Principled Syntactic Code Completion using Placeholders

Luís Eduardo de Souza Amorim   Sebastian Erdweg   Guido Wachsmuth   Eelco Visser

Delft University of Technology, Netherlands

l.e.desouzaamorim-1@tudelft.nl, s.t.erdweg@tudelft.nl, guwac@acm.org, e.visser@tudelft.nl

Abstract

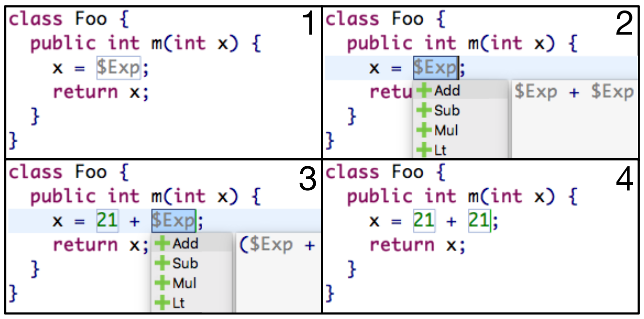
Principled syntactic code completion enables developers to change source code by inserting code templates, thus increasing developer efficiency and supporting language exploration. However, existing code completion systems are ad-hoc and neither complete nor sound. They are not complete and only provide few code templates for selected programming languages. They also are not sound and propose code templates that yield invalid programs when inserted. This paper presents a generic framework that automatically derives complete and sound syntactic code completion from the syntax definition of arbitrary languages. A key insight of our work is to provide an explicit syntactic representation for incomplete programs using placeholders. This enables us to address the following challenges for code completion separately: (i) completing incomplete programs by replacing placeholders with code templates, (ii) injecting placeholders into complete programs to make them incomplete, and (iii) introducing lexemes and placeholders into incorrect programs through error-recovery parsing to make them correct so we can apply one of the previous strategies. We formalize our framework and provide an implementation in Spoofax.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments

Keywords Code Completion, Language Workbenches, IDEs

1. Introduction

Code completion, also known as content completion or content assist, is an editor service that proposes and performs expansion of the program text. Code completion helps the programmer to avoid misspellings and acts as a guide to discover



```
class Foo {
  public int m(int x) {
    x = $Exp;
    return x;
  }
}
```

```
class Foo {
  public int m(int x) {
    x = $Exp;
    retu
  }
}
```
```
$Exp + $Exp
```

```
class Foo {
  public int m(int x) {
    x = 21 + $Exp;
    return x;
  }
}
```
```
($Exp +
```

```
class Foo {
  public int m(int x) {
    x = 21 + 21;
    return x;
  }
}
```

Figure 1. (1) Incomplete program with explicit placeholders. (2) Triggering completion for a placeholder. (3) After selecting a proposal, showing completions for nested placeholders. (4) Completing a nested placeholder by typing.

language features and APIs. Most mainstream integrated development environments (IDEs) provide some form of code completion and industrial studies indicate that code completion is one of the most frequently used IDE services [1].

There are two classes of code completion: syntactic and semantic. Syntactic code completion considers the syntactic context at the cursor position and proposes code templates for syntactic structures of the language. For example, most IDEs for Java support syntactic code completion with class and method templates. Semantic code completion also uses the cursor position to propose templates, but by applying semantic analysis to the program, the IDE can propose code templates or identifiers that do not violate the language's name binding or typing rules. For example, in this case IDEs for Java may suggest variables or methods that are visible in the current scope and have the expected type at the cursor position.

In this paper, we focus on *syntactic code completion*. Even for mainstream languages in mainstream IDEs, syntactic code completion is often ad-hoc and unreliable. Specifically, most existing services for syntactic code completion are incomplete and only propose code templates for selected language constructs of a few supported languages, thus inhibiting exploring the language's syntax. Moreover, most existing services are unsound and propose code templates that yield syntax errors when inserted at the cursor position.

To address these shortcomings, we present a generic code-completion framework that derives sound and complete syntactic code completion from syntax definitions. From the syntax definition, we derive code templates and applicability conditions for them to ensure soundness. To support completeness and propose all language structures, we represent incomplete program text explicitly using placeholders that we automatically introduce into the syntax definition. This allows our code templates to yield incomplete programs that can be subsequently completed.

Figure 1 illustrates our use of placeholders in a Java-like program. The first box shows an incomplete program with an expression placeholder. The program is syntactically correct since we introduce the placeholder as part of the language. The second box shows that placeholders give rise to completion proposals, which may themselves be incomplete (contain placeholders). After selection of a proposal, the developer can expand or textually replace the placeholders inserted by the template.

In addition to enabling step-wise code completion, explicit placeholders allow us to address two important practical challenges of code completion: inferring completion opportunities in complete program texts and generating completion proposals while recovering from syntax errors. Complete programs do not contain placeholders, yet code completion can be useful for adding list elements or optional constructs. For example, we may want to use syntactic code completion to add modifiers like `public` to a method or to add statements to a method's body. Instead of developing such support for complete programs directly, we provide our solution using placeholder inference (to make the program incomplete) followed by regular syntactic code completion of the inferred placeholder.

Incorrect programs contain syntax errors but are important to support because incorrect programs occur frequently during development. Again, instead of developing syntactic code completion for incorrect programs directly, we decompose this activity. We use error-recovery parsing [3, 4] to insert lexemes into the program text. However, since we made placeholders part of the language, an error-recovering parser will also consider placeholders for insertion, thus yielding incomplete programs. A developer can select one of multiple alternative recoveries and can use regular syntactic code completion for placeholders in the selected recovered program. Fig. 2 shows all transitions between complete, incomplete, and incorrect programs.

We present a formalization of our completion framework and the involved algorithms. We describe completeness, formally define soundness, and prove soundness for our algorithms. We also implemented our framework as part of the Spoofax language workbench [7], which we used to derive syntactic code completion for a subset of Java containing classes, methods, statements and expressions, Pascal, and IceDust [6], a domain-specific language for data modeling.

In summary, we make the following contributions:

- An analysis of syntactic code completion in IDEs and language workbenches, revealing completeness and soundness issues (Section 2).
- A sound and complete approach for completing incomplete programs by rewriting placeholders (Section 3).
- An algorithm for inferring placeholders, yielding support for expanding complete programs (Section 4).
- An extension of error-recovery parsing for inserting placeholders, yielding syntactic code completion for incorrect programs (Section 5).
- Throughout, we develop a formal framework for reasoning about syntactic code completion and we describe how we realized the algorithms in Spoofax.

2. State of the Art of Syntactic Completion

In this section, we motivate our work on syntactic code completion by presenting examples collected from state-of-the-art IDEs and language workbenches. We observe that state-of-the-art solutions are unsound, incomplete, language-dependent, and do not support incorrect programs well.

