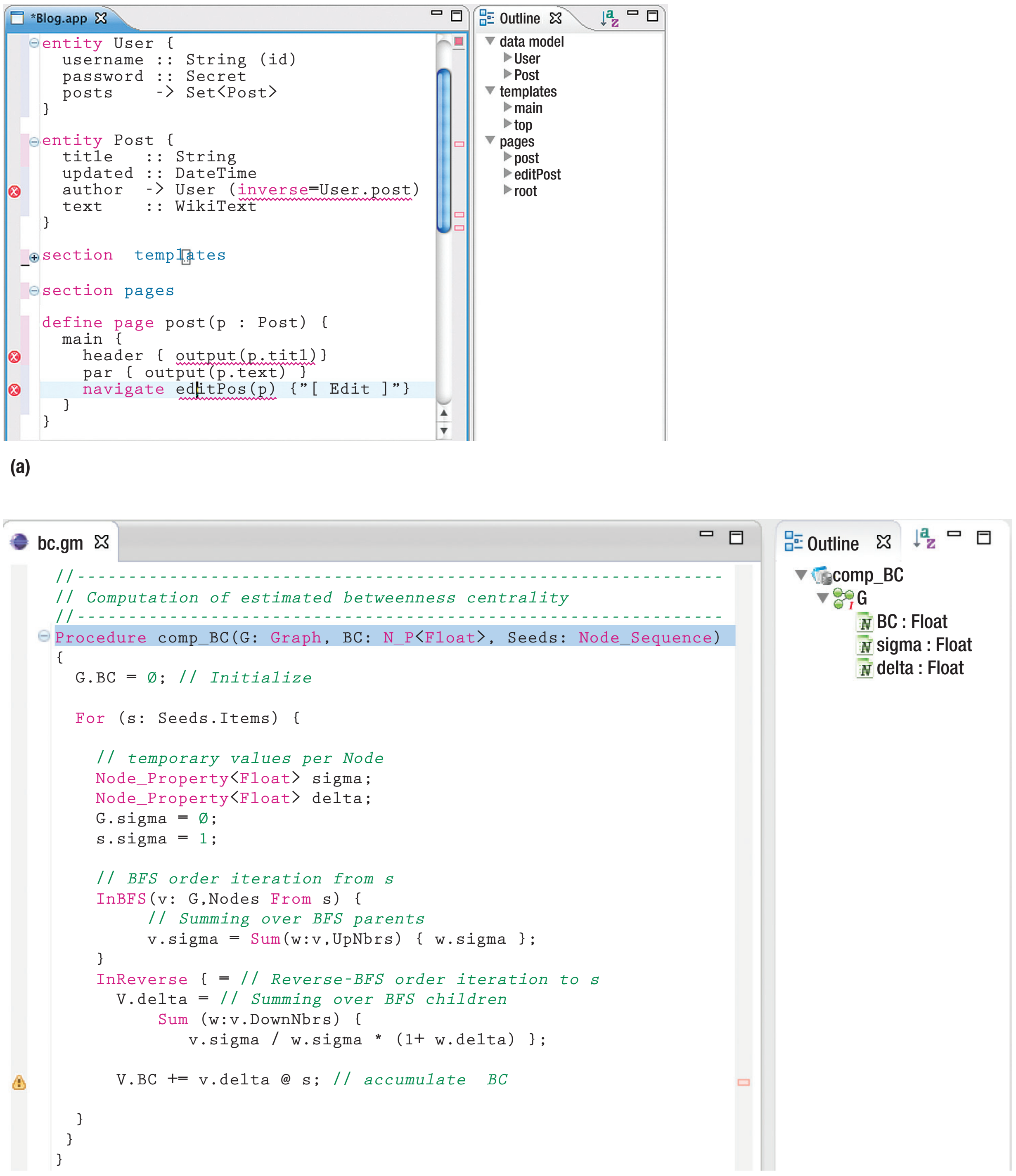
Spoofax in Practice

Spoofax is used to develop programming languages in education, research, and industry. We use Spoofax in two MSc courses at the Delft University of Technology. In the compiler construction course, students build a compiler for a Java subset with Spoofax. In the language engineering project course, students use Spoofax to design domain-specific languages (DSLs).

