UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Dmitri Gabbasov

# Adding Nim programming language support to IntelliJ IDEA

Master's Thesis (30 ECTS)

Supervisor: Vesal Vojdani

Tartu 2016

## Adding Nim programming language support to IntelliJ IDEA

**Abstract:**
Nim is a programming language that takes inspiration from such languages as C, Python and Lisp. It is mainly a low-level systems programming language, albeit its flexible syntax and built-in support for garbage collection have facilitated its use in web application development among other areas. IntelliJ IDEA is a popular integrated development environment (IDE) created by JetBrains and used to develop in languages such as Java, PHP, Python, C++, Ruby and others. The aim of this thesis is to create a plugin for IntelliJ IDEA that would allow writing applications in Nim. The main focus is on providing symbol navigation and code completion.

**Keywords:** Nim, Nim IDE, IntelliJ IDEA, IntelliJ plugin

**CERCS:** P170

## Nim keele toe lisamine IntelliJ IDEA arenduskeskkonnale

**Lühikokkuvõte:**
Nim on programmeerimiskeel, mis on inspireeritud selliste keelte poolt nagu C, Python ja Lisp. Tegu on eelkõige madala taseme süsteemide programmeerimiskeelega, kuid selle paindlik süntaks ja automaatse mäluhalduse tugi võimaldavad ka näiteks veebirakenduste kirjutamist. IntelliJ IDEA on firma JetBrains poolt loodud arenduskeskkond (IDE), mis võimaldab arendada rakendusi erinevates keeltes (nt. Java, PHP, Python, C++, Ruby jt). Selle töö eesmärgiks on luua IntelliJ arenduskeskkonnale laiendus, mis võimaldaks Nim keeles rakenduste kirjutamist. Põhiline eesmärk on realiseerida sümbolite järgi navigeerimine ja automaatne sümbolite sisestamine.

**Võtmesõnad:** Nim, Nim IDE, IntelliJ IDEA, IntelliJ laiendus

**CERCS:** P170

# Contents

**Introduction** **4**

**1 The Nim programming language** **5**

1.1 Variables and constants . . . . . 5

1.2 Procedures . . . . . 6

1.3 Operators . . . . . 8

1.4 Type definitions . . . . . 8

1.5 Generics . . . . . 9

1.6 Templates . . . . . 9

1.7 Modules and visibility . . . . . 10

**2 IntelliJ IDEA plugin** **11**

2.1 Lexer . . . . . 11

2.2 Parser . . . . . 12

2.2.1 BNF grammar . . . . . 12

2.2.2 AST and PSI tree . . . . . 15

2.3 Reference resolution . . . . . 16

2.4 PSI stubs and stub indices . . . . . 20

2.5 Type deduction . . . . . 23

2.6 Procedure overload resolution . . . . . 25

2.7 Code completion . . . . . 26

**3 Conclusion** **29**

# Introduction

Nim [1] is a statically typed, imperative programming language that aims to achieve high runtime efficiency while being expressive and elegant in its syntax. Nim's history can be traced back to as far as 2008, which is when Andreas Rumpf – the author of Nim – made the first commit to the Nim compiler repository [2]. The language might not have gathered much mainstream popularity, but it has seen a significant amount of work and comes with a comprehensive standard library.

IntelliJ IDEA [3] is an integrated development environment (IDE) made by the Czech company JetBrains. Even though it is primarily a Java IDE, JetBrains has reused the platform that it is built on to create IDEs for other languages. There are separate products for PHP (PhpStorm), Python (PyCharm), Ruby (RubyMine), HTML/JavaScript (WebStorm), Objective-C (AppCode) and C++ (CLion). The platform itself, however, is extensible and the main IDE – IntelliJ IDEA – can support nearly any language through the use of *custom language plugins*.

As far as availability of IDEs for Nim goes, there is an advanced text editor called Aporia [4] that is developed by one of the people behind the Nim compiler. There are also plugins for several popular text editors (Vim, Emacs, Sublime Text, TextMate and others), however, many of them provide only basic syntax highlighting. Aporia, as well as some of the text editor plugins, use a tool called *nimsuggest* [5] to achieve symbol navigation and code completion. Nimsuggest is a command-line utility that can be invoked with the path to the source files and the desired operation (find definition, find usages etc) as arguments. The tool will then analyze the given files, and output an answer. Even though nimsuggest is developed alongside the Nim compiler (they share a large amount of source code) and can provide a decent amount of information, it still cannot be used for all the features that an IDE is expected to have. For instance, syntax highlighting in IntelliJ can be implemented in such a way that it takes into account which identifiers refer to types, and gives them a different appearance. For nimsuggest based editors, that would mean executing the command line utility for every word inside a file, which is hardly an elegant solution.

The aim of this thesis is to create a plugin for IntelliJ IDEA that would facilitate writing Nim applications. The goal is for the plugin to provide an experience similar to that of writing Java applications in the same IDE. The first chapter describes the Nim programming language, giving a brief overview of its syntax. In the second chapter we describe the implementation details of the plugin. We conclude by outlining some further work that remains to be done. The appendix at the end of the thesis contains instructions on how to obtain the source code of the plugin.

# 1 The Nim programming language

According to Nim's creator, Andreas Rumpf, the language was mostly inspired by C, Python and Lisp. More precisely, the goal was for the language to be "as fast as C, as expressive as Python and as extensible as Lisp" [6]. Nim is an imperative language and is statically typed. It is important to distinguish between the language itself, and things that are related to it, but do not necessarily define it (most notably a compiler and a standard library). In this thesis we only consider the official (and most likely the only) Nim compiler, as well as the official standard library that comes with it. The official Nim compiler translates Nim source to C code (or, optionally, to C++) and then uses an existing C compiler to produce binaries.

Nim is a rich language with a lot of features. Here we will glance over some of the concepts that the reader should get familiar with before reading the rest of this thesis. Some things are not going to be covered, mostly due to the fact that they are not relevant from the viewpoint of static analysis (like the fact that Nim supports automatic memory management using a garbage collector), and so we recommend checking the Nim language manual [7] for more details.

A Nim file consists of a series of statements. Scoping is based on indentation (like in Python). There is no need for a main function, executable statements can appear at the top level. Nim also has local variable type inference, meaning that declaring types of local variables is optional.

```nim
# This is a comment.

var name = "John"

proc greet(name: string) =
  echo("Hello, ", name, "!")

greet(name)
```

The above program will output "Hello, John!" to the standard output.

Let us take a closer look at the various kinds of statements.

## 1.1 Variables and constants

One of the simplest declaration statements in Nim are the constant and variable declarations. There are a total of three of those: one for declaring constants, one for declaring immutable variables and one for declaring normal (mutable) variables.

```nim
const
```

```nim
  PI = 3.14159265
  E = 2.7

let
  alpha = "foo"

var
  bravo, charlie: int
  delta: string = "bar"
```

A single `const`, `let` or `var` section can declare multiple constants / variables. The first section (`const`) declares constants `PI` and `E`. Constants are evaluated at compile time, meaning they must be initialized with constant expressions. The second statement (`let`) declares a single *immutable* variable `alpha`. Immutable variables cannot be reassigned after they have been defined. Also note that the variable does not declare its type; the compiler can infer it automatically (in this case it's `string`). The third section (`var`) declares mutable variables `bravo`, `charlie` and `delta`. Two of them don't have an explicit initializer, meaning that they will be initialized to the default value (0 for integers). When declaring only a single constant or variable, one can keep the whole thing on a single line:

```nim
const PI = 3.14
```

Let us note here that Nim is very capable of evaluating arbitrary expressions at compile time, e.g. the following is a valid constant declaration:

```nim
proc repeat(c: char, n: int): string =
  var s = ""
  for i in 1..n:
    s.add(c)
  return s

const foo = repeat('A', 3)

echo(foo)
```

The compiler will successfully evaluate the value of `foo` (which is `"AAA"`) and embed it directly into the C source that it outputs.