Soundness. Existing IDEs and language workbenches often propose unsound completions that yield syntax errors when inserted. For example, as shown in Figure 3, Eclipse largely ignores the syntactic context at the cursor position and proposes the insertion of an *else*-block without a corresponding *if*-statement, yielding a syntax error after insertion. IntelliJ provides code templates that are sensitive to the syntactic and typing context, but may yield syntax errors nonetheless. For example, the live template `lst` inserts an

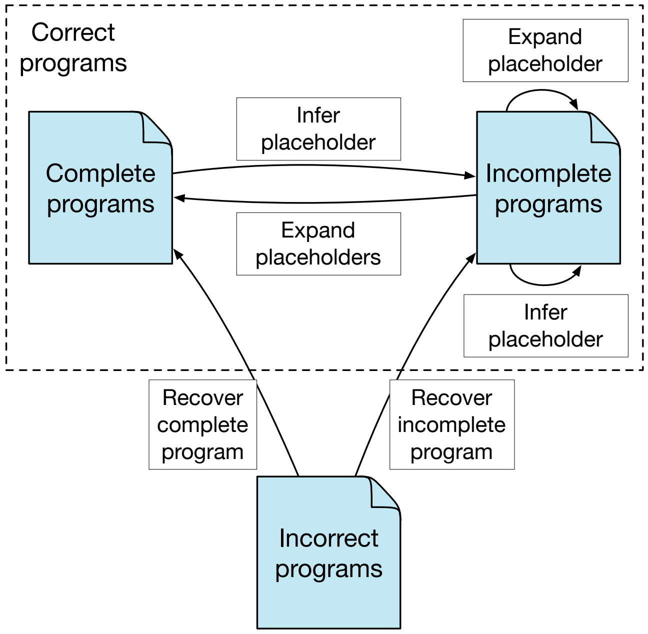
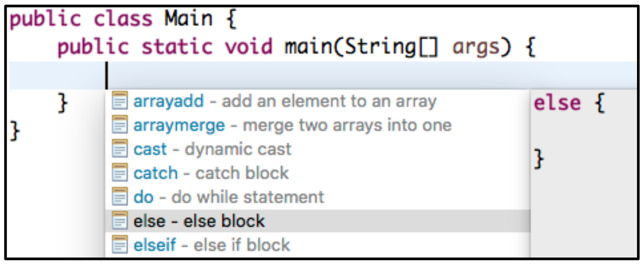


Figure 2. Separation of concerns in code completion.



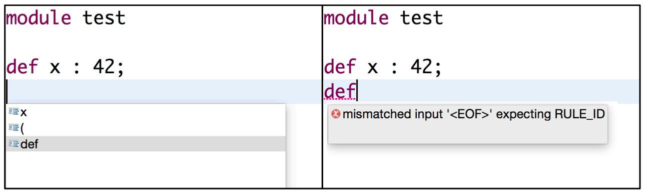
```
public class Main {
    public static void main(String[] args) {

    }
}
```

arrayadd - add an element to an array
arraymerge - merge two arrays into one
cast - dynamic cast
catch - catch block
do - do while statement
else - else block
elseif - else if block

```
else {

}
```

Figure 3. Eclipse: Unsound completion yields syntax error.

expression for fetching the last element of an array, but yields the invalid fragment `[.length - 1]` when no array is available in the current scope. Language workbenches have similar issues and also propose invalid completions. For example, as shown in Figure 4, Xtext [5] only proposes the next keyword, but not a complete *def*-template.



```
module test

def x : 42;
```

x
(
def

```
module test

def x : 42;
def
```

mismatched input '<EOF>' expecting RULE_ID

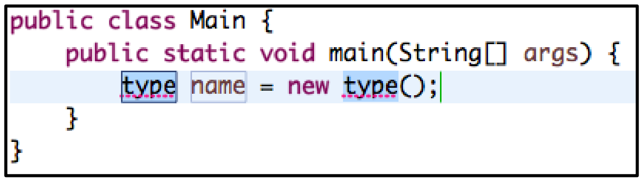
Figure 4. Xtext: Unsound completion yields syntax error.

Completeness. IDE developers define code templates manually. As a consequence, the set of available templates is limited. Many language constructs are not available through syntactic code completion, and changes to a language are often not reflected in the code templates. For example, the proposal list of Eclipse 4.5.2 shown in Figure 3 does not provide a code template for *try-with-resource* statements.

Besides missing templates, existing IDEs and language workbenches also have no way to represent partial completions that users can subsequently complete to form complex constructs. Instead, existing systems always generate complete programs with concrete "dummy" constructs as subexpressions. For example, as shown in Figure 5, Eclipse's proposal for constructing and storing a new object yields a complete program using `type` as a "dummy" class name and `name` as a "dummy" variable name.

If the developer leaves the IDE's completion mode, the "dummy" constructs become part of the program. This inhibits partial completions such as for assignment statements `type name = exp`, where `exp` will be interpreted as a variable reference rather than as a placeholder for arbitrary expressions.

Incorrect Programs. In current IDEs and language workbenches, error-recovery parsing and code completion are largely orthogonal. For example, Eclipse uses error-recovery parsing to identify the syntactic context at the cursor position and compute corresponding proposals. However, Eclipse does not actually offer support for recovering from the syntax



```
public class Main {
    public static void main(String[] args) {
        type name = new type();
    }
}
```

Figure 5. Eclipse: Completion with "dummy" constructs.

error itself. Similarly, code completion and error recovery are orthogonal in IntelliJ and the previous version of the Spoofax language workbench [3, 7]. Instead, error recovery should yield a list of alternative recovery proposals for the user to select from. If the user selects an incomplete program as recovery, the user can continue to expand the program in subsequent code completion steps.

Summary. Based on this discussion, we derive the following requirements for principled syntactic code completion:

- Proposals need to be *sound* such that code completion does not introduce syntax errors.
- Proposals need to be *complete*, meaning that code templates exist for all language constructs and that developers can use iterative code completion.
- Code completion should propose *recoveries* for incorrect programs and allow the iterative completion of recovered programs.

In the remainder of this paper, we present a generic framework for syntactic code completion that satisfies these requirements. Our framework is language-independent and automatically derives principled code-completion support from a language's syntax definition.

3. Completion by Rewriting Placeholders

In most editors, programs in an incomplete state are incorrect, as they contain syntax errors indicating missing elements in the source code. In this section, we present a formal model for syntactic code completion for a subset of incomplete programs where missing elements correspond to entire structures from the language. We introduce a valid representation for this subset, representing these structures by explicit placeholders. As the programs in the subset are syntactically correct considering our representation, our framework models sound and complete syntactic code completion deriving completion proposals as placeholder expansions. Finally, we present an instantiation of our formal model, describing our implementation of syntactic code completion in Spoofax.

3.1 Representing Complete and Incomplete Programs

We consider abstract syntax trees as our primary program representation. We define representations for complete and incomplete programs as terms over a signature Σ:

Definition 1 (Signature). *A signature $\Sigma = \langle S, C\rangle$ is a pair consisting of a set of sorts $s \in S$ and a set of constructor* *declarations* $(c : s_1 \times \ldots \times s_n \to s_0) \in C$ *with zero or more arguments and all* $s_i \in S$*. The set of sorts must contain the predefined sort* $\mathsf{LEX} \in S$ *for representing lexemes.*

Given a signature $\Sigma = \langle S, C \rangle$, we define the well-formed terms T_Σ over Σ as follows:

Definition 2 (Well-formed terms)**.** *For each sort* $s \in S$*, the set of well-formed terms* T_Σ^s *of sort* s *is the least set such that*

$$\frac{s \text{ is a string literal}}{s \in T_\Sigma^{\mathsf{LEX}}} \tag{1}$$

$$\frac{\begin{array}{c}(c : s_1 \times \ldots \times s_n \to s) \in C \\ t_i \in T_\Sigma^{s_i} \quad \forall 1 \leq i \leq n\end{array}}{c(t_1, \ldots, t_n) \in T_\Sigma^s} \tag{2}$$

The family T_Σ *of well-formed terms is then defined as*

$$T_\Sigma = (T_\Sigma^s)_{s \in S}.$$

Well-formed terms represent complete programs over Σ. For example, given a signature for a statement in an imperative programming language, the term

`Assign("x", Add(Int("21"), Int("21")))`

represents a complete program.

We represent incomplete structures in programs by means of explicit placeholders. We introduce an explicit placeholder $\$s$ for each sort s as a nullary constructor:

Definition 3 (Placeholders)**.** *Given a signature* $\Sigma = \langle S, C \rangle$*, we define placeholders as the set of nullary constructor declarations*

$$S^\$ = \{\$s : s \mid s \in S\}$$

and the placeholder-extended signature

$$\Sigma^\$ = \langle S, C \cup S^\$ \rangle.$$

Well-formed terms over an extended signature $\Sigma^\$$ represent incomplete programs. For example, the term

`Assign("x", $Exp)`

represents an incomplete program, where we use the placeholder `$Exp` of sort `Exp` instead of a concrete argument term for `Assign`. According to our definition, every complete program is also an incomplete program. However, a program is properly incomplete if $t \in T_{\Sigma^\$}$ and $t \notin T_\Sigma$, that is, t contains at least one placeholder.