Mobl is a DSL for mobile Web applications. It integrates languages for user interface design, styling, data modeling, querying, and application logic into a single, unified language that's flexible and expressive and that enables early error detection.[6]

WebDSL is the largest, most complex DSL designed with Spoofax to date (see Figure 4a). A DSL for Web information systems, WebDSL maintains separation of concerns while integrating its sublanguages, enabling consistency checking and reusing common language concepts.[5]

SugarJ is an extensible programming language on top of Java and the Spoofax metalanguages SDF and Stratego. (Stratego unifies program transformation and code generation.[7]) It supports syntactic extensibility of the host language in the form of language libraries.[8]



```
entity User {
  username :: String (id)
  password :: Secret
  posts    -> Set<Post>
}

entity Post {
  title   :: String
  updated :: DateTime
  author  -> User (inverse=User.post)
  text    :: WikiText
}

section  templates

section pages

define page post(p : Post) {
  main {
    header { output(p.titl)}
    par { output(p.text) }
    navigate editPos(p) {"[ Edit ]"}
  }
}
```

(a)

```
//-----------------------------------------------------------------
// Computation of estimated betweenness centrality
//-----------------------------------------------------------------
Procedure comp_BC(G: Graph, BC: N_P<Float>, Seeds: Node_Sequence)
{
  G.BC = Ø; // Initialize

  For (s: Seeds.Items) {

    // temporary values per Node
    Node_Property<Float> sigma;
    Node_Property<Float> delta;
    G.sigma = Ø;
    s.sigma = 1;

    // BFS order iteration from s
    InBFS(v: G.Nodes From s) {
         // Summing over BFS parents
         v.sigma = Sum(w:v.UpNbrs) { w.sigma };
    }
    InReverse { = // Reverse-BFS order iteration to s
      V.delta = // Summing over BFS children
          Sum (w:v.DownNbrs) {
             v.sigma / w.sigma * (1+ w.delta) };

      V.BC += v.delta @ s; // accumulate  BC

    }
   }
 }
```

(b)

FIGURE 4. Eclipse IDEs for (a) WebDSL and (b) Green-Marl. Both DSLs are developed in Spoofax. Spoofax automatically derives efficient implementations for IDE features.

```
static void gm_expand_argument_list(std::list<ast_argdecl*>& A) {
   std::list<ast_argdecl*> s; // temp;
   std::list<ast_argdecl*>::iterator I;

   // expand x,y : INT -> x:INT, y:INT
   for (I = A.begin(); I != A.end(); I++) {
     ast_argdecl *a  = *I;
      ast_idlist* idl = a->get_idlist();
      ast_typedecl* t = a->get_type();
      if (idl->get_length() == 1) {
         s.push_back(a);
      } else {
         for  (int i = 0; i < idl->get_length();  i++) {
            ast_id* I =  idl->get_item(i)->copy();
            ast_idlist* IDL = new ast_idlist();
            IDL->add_id(I);
            ast_typedecl* T = t->copy();

            ast_argdecl *aa = ast_argdecl::new_argdecl(IDL, T);
            s.push_back(aa);
         }

         delete a;
      }
   }

   // new clear A, and put contents of S into A
   A.clear();
   for (I = s.begin(); I !=  s.end(); I++) {
      A.push_back(*I);
   }
}
```
(a)

```
strategies

   normalize-all = innermost(normalize)

rules

  normalize:
   [ ArgDecl([n1, n2 | n*], ty) | arg* ] ->
   [ ArgDecl([n1], ty), ArgDecl([n2 | n*], ty) | arg* ]
```
(b)

FIGURE 5. A normalization step in the Green-Marl compiler, splitting argument declarations into separate declarations. (a) The manual implementation in C++. (b) The declarative specification in Spoofax.

Developed by Stanford University's Pervasive Parallelism Laboratory, *Green-Marl* is a DSL for efficient graph data analysis.[9] Its original compiler is implemented as a command-line tool in C++. In 2012, we collaborated with Oracle Labs to evaluate the Spoofax workbench's applicability to DSLs under development at Oracle. As part of this collaboration, we redesigned Green-Marl with Spoofax and developed a Green-Marl plugin for Eclipse (see Figure 4b).

Although the Spoofax version provides additional IDE features, it's significantly smaller than the handwritten command-line compiler. The syntax definition in SDF3 is only half the size of the YACC-based syntax definition from the hand-written compiler. The specification of Green-Marl's type system in NaBL and TS (Spoofax's metalanguage for type specification) is an order of magnitude smaller than the manually implemented static analysis in C++.

