

MASTER THESIS - FALL 2016

Tifig

Author:
Toni Suter

Supervisor:
Prof. Peter Sommerlad

March 20, 2017

Abstract

In 2014, Apple introduced a new programming language called Swift which replaced Objective-C as the default programming language to develop applications for Apple's platforms. Since then, the language has evolved a lot and was open-sourced at the end of 2015. There are now official releases for both macOS and Ubuntu and there are efforts from the community to bring the language to other platforms as well.

With its Xcode IDE (integrated development environment) Apple focuses mainly on the development of iOS and macOS apps. However, Xcode is not available on platforms other than the Mac and there aren't a lot of alternatives yet. Additionally, many programmers are interested in using Swift for other areas such as web development.

The main goal of this project is to create a cross-platform Swift IDE based on Eclipse which contains the basic components required to develop Swift programs.

Over the course of the term project a simple Swift IDE called *Tifig* has been developed. The user can edit source files, build projects and run the resulting executables all from within the IDE. Every time the user changes a source file, the code is re-parsed and syntax errors are reported in the form of markers in the editor.

In the subsequent master thesis the parser has been further improved and updated for Swift 3. Additionally, an indexer has been implemented. The semantic knowledge that is obtained by the indexer allowed for the development of the code navigation features *Open Type* and *Jump to Definition*. A lot of Swift's core types and operators are not part of the language itself, but are instead declared in the standard library. For this reason, Tifig also indexes the standard library and makes its public symbols available in each project.

Management Summary

Tifig is a simple, cross-platform Swift IDE [App17b] based on the Eclipse platform [ecl17c]. I developed it during my term project and extended it in my subsequent master thesis. This management summary gives an overview over the motivation and the goals for this project. It also describes the results of the project as well as work that could be done to further improve Tifig in the future.

Motivation

Swift is a relatively new programming language that was originally invented by Apple and is now available as an open-source project. Apple also develops Xcode [App17g] which is probably the most well-known IDE with support for Swift. Other than that, there are not a lot of compelling options yet.

While Xcode is a very powerful and mature IDE, it has a few drawbacks. It is heavily focused on the development of iOS and macOS apps and is less suited for other areas such as web development. Additionally, it doesn't yet support the refactoring of Swift code and is only available on macOS.

Therefore, this is a good time to develop a new, cross-platform Swift IDE. The Eclipse platform is a good foundation to build on, since it is very extendable and there are already a lot of other well-known IDEs such as Eclipse JDT [ecl17b] and Eclipse CDT [ecl17a] that are based on it.

Goals

The goal of the term project was to create an Eclipse-based Swift IDE with the following features:

- Wizards to create new Swift projects & files
- Source code editor with syntax-highlighting support for Swift
- Automatic parsing of the code and reporting of syntax errors
- Building & running of Swift programs from within the IDE

Afterwards, the following goals were set for the subsequent master thesis:

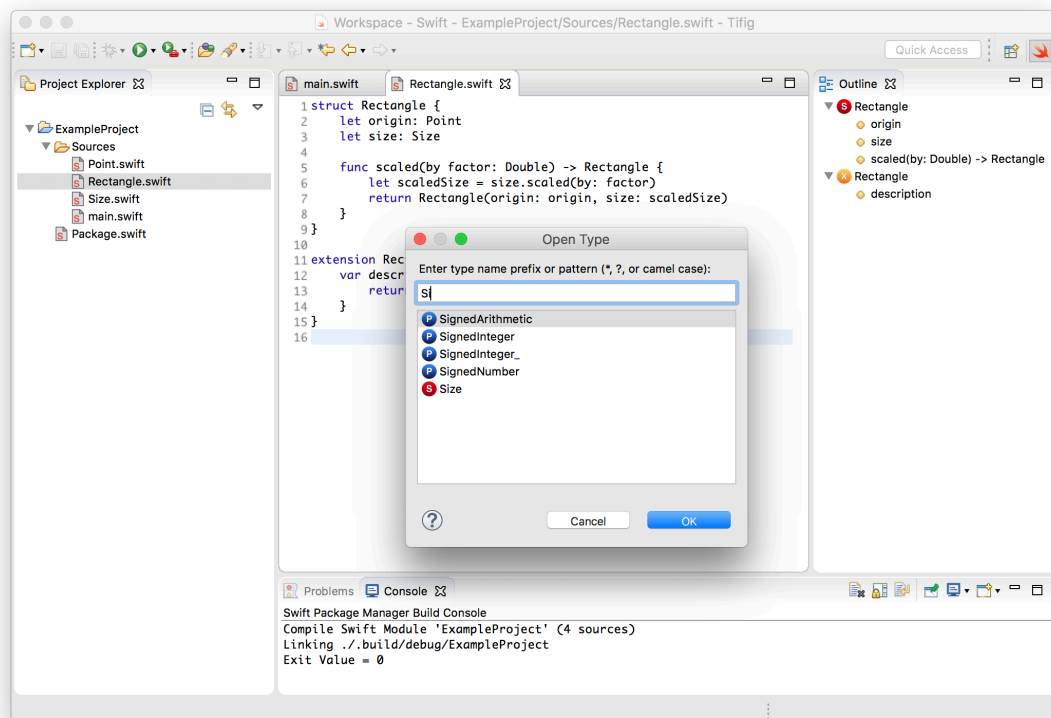
- Improve the parser to make it fully compatible with Swift 3
- Develop an indexer for the Swift programming language

- Integrate Swift’s standard library in the indexing process
- Add IDE features that rely on the index (e.g., *Open Type*, *Jump to Definition*)
- Add refactoring support

Results

Most of the project goals were achieved and the result is a simple Swift IDE called *Tifig*. It automatically parses and indexes Swift code as it is entered by the user. In addition to the user’s code, Tifig also indexes the Swift standard library and makes its public symbols available in each project. Errors are reported in the form of markers in the editor and programs are built with the help of the Swift Package Manager. Finally, the two features *Open Type* and *Jump to Definition* have been implemented in order to make it more convenient to navigate a Swift project. A screenshot of Tifig is shown below:

Screenshot of Tifig



Unfortunately, the development of the indexer took longer than expected and there was not enough time to implement refactoring support.

Further work

While Tifig is already a functioning IDE for small projects, there is still a lot of room for improvement. The following list contains a few things that could be added or improved in the future:

- Improve the accuracy of reported syntax / semantic errors
- Improve indexer (e.g., better generics support, support for partial imports, etc.)
- Add more code navigation features (e.g., *Open Call Hierarchy*)
- Add support for auto-completion in the Swift editor
- Add support for debugging
- Add support for automated refactorings

Contents

1	Task Description - Term Project	4
1.1	Motivation	4
1.2	Project Goals	4
1.3	Expected Results	4
1.3.1	Optional features	5
1.4	Time management	5
1.5	Deliverables	5
2	Task Description - Master Thesis	6
2.1	Motivation	6
2.2	Project Goals	6
2.3	Expected Results	6
2.3.1	Optional Features	7
2.4	Time management	7
2.5	Deliverables	7
3	Analysis	8
3.1	Introduction to Swift	8
3.1.1	Development	8
3.1.2	Vision	8
3.1.3	Type Safety	9
3.1.4	Variables / Properties	9
3.1.5	Standard Library Types	11
3.1.6	Tuples	11
3.1.7	Functions	11
3.1.8	Closures	12
3.1.9	Optionals	13
3.1.10	Extensions	15
3.1.11	Operators	16
3.1.12	Pattern Matching	21
3.1.13	Protocol-Oriented Programming	24
3.1.14	Memory Management	24
3.1.15	Interoperability with Objective-C and C	25
3.2	Overview	25
3.2.1	Features	25
3.2.2	Components	26
3.3	Conclusion	28
4	Lexer	29
4.1	Keywords	29