3.2 Terms with Source Regions

The goal of our formalization is to provide a formal framework for syntactic code completion. Since code completion is sensitive to the cursor position in the source code, we need to extend our representation of terms with source regions. This will later enable us to map the cursor position to the corresponding subterm.

A term's source region identifies the region of the original source file to which the term corresponds. Later, when we use code completion to synthesize terms, we will also need empty source regions that have no correspondence in the source file.

Definition 4 (Source region)**.** *A source region* r *is an interval* $[m, n] = \{x \in \mathbb{N} \mid m \leq x \leq n\}$ *starting at character offset* m *and ending at character offset* n*. We define* $r_1 < r_2$ *to mean* $\forall x_1 \in r_1.\ \forall x_2 \in r_2.\ x_1 < x_2$*.*

Note that $\emptyset$ denotes an empty source region of the source file. We use $\emptyset$ to denote the source region of synthesized terms. Note furthermore that $r_1 < r_2$ expresses that r_1 precedes r_2 and the two regions may not touch and not overlap. Finally, the empty region is not affected by the ordering, $\emptyset < r$ and $r < \emptyset$ for all r.

We define an augmented set of well-formed terms that associates a source region with each subterm:

Definition 5 (Well-formed terms with source regions)**.** *For each sort* $s \in S$*, the set of well-formed terms with source regions* $T_\Sigma^{R,s}$ *of sort* s *is the least set such that*

$$\frac{s \text{ is a string literal}}{s^r \in T_\Sigma^{R,\mathsf{LEX}}} \tag{3}$$

$$\frac{\begin{array}{c}(c : s_1 \times \ldots \times s_n \to s) \in C \\ \forall 1 \leq i \leq n : t_i^{r_i} \in T_\Sigma^{R,s_i} \\ \forall 1 \leq i < j \leq n : r_i < r_j \\ r_1 \cup \cdots \cup r_n \subseteq r\end{array}}{c(t_1^{r_1}, \ldots, t_n^{r_n})^r \in T_\Sigma^{R,s}} \tag{4}$$

The family T_Σ^R *of well-formed terms with source regions is then defined as*

$$T_\Sigma^R = (T_\Sigma^{R,s})_{s \in S}.$$

The first two preconditions of Equation 4 ensure that the terms in T_Σ^R are well-formed as before. The latter two preconditions ensure that the annotated source regions are well-formed. That is, the region of each left-sibling precedes the region of each right-sibling and the region of a parent term includes the regions of all its subterms. The well-formedness of source regions allows us to efficiently navigate within terms to identify the subterm corresponding to a cursor position. Finally, note that $T_{\Sigma^\$}^R$ denotes the set of terms of incomplete programs with source regions.

3.3 Completing Incomplete Programs

We are now ready to define code completion for incomplete programs where we replace explicit placeholders by proposed terms. We divide the definition of code completion into three functions $\mathsf{replace}$, $\mathsf{propose}$, and $\mathsf{complete}$. Function $\mathsf{replace}$ takes a term t^r and replaces its subterm u^p by term v. We use source regions r and p to navigate in t and to uniquely identify subterm u.

Definition 6 (Function $\mathsf{replace}$)**.**

$$\mathsf{replace}(t^r, u^p, v) = \begin{cases} v^\emptyset, & \textit{if } t^r = u^p \\ \mathsf{replace}(t_i^{r_i}, u^p, v), & \textit{if } t^r \neq u^p, t = c(t_1^{r_1}, \ldots, t_n^{r_n}), \\ & p \subseteq r_i \\ t^r, & \textit{otherwise} \end{cases}$$

If the current term t^r equals u^p including the source region, we yield the replacement v. We recursively impose the empty source region on term v to mark it as being synthesized. We use the source region p of u to navigate to and recurse on subterm $t_i^{r_i}$ of t^r such that p is included in r_i. If we cannot find an appropriate subterm, we yield the current term t^r unchanged.

We are not only interested in defining functions like replace but also in the metatheoretical properties of these functions. In particular, we want to reason about the soundness of code completion, which means that code completion yields well-formed terms. Thus, before moving on to the other functions, we define precisely when an application of replace is sound.

Theorem 3.1 (Soundness of replace). *Given $t^r \in T_\Sigma^{R,s}$ for some sort s, a replacement* $\mathsf{replace}(t^r, u^p, v) = w^q$ *is sound iff $w^q \in T_\Sigma^{R,s}$. If $u^p \in T_\Sigma^{R,s'}$ for some sort s' and $v \in T_\Sigma^{s'}$, then* $\mathsf{replace}(t^r, u^p, v)$ *is sound.*

That is, a replacement is sound if it yields well-formed terms of the same sort as input t^r. Specifically, a replacement of u by v in t is sound if u and v have the same sort. For the proof of this theorem it is important that we impose the empty region on v, such that the result of the replacement has well-formed regions.

We will use replace to inject proposed code fragments in place of placeholders $\$s$. Function proposals computes a list of proposed code fragments for a given sort s. Here, we only specify proposals abstractly, reasoning about its soundness.

Definition 7 (Proposals function). *Given a signature $\Sigma = \langle S, C \rangle$, a proposal function* $\mathsf{proposals} : S \to (T_{\Sigma\$})^*$ *maps each sort $s \in S$ to a sequence of proposed terms. A proposal function* proposals *is sound iff for all $s \in S$, the terms proposed for s have sort s:* $\mathsf{proposals}(s) \in (T_{\Sigma\$}^s)^*$.

Our proposal function permits context-free syntactic code-completion proposals based on the expected sort. Based on proposals and replace, we can model code completion by (i) navigating to the placeholder at the current cursor position $c \in \mathbb{N}$, (ii) getting proposals for that placeholder, (iii) replacing the placeholder by one of the proposed terms. Function propose performs steps (i) and (ii). That is, propose takes a term $t^r \in T_{\Sigma\$}^R$ with placeholders as well as source regions and it takes a cursor position $c \in \mathbb{N}$. It finds and yields the term at the cursor position together with a possibly empty list of proposals for it.

Definition 8 (Function propose).

$$\mathsf{propose}(t^r, cur) = \begin{cases} \langle \$s^r, \mathsf{proposals}(s) \rangle, & \text{if } t = \$s,\ cur \in r \\ \mathsf{propose}(t_i^{r_i}, cur), & \text{if } t = c(t_1^{r_1}, \ldots, t_n^{r_n}),\ cur \in r_i \\ \langle t^r, \varepsilon \rangle, & \text{otherwise} \end{cases}$$

Finally, function complete uses propose and replace to implement full code completion. To model the user's behavior, we use an oracle $\phi : (T_\Sigma)^+ \to T_\Sigma$ to select one of the proposed terms.

Definition 9 (Function complete).

```
complete($t^r$, $cur$, $\phi$) =
  let $\langle u^p, ts \rangle$ = propose($t^r$, $cur$) in
    if $ts = \varepsilon$
      then $t^r$
      else replace($t^r$, $u^p$, $\phi(ts)$)
```

Theorem 3.2 (Soundness of complete). *Given $t^r \in T_\Sigma^{R,s}$ for some sort s and arbitrary cur and ϕ, a completion* $\mathsf{complete}(t^r, cur, \phi) = w^q$ *is sound iff $w^q \in T_\Sigma^{R,s}$. If function* proposals *is sound, then function* complete *is sound for all $t^r \in T_\Sigma^{R,s}$.*

That is, a completion is sound if the resulting term is well-formed and has the same sort as the input. Specifically, for any sound proposal function that only proposes terms of the required sort, code completion is indeed sound. This holds because replace is sound and for all proposal $\langle \$s^r, ts \rangle$, the sort of terms $t \in ts$ is s.