Because of the Spoofax version's declarative nature, it improves extensibility and maintainability. For example, declarative name-binding and typing rules reveal cases that the manual implementation doesn't cover. Figure 5 provides another example, showing the implementation of a normalization step in C++ and the specification of the same normalization step in Spoofax. We can easily extend the latter by adding new `normalize` rules.

Implementation

Spoofax is based on several language-parametric runtime systems. Spoofax maps declarative language designs to parameters for these runtime systems. We bootstrap Spoofax's metalanguages to derive Spoofax-based IDEs for these languages.

Parsing and Error Recovery

From a syntax definition in SDF3, we derive a permissive grammar that also accepts programs with minor errors by adding recovery productions.[10] We generate a parse table from the permissive grammar. This parse table is interpreted by a Java implementation of a scannerless generalized LR parsing algorithm,[11] which we extend with a selective form of backtracking that's used only for error recovery. We ignore all recovery productions during normal parsing; we employ backtracking to apply the recovery rules only when an error is detected.[10]

Formatting

Spoofax provides Stratego to specify program transformation and code generation; we also use it as an implementation language for IDE features. To derive an implementation of a formatter, we translate each template production from a syntax definition in SDF3 to a Stratego rewrite rule. These rules match abstract representations and produce concrete representations by combining concrete syntax fragments, whitespace, and formatted subelements. The rewrite rules are desugared into core constructs, which can be interpreted by the Stratego interpreter or compiled into Java.

Incremental Name and Type Analysis

Name and type analysis involves two phases. The collection phase analyzes lexical scopes, collects information about binding instances, and creates deferred analysis tasks in a top-down traversal.[1] We derive this traversal by generating Stratego rewrite rules from declarative name-binding and scope rules in NaBL and typing rules in TS. The resulting Stratego rules are either desugared and interpreted, or compiled into Java.

Each analysis task captures a name resolution or type analysis step. Tasks might depend on other tasks and are evaluated by an incremental task engine in the evaluation phase.[4] The task engine is implemented in Java and Stratego and is integrated into the Spoofax runtime environment.

Incremental analysis is supported at the file level by the collection phase and at the task level by the evaluation phase. When a file changes, only this file is recollected, and only the tasks affected by the changes in the collected data are reevaluated. Consequently, the analysis neither reparses nor retraverses unchanged files, even if they're affected by changes in other files.

Spoofax is used to develop programming languages in education, research, and industry.

Origin Tracking

For several IDE features, such as syntax highlighting, outline views, and content completion, we generate declarative default specifications. These can be customized with additional, user-defined specifications. We combine default and user-defined specifications into a single specification, which the Spoofax runtime system interprets.

Many of these features also rely on generic or generated Stratego rules, including outline views, constraint checks, reference resolution, and name-based content completion. To support source code navigation and precise error marking, the Spoofax runtime system keeps track of original source positions. Both interpreted and compiled Stratego rules implicitly pass along position information to maintain relations between the original source positions and rewrite results.

Bootstrapping

The Spoofax project started with the development of Eclipse editors dedicated to SDF2 and Stratego. It later evolved to a language workbench, providing IDE support for languages built with Spoofax. The metalanguage editors for SDF2, Stratego, and Spoofax's editor description language[12] were fully bootstrapped. Their syntax was defined in SDF2, their analyses and code generations were implemented in Stratego, and their editor services were specified in the editor description language.

With the emergence of metalanguages such as SDF3, NaBL, and TS, we started a new bootstrapping cycle. SDF3, NaBL, and TS are already completely bootstrapped. We recently started migrating the syntax definition of Stratego to SDF3 and to express name and type analysis in NaBL and TS, leaving Stratego only for transformations such as normalization and code generation.

Availability

Spoofax is developed at the Delft University of Technology's Software Language Design and Engineering Lab. All lab results are available for application, reproduction, and further research through open source software distributions. We continuously build Spoofax releases, which are available at Eclipse update sites for stable, unstable, and nightly releases. We emphasize development of research software to the extent that it's usable in production systems. Furthermore, the SugarJ team and Oracle Labs contribute to Spoofax.