## 1.2 Procedures

Functions in Nim are called procedures. The following is an example of a procedure definition.

```nim
proc count(st: string, ch: char): int =
  for c in st:
    if c == ch:
      result += 1
```

Note the use of the implicit `result` variable. Every procedure that returns something (i.e. one that has a return type other than `void`) has an implicit variable called `result`, which represents the value that will be returned from the procedure. There is no need to explicitly return it. The initial value of the `result` variable will be the default value of its type. In the example above, `result` has the type `int`, which has a default value of 0. It is also possible to use explicit `return` statements.

There is also a certain amount of flexibility in the way one writes the signature of a procedure. In the following example the parameters that have the same type have been combined, and the return type has been omitted (meaning that it is `void`).

```
proc foo(a, b, c: int; d: string) = discard
```

The `discard` statement is similar to the `pass` statement in Python – it does nothing. Parameters can also have default values, in which case one can omit the type of the parameter:

```
proc foo(a: int, b = "bar") =
  discard
foo(1, "bah")
foo(1)
```

Procedures can be overloaded. This means that it is possible to define multiple procedures with the same name, but different parameter types. When calling an overloaded procedure, the given arguments will be considered in order to pick the right overload. Details on how overloads are picked will be discussed in section 2.6. The following is an example where the `foo` procedure has been overloaded.

```
proc foo(s: string) = discard
proc foo(i: int) = discard
foo("bar")
foo(42)
```

Procedures can be invoked using the so called method call syntax. This is illustrated by the following example.

```
proc count(st: string, ch: char): int =
  for c in st:
    if c == ch:
      result += 1

let word = "banana"
let n = word.count('a')
```

Instead of passing the first argument as one would normally, the argument is placed to the left of the method call, with a dot in between. This syntax looks similar to how methods are invoked on objects in other programming languages.

Another way to call procedures is using the so called command invocation syntax. This allows one to leave out the parenthesis around the arguments, so instead of writing `echo("foo")` one can write `echo "foo"`. However, this syntax has a limitation. If the call is a statement, multiple arguments can be passed to it, if the call is an expression used in another statement, then only one argument may be passed. The following example illustrates this.

```
echo "one", "two"            # ok
let n = count "banana", 'a'  # not ok
let m = "banana".count 'a'   # ok
```

The second line is not syntactically valid, as the invocation of `count` is part of another statement (a `let` statement), and so only one argument may be passed to it.

## 1.3 Operators

Nim supports defining custom operators, which can be invoked in prefix and infix form. Operators are defined in exactly the same way as procedures, the name just has to be enclosed between accents, as shown in the following example.

```
proc `*`(st: string, n: int): string =
  result = ""
  for i in 1..n:
    result.add(st)

echo "na" * 8, " batman"  # outputs "nananananananana batman"
```

## 1.4 Type definitions

Nim comes with a variety of built-in types. There are integer types (of various sizes), a boolean type, a character type, floating point types, a string type, heterogeneous and homogeneous structured types, references, pointers, function types, enumerations and more. It is also possible to define new types; the following is an example of a few type definitions.

```
type
  TCustomer = object         # objects are heterogeneous structures
    name: string
    time_joined: uint64
    balance: int
    active: bool

  PCustomer = ref TCustomer  # a TCustomer reference

  Utf8String = string        # an alias

  ColorComponent = enum      # an enumeration
    ccRed, ccGreen, ccBlue
```

## 1.5 Generics

Nim also supports generics (type parameters). In particular, one can have type parameters for procedures and type definitions. Following example demonstrates a procedure `max`, that will return the larger of the two arguments that are passed to it.

```
proc max[T](a, b: T): T =
  if a < b:
    return b
  return a
echo max(3, 42)
echo max('a', 'x')
```

The above procedure can be called with any type for which the *less than* operator is defined. Type definitions can use type parameters as follows:

```
type
  Node[T] = object
    left, right: ref Node[T]
    data: T

let node = Node[string](data: "foo")
echo node.data
```

This defines a `Node` type, that takes a type parameter for the type of the data that it contains.

## 1.6 Templates

Nim has a concept of *templates*, which are a form of macros that operate on the AST of the program. The following is an example of a template.

```
template `!=` (a, b: untyped): untyped =
  not (a == b)

assert(5 != 6) # the compiler rewrites this to: assert(not (5 == 6))
```

The parameters of a template can have special types *untyped*, *typed* and *typedesc*. An *untyped* parameter can be passed any expression, even one that contains unbound symbols (i.e. symbols that are not defined). Following example showcases the usage of untyped template parameters.

```nim
template declareInt(x: untyped) =
  var x: int

declareInt(foo)
foo = 3
```

The invocation of the `declareInt` template will be substituted by a variable declaration, and `foo` will be the name of the variable. In contrast, *typed* parameters cannot be passed arguments that contain unresolved symbols. Finally a *typedesc* parameter must be passed an expression that evaluates to a type, like in the following example.

```nim
template declareVar(x: untyped, t: typedesc) =
  var x: t

declareVar(foo, string)
foo = "bar"
```

## 1.7 Modules and visibility

Every Nim file constitutes a module, and all symbols declared in that file belong to that module. The name of the file is the name of the module. By default, all symbols declared in a module are only accessible from within that module. In order to export symbols, they must be declared with an asterisk, as shown in the following example.

```nim
type
  ExportedType* = object
    exportedField*: string

var exportedVar*: int

const exportedConst* = 42

proc exportedProc*() = discard
```

In order to use symbols from another module, one has to import it.

```nim
import foo

exportedProc() # this procedure is defined in the 'foo' module
```

Special options can be passed to the Nim compiler to specify the directories where modules are searched for.

# 2 IntelliJ IDEA plugin

We will now describe the implementation of the plugin. Getting started with a custom language IntelliJ plugin is straightforward thanks to their documentation [8]. It provides enough information on how to implement some of the basic features. However, on several occasions, when trying to achieve something that wasn't covered in the quick start guide, we have had to read the code of the Java language implementation or sometimes just debug IntelliJ code to see how it executes and how to manipulate the flow in a way that would give us the desired result.

## 2.1 Lexer

The very first components that need to be implemented for any type of static language analysis are a lexer and a parser. Our plugin uses a library called JFlex [9] to generate the lexer from a declarative specification. The task of the lexer is to take a string of source code, and output a stream of tokens, that will then be used by the parser. Our lexer is described in a file called `Nim.flex`. Here is a small excerpt of a lexer definition:

```
WHITE_SPACE_CHAR = [\ \n\r\t\f]
LINE_COMMENT     = "#" [^\r\n]*
INT_LITERAL      = [0-9]+
FLOAT_LITERAL    = [0-9]+ "." [0-9]+

<YYINITIAL> {
  {WHITE_SPACE_CHAR}+ { return WHITE_SPACE; }
  {LINE_COMMENT}      { return LINE_COMMENT; }
  {FLOAT_LITERAL}     { return FLOAT_LITERAL; }
  {INT_LITERAL}       { return INT_LITERAL; }
  ";"                 { return T_SEMICOLON; }
  ":"                 { return T_COLON; }
  ","                 { return T_COMMA; }
  "if"                { return T_IF; }
  "import"            { return T_IMPORT; }
}
```

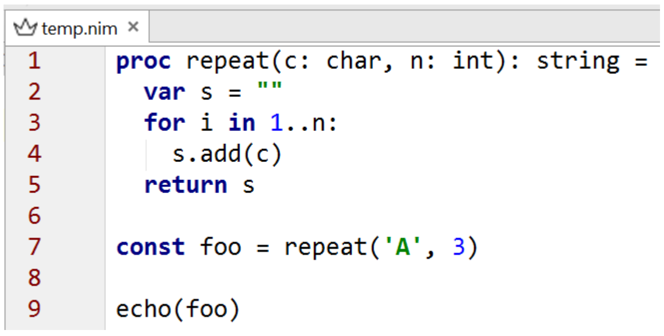
The first four lines define a few regular expressions to use later. The block that follows consists of a number of rules, each rule specifies a regular expression which, when matched, will make the lexer return a token of the specified kind. For example, a single semicolon character is converted into a token of type `T_SEMICOLON`, whereas consecutive digits like `123` will be converted into a single token of type `INT_LITERAL`. The `<YYINITIAL>` part at the start of the block means that the rules inside the block will only be matched if the current state of the lexer is `YYINITIAL` (this is the default state for a lexer). The state can be changed by the rules, meaning that it is possible to have a stateful lexer.