Contents

4.2	Identifiers	30
4.3	Operators	31
4.4	Literals	33
4.4.1	Integer Literals	33
4.4.2	Floating-Point Literals	33
4.4.3	String Literals	34
4.4.4	Boolean Literals	34
4.4.5	Nil Literal	34
4.4.6	Compiler Literals	35
4.5	Punctuation	35
4.6	Comments	35
4.6.1	Single-line Comments	36
4.6.2	Multi-line Comments	36
4.7	Implementation Status	36
5	Parser	37
5.1	Architecture	37
5.1.1	Parser Modules	38
5.1.2	Recursive-Descent Parsing	41
5.1.3	Speculative Parsing and Backtracking	43
5.2	AST	44
5.2.1	Requirements	45
5.2.2	Structure	45
5.2.3	Visiting an AST	46
5.3	Error handling	47
5.4	Testing	49
5.5	Implementation Status	50
6	Indexer	51
6.1	The job of an Indexer	51
6.2	Architecture Overview	52
6.3	Definition Pass	54
6.3.1	Bindings	55
6.3.2	Unavailable Declarations	55
6.3.3	Conditions	56
6.3.4	Extensions	57
6.3.5	Implicit Operator Bindings	57
6.3.6	Implicit Variable Bindings	57
6.3.7	Implicit Closure Parameters	58
6.3.8	Imports	59
6.3.9	Standard Library	59
6.4	Type-Annotation Pass	60
6.4.1	Index Types	60
6.4.2	Tasks of the Type-Annotation Pass	66
6.5	Type-Check Pass	71
6.5.1	Type Inference in Swift	71
6.5.2	Implementation Approach	76

Contents

6.6	Constraint-Based Type Checker	78
6.6.1	Overview	78
6.6.2	Example 1: Literals	80
6.6.3	Example 2: Overload Resolution	84
6.6.4	Example 3: Binary Expressions	89
6.6.5	Example 4: Explicit Member Expression	95
6.6.6	Example 5: Implicit Member Expression	99
6.6.7	Example 6: Optionals	102
6.6.8	Example 7: Initializer Call	105
6.6.9	Example 8: Generic Function	108
6.6.10	Solver Algorithm	111
6.6.11	Ranking Rules	113
6.6.12	Contextual Type Constraints	115
6.6.13	Pattern Matching	117
6.6.14	Conversions	118
6.7	Testing	121
6.7.1	Single-File Test Cases	121
6.7.2	Multi-File Test Cases	122
6.7.3	Multi-Module Test Cases	123
6.8	Implementation Status	124
7	User Interface	125
7.1	Wizards	125
7.1.1	Project Wizard	125
7.1.2	File Wizards	126
7.2	Project Nature	127
7.3	Swift Perspective	127
7.4	Editor	128
7.4.1	Auto Indenting	128
7.4.2	Syntax Highlighting	129
7.4.3	Reconciler	130
7.5	Type Information Hover	134
7.6	Open Type Dialog	135
7.7	Builder	136
7.8	Launcher	137
7.9	Implementation Status	139
8	Conclusion	140
8.1	Results	140
8.2	Outlook	141
8.3	Acknowledgements	141
	Bibliography	142

1 Task Description - Term Project

This section outlines the goals and the scope of the term project.

1.1 Motivation

In 2014 Apple introduced a new programming language called Swift. It is a modern, statically-typed language that is meant to replace Objective-C as the default programming language to develop applications for Apple's platforms. On December 3rd 2015, Swift was made open source under the Apache 2.0 license [Fou04].

Apple provides the Xcode IDE as the default tool to program in Swift. While Apple and the open source community are already working on porting the language and its standard library to Linux, there seem to be no plans to do the same for Xcode.

Thus, it would be great to have a cross-platform alternative to Xcode. The Eclipse platform with its plug-in system seems like a good foundation to build on.

1.2 Project Goals

The main goal of this term project is to create a collection of plug-ins for the Eclipse platform that add support for the Swift programming language. Since there will not be enough time to develop all the features that are expected from a modern IDE, I want to focus on creating a good foundation, which can later be extended during my upcoming master thesis.

1.3 Expected Results

- **Wizards**
The wizards will allow the creation of new Swift projects and files, classes, etc.
- **Perspective**
The perspective allows programmers to configure the views they want to see in the workbench while writing Swift code.
- **Editor**
The editor will likely be the bulk of the work. A parser will be required to support features like syntax highlighting, code formatting and static analysis / refactorings.