Oracle Labs also funds research and development for Spoofax. The

Netherlands Organization for Scientific Research awarded Eelco Visser a €1.5-million Vici grant to develop methods and techniques for automatically verifying language definitions. These techniques will be integrated in Spoofax and should let language developers easily detect semantic errors early.

Comparison

Spoofax is a language workbench for developing textual programming languages. We compare Spoofax to other notable tools for creating and using textual languages.

Parser Generators

Parser generators typically support particular parsing algorithms, which work for only a subclass of the set of all context-free grammars, such as LL(1), LR(1), and LALR(1). Grammar class restrictions can seem arbitrary to prospective users. From an implementation viewpoint, the restrictions make sense for the algorithms used for these parsers. However, from a usability viewpoint, the restrictions reveal a leaky abstraction. The implementation directs and restricts how a grammar can be written, forcing language engineers to focus on a parser implementation's complex inner workings. Instead of focusing on language design, engineers must deal with parsing algorithms' idiosyncrasies. Factorizing and massaging syntax definitions lead to specifications that don't correspond with the high-level declarative description of the language's natural grammar.[13]

Language Workbenches

Many language workbenches provide high-level mechanisms for implementing programming languages.[14] We compare Spoofax to the textual language workbenches Xtext[15] and Rascal[16] and to the projectional language workbench MPS (Meta Programming System).[17]

Xtext is a mature open source framework for developing programming languages and DSLs. It lets developers reuse established and commonly understood default semantics for many language aspects. It relies on the ANTLR (Another Tool for Language Recognition) parser generator and inherits its grammar class restriction, requiring language engineers to factorize and massage their syntax definitions and preventing them from freely composing syntax definitions. Name bindings are specified as cross-references in the grammar. A simple resolution algorithm then automatically resolves references. Scoping or visibility can't be defined in the grammar but must be implemented in Java with the help of a scoping API with default resolvers. Common constraint checks such as duplicate definitions, use-before definitions, and unused definitions must also be specified manually. This increases the manual implementation effort.

Rascal is an extensible metaprogramming language and IDE for source code analysis and transformation. Rascal employs GLL (generalized LL) parsing, which supports debugging of syntax definitions. Spoofax derives efficient implementations from declarative specifications in different metalanguages. In contrast, Rascal provides a single metalanguage to support programmatic encodings of name and type analyses and custom IDE features for programming languages.

MPS is an open source language workbench developed by JetBrains. It provides projectional editors with which users edit a projection of the abstract representation in a standard, fixed layout. This allows for integrated textual, symbolic, and tabular notation. Owing to MPS's projectional nature, parsing and name binding are no longer needed, and it provides wide-ranging support for composing and extending languages and editors. Similar to Spoofax, MPS provides declarative metalanguages for testing language definitions and typing rules.

Programming language designers want to get usable, reliable realizations of their language design ideas into the hands of programmers as efficiently as possible. To achieve this goal, they need to produce several artifacts:

- a compiler or interpreter that allows programmers to execute programs in the language,
- an IDE that supports programmers in constructing programs in the language,
- a high-level specification of the language that documents its intent for programmers, and
- validation, via automated testing or formal verification, that the language designs and implementations are correct and consistent.

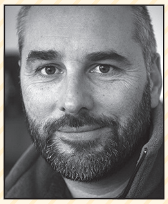
Existing tools generally require the designer to create each of these artifacts separately, even though they reflect the same underlying design. Consequently, a compiler or interpreter is often the only artifact produced; documentation, IDE, and—especially—validation are typically omitted. For example, language implementations rarely formally guarantee semantic correctness properties such as type soundness and behavior preservation of transformations, because current implementation tools don't provide support for verification. This can lead to subtle errors in languages that are discovered late.

Our vision is a language designer's workbench as a one-stop shop for language design implementation and validation.[18] The key to realizing this vision is to conceptualize the subdomains of language definition as a collection of declarative, multipurpose metalanguages, so that a single language definition can be used as the source for the implementation of efficient and scalable compilers and IDEs, the verification or testing of correctness properties, and as a source of technical documentation for users of the language.

ABOUT THE AUTHORS

GUIDO H. WACHSMUTH is an assistant professor in the Software Language Design and Engineering program at the Delft University of Technology. He designs and implements declarative metalanguages for Spoofax. His research focuses on new linguistic abstractions in metalanguages, which enable declarative definitions of software languages and form a single basis for efficiently executing various language-processing tasks. Wachsmuth received a PhD in computer science from Humboldt Universität zu Berlin. He's a member of the ACM Special Interest Group on Programming Languages. Contact him at guwac@acm.org.