It is important to note, that our lexer is not as strict as the lexer that's used by the Nim compiler. For instance, the following is the regular expression that we use to match string literals:

```
STRING_LITERAL = \" (\\. | [^\\\"\r\n])* \"?
```

Not only does it make the ending quote optional, but also accepts arbitrary escape sequences within a string, something that the compiler would treat as a syntax error. For example, the string literal `"foo \d bar"` is successfully recognized as such by our lexer, but rejected by the compiler with an error message saying "invalid character constant". This leniency is intended, as it allows us to still parse a file and collect information about its structure, even when there are trivial syntax errors in the code.

The lexer specification is processed by JFlex as part of the build process and a class called `NimLexer` is generated. Using this class it is then possible to turn a string into a stream of tokens. Without even writing a parser, we can already utilize the lexer to implement basic syntax highlighting. All that's required is creating a class that extends `SyntaxHighlighterBase` and overriding two methods – one that returns an instance of a lexer, and one that maps a token type to a set of text attributes that describe the color and font style for rendering tokens of that type. Our syntax highlighter is realized by the class `NimSyntaxHighlighter`. Figure 1 depicts a Nim file open in IntelliJ with syntax highlighting in effect.



```
proc repeat(c: char, n: int): string =
  var s = ""
  for i in 1..n:
    s.add(c)
  return s

const foo = repeat('A', 3)

echo(foo)
```

Figure 1: A Nim file with syntax highlighting in effect.

## 2.2 Parser

### 2.2.1 BNF grammar

Like the lexer, the parser is also generated from a declarative specification. More precisely, it is generated from a BNF grammar. The task of the parser is to take a token stream and build an abstract syntax tree (AST). We use a library called Grammar-Kit [10] to generate the parser. Grammar-Kit is developed specifically for writing custom language plugins for IntelliJ, and so the parser that it generates integrates very well into the IntelliJ architecture.

The BNF grammar is defined in the file `Nim.bnf`. It has been ported from the actual parser code that's used by the Nim compiler [11]. A large part of Nim's syntax can be represented with BNF rules, but there are also some things that need to be parsed manually. For this purpose, Grammar-Kit allows one to define so called *external rules*, which are basically production rules that are implemented in normal Java code. We use external rules to be able to specify a grammar that takes into account the indentation of tokens. We also use external rules for a couple other things, like parsing binary operator expressions.

In order to be able to take indentation into account, the parser keeps track of the *current* indentation level. At certain points the indentation level of a specific token is matched against the current level in order to decide whether that token can be matched or not. At certain other points the current level is changed according to the level of a specific token. The indentation level of a token is the number of spaces between it and the preceding newline character. A token that follows some other token, without any newlines in between, does not have a valid indentation level. A token at the very beginning of a line (at column 0) has an indentation level of 0 (which is a valid indentation level). It is important to understand that the indentation level is a property of a token, i.e. there are no separate tokens for representing an increase and a decrease in indentation. Also, we currently compute the indentation level of every token in the parser (by analyzing the whitespace between tokens), but it could in fact be done in the lexer, so that each token would already have it as a precomputed property.

Here is an example of BNF grammar that declares two rules used for parsing constant definitions:

```
ConstSect ::= T_CONST (&INDNONE ConstDef
                     | <<indented ConstDef (&INDEQ ConstDef)*>>)

ConstDef ::= Identifier OPERATOR? Pragma? (T_COLON &OPTIND TypeDesc)?
             T_EQ &OPTIND expr
```

The tokens `T_CONST`, `OPERATOR`, `T_COLON` and `T_EQ` correspond to those produced by the lexer and represent the word "const", an operator, a colon and an equals sign respectively. `Identifier` is a rule that matches a valid identifier, `Pragma` matches a pragma, `TypeDesc` matches a type (which can in fact be any expression) and `expr` matches an expression. The rule `INDNONE` is an external rule that matches iff the next token doesn't have a valid indentation level, this means that the next token has to be on the same line as the previous token. The ampersand (&) in front of `INDNONE` means that the rule is matched without advancing the position in the token stream, but since `INDNONE` doesn't consume tokens anyway, then the ampersand is not really necessary (it is kept merely for semantic accuracy – to signify that `INDNONE` will only check some preconditions and not consume anything). The `<<indented ...>>` rule is also an external rule, here is its definition in Java code:

```java
public static boolean indented(PsiBuilder builder, int level, Parser parser) {
  ParserState state = getParserState(builder);
  int tokIndent = state.getTokenIndent();
  if (tokIndent > state.currentIndent) {
    int prevIndent = state.currentIndent;
    state.currentIndent = tokIndent;
    boolean result = parser.parse(builder, level + 1);
    state.currentIndent = prevIndent;
    return result;
  }
  return false;
}
```

As one can see, the rule first makes sure that the next token has an indentation level greater than the current one (if that's not the case then the rule doesn't match), after which it changes the current level to the new one and invokes the parser that was passed to the rule (in the case of our example, the passed parser is one that matches `ConstDef (&INDEQ ConstDef)∗`). In the end, the indentation level is reset to its previous value.

The `INDEQ` rule matches if the indentation level of the next token is equal to the current indentation level. This means that `ConstDef`s will be consumed as long as they have the right indentation level. For example, in the following code, only two constant definitions are matched:

```
const
  PI = 3.14
  E = 2.7
FOO = "bar"
```

The fourth line is not part of the `const` section (it is a separate assignment expression).

Finally, the `OPTIND` rule matches iff the next token either doesn't have a valid indentation level or if its indentation level is greater than the current level. In other words, the token can have optional indentation.

It is also important to understand, that the indentation level only matters in select places. The following code is still valid, even though the indentation may seem to be out of place:

```
const
  PI
```

```
= 3.14
  E = 2.7
```

This is because there are no restrictions on the indentation level of the equals sign (the `T_EQ` token in the grammar).