Code completion should also be *complete*. That is, starting from some placeholder $\$s$, all terms of sort s should be constructible through code completion (and by typing lexemes of sort LEX). Complete completion enables a purely projectional user interaction where no typing is necessary except for names of variables etc.

3.4 Implementation in Spoofax

As an instantiation of our formal model for syntactic code completion, we implemented a generic completion framework in the Spoofax Language Workbench. Spoofax provides the syntax definition formalism SDF3 for specification of syntax. The distinguishing feature of SDF3 is the introduction of explicit layout specified in a template as the body of a context-free production [23]. A template production defines the usual sequence of symbols of a production and an abstract syntax tree constructor. In addition, the whitespace between the symbols is interpreted as a specification-by-example for the purpose of producing a pretty-printer. Thus, a single syntax definition serves to define a grammar, a scannerless generalized parser for that grammar, an abstract syntax tree schema (in the form of an algebraic signature), *and* a pretty-printer mapping ASTs to text.

We support explicit placeholders as part of a language by automatically extending the language's syntax definition with extra template productions. As specified in the formalization, the goal is to allow a placeholder to appear whenever it is possible to parse a non-terminal at a certain position in the program. The second box of Fig. 6 illustrates the generated template productions from the regular productions defined in the first box.

In our implementation, we instantiate the abstract function proposals as the function templates returning a list of proposals for a sort $s \in \Sigma$.

Definition 10 (Templates function)**.** *Given a signature* $\Sigma = \langle S, C \rangle$*, we define the set of concrete proposals returned by function* templates : $S \to (T_{\Sigma\$})^*$ *such that:*

$$\frac{c : s_1 \times \ldots \times s_n \to s \in \Sigma}{c(\$s_1, \ldots, \$s_n) \in \mathsf{templates}(s)}$$

We can reason about the soundness of our function templates based on the definition of the abstract function proposals.

Theorem 3.3. *Our implementation of the function* proposals *provided by the function* templates *is sound.*

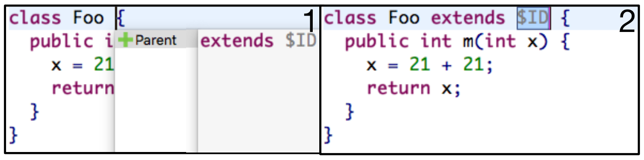
Proof. By the definition of templates, all the terms that we generate as an expansion for a placeholder of sort s have sort s. Thus, according to the soundness criterion of the abstract function proposals, templates is sound. □

```
context-free syntax  // regular syntax rules

 Statement.Assign  = [[VarRef] = [Exp];]
 Statement.If      = [if([Exp]) [Statement]
                      else [Statement]]
 Statement.While   = [while([Exp]) [Statement]]
 Statement.Block   = [{
                        [{Statement "\n"}*]
                      }]
 Statement.VarDecl = [[Type] [ID];]
```

```
context-free syntax  // derived syntax rules

 VarRef.VarRef-Plhdr        = [$VarRef]
 Exp.Exp-Plhdr              = [$Exp]
 Statement.Statement-Plhdr = [$Statement]
 Type.Type-Plhdr            = [$Type]
 ID.ID-Plhdr                = [$ID]
```

```
rules // derived rewrite rules

 rewrite-placeholder:
   Statement-Plhdr() -> Assign(VarRef-Plhdr(),
                               Exp-Plhdr())

 rewrite-placeholder:
   Statement-Plhdr() -> If(Exp-Plhdr(),
                           Statement-Plhdr(),
                           Statement-Plhdr())

 rewrite-placeholder:
   Statement-Plhdr() -> While(Exp-Plhdr(),
                              Statement-Plhdr())

 rewrite-placeholder:
   Statement-Plhdr() -> Block([])

 rewrite-placeholder:
   Statement-Plhdr() -> VarDecl(Type-Plhdr()
                                , ID-Plhdr())
```

Figure 6. Extending the grammar with placeholder productions and automatically generating rewrite rules for placeholder expansion from the syntax definition.



```
class Foo {                              1
  public i  +Parent   extends $ID
    x = 21
    return
  }
}
```

```
class Foo extends $ID {                  2
  public int m(int x) {
    x = 21 + 21;
    return x;
  }
}
```

Figure 7. Inferring a placeholder inside an optional node.

Moreover, since the templates function generates all alternatives for a sort s, it is straightforward to establish that our automatically derived completions are complete.

As specified in the function templates, in Spoofax we not only automatically derive placeholders from the SDF3 but also derive their respective proposals. Each template production with a constructor in the syntax definition defines a possible placeholder expansion. The last box in Fig. 6 shows an example of generated rewrite rules in the Stratego transformation language [22] that transform a placeholder of sort s into all its abstract expansions. As a design decision, placeholder expansions do not include placeholders for nullable symbols such as lists with zero or many elements or optional nodes, generating empty lists or optionals by default and expanding them by placeholder inference as we will present in Section 4.

Spoofax constructs source regions as attachments of terms when parsing a program and imploding the parse tree. We use this information to navigate to a placeholder in the program, as specified by the function replace. We produce concrete proposals by pretty-printing the abstract expansions collected from the rewrite rules using the generated pretty-printer from SDF3. Completing the program replaces the placeholder text by the pretty-printed text of its selected expansion. Thus, completing a program preserves its structure except for the placeholder being expanded.

4. Code Expansion by Placeholder Inference

In this section, we investigate how to use placeholders to propose expansions of complete programs. A complete program contains no placeholders thus, the method described in the previous section fails to generate any proposals. However, we want to use code completion to propose expansions of complete programs. In this paper, we focus on adding elements to lists and adding previously missing optional elements. For example, class `Main` in Fig. 7 does not define the optional extends clause. An invocation of code completion should propose defining the optional element. To this end, we introduce a method for expanding complete programs by inferring and inserting placeholders.

4.1 Placeholder Inference for Optionals

We first investigate placeholder inference for optionals, which we represent according to the following definition.

Definition 11 (Optional terms)**.** *Given a sort* s*,* $Opt(s)$ *is the sort of optional terms. For each* s*, constructor* $Some_s : s \to$

$Opt(s)$ indicates the presence of a term of sort s whereas constructor $None_s : Opt(s)$ indicates its absence.

Placeholder inference for optionals is similar to completion for explicit placeholders, where term `None`$_s$`()` plays the role of placeholder $\$s$. Thus, in a first approximation we could extend function propose to generate proposals for `None`$_s$`()` terms as well as placeholders. However, it is not as straightforward as that. Since a `None`$_s$`()` term corresponds to the empty string, it does not have a source region. Hence, in contrast to a placeholder, we cannot select a `None`$_s$`()` term using the cursor. Furthermore, there may be multiple `None`$_s$`()` terms that are candidates for expansion in the area around the cursor. For example, consider the term `VarDecl(Ident("x", None`$_{type}$`()), None`$_{exp}$`())` that represents a variable declaration for an identifier with optional type and optional initializer. When the cursor is placed after the identifier and code completion is triggered, we would like to see proposals for expanding the type as well as the initializer.

To formalize this we need the notion of adjacency of terms to the cursor. A subterm t_i of a term t is adjacent to the cursor if none of the other subterms of t capture the cursor in their source region. Since terms like `None`$_s$`()` have empty source regions, multiple subterms can be adjacent to the cursor simultaneously.

Definition 12 (Function $\mathsf{adjacent}$)**.**

$$\mathsf{adjacent}(t, cur) = \begin{cases} \{t_i^{r_i} \mid r_1 \cup \ldots \cup r_{i-1} < \{cur\} < r_{i+1} \cup \ldots \cup r_n\}, \\ \qquad \textit{if}\ t = C(t_1^{r_1}, \ldots, t_n^{r_n}) \\ \emptyset, \textit{otherwise} \end{cases}$$

Function $\mathsf{infer_o}$ infers completion proposals for *all* optionals that are adjacent to the cursor.