- **Builder**

The builder is responsible for the correct invocation of the Swift compiler.

- **Launch Configuration**

The launch configuration knows how to launch a Swift application.

1.3.1 Optional features

The following features are expected in a modern IDE but are considered optional for this term project due to the limited time available:

- **Debugging**
- **Refactorings**
- **Interoperability with C/C++ (Eclipse CDT)**

1.4 Time management

The project started on February 29th 2016. It will end on July 18th 2016 at 12:00 p.m., which is when the final release has to be submitted completely.

1.5 Deliverables

The following items will be included in the final release of the project:

- 2 printed copies of the documentation
- PDF of poster for presentation
- 2 CDs that contain the code, project resources, documentation
- 1 CD for archive with the documentation and abstract without personal information

2 Task Description - Master Thesis

This section outlines the goals and the scope of the master thesis.

2.1 Motivation

This master thesis is based on my previous term project in which I developed an Eclipse-based Swift IDE called Tifig. Currently, Tifig is still a very basic IDE. It has a parser that checks the syntax and it can build and run programs using the Swift Package Manager. However, the semantic analysis part is far from complete and there are a lot of opportunities for improvement. In addition the Swift 3 language definition was still in flux and changed in the last weeks of the term project. Not all of those changes have made it into Tifig's parser yet.

2.2 Project Goals

The goal of this master thesis is to extend the capabilities of Tifig. The main focus will be on improving the indexer and on adding refactoring support. Additionally, the remaining issues of the parser should be resolved in order to support the full Swift 3 feature set.

2.3 Expected Results

Parser Improvements

- Implement Swift 3 grammar changes that happened since the term project
- Improve the error reporting of the parser

Semantic Analysis and Symbol Table Improvements

- Improve and extend the implementation and the tests of the indexer
- Add imported symbols and standard library symbols to the index
- Report semantic errors in the UI
- Add more IDE features that rely on the index (e.g., Open Type, Open Call Hierarchy)

Refactoring Support

- Add the ability for plug-ins to programmatically modify a Swift program's AST and to reflect those changes in the Editor and source code.
- Implement some useful refactorings (e.g., extract method, rename symbol)

2.3.1 Optional Features

The following features are expected in a modern IDE but are considered optional for this master thesis due to the limited time available:

Debugging Support

- Set breakpoints
- Step through the statements of a program
- Inspect local variables

Integration with Eclipse CDT / Codeloop

- Add the ability to develop called C/C++ code from the Swift language within a single IDE

Add Unit Testing Support

2.4 Time management

The project started on September 1st 2016. It will end on February 28th 2017 at 12:00 p.m., which is when the final release has to be submitted completely.

2.5 Deliverables

The following items will be included in the final release of the project:

- 2 printed copies of the documentation
- Poster for presentation
- 2 CDs that contain the code, project resources, documentation
- 1 CD for archive with the documentation and abstract without personal information

3 Analysis

The chapters 1 and 2 described the motivation for this project and gave an overview of the project goals. This included a list of the major components that should be implemented as part of this project. This chapter gives a short introduction to Swift and describes the components in more detail.

3.1 Introduction to Swift

This section gives a short introduction to the Swift programming language. It is by no means a comprehensive tutorial but rather a quick overview of the language’s key characteristics and goals.

3.1.1 Development

The development of the Swift programming language started internally at Apple in 2010 [Lat10]. For the first few years Chris Lattner was the lead developer of the project [Lat17a]. He also gave the first public demo of Swift on June 2, 2014 at Apple’s annual Worldwide Developers Conference (WWDC) [App14]. Since Lattner left Apple in early 2017, Ted Kremenek has taken over his role as lead developer [Lat17b].

Since its inception, Swift has undergone many significant changes and was open-sourced on December 3, 2015 [swi15]. Apple is the project lead, but there are already many contributions from non-Apple contributors and the community is very active [swi17a]. On September 13, 2016, Swift 3.0 was released [Kre16]. It was the first major release since Swift was open-sourced. At the time of writing, Swift 4.0 is being developed, which is expected to be released in the fall of 2017 [Kre17].