### 2.2.2 AST and PSI tree

Feeding the BNF grammar to Grammar-Kit will generate the parser class, in our case `NimParser`. The class can process a stream of tokens and produce an abstract syntax tree. The AST, however, is a generic structure and is not tied to Nim at all. In order to have a better API for static analysis, we need to build a specialized tree structure on top of the AST. IntelliJ refers to the specialized tree as a PSI tree, which stands for *program structure interface* tree. Building the PSI tree involves creating interfaces and classes for relevant PSI tree nodes and providing the logic that instantiates the right class for a given AST node. The BNF grammar that we feed to Grammar-Kit in fact already describes the PSI structure that we want. Grammar-Kit can generate all the PSI interfaces and classes by itself. For instance, the `ConstDef` rule mentioned in the previous BNF grammar example would cause the following Java classes to be generated:

```java
public interface ConstDef extends PsiElement {
  Expression getExpression();
  Identifier getIdentifier();
  Pragma getPragma();
  TypeDesc getTypeDesc();
}
```

```java
public class ConstDefImpl extends ASTWrapperPsiElement implements ConstDef {
  public ConstDefImpl(ASTNode node) {
    super(node);
  }
  @Override
  public Expression getExpression() {
    return findChildByClass(Expression.class);
  }
  @Override
  public Identifier getIdentifier() {
    return findNotNullChildByClass(Identifier.class);
  }
  @Override
  public Pragma getPragma() {
    return findChildByClass(Pragma.class);
  }
  @Override
  public TypeDesc getTypeDesc() {
    return findChildByClass(TypeDesc.class);
  }
}
```

A generated factory method will also contain the following lines for each rule:

```java
public static PsiElement createElement(ASTNode node) {
  IElementType type = node.getElementType();
  ...
  else if (type == CONST_DEF) {
    return new ConstDefImpl(node);
  }
  ...
}
```

One can see that Grammar-Kit does a decent job of generating a lot of code for us. However, we would like to have additional methods on those PSI interfaces and classes, for instance we want the `ConstDef` interface to have a method that would return the type of the constant (int, string, user defined type etc). Even though Grammar-Kit *does* have a way of adding auxiliary methods to PSI interfaces, those ways are cumbersome, as they entail polluting the BNF grammar with verbose meta information, and all the logic will need to be put into static methods of some utility classes, that will then be called by the generated PSI classes. We would much rather add methods directly to the PSI classes. This calls for us having to write all the PSI interfaces and classes ourselves, which is in fact what we ultimately decided to do. This simply gives us more flexibility and control over the structure of those classes. Grammar-Kit still generates the parser for us (which builds the AST), we only write the PSI class factory method and the PSI interfaces and classes.

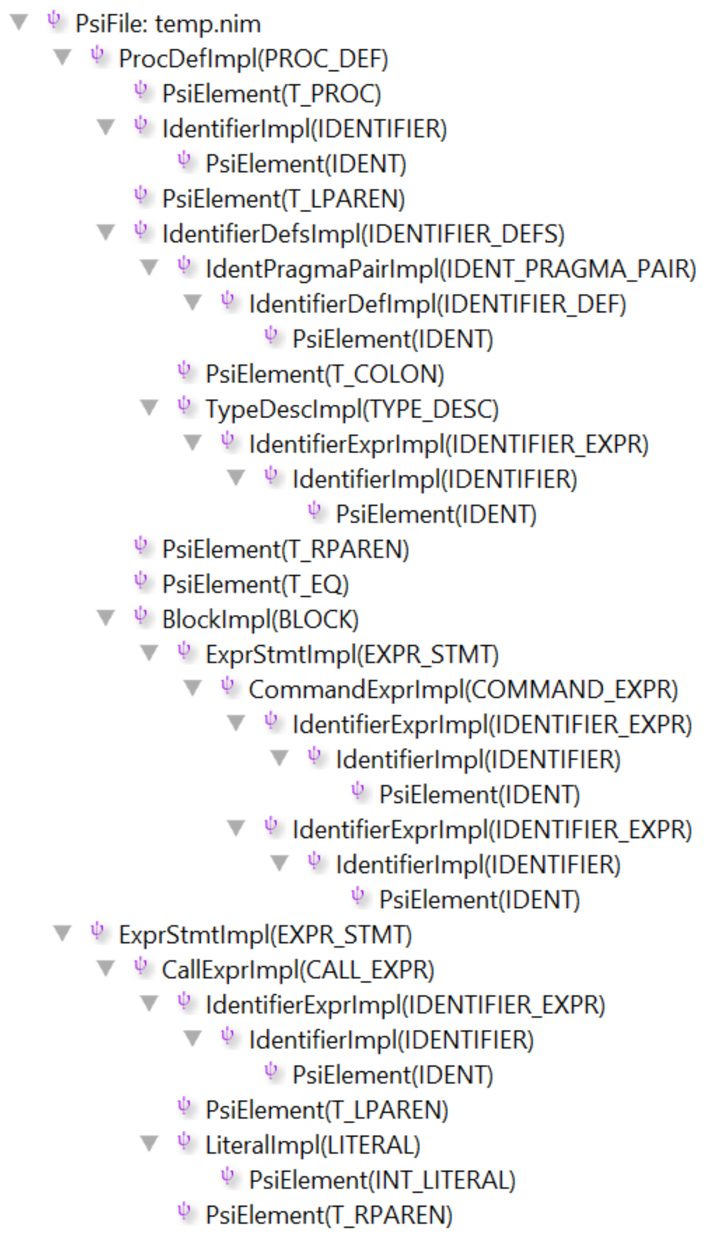
The following piece of Nim code will produce a PSI tree as shown in figure 2.

```nim
proc foo(x: int) =
  echo x
foo(32)
```

The root of the tree is a `PsiFile` node, it corresponds to a single file. It has two child nodes – `ProcDef`, which corresponds to the definition of the `foo` procedure, and `ExprStmt`, which corresponds to the statement that invokes `foo`. Each of these nodes in turn consists of various other nodes. Leaf nodes (those of type `PsiElement` in the figure) correspond to tokens produced by the lexer.

## 2.3 Reference resolution

The next step in implementing our plugin is adding support for resolving references to different symbols in the code. This is one of the major functionalities of the plugin and



```
PsiFile: temp.nim
  ProcDefImpl(PROC_DEF)
    PsiElement(T_PROC)
    IdentifierImpl(IDENTIFIER)
      PsiElement(IDENT)
    PsiElement(T_LPAREN)
    IdentifierDefsImpl(IDENTIFIER_DEFS)
      IdentPragmaPairImpl(IDENT_PRAGMA_PAIR)
        IdentifierDefImpl(IDENTIFIER_DEF)
          PsiElement(IDENT)
      PsiElement(T_COLON)
      TypeDescImpl(TYPE_DESC)
        IdentifierExprImpl(IDENTIFIER_EXPR)
          IdentifierImpl(IDENTIFIER)
            PsiElement(IDENT)
    PsiElement(T_RPAREN)
    PsiElement(T_EQ)
    BlockImpl(BLOCK)
      ExprStmtImpl(EXPR_STMT)
        CommandExprImpl(COMMAND_EXPR)
          IdentifierExprImpl(IDENTIFIER_EXPR)
            IdentifierImpl(IDENTIFIER)
              PsiElement(IDENT)
          IdentifierExprImpl(IDENTIFIER_EXPR)
            IdentifierImpl(IDENTIFIER)
              PsiElement(IDENT)
  ExprStmtImpl(EXPR_STMT)
    CallExprImpl(CALL_EXPR)
      IdentifierExprImpl(IDENTIFIER_EXPR)
        IdentifierImpl(IDENTIFIER)
          PsiElement(IDENT)
      PsiElement(T_LPAREN)
      LiteralImpl(LITERAL)
        PsiElement(INT_LITERAL)
      PsiElement(T_RPAREN)
```

Figure 2: Example of a PSI tree.

nearly every other feature depends on proper reference resolution. For instance, in the following code snippet, we want IntelliJ to know that the `foo` call on line 2 refers to the `foo` procedure defined on line 1.

```
1  proc foo(): string = return "bar"
2  echo foo()
```

Similarly we want IntelliJ to know that in the following example it is the second overload that gets called on line 4:

```
1  proc foo(i: int) = echo "got an int"
2  proc foo(s: string) = echo "got a string"
3  let x = "magic"
4  foo(x)
```

PSI elements that can possibly form a reference to a symbol need to implement the `getReference()` method of the `PsiElement` interface and return a `PsiReference`. In our case, we have an element type called `Identifier`, which almost always needs to be resolved to something. `Identifier` overrides the mentioned method. An implementation of `PsiReference` needs to implement its `resolve()` method, whose task it is to find the definition of the symbol that is being referenced. Resolution mostly involves walking up the PSI tree in search of suitable symbols as well as querying some global indices for symbols that are defined in other files. We have several implementations of `PsiReference`. There is a `ProcReference`, which only resolves to procedures (and not local variables, parameters, types etc). There is a `TypeReference`, which resolves strictly to type definitions. There is also a `MemberReference`, which resolves to object fields as well as procedures that can be called as methods in the given context. And finally there is an `IdentifierReference`, which resolves to just about anything. `Identifier` will inspect its surrounding context and return an instance of one of these reference implementations.