Definition 13 (Function $\mathsf{infer_o}$)**.**

$$\mathsf{infer_o}(t^r, cur) = \begin{cases} \{\ \langle t^r, \mathsf{proposals_o}(s)\rangle\ \}, \\ \qquad \textit{if}\ t = None_s() \\ \bigcup\{\mathsf{infer_o}(t_i^{r_i}, cur) \mid t_i^{r_i} \in \mathsf{adjacent}(t, cur)\}, \\ \qquad \textit{if}\ t = C(t_1^{r_1}, \ldots, t_n^{r_n}) \end{cases}$$

$$\mathsf{proposals_o}(s) = \{Some_s(t) \mid t \in \mathsf{proposals}(s)\}$$

In the first case, if the current term is `None`$_s$`()`, we generate replacement proposals for sort s. This corresponds to the case for placeholders $\$s$ of propose except that here we generate proposals of optional terms using $\mathsf{proposals_o}$. The second case applies inference recursively to those subterms $t_i^{r_i}$ that are adjacent to the cursor.

4.2 Placeholder Inference for Lists

We now consider placeholder inference for list terms, which we represent according to the following definition.

Definition 14 (List terms)**.** *Given a sort s, $List(s)$ is the sort of list terms. For each s, constructor $Cons_s : s \times List(s) \rightarrow List(s)$ indicates a non-empty list with head and tail and $Nil_s : List(s)$ indicates an empty list.*

Placeholder inference for lists generates proposals for inserting elements into a list. To that end, inference selects the sublist that directly *follows* the cursor (modulo layout) and generates proposals for the syntactic sort of the elements in the list. When the user selects a proposal, we insert the selected element at the cursor position. For example, in Fig. 8, the cursor is positioned between two statements in a list of statements. Code completion proposes the insertion of a new statement at the cursor position.

Function $\mathsf{infer_*}$ generates completion proposals for list elements.

Definition 15 (Function $\mathsf{infer_*}$)**.**

$$\mathsf{infer_*}(t^r, cur) = \begin{cases} \{\ \langle t^r, \mathsf{proposals_*}(s, t)\rangle\ \}, \\ \qquad \textit{if}\ t = Nil_s() \\ \{\ \langle t^r, \mathsf{proposals_*}(s, t)\rangle\ \} \cup \mathsf{infer_*}(hd^p, cur) \\ \qquad \textit{if}\ t = Cons_s(hd^p, tl^q), \{cur\} < p \\ \mathsf{infer_*}(hd^p, cur) \\ \qquad \textit{if}\ t = Cons_s(hd^p, tl^q), cur \in p \\ \mathsf{infer_*}(tl^q, cur) \\ \qquad \textit{if}\ t = Cons_s(hd^p, tl^q), \{cur\} > p \\ \bigcup\{\mathsf{infer_*}(t_i^{r_i}, cur) \mid t_i^{r_i} \in \mathsf{adjacent}(t, cur)\}, \\ \qquad \textit{if}\ t = C(t_1^{r_1}, \ldots, t_n^{r_n}), C \neq Cons \end{cases}$$

$$\mathsf{proposals_*}(s, tl) = \{Cons_s(hd, tl) \mid hd \in \mathsf{proposals}(s)\}$$

In the first case, if the current term is the empty list `Nil`$_s$`()`, we generate proposals for element sort s. This corresponds to the `None`$_s$`()` case of $\mathsf{infer_o}$ and to the case for placeholders $\$s$ of propose except that here we generate proposals for list terms using $\mathsf{proposals_*}$. Specifically, we propose to replace the empty list with a singleton list where the head element is a proposal for sort s. In the second case of $\mathsf{infer_*}$, if the current term is a `Cons` term and the cursor to the left of the head element, we propose to prepend another element. We also recursively infer completions in the head element to support proposals for nested lists. In the third case, if the cursor is within the source region of the head element, we recursively infer proposals there. Otherwise, if the cursor is to the right of the head element, we recursively infer proposals for the tail of the list. Finally, for terms that are not lists we recursively infer proposals for all subterms that are adjacent to the cursor.

We illustrate a concrete example in Fig. 8. Note that the cursor is at position 58, that is, we want to add a statement in between the two existing statements for variable declaration and assignment. We start computing proposals by applying $\mathsf{infer_*}$ to node `Method`$^{[16,92]}$. Since that node is not a list, the function reaches the fourth case, returning the union of a recursive application of $\mathsf{infer_*}$ on all adjacent children. However, node `Cons`$^{[39,74]}$ is the only adjacent subterm.

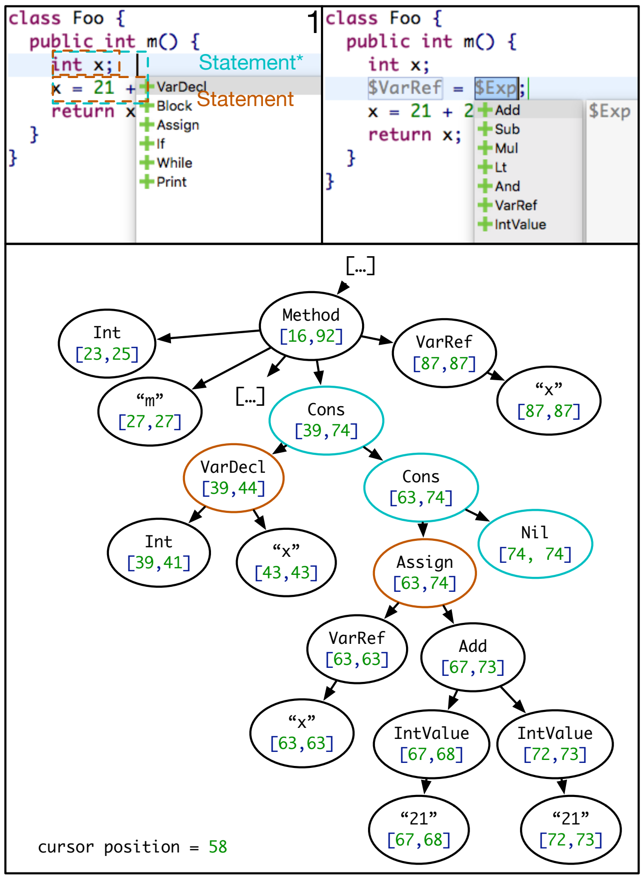

```
class Foo {
  public int m() {
    int x;
    x = 21 + 
    return x
  }
}
```
```
class Foo {
  public int m() {
    int x;
    $VarRef = $Exp;
    x = 21 + 2
    return x;
  }
}
```

Figure 8. Placeholder inference inside a list. At the bottom, excerpt of the AST with source regions before expansion.

Since that node is a non-empty list, we check whether the cursor is before or after the head of the list. Since the head element `VarDecl`$^{[39,44]}$ precedes the cursor at 58, the third case of infer$_*$ applies and recurses into the tail of the list `Cons`$^{[63,74]}$. This time, the cursor position precedes the head element `Assign`$^{[63,74]}$ and the second case of infer$_*$ applies. Thus, we propose completions that prepend a statement to `Cons`$^{[63,74]}$. The recursive call in the second case of infer$_*$ does not yield any additional completions because the head element does not contain a nested list adjacent to the cursor.

4.3 Code Expansion by Placeholder Inference

Similar to code completion, we can combine the two inference functions infer$_\circ$ and infer$_*$ together with replace. Since we do not rewrite incomplete program fragments but insert code into complete program fragments, we call this *code expansion* rather than code completion.

The following function $expand$ defines code expansion formally. To model the user's behavior, here we use two oracle functions ϕ_1 and ϕ_2. Through the first oracle ϕ_1, the user selects which one of the subterms adjacent to the cursor to expand. Through the second oracle ϕ_2, the user selects the expansion for the selected subterm.