3.1.2 Vision

Apple’s vision for Swift is to create a safe programming language that is friendly to beginners but also provides modern features that make programming easier, more flexible and more fun. Apple also states that Swift is “the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language” and that it is “designed to scale from ‘hello, world’ to an entire operating system” [tsp17a].

3.1.3 Type Safety

Swift is statically typed, which means that the type of each expression has to be known at compile time. This way, a lot of bugs can be caught and fixed early. Luckily, explicit type annotations are not always necessary, because the compiler can often infer the types automatically.

3.1.4 Variables / Properties

Variables / Properties are introduced with a `let`- or `var`-declaration. If such a declaration is located within a named type (e.g., a struct or a class) it is considered to be a *property* of that type. However, a `let`- or `var`-declaration can also appear in the global scope or in a local scope in which case it is considered to be a *global variable* or a *local variable*, respectively.

There are three different kinds of variable / property declarations which are described in the following sections.

Stored Variables / Properties

Listing 3.1 shows a few different examples of declarations of stored variables:

Listing 3.1: *Stored Variables*

```

1  var x = 41                // x is of type Int
2  x += 1
3
4  let y = "hello"           // x is of type String
5  y = y.uppercased()        // error: cannot assign to value: 'y' is a 'let' constant
6
7  let a1 = 1.5, b1 = true    // a1 is of type Double, b1 is of type Bool
8
9  let (a2, b2) = (1.5, true) // a2 is of type Double, b2 is of type Bool

```

On line 1, an `Int` variable called `x` is declared. This variable is mutable, because we used the `var` keyword. Thus, the subsequent assignment `x += 1` works and `x` has the value 42 after the assignment.

On line 4, a `String` variable called `y` is declared. This variable is immutable, because we used the `let` keyword. Therefore, the assignment `y = y.uppercased()` results in a compilation error.

On line 7 two immutable variables called `a1` and `b1` are declared in the same declaration.

Finally, on line 9 two immutable variables called `a2` and `b2` are declared in the same declaration. However, this time there is a tuple pattern on the left hand side of the equals sign that is matched against a tuple expression on the right hand side.

Computed Variables / Properties

Since the values of computed variables / properties usually change during the execution of the program, they always have to be declared with the `var` keyword. Listing 3.2 shows an example of a read-only, computed variable:

Listing 3.2: *Read-only, computed variable*

```
1 import Darwin          // import required for arc4random()
2
3 var rand: Int {
4     return Int(arc4random() % 100)
5 }
6
7 print(rand, rand)      // prints two random numbers between 0 and 99
```

The computed variable `rand` produces a random number between 0 and 99 every time it is evaluated.

Listing 3.3 shows an example of a read-write, computed variable:

Listing 3.3: *Read-write, computed variable*

```
1 import Foundation      // import required for sqrt()
2
3 var radius = 5.5
4
5 var area: Double {
6     get {
7         return radius * radius * Double.pi
8     }
9
10    set {
11        radius = sqrt(newValue / Double.pi)
12    }
13 }
```

When the computed variable `area` is accessed, its value is automatically computed from the value of the stored variable `radius`. On the other hand, when a new value is assigned to `area`, its setter is executed which updates the stored variable `radius`. Note that the `newValue` variable is implicitly available in the setters of computed variables.

Observed Variables / Properties

Observed variables / properties always have to be declared with the `var` keyword, because otherwise their value could not change and it would make no sense to observe them. Listing 3.4 shows an example of an observed variable:

Listing 3.4: *Observed Variable*

```
1 var x = 0 {
2     willSet {
3         print("value will change from \(x) to \(newValue)")
4     }
5     didSet {
6         print("value did change from \(oldValue) to \(x)")
7     }
8 }
9
10 x = 5      // prints 'value will change from 0 to 5'
11           // and 'value did change from 0 to 5'
```

The `willSet`-clause is executed shortly before the value is updated. Within that code block the `newValue` variable is implicitly available to refer to the new value of the variable. Similarly, the `didSet`-clause is executed shortly after the value is updated and within its code block we can use the implicit `oldValue` variable to refer to the old value of the variable.