Let's have a look at the following example.

```
1  var foo: string
2  add(foo, "bar")
3  echo foo.find('a')
```

There are several `Identifier` elements here. The word `string` on line 1 is an `Identifier` and invoking `getReference()` on it will return a `TypeReference`, that is because it can tell that it needs to resolve to a type (if there was a variable called *string* it would *not* interfere with the type `string`). The word `add` on line 2 is also an `Identifier` and it gives a `ProcReference`, because, once again, it sees that its being used as part of a function call. The word `foo` on line 2 is an `Identifier` that gives an `IdentifierReference`, because in that context, all sorts of symbols are allowed. And lastly, the word `find` on line 3 is an `Identifier` that gives a `MemberReference`, because it can tell, that it is being used to refer to a member of `foo`.

When an `IdentifierReference` is asked to resolve itself, it will walk up the PSI tree starting at the `Identifier` that created it. It will inspect each PSI element along the way and ask the element to process the declarations that it contains, if any. Here is part of the code of the `resolve` method that showcases the tree traversal.

```
Identifier identifier = getElement(); // This is the element that we start from
SymbolResolver resolver = SymbolResolver.forName(identifier.getText());
PsiElement prevParent = identifier;
PsiElement scope = identifier;
while (scope != null) {
  if (!scope.processDeclarations(resolver, prevParent, identifier))
    break;

  prevParent = scope;
  scope = scope.getContext(); // getContext() returns the parent PsiElement
}
return resolver.getResolvedTarget();
```

On the second line we create a `SymbolResolver`, which will perform the actual checks against elements that are passed to it and determine whether an element is the one we are looking for (e.g. a variable declaration with a name that matches our `Identifier`). Inside the while loop we call the `processDeclarations` method on each element in the chain of parents. We pass along the symbol resolver, the previous parent (which, by necessity, is a child of the current element) and, additionally, the initial `Identifier` that we started from. Inside the `processDeclarations` method, the element then has the choice of passing itself or any of its children to the symbol resolver.

Here is how the `processDeclarations` method is implemented in the `ConstDef` element, which represents a constant definition (e.g. `const PI = 3.14`).

```
public boolean processDeclarations(PsiScopeProcessor processor,
    PsiElement lastParent, PsiElement place) {
  if (lastParent != null)
    return true;
  return processor.execute(this);
}
```

The initial *if* condition is necessary so that if the resolution begins from within the constant definition itself (e.g. from within the initializer expression) then the constant would not be considered as a candidate for matching any possible symbols. The convention, that is currently used, is that `lastParent` is `null`, if the node is being descended (parent of `ConstDef` calls `processDeclarations` on it), and not `null`, if the node is being ascended (child of `ConstDef` is being resolved). If `lastParent` is `null`, `ConstDef` passes itself to the processor, which is actually the symbol resolver. The `execute` method of our

SymbolResolver class determines the type of the element that is passed to it, and, if that element happens to be a symbol declaration, checks whether the name matches the one we are trying to resolve. If it does, the element will be remembered, and can be retrieved using the getResolvedTarget method. The return value of the method indicates whether the search should continue (true) or it should stop because the symbol has been found (false).

Another example of the processDeclarations method is the following, which is taken from an element called Block. Block is a collection of statements, it is used for the bodies of if-statements, loops, procedures etc.

```java
public boolean processDeclarations(PsiScopeProcessor processor,
    PsiElement lastParent, PsiElement place) {
  for (Statement statement : getStatements()) {
    if (statement == lastParent)
      return true;
    if (!statement.processDeclarations(processor, null, place))
      return false;
  }
  return true;
}
```

We iterate over all the statements inside the block, and ask each one to process its declarations. In case we are ascending the PSI tree, then we only want to process the statements that precede the one where the resolution started from, so we exit the loop when we reach the statement that was the previous parent.

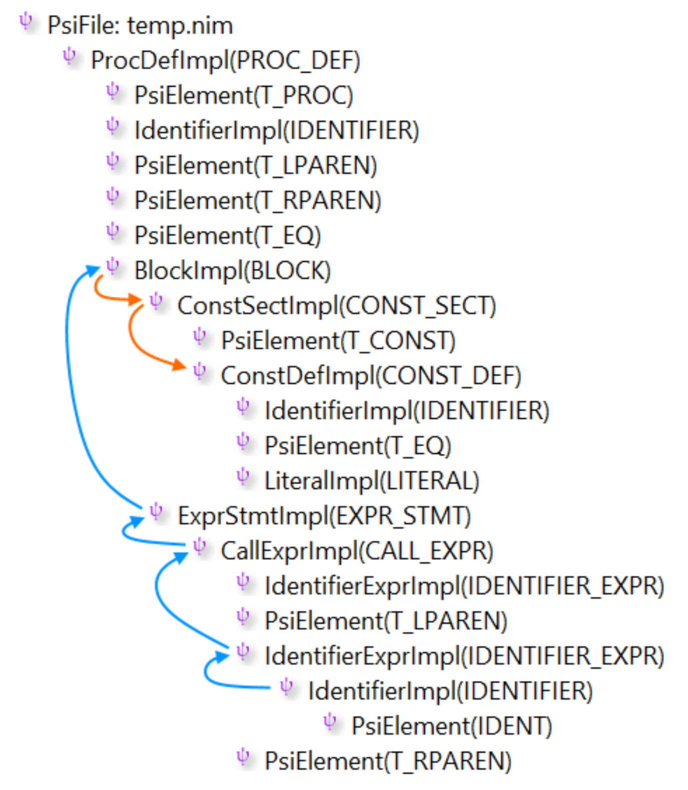
Resolving the argument of the echo call in the following code will cause the PSI tree to be traversed as shown in figure 3. The blue arrows indicate the path that is realized by the loop in the resolve method, the orange arrows indicate the path that is realized by the recursive processDeclarations calls.

```nim
proc foo() =
  const PI = 3.14
  echo(PI)
```

As mentioned earlier, reference resolution is a core functionality of a language plugin, and enables a lot of features. For instance it is now possible to search for usages of a symbol and perform rename refactoring. Figure 4 shows finding of usages in action.