Definition 16 (Function $expand$)**.**

```
$expand(t^r, cur, \phi_1, \phi_2) =$
  let $props = \text{infer}_*(t^r, cur) \cup \text{infer}_\circ(t^r, cur)$ in
    if $props = \emptyset$
      then $t^r$
      else let $\langle u^p, ts\rangle = \phi_1(props)$ in
        if $ts = \varepsilon$
          then $t^r$
          else replace$(t^r, u^p, \phi_2(ts))$
```

Theorem 4.1 (Soundness of $expand$)**.** *Given $t^r \in T_\Sigma^{R,s}$ for some sort s and arbitrary cur, ϕ_1, and ϕ_2, an expansion $expand(t^r, cur, \phi_1, \phi_2) = w^q$ is sound iff $w^q \in T_\Sigma^{R,s}$. If function* proposals *is sound, then function expand is sound for all $t^r \in T_\Sigma^{R,s}$.*

That is, an expansion is sound if the resulting term is well-formed and has the same sort as the input. Specifically, for any sound proposal function that only proposes terms of the required sort, code expansion is indeed sound. This holds because replace is sound and we have setup proposals$_\circ$ and proposals$_*$ such that for all proposal $\langle u, ts\rangle$, the sort of terms $t \in ts$ is identical to the sort of u.

A pragmatic concern when inserting elements into a list is the formatting of the source code. Our formal model abstracts from this issue by considering ASTs only. As illustrated in Fig. 9, our implementation in Spoofax preserves the layout of all existing code and only formats the inserted element, also inserting list separators as needed.

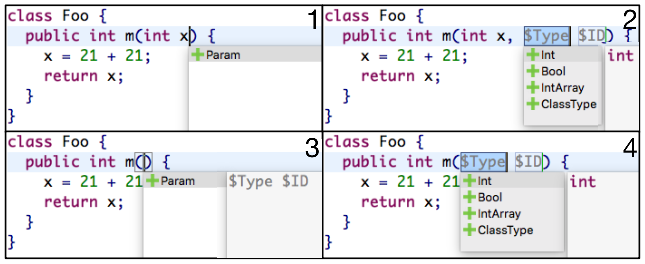

```
class Foo {
  public int m(int x) {
    x = 21 + 21;
    return x;
  }
}
```
```
class Foo {
  public int m(int x, $Type $ID) {
    x = 21 + 21;
    return x;
  }
}
```
```
class Foo {
  public int m() {
    x = 21 + 21
    return x;
  }
}
```
```
class Foo {
  public int m($Type $ID) {
    x = 21 + 21
    return x;
  }
}
```

Figure 9. Inserting an element into a list: Spoofax preserves the surrounding layout and inserts list separators as needed.

5. Code Completion for Incorrect Programs

In this section, we consider syntactic code completion for syntactically incorrect programs, i.e. for which parsing fails. Such syntactic errors occur frequently during editing. For example, when the developer writes an assignment statement, the program text remains incorrect until the developer terminates the statement with a semicolon. We want to provide code completion for incorrect programs to assist developers in completing code fragments as they write them. Specifically, we address the following scenario:

- We only consider syntax errors at the cursor position; we ignore errors elsewhere in the program text.

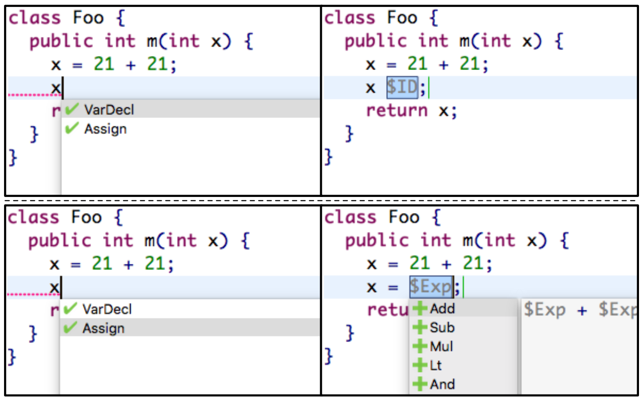
- We only consider the insertion of symbols into the program text; we ignore other forms of manipulation such as symbol removal.
- Soundness applies as before: The proposed recoveries must yield correct programs at the cursor position.
- We relax the requirement on completeness: Not all programs are necessarily constructible from the proposed recoveries.

Figure 10 illustrates the expected behaviour of the completion framework for an incorrect program using the grammar of Figure 6. Here, `x` is the first symbol of a statement and the framework should propose all statements that can start with symbol `x`. As shown in the top-right and bottom-right boxes, upon selection of a proposal, the framework inserts the missing symbols to make the program *syntactically* correct. We insert placeholders for subterms, thus allowing the user to subsequently complete the program as described in Section 3. In the remainder of this section, we present our solution for computing proposals based on the insertion of missing symbols. Placeholders play a crucial role for our solution as we will discuss in Section 5.2.

5.1 Constructing Proposals by Inserting Symbols

To construct the list of proposals, we compute all possible ways to recover a correct program by inserting symbols at the cursor position. To perform symbol insertions, we use an error-recovering technique based on permissive grammars and insertion productions [3].

Permissive grammars are grammars that can parse a more relaxed version of the input by either skipping individual symbols or simulating the insertion of missing symbols. Here, we only consider the insertion of missing symbols as an alternative to fix the error. In addition to regular productions, a permissive grammar for completion contains *insertion productions* that denote which symbols may be inserted. For example, Figure 11 shows the insertion productions for our imperative language from Figure 6. An insertion production recognizes the *empty string* — the right-hand side of the production is empty. Thus, if a regular production expects some symbol, which is not present in the text, the insertion production can parse the empty string to pretend it is there anyway. We automatically generate such insertion productions for each lexeme and placeholder of the grammar.



```
class Foo {
  public int m(int x) {
    x = 21 + 21;
    x
    r
  }
}
```

```
class Foo {
  public int m(int x) {
    x = 21 + 21;
    x $ID;
    return x;
  }
}
```

```
class Foo {
  public int m(int x) {
    x = 21 + 21;
    x
    r
  }
}
```

```
class Foo {
  public int m(int x) {
    x = 21 + 21;
    x = $Exp;
    retu
  }
}
```

Figure 10. Fixing syntax errors by code completion.

```
// derived insertion rules for placeholders
 context-free syntax

  VarRef.VarRef-Plhdr       = {symbol-insertion}
  Exp.Exp-Plhdr             = {symbol-insertion}
  Statement.Statement-Plhdr = {symbol-insertion}
  Type.Type-Plhdr           = {symbol-insertion}
  ID.ID-Plhdr               = {symbol-insertion}

// derived insertion rules for literals
 lexical syntax

  "="      = {symbol-insertion}
  "if"     = {symbol-insertion}
  "else"   = {symbol-insertion}
  "while"  = {symbol-insertion}
  "("      = {symbol-insertion}
  ")"      = {symbol-insertion}
  "{"      = {symbol-insertion}
  "}"      = {symbol-insertion}
  ";"      = {symbol-insertion}
```

Figure 11. Extending the grammar with insertion rules.

To compute the list of proposals, we use generalized parsing [18, 20, 21] on the permissive grammar. Generalized parsing supports ambiguous inputs and constructs a parse forest with one AST for each possible parse result. Thus, if alternative insertions lead to a correct parse result, we retrieve all alternatives from the generalized parser. Generalized parsers typically compact the parse forest, using ambiguity nodes `amb` to denote alternative subtrees.