3.1.5 Standard Library Types

Many of the types that seem to be built into the Swift programming language are actually declared in the standard library. This includes the types `Int`, `Double`, `Bool`, `String`, `Array` and `Dictionary`, among others. All of these types are structs with value semantics [Gal16].

Two notable exceptions are tuple types and function types, which are built into Swift.

3.1.6 Tuples

Tuples provide a way to quickly group multiple values without defining a new named type. Tuples have a fixed number of elements and their elements can have different types. An example of this is shown in Listing 3.5:

Listing 3.5: *Tuple Example*

```
1 let tuple = (8640, "Rapperswil") // tuple is of tuple type (Int, String)
2 print(tuple.0)                  // prints '8640'
3 print(tuple.1)                  // prints 'Rapperswil'
```

The example shows that individual elements of a tuple can be accessed by element index. Alternatively, the elements of a tuple can be named. An example of this is shown in Listing 3.6:

Listing 3.6: *Tuple with named elements*

```
1 let tuple = (zip: 8640, name: "Rapperswil") // tuple is of tuple type (zip: Int, name: String)
2 print(tuple.zip)                          // prints '8640'
3 print(tuple.name)                         // prints 'Rapperswil'
```

Note that there is no such thing as a single-element tuple. Instead, a parenthesized expression that only contains a single element has the same type and value as the element itself.

3.1.7 Functions

In Swift, functions can be declared in many different locations. Apart from declaring methods within named types it is also valid to declare free functions at the file level or to declare a function in the local scope of another function. The declaration syntax always looks the same. Listing 3.7 shows a free function that computes the factorial of a number:

Listing 3.7: *Factorial function in Swift*

```

1 func factorial(_ n: Int) -> Int {
2     guard n > 0 else {
3         return 1
4     }
5     return n * factorial(n - 1)
6 }
7
8 print(factorial(5)) // 120

```

Note that the return type is specified after an arrow (\rightarrow) which follows the parameter list. Each parameter can have an external name and an internal name. The internal name is used to refer to the parameter from within the function body. If a parameter has an external name, this name has to be specified in an argument label at the call site. In the example above, the parameter doesn't have an external name because it was suppressed with the underscore `_`. By default, the external name and the internal name of a parameter are the same. Alternatively, it is also possible to specify two parameter names in which case the first one is the external name and the second one is the internal name. This can lead to code that is more readable, especially if a function has many parameters. Listing 3.8 shows an example:

Listing 3.8: *External and internal parameter names*

```

1 func greet(person: String, from hometown: String) {
2     print("Hello, \(person) from \(hometown)!")
3 }
4
5 greet(person: "Tim", from: "Cupertino") // Hello, Tim from Cupertino!

```

The first parameter only specifies the name `person`. Therefore, both its internal and external name is `person`. For the second parameter two different names are specified. The external parameter name is `from` and the internal parameter name is `hometown`.

Swift functions are first class values of the language. They can be stored in variables, passed as arguments to functions and returned from functions. This also means that functions have a type. The `factorial()` function in Listing 3.7 has the type `(Int) -> Int` and the `greet()` function in Listing 3.8 has the type `(String, String) -> ()`. Note that a function that doesn't return anything implicitly has the return type `()`.

Listing 3.9 shows an example of how to use the higher-order method `map()` to transform an array of numbers [Har96]:

Listing 3.9: *Passing a function to map()*

```

1 func square(_ n: Int) -> Int {
2     return n * n
3 }
4
5 let numbers = [0, 1, 2, 3, 4, 5]
6 let squares = numbers.map(square)
7 print(squares) // 0, 1, 4, 9, 16, 25

```

3.1.8 Closures

Closures are expressions that can be used in the same places as functions. Listing 3.10 shows code that is very similar to the example in Listing 3.9, but this time a closure is

passed to the `map()` method:

Listing 3.10: *Passing a closure to `map()`*

```
1 let numbers = [0, 1, 2, 3, 4, 5]
2 let squares = numbers.map({ (n: Int) -> Int in return n * n })
3 print(squares) // 0, 1, 4, 9, 16, 25
```

Listing 3.11 shows another version of the same code. This time the closure’s parameter type and return type is inferred from the context. Additionally, the expression `n * n` is now returned implicitly. This is only possible if the body of a closure consists of a single expression:

Listing 3.11: *Passing a closure to `map()`*

```
1 let numbers = [0, 1, 2, 3, 4, 5]
2 let squares = numbers.map({ n in n * n })
3 print(squares) // 0, 1, 4, 9, 16, 25
```

Finally, to make the code even more concise, we can use the shorthand parameter names `$0`, `$1`, `$2` and so on to refer to the first, second or third parameter, respectively. Additionally, if a closure is the last argument in a function call, we can use the trailing closure syntax which allows us to write the closure after the function call’s parentheses. If the closure is the only argument in a function call, the parentheses can be omitted entirely. This is shown in Listing 3.12:

Listing 3.12: *Passing a closure to `map()`*

```
1 let numbers = [0, 1, 2, 3, 4, 5]
2 let squares = numbers.map { $0 * $0 }
3 print(squares) // 0, 1, 4, 9, 16, 25
```

3.1.9 Optionals

Optionals are a core concept in the Swift programming language. The goal of this feature is to improve the safety of programs by formalizing the notion of optionality so that it can be enforced by the compiler. In languages like Objective-C and Java, you never know whether it is really safe to dereference e.g., a parameter because it might be `null`. Good programmers insert `null` checks to avoid exceptions. However, that requires a lot of discipline and is therefore error-prone.

In Swift, a variable can only become `nil`, if its value is wrapped in an optional. For example, a `String` cannot be `nil`, but an `Optional<String>` can. The compiler prohibits access to properties or methods of an optional. Instead the programmer has to unwrap the underlying value first.

Another benefit of optionals is that they also work with basic types such as integers and floating-point numbers which means that programmers don’t have to resort to arbitrary sentinel values such as `-1` [tsp17e].

Swift provides different language constructs to unwrap optionals which are described in the following sections.

Optional Binding

Optional Binding is a special kind of condition that can be used in `if`, `guard` and `while` statements. An example of this is shown in Listing 3.13:

Listing 3.13: *Optional Binding*

```

1 // String? is syntactic sugar for Optional<String>
2 func createGreeting(name: String? = nil) -> String {
3   // here, the type of 'name' is Optional<String>
4   if let name = name {
5     // here, the type of 'name' is String
6     return "Hi \(name)!"
7   }
8   return "Hello there!"
9 }
10
11 print(createGreeting(name: "Toni")) // Hi Toni!
12 print(createGreeting())           // Hello there!

```

If the optional parameter `name` contains a value, the condition `let name = name` creates a new `name` variable that *shadows* the parameter and contains the unwrapped value of the optional. Note that the new `name` variable is only in scope within the *then clause* of the `if` statement.

On the other hand, if the optional parameter `name` is `nil`, the condition is considered to be false and the *then clause* of the `if` statement is not executed.

Optional Chaining

Sometimes we might want to access a property or a method of an optional without unwrapping it first. This can be done with *optional chaining*. An example of this is shown in Listing 3.14:

Listing 3.14: *Optional Chaining*

```

1 import Foundation // required for sqrt()
2
3 struct Vec2D {
4   var x: Double
5   var y: Double
6
7   func length() -> Double {
8     return sqrt(x * x + y * y)
9   }
10 }
11
12 var vec: Vec2D? = Vec2D(x: 2.5, y: 4.0)
13 let len = vec?.length() // len is of type Optional<Double>

```

The expression `vec?.length()` is an example of *optional chaining*. The type of this expression is `Optional<Double>`. If `vec` is `nil`, the value of the entire expression is `nil` as well. Otherwise, the value of the expression is the result of calling the method `length()`, wrapped in an optional.