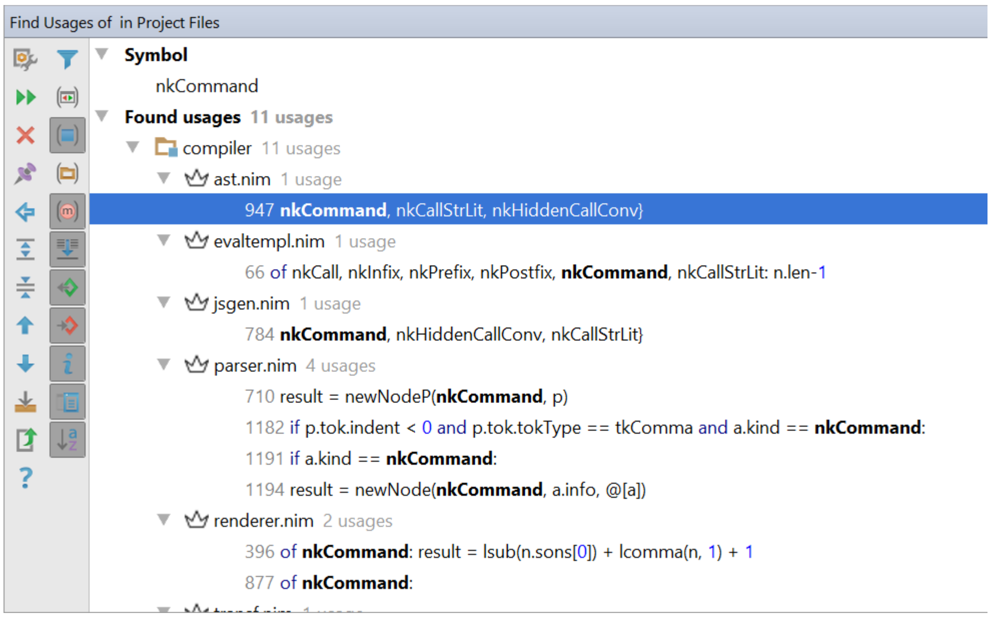
## 2.4 PSI stubs and stub indices

IntelliJ has a concept of so called PSI stubs and stub indices. The IDE scans every file in the project and in the standard library once, and for each file builds a PSI stub tree. A stub tree mirrors a normal PSI tree, but contains far fewer elements. It usually only



```
PsiFile: temp.nim
  ProcDefImpl(PROC_DEF)
    PsiElement(T_PROC)
    IdentifierImpl(IDENTIFIER)
    PsiElement(T_LPAREN)
    PsiElement(T_RPAREN)
    PsiElement(T_EQ)
    BlockImpl(BLOCK)
      ConstSectImpl(CONST_SECT)
        PsiElement(T_CONST)
        ConstDefImpl(CONST_DEF)
          IdentifierImpl(IDENTIFIER)
          PsiElement(T_EQ)
          LiteralImpl(LITERAL)
      ExprStmtImpl(EXPR_STMT)
        CallExprImpl(CALL_EXPR)
          IdentifierExprImpl(IDENTIFIER_EXPR)
          PsiElement(T_LPAREN)
          IdentifierExprImpl(IDENTIFIER_EXPR)
            IdentifierImpl(IDENTIFIER)
              PsiElement(IDENT)
          PsiElement(T_RPAREN)
```

Figure 3: Traversing the PSI tree.



```
Find Usages of  in Project Files
Symbol
    nkCommand
Found usages 11 usages
  compiler 11 usages
    ast.nim 1 usage
        947 nkCommand, nkCallStrLit, nkHiddenCallConv}
    evaltempl.nim 1 usage
        66 of nkCall, nkInfix, nkPrefix, nkPostfix, nkCommand, nkCallStrLit: n.len-1
    jsgen.nim 1 usage
        784 nkCommand, nkHiddenCallConv, nkCallStrLit}
    parser.nim 4 usages
        710 result = newNodeP(nkCommand, p)
        1182 if p.tok.indent < 0 and p.tok.tokType == tkComma and a.kind == nkCommand:
        1191 if a.kind == nkCommand:
        1194 result = newNode(nkCommand, a.info, @[a])
    renderer.nim 2 usages
        396 of nkCommand: result = lsub(n.sons[0]) + lcomma(n, 1) + 1
        877 of nkCommand:
```

Figure 4: Finding the usages of a symbol.

contains elements that represent global symbol definitions. For example, there is no need to store the bodies of procedures in it, for there is nothing inside a procedure that could be referenced from outside.

The real value of the stubs lies in the ability to index them based on a certain key. For instance, we have a stub index called `TypeIndex`, and every `TypeDef` stub element that occurs in the top-level file scope will add itself to that index. The name of the type will be used as the key. Similarly, we have a `RoutineIndex`, that contains all procedures indexed by their name.

A PSI element can, at any moment, be backed by either a stub, or an AST node. In a stub tree, every PSI element is backed by a PSI stub. In a full PSI tree, every element is backed by an AST node. Naturally, a stub-backed PSI element does not contain all the information that might be necessary for different PSI operations (e.g. getting the types of the parameters of a procedure). However, the API is designed in such a way that whenever a PSI element tries to access the AST and it is not backed by an AST node, the file that the element belongs to will be loaded and parsed, and the AST node will be injected into the PSI element. This happens seamlessly, without us having to write any code for it.

During reference resolution, if a symbol is not found in the file where the resolution starts from, some of the indices are queried instead. For instance, `TypeReference` will query the `TypeIndex` for a type definition with the name that it's trying to resolve. The index will return a list of `TypeDef` elements (if it has any for the given key) that are backed by stubs. We can then operate on the elements directly if we wish to process them further, keeping in mind that certain operations will end up loading the originating file as described above. Here is the whole `resolve` method of the `TypeReference` class, demonstrating the use of `TypeIndex`.

```java
public PsiElement resolve() {
  PsiElement entrance = getElement();
  String name = entrance.getText();
  SymbolResolver resolver = SymbolResolver.forName(name)
      .withFilter(el -> el instanceof TypeDef || el instanceof GenericParam);
  PsiElement prevParent = entrance;
  PsiElement scope = entrance;
  while (scope != null) {
    if (!scope.processDeclarations(resolver, prevParent, entrance))
      return resolver.getResolvedTarget();
    prevParent = scope;
    scope = scope.getContext();
  }

  GlobalSearchScope searchScope = ImportProcessor.buildImportScope(entrance);
  List<TypeDef> typeDefs = TypeIndex.getInstance().get(name, searchScope);
  for (PsiElement typeDef : typeDefs)
    if (!typeDef.processDeclarations(resolver, null, entrance))
      break;
  return resolver.getResolvedTarget();
}
```

The first part of the method is almost the same as the previous example of the `resolve` method of `IdentifierReference`. After the tree traversal is done and the symbol still hasn't been resolved, we query the `TypeIndex` for type definitions with the name that we are trying to find. Note that we also use a search scope, which is constructed based on the imports that are present in the current file. This makes sure we do not consider symbols that are not actually in scope.

The small size of the stub trees allows the IDE to keep many of them in memory for fast access. In fact, all the stub trees of all the files in a project and in the SDK are kept in memory while a project is open. This provides us with the ability to quickly resolve global symbols without having to load and parse any files. Additionally, stub trees are serialized and stored in the file system, so the project doesn't need to be rescanned every time it is opened.

## 2.5 Type deduction

Often, when resolving symbols, we need to know the type of a certain expression. For instance, when a `MemberReference` wants to resolve a field of an object, it needs to know the type of the object whose field is being accessed. In the following example, in order to resolve `name` on line 4, we need to first determine the type of `joe`.

```nim
1 type Person = object
2   name: string
3 var joe: Person
4 echo joe.name
```

In this case it is still fairly easy, as we just need to first resolve the symbol `joe` to its definition, and then resolve the symbol `Person` that is used as the type of the variable. However, it can be far more complex, as illustrated in the following example.

```nim
type
  A = object
    property: string
  B = object
    property: int
proc getObject(prop: string): A = A(property: prop)
proc getObject(prop: int): B = B(property: prop)
const arg = "Joe"
```

```
9  echo getObject(arg).property
```

In order to resolve the identifier `property` on line 9, we need to know the type of the expression `getObject(arg)`. There are two procedures called `getObject`, in order to know which one is being invoked, we need to know the type of the argument that is passed to it. The argument is `arg`, which we need to resolve in order to find its type, in this case it is a constant that is initialized with a string literal, so its type is string. Now we can pick the right procedure and examine its return type, in this case we call the procedure that's defined on line 6, which returns `A`. We can now resolve `A` and examine its fields to find `property`.