Figure 12 shows the parse forest we retrieve for parsing the program from Figure 10 using the permissive grammar. The parser found two alternatives for completing the program. First, we can interpret the lexeme `x` as a class type, insert an `ID` placeholder, and a semicolon lexeme. Or, we can interpret `x` as a variable reference, insert an equality lexeme, an `Exp` placeholder, and a semicolon lexeme. In Figure 12, we mark

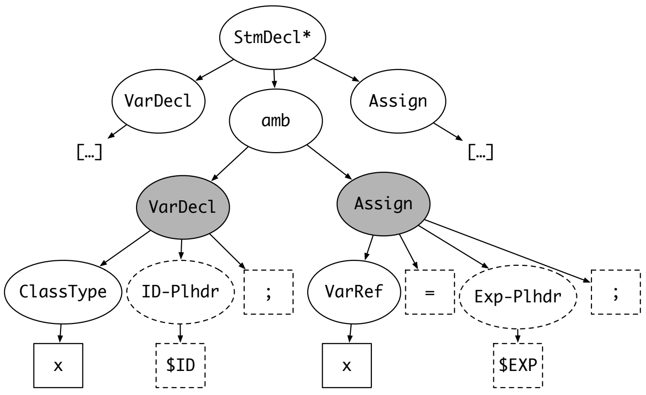


Figure 12. Recovered AST with inserted nodes (dashed line) and proposals (shaded fill).

the inserted symbols using dashed shapes and we use shaded nodes to mark nodes that become proposals. To avoid an excessive search for possible recoveries, we limit the search space using placeholders.

5.2 Limiting the Search Space for Recoveries

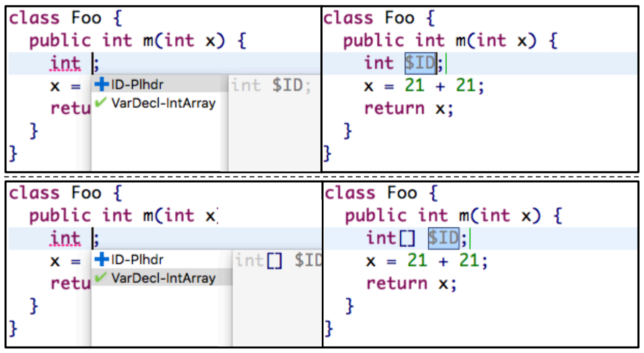
Insertion productions indicate to the parser to insert missing symbols. However, arbitrarily applying insertion productions would lead to non-termination. In our example, we could keep inserting symbols to add more statements to the list or even construct additional classes. This happens because insertion productions produce the empty string. As a result, the parser does not consume any input when applying insertion productions, leading to an infinite number of possible parses. Hence, we need to restrict the application of insertion productions to guarantee termination.

First, recall that we are merely interested in code completion, rather than error-recovery parsing in general. Thus, we can restrict insertions to the cursor position modulo layout. Conversely, we prohibit the parser from applying insertion productions elsewhere in the program.

Second, we assume that part of a proposal is already in the input. Therefore, fixing the error preserves the existing fragments of a proposal and only adds the missing symbols necessary to finish a structure. From this restriction, we disallow the application of regular productions on only recovered nodes. Note that error recovery can recover either placeholders or literal strings from the program. Recovering placeholders is essential to limit the search space as we do not need to recursively recover complex subterms that may be constructed by placeholder expansion later.

Third, we define our recovery approach as greedy, assuming that proposals contain as many symbols as possible from the input. A proposal can also include multiple nodes implying that the AST of the program contains an erroneous branch. Thus, we construct a single proposal as the smallest subtree containing all proposal nodes and we construct multiple proposals by flattening ambiguities containing multiple proposal nodes. By doing that, we guarantee that completing the program chooses only one alternative of the ambiguity and proposals only fix a single node (or branch). Most importantly, we guarantee that our strategy is sound as selecting a proposal does not introduce errors, but introduces a fix instead. The list of proposals is partially complete, i.e., we produce all fixes that include the elements that are already part of the input.

Figure 13 shows examples of the third restriction. In the top program, we do not create two proposals, for example, using *int* as the prefix of a variable declaration, and semicolon as a suffix for an assignment. Instead, we use both symbols as part of only one proposal for a variable declaration. Moreover, at the bottom, we show an example of a proposal that *fixes* more than a single node of the program. In this case, we change the inner node for the type of the variable declaration to array of integers, and add the missing placeholder for the identifier to complete the variable declaration itself.



```
class Foo {
  public int m(int x) {
    int ;
    x =
    retu
  }
}
```

```
class Foo {
  public int m(int x) {
    int $ID;
    x = 21 + 21;
    return x;
  }
}
```

```
class Foo {
  public int m(int x
    int ;
    x =
    retu
  }
}
```

```
class Foo {
  public int m(int x) {
    int[] $ID;
    x = 21 + 21;
    return x;
  }
}
```

Figure 13. At the top the proposal inserts the infix of an incomplete variable declaration (in this case either a single placeholder `$ID`). At the bottom, an example of a nested proposal, where it is necessary to fix multiple nodes in the AST to recover from the error (turn the Int type into IntArray and insert `$ID`).

5.3 Implementation in Spoofax

To implement the syntactic code completion for incorrect programs in Spoofax, we use the scannerless generalized-LR parsing algorithm (SGLR). The complete algorithm for SGLR is described in [20, 21] but as we are only interested in restricting the application of grammar rules to construct proposals, we only modified the reducer method of SGLR, applying the restrictions on the search space of possible recoveries we described before.

SGLR is a generalized shift-reduce parser that handles multiple stacks in parallel. Each conflict action in the parse table generates a new stack so that parsing can continue with that action. Given a parse table for some grammar and a string, the parser returns a parse forest containing all possible alternatives to parse the string according to the grammar described in the table. This makes SGLR a perfect match for collecting all possible recoveries as ambiguities in the resulting parse tree.

To produce the list of proposals, we do a traversal on the resulting parse tree, collecting all proposals as described before and pretty-printing them to present to the user. A selected proposal only adds the missing fragments necessary to fix the program. SGLR deals with other errors in the program using its regular error recovering strategy based on permissive grammars [3].

With the approach described in this section, the completion framework can handle incorrect programs since the parser is able to construct a list of proposals by inserting missing symbols, fixing an error at the cursor position. From there, the framework provides syntactic code completion following the approach we described before, either by expanding explicit placeholders or by placeholder inference as illustrated in Figure 2.

Our solution also preserves the generic aspect of the completion framework, as we derive insertion productions from the syntax definition. If the error at the cursor position does not follow the assumptions we made previously, error recovery does not produce any proposal. Moreover, syntax errors that occur in other locations and do not influence code completion are just preserved since insertion productions to create proposals are only applied at the cursor position.

6. Evaluation

We have applied our approach to generate syntactic code completion for Pascal, a subset of Java, and IceDust, a domain-specific language for data modelling [6]. We automatically generate placeholder transformations and construct the proposals with the pretty-printer generated from the syntax definition. Recovered proposals are constructed by our adapted version of SGLR.

We observed that the way production rules are organized in the syntax definition directly affects the number of placeholders and proposals for each placeholder. Moreover, when considering placeholder inference, inferring a placeholder when multiple optionals and empty lists are adjacent to the cursor makes the list of proposals even larger. Ideally, it should not be necessary to massage the grammar to produce better proposals. However, in the current implementation the grammar structure can affect the generated proposals.

In general, the completion framework produced acceptable proposals for all languages we evaluated. Deriving syntactic code completion from the syntax definition allowed us to implement the completion service for each of these languages without additional effort.

7. Related Work

We have implemented a generic content completion framework that is able to derive sound and complete syntactic code completion from language definitions. We adopt placeholders to represent incomplete structures for a program in a textual editor, similar to structural editors. For programs that still contain syntax errors due to incomplete structures we construct the list of proposals by error recovery. We compare our approach to projectional editors in the literature, textual language workbenches and discuss syntactic error recovery.

Syntactic Completion in Textual Language Workbenches Textual language workbenches such as Spoofax [7] and Xtext [5] derive syntactic completions from the syntax definition. However, these language workbenches currently do not have a representation of incomplete programs. In the case of Xtext, proposals involve only the following token that can appear in the input. Since non-terminal symbols can reference each other in the syntax definition, proposals may also involve predefined names or types. To extend the automatically generated completions, the language engineer can customize code templates in automatically generated Java methods.