GABRIËL D.P. KONAT is a PhD student in the Software Language Design and Engineering program at the Delft University of Technology. His research interests include software language engineering, domain-specific languages, and declarative metalanguages and their efficient implementation. He currently works on incremental name and type analyses, which can be automatically derived from declarative specifications in Spoofax. Konat received an MSc in computer science from the Delft University of Technology. He's a member of the ACM Special Interest Group on Programming Languages. Contact him at gkonat@acm.org.



EELCO VISSER is an Antoni van Leeuwenhoek Professor of computer science at the Delft University of Technology, where he leads the Software Language Design and Engineering program. That program develops Spoofax and many domain-specific languages (DSLs), including DSLs for syntax definition, program transformation, Web application development, and mobile phone applications. Visser received a PhD in computer science from the University of Amsterdam. He's a member of IEEE and the ACM Special Interest Group on Programming Languages. Contact him at visser@acm.org.

References

1. T. Vollebregt, L.C.L. Kats, and E. Visser, "Declarative Specification of Template-Based Textual Editors," *Proc. 2012 Workshop Language Descriptions, Tools, and Applications* (LDTA 12), 2012, article 8.
2. L.C.L. Kats, R. Vermaas, and E. Visser, "Testing Domain-Specific Languages," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 11), 2011, pp. 25–26.
3. G.D.P. Konat et al., "Declarative Name Binding and Scope Rules," *Software Language Engineering*, LNCS 7745, Springer, 2012, pp. 311–331.
4. G. Wachsmuth et al., "A Language Independent Task Engine for Incremental Name and Type Analysis," *Software Language Engineering*, M. Erwig et al., eds., LNCS 8225, Springer, 2013, pp. 260–280.
5. D.M. Groenewegen et al., "WebDSL: A Domain-Specific Language for Dynamic Web Applications," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 08), 2008, pp. 779–780.
6. Z. Hemel and E. Visser, "Declaratively Programming the Mobile Web with Mobl," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 11), 2011, pp. 695–712.
7. M. Bravenboer et al., "Stratego/XT 0.17. A Language and Toolset for Program Transformation," *Science of Computer Programming*, vol. 72, nos. 1–2, 2008, pp. 52–70.
8. S. Erdweg et al., "Library-Based Model-Driven Software Development with SugarJ," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 11), 2011, pp. 17–18.
9. S. Hong et al., "Green-Marl: A DSL for Easy and Efficient Graph Analysis," *Proc. Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 12), 2012, pp. 349–362.
10. M. de Jonge et al., "Natural and Flexible Error Recovery for Generated Modular Language Environments," *ACM Trans. Programming Languages and Systems*, vol. 34, no. 4, 2012, article 15.
11. E. Visser, *Scannerless Generalized-LR Parsing*, tech. report P9707, Programming Research Group, Univ. Amsterdam, 1997.
12. L.C.L. Kats and E. Visser, "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 10), 2010, pp. 444–463.
13. L.C.L. Kats, E. Visser, and G. Wachsmuth, "Pure and Declarative Syntax Definition: Paradise Lost and Regained," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 10), 2010, pp. 918–932.
14. S. Erdweg et al., "The State of the Art in Language Workbenches," *Software Language Engineering*, M. Erwig et al., eds., LNCS 8225, Springer, 2013, pp. 197–217.
15. M. Eysholdt and H. Behrens, "Xtext: Implement Your Language Faster Than the Quick and Dirty Way," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 10), 2010, pp. 307–309.
16. P. Klint, T. van der Storm, and J. Vinju, "EASY Meta-programming with Rascal," *Generative and Transformational Techniques in Software Engineering III*, J.M. Fernandes et al., eds., LNCS 6491, Springer, 2011, pp. 222–289.
17. M. Völter and V. Pech, "Language Modularity with the MPS Language Workbench," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 1449–1450.
18. E. Visser, et al., "A Language Designer's Workbench: A One-Stop Shop for Implementation and Verification of Language Designs," to be published in *Proc. Symp. New Ideas in Programming and Reflections on Software* (Onward! 14), 2014.