We have a class called `Type` that represents a concrete type in Nim. In most cases it will contain a reference to the type definition PSI element (`TypeDef`), but it can also represent generic type parameters. Every PSI element that represents an expression has a `getType` method that returns a `Type`. It is up for various expressions classes to implement the method. For instance, `IdentifierExpr` (an expression that consists of a single identifier) will simply resolve the identifier and, if it's something that can have a type (a variable, a parameter, a constant etc), will return that type. `CallExpr` (an expression that represents a procedure call) will resolve the procedure and return its return type. The type deduction logic has not been implemented for all possible expressions yet, for example, binary operator expressions do not currently resolve their type.

Nim happens to be such a flexible language, that even types can be represented by arbitrary expressions. This primarily occurs when templates and macros are used. Consider the following code snippet.

```
template ‘*‘(t: typedesc, n: int): typedesc =
  array[0..n-1, t]

var foo: int * 3
```

The variable `foo` has the type `int * 3`, which is an invocation of the template defined above it. The template invocation will be substituted with the type `array[0..2, int]`.

This example illustrates the need for us to be able to *evaluate any expression as a type*. Therefore, every expression also has an `asType` method, which should return the type that is represented by the expression (*not* the type *of* the expression). The easiest case is once again the `IdentifierExpr` element, as it just needs to resolve the identifier and if it points to a type definition, return the type for that definition. At this moment, this capability has only been implemented for a handful of expression types (e.g. the type of `foo` will not be deduced in the code snippet above).

## 2.6 Procedure overload resolution

Compared to other types of symbols, procedures require more work when resolving references to them. This is caused by the fact that procedures can be overloaded, meaning that there can be several procedures with the same name, that differ in the types of their parameters. Therefore, we always need to take into account the types of the arguments that are passed to a procedure invocation. Nim documentation describes a specific algorithm of how argument and parameter types are matched.

When an argument is compared to the formal type of a parameter, it either doesn't match or it matches according to one of six possible categories.

1. Exact match – the formal parameter type and the type of the argument are exactly the same.
2. Literal match – the argument is an integer literal and the parameter is of integer type such that the argument belongs to its range, or the argument is a floating point literal and the parameter is of floating point type such that the argument is in its range.
3. Generic match – the parameter has a generic type and the type of the argument satisfies the constraints of the generic type (if any).
4. Subrange or subtype match – the type of the argument is `range[T]` and the type of the parameter is exactly `T`, or the type of the argument is a subtype of the parameter type.
5. Integral conversion match – the parameter as well as the argument are of some integer or floating point type, and the argument is convertible to the parameter type.
6. Conversion match – the argument is convertible to the parameter type, possibly via a user-defined converter.

If at least one argument does not match at all, then that overload is not considered (i.e. each argument needs to match according to one of the six categories). The overload that has the highest number of exact matches is the one that is going to be chosen. If two or more overloads have the same number of exact matches, then the number of literal matches in considered, and so on. If all categories are tied, then the call is ambiguous; this is a compile time error. We have currently only implemented *exact*, *subtype* and *generic* matching (without the constraint checks) and so overload resolution might not always be correct. The implementation of overload resolution can be found in the class `RoutineResolver`.

## 2.7 Code completion

IntelliJ supports two ways of implementing code completion – *reference completion*, which is simpler and good enough for most cases, and *contributor-based completion*, which is supposed to provide more fine-grained control over how and when does code completion get invoked. We only make use of reference completion. All that's needed for it, is to implement the `getVariants` method of the `PsiReference` interface in our reference classes. Whenever code completion needs to be invoked (either at the request of the user, or automatically by the IDE), IntelliJ will insert a dummy identifier at the current position of the cursor, and check whether the identifier returns some sort of a reference (in our case, either `IdentifierReference`, `ProcReference`, `TypeReference` or `MemberReference`). If it does, the `getVariants` method will be invoked on it. The method should return a list of symbols that will be shown in the code completion pop-up. The operation of the `getVariants` method is similar to that of the `resolve` method, except that, instead of searching for a symbol with a specific name, we must collect *all* symbols that are in scope and are valid candidates in the current context. The following is an example of how the method is implemented in our `TypeReference` class.

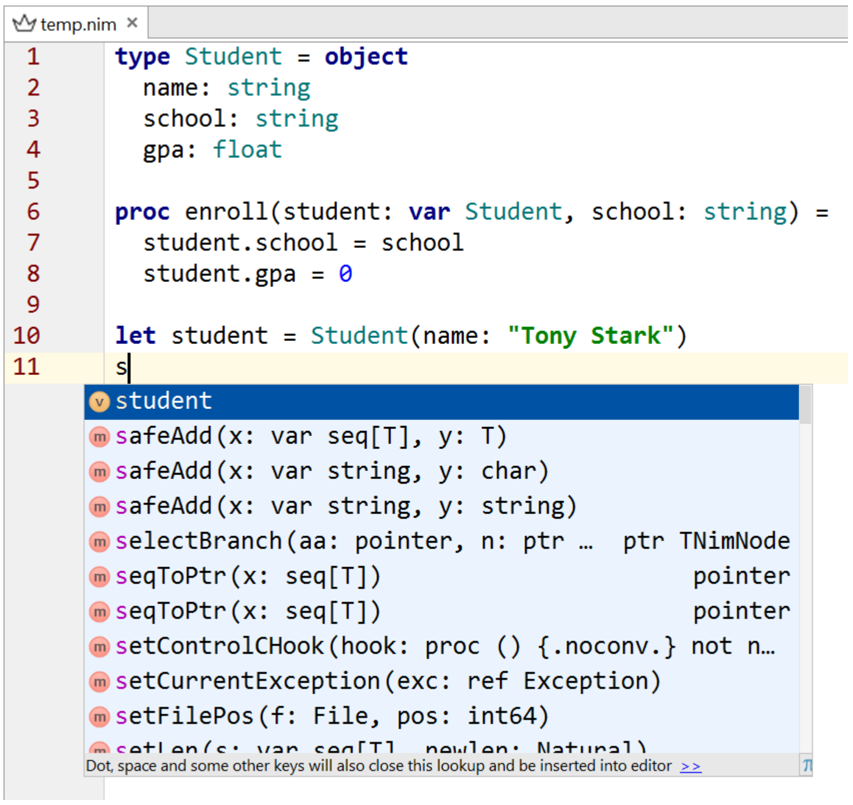
```java
public Object[] getVariants() {
  SymbolCollector collector = SymbolCollector.withFilter(
      el -> el instanceof TypeDef || el instanceof GenericParam);
  PsiElement entrance = getElement();
  PsiElement prevParent = entrance;
  PsiElement scope = entrance;
  while (scope != null) {
    scope.processDeclarations(collector, prevParent, entrance);
    prevParent = scope;
    scope = scope.getContext();
  }

  Project project = entrance.getProject();
  TypeIndex typeIndex = TypeIndex.getInstance();
  GlobalSearchScope searchScope = ImportProcessor.buildImportScope(getElement());
  List<TypeDef> extra = typeIndex.getAllKeys(project).stream()
      .flatMap(key -> typeIndex.get(key, project, searchScope).stream())
      .collect(Collectors.toList());
  for (TypeDef typeDef : extra)
    typeDef.processDeclarations(collector, null, entrance);

  return collector.getLookupElements().toArray();
}
```

The first half of the method walks up the PSI tree, similar to the `resolve` method, except it never breaks out of the loop until the root of the tree has been reached. The `SymbolCollector` class collects all the symbols that are passed to it (and that match the given filter) and can later return all the elements as a list of `LookupElements`. A `LookupElement` contains information about what text and icon to display in the pop-up, what font style should be used etc. The second half of the method queries our type stub index for all available type definitions that should be in scope according to the list of imported modules.