As for the old implementation of syntactic code completion in Spoofax, a descriptor language for editor services contains the specification of completion templates, defining expansions given the context of a non-terminal symbol from the grammar. Placeholders inside proposals contain default strings, and the possibility to directly navigate to them is lost when leaving the completion mode. Furthermore, completing the program might lead to syntax errors, as the framework calculates proposals based on whether it is possible to parse a non-terminal symbol at the cursor position, possibly inserting incomplete structures.

Projectional Editing with Placeholders Placeholders allow for directly manipulating the AST of a program, a characteristic of projectional/structural editors. The Generic Syntax-directed Editor (GSE) [11] is part of the ASF+SDF Meta-environment [8] and generates an interactive editor that is hybrid, i.e., both textual and projectional from the language specification extended with placeholders.

In GSE, the editor uses the cursor position to determine the smallest node in the AST being edited. Only the content inside the focus is actually parsed, with the guarantee that the remainder of the input is syntactically correct. Whenever the focus is in a placeholder, the editor can expand the node following the grammar rules for the placeholder. However, one of the consequences for supporting hybrid editing is that GSE stores both the textual and abstract representation of the program in memory, creating a two-way mapping between them. Our approach is only based on textual editors, and we rely on source positions mapped as attachments to nodes in the AST, constructing them whenever parsing the program.

Another issue is that GSE does not support error recovery, so a focus is either syntactically correct or it is not. Thus, to properly provide code completion the user needs to first manually fix syntax errors. In addition, we only provide a single editor operation (control + space) to invoke the completion framework, whereas GSE uses the focus to determine the completions for a placeholder. Furthermore, our approach supports free textual editing, without any need for substring parsing to keep track of focuses.

Language workbenches such as CENTAUR [2], MPS [24] and mbeddr [25] generate projectional editors from language specifications. In such editors, the user edits the program by manipulating the AST directly instead of editing pure text. Proposals are automatically derived from the projections defined by the language engineer, making the completion service sound by definition. Code completion also alleviates the problems when writing programs in projectional editors, since the normal editing behaviour does not resemble classical text editing [26].

The Synthesizer Generator [15] has a representation for unexpanded terms as *completing operators*. Completing operators act as placeholders and can be structurally edited by specifying rules as commands that insert code templates. In our implementation, code templates are defined by the grammar, whereas the definition of code templates is disjoint from parsing rules. Moreover, since the language definition is based on attribute grammars [9], template proposals can also use semantic information by evaluating attributes derived from syntactic sorts of completion operators. Proxima [17] also uses placeholders as *holes* that can appear inside textual or structural presentation elements. As our solution is implemented in a textual editor, we only support placeholders as part of textual elements of the program.

Error Recovery To handle incorrect programs, our approach recovers missing symbols to construct a valid AST from which the framework creates an expansion proposal. There are different approaches to support error recovery from syntax errors [4]. The current approach implemented in the Spoofax language workbench is based on island grammars [12, 13] and recovery rules providing error recovery for a generalized parsing algorithm [3].

In the generalized scenario of SGLR, it is necessary to investigate multiple branches, and the detection of syntax errors occurs at the point where the last branch failed. This point might not even be local to the actual root cause of the error, making error reporting more difficult. Scannerless parsing also contributes to make the recovery strategy more complex. Common strategies based on token insertion or deletion to fix the error are ineffective when considering single characters.

Our approach benefits from the assumptions that we know the error location and that only missing elements contribute to the error. Therefore, it is not necessary to skip parts of the input nor backtrack to find the actual error location. Furthermore, we benefit from the fact that SGLR constructs a parse forest as result. Thus we return all possible fixes, reporting them to the user as proposals.

8. Future Work

Character-based Completions Our current recovery strategy does not recover from incomplete words, producing only insertion symbols for literals and placeholders. The completion framework could handle partial keywords by manipulating the input to reconstruct keywords and use them as a starting point for recovering a proposal.

Inlining and Ordering Proposals The current approach might generate too many proposals depending on the productions in the grammar. For this issue, ordering suggestions might improve the final user experience [16]. Inlining proposals can also improve the framework for cases when it is necessary many placeholder replacements to create a final code template.

Semantic Completions The completions in this paper are restricted to *syntactic* completions. Mainstream IDEs typically have spent more effort in the support for *semantic* completions, i.e. proposing names (e.g. of variables or methods) that are valid to use in the cursor context. In future work, we plan to explore providing *generic* support for such semantic completions based on our work on name [10, 14] and type resolution [19]. Using the results of name and type resolution, we can propose completions for lexical placeholders to insert declared names. Moreover, semantic information can also be used to filter the list of syntactic proposals such that sound content completion guarantees the absence of syntactic *and* semantic errors.

9. Conclusion

Code completion avoids misspellings and enables language exploration. However, the support for syntactic completion is not fully implemented by most IDEs. The completion implementation is ad-hoc, unsound and incomplete.

The separation of programs into different states allowed us to provide code completion with a "divide and conquer" strategy. For correct programs, we implement code completion by expanding placeholders that can appear implicitly or explicitly in programs. For incorrect programs, we used the nature of errors in the completion scenario to propose an adapted error recovery strategy to construct the list of proposals.

Finally, our formalization allowed us to reason about soundness and completeness of code completion. We implemented the framework by modifying the scannerless GLR parsing algorithm and by generating placeholders and its expansions from syntax definitions in the Spoofax Language Workbench. Our framework addresses the requirements derived from the analysis of state-of-the-art implementations of syntactic code completion (Section 2). This work opens up a path to rich editing services based on the (context-sensitive) structure of a program in purely textual IDEs.

Acknowledgements

The work presented in this paper was partially funded by CAPES, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil and by the NWO VICI *Language Designer's Workbench* project (639.023.206). We would like to thank Daco Harkes, Daniël Pelsmaeker, Danny Groenewegen, Gabriël Konat and Sven Keidel for testing the implementation, and the anonymous reviewers for their comments on the paper.

References

[1] S. Amann, S. Proksch, S. Nadi, and M. Mezini. A study of visual studio usage in practice. In *SANER*, 2016.

[2] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 14–24, New York, USA, 1988. ACM.

[3] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems*, 34(4):15, 2012.

[4] P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software: Practice and Experience*, 25(6):657–679, 1995.

[5] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[6] D. Harkes, D. M. Groenewegen, and E. Visser. IceDust: Incremental and eventual computation of derived values in persistent object graphs. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[7] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[8] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, Apr. 1993.

[9] D. E. Knuth. Semantics of context-free languages. In *In Mathematical Systems Theory*, pages 127–145, 1968.

[10] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012.

[11] J. Koorn. GSE: a generic text and structure editor. In *University of Amsterdam*, pages 168–177, 1992.

[12] L. Moonen. Generating robust parsers using island grammars. In *WCRE*, page 13, 2001.

[13] L. Moonen. Lightweight impact analysis using island grammars. In *10th International Workshop on Program Comprehension (IWPC 2002), 27-29 June 2002, Paris, France*, pages 219–228. IEEE Computer Society, 2002.

[14] P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015.

[15] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, Apr. 1984.

[16] R. Robbes and M. Lanza. How program history can improve code completion. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L Aquila, Italy*, pages 317–326. IEEE, 2008.

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

[18] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[19] H. van Antwerpen, P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In M. Erwig and T. Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016.

[20] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[21] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[22] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003.

[23] T. Vollebregt, L. C. L. Kats, and E. Visser. Declarative specification of template-based textual editors. In A. Sloane and S. Andova, editors, *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, page 8. ACM, 2012.

[24] M. Völter. Language and IDE modularization and composition with MPS. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, volume 7680 of *Lecture Notes in Computer Science*, pages 383–430. Springer, 2011.

[25] M. Völter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Journal of Automated Software Engineering*, 2013.

[26] M. Völter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 41–61. Springer, 2014.