It is worth pointing out that getting the list of all possible completion candidates can involve a lot of work. For instance, when `MemberReference` is asked to produce a list of procedures that could be called on an object using the method invocation syntax, it first has to ask the `RoutineIndex` for all procedures that are currently in scope, and then needs to match the receiver expression (the thing that the method is being called on) against the first parameter of every single procedure. Keep in mind, that the index will return a list of PSI elements that are backed by *stubs*, and they do not have enough information to be able to properly resolve the types of parameters, meaning that for every procedure we will actually load and parse the file were that procedure is defined, in order to be able to get full type information. The reason, why routine stub elements do not store the parameter type in the stub tree, is largely due to the fact that a type in Nim can be represented by any arbitrary expression, and therefore every expression element would have to be able to operate off a stub and also be able to serialize and deserialize itself. This is in contrast to some other programming languages, where types can only be expressed by a small number of possible forms; for instance, in Java, the formal type of every method parameter will be represented by either a primitive type name, a class name (possibly with generic parameters), a name referring to a generic type parameter or an array, all of which could be stored in a simple structure (even just a string) and put into the stub tree. Also, it is not possible to resolve the type when building the stub tree, as no stub indexes are available at that time.



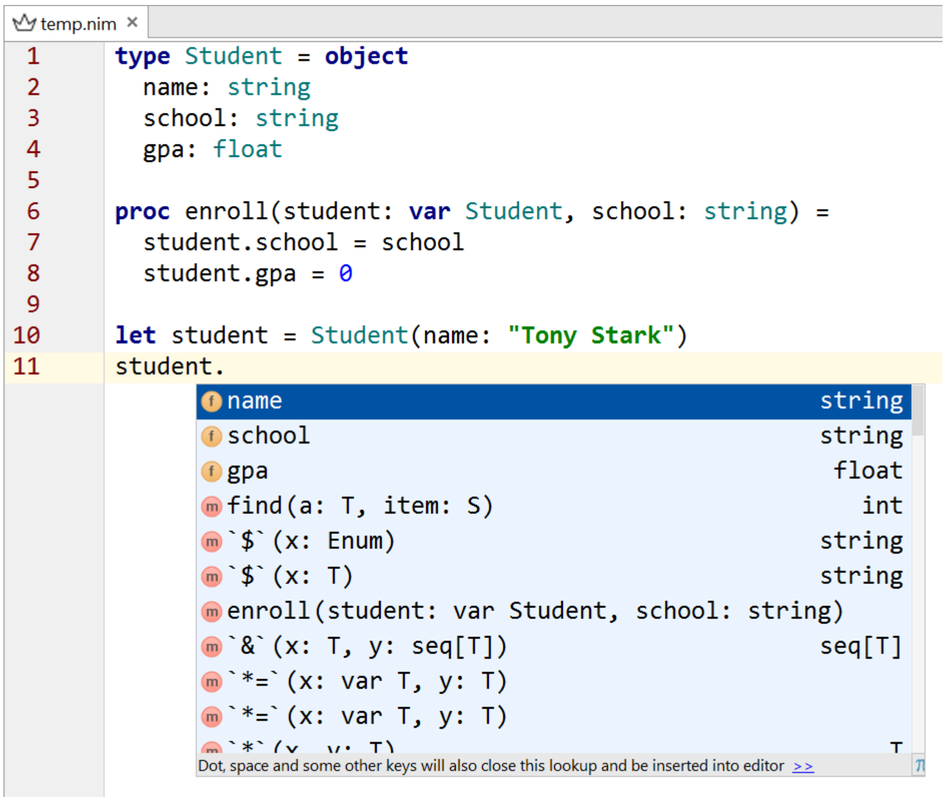
```
type Student = object
  name: string
  school: string
  gpa: float

proc enroll(student: var Student, school: string) =
  student.school = school
  student.gpa = 0

let student = Student(name: "Tony Stark")
s
```

```
student
safeAdd(x: var seq[T], y: T)
safeAdd(x: var string, y: char)
safeAdd(x: var string, y: string)
selectBranch(aa: pointer, n: ptr …  ptr TNimNode
seqToPtr(x: seq[T])                         pointer
seqToPtr(x: seq[T])                         pointer
setControlCHook(hook: proc () {.noconv.} not n…
setCurrentException(exc: ref Exception)
setFilePos(f: File, pos: int64)
setLen(s: var seq[T], newlen: Natural)
Dot, space and some other keys will also close this lookup and be inserted into editor >>
```

Figure 5: An `IdentifierReference` is being completed.



```
type Student = object
  name: string
  school: string
  gpa: float

proc enroll(student: var Student, school: string) =
  student.school = school
  student.gpa = 0

let student = Student(name: "Tony Stark")
student.
```

```
name                                        string
school                                      string
gpa                                          float
find(a: T, item: S)                            int
`$`(x: Enum)                                string
`$`(x: T)                                   string
enroll(student: var Student, school: string)
`&`(x: T, y: seq[T])                        seq[T]
`*=`(x: var T, y: T)
`*=`(x: var T, y: T)
`*`(x, y: T)                                     T
Dot, space and some other keys will also close this lookup and be inserted into editor >>
```

Figure 6: A `MemberReference` is being completed.

# 3 Conclusion

In this thesis we created a plugin for IntelliJ IDEA that facilitates writing Nim programs. The plugin provides symbol navigation and code completion as its primary features. Additionally, one can search for usages of a symbol and perform rename refactoring.

At this stage, there are still areas where the plugin could be improved. Type deduction is currently somewhat limited, for instance there is no special support for generics, and not all expressions can currently be evaluated as types. Procedure overload resolution is also lacking, as certain match categories haven't been implemented. The Nim compiler contains inside it a small virtual machine, that allows it to evaluate constant expressions at compile time. This is also needed in order to deduce the types of compile time constants. Our plugin currently lacks that functionality.

In addition to improving the core language support functionality, other auxiliary features could be added. For instance, automatic addition of import statements when a symbol, that's declared in a non-imported module, is used. Integration with the Nim build system could also be of use, as it would allow to compile and run programs directly from the IDE. Nim also has a convention for writing documentation comments. The standard library contains a lot of documentation in the code. IntelliJ fully supports the concept of showing symbol documentation in the editor, and so implementing support for that would also be of great value. Lastly, IntelliJ has a nice way of showing errors and warnings inline inside the editor. Utilizing that to point out invalid code would be another worthwhile addition.

## References

[1] Nim website.
`http://nim-lang.org/`

[2] First commit in the Nim repository.
`https://github.com/nim-lang/Nim/commit/405b86068e6a3d39970b9129ceec0a9108464b28`

[3] IntelliJ IDEA website.
`https://www.jetbrains.com/idea/`

[4] Aporia IDE on GitHub.
`https://github.com/nim-lang/Aporia`

[5] Nimsuggest on GitHub.
`https://github.com/nim-lang/nimsuggest`

[6] Andreas Rumpf. OSCON 2015. Nim: An Overview.
`https://www.youtube.com/watch?v=4rJEBs_Nnaw`

[7] Nim Manual.
`http://nim-lang.org/docs/manual.html`

[8] IntelliJ Platform SDK DevGuide. Custom Language Support.
`http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support.html`

[9] JFlex.
`http://jflex.de/`

[10] Grammar-Kit.
`https://github.com/JetBrains/Grammar-Kit`

[11] Nim compiler source on GitHub. *parser.nim.*
`https://github.com/nim-lang/Nim/blob/devel/compiler/parser.nim`

# Appendix

## Source code

The source code for the plugin is available on BitBucket at the following URL: `https://bitbucket.org/dmitri_gb/idea-nim/`. Build instructions are included in the repository.