Nil Coalescing Operator

The *nil coalescing operator* is a convenient way to provide a default value that can be used when an optional is `nil`. An example of this is shown in Listing 3.15:

Listing 3.15: *Nil Coalescing Operator*

```
1 print("Please enter your name:")
2 let name = readLine() ?? "user"    // name is of type String
3 print("Hello \(name)!")
```

The standard library function `readLine()` reads a line from `stdin`. The return type of this function is `Optional<String>`, because the input might be EOF (end of file) before a line can be read. In such a situation we can use the operator `??` (nil coalescing operator) in order to provide a default value. In the example above, the variable `name` is set to the name that was entered or to the default value `"user"` if `readLine()` returned `nil`.

Force Unwrapping

Force Unwrapping is another way to unwrap an optional. An example of this is shown in Listing 3.16:

Listing 3.16: *Force Unwrapping*

```
1 var optInt: Int? = 42
2 var x = optInt!    // x has type Int and value 42
3
4 optInt = nil
5 x = optInt!        // fatal error: unexpectedly found nil while unwrapping an Optional value
```

If the optional `optInt` contains a value, the expression `optInt!` evaluates to the unwrapped value of that optional. Otherwise, the program is aborted with a fatal error. Thus, this feature should only be used, if you are absolutely sure that an optional contains a value.

3.1.10 Extensions

Extensions are a way of adding new members (e.g., computed properties, methods, initializers) to an existing named type (i.e., structs, classes, enums and protocols). This feature serves two main purposes. Firstly, it allows programmers to add functionality to types that they don't control (e.g., types that are imported from an external framework or from the standard library). Secondly, it is sometimes useful to divide the declaration of your own types into multiple extensions in order to group the members a certain way or to spread the declaration across multiple files.

Most of Swift's core types such as `Int`, `Double`, `String` and `Array` are regular struct types that are declared in the standard library. This means that they can be extended as well. An example of this is shown in Listing 3.17:

Listing 3.17: *Extensions in Swift*

```

1 extension String {
2     func isPalindrome() -> Bool {
3         let lowercased = self.lowercased()
4         return lowercased == String(lowercased.characters.reversed())
5     }
6 }
7
8 print("Anna".isPalindrome()) // true
9 print("John".isPalindrome()) // false

```

In this example, the function `isPalindrome()` is added to the standard library type `String`. Of course we could provide this functionality without using an extension (e.g., by making `isPalindrome()` a free function that takes a `String`), but one could argue that the use of an extension makes the code more uniform, because it lets us treat `isPalindrome()` like any other method of the type `String`.

Another important feature of extensions is that they can add protocol conformance to an existing type. For example, let's assume that there is a new protocol called `Squarable` which requires a single method `squared()`. If we need to, we can make an existing type conform to this protocol through an extension. An example of this is shown in Listing 3.18:

Listing 3.18: *Extensions in Swift*

```

1 protocol Squarable {
2     func squared() -> Self
3 }
4
5 extension Int: Squarable {
6     func squared() -> Int {
7         return self * self
8     }
9 }
10
11 func printSquare(_ n: Squarable) {
12     print(n.squared())
13 }
14
15 printSquare(5) // 25

```

This is sometimes called *retroactive modelling* [App15].

3.1.11 Operators

Swift not only supports overloading of existing operators but it also provides the ability to declare entirely new operators. Most operators that are available in Swift by default, are declared in the standard library. Note that this works for unary (i.e., prefix or postfix) and binary (i.e., infix) operators but not for ternary operators. There is only one ternary operator and it is the conditional expression operator `a ? b : c` that is also common in other languages. This operator cannot be overloaded and we cannot add a new ternary operator.

Overloading an existing operator

To overload an existing operator, we declare a function that has the same name as the operator. This function can either be a free function or a static method on one of the operand types. For example, the standard library protocol `Equatable` has a single requirement, which says that any type that conforms to this protocol must overload the infix operator `==`. An example of this is shown in Listing 3.19:

Listing 3.19: *Overloading the == operator using a free function*

```

1 struct Point: Equatable {
2   var x: Int
3   var y: Int
4 }
5
6 func ==(lhs: Point, rhs: Point) -> Bool {
7   return lhs.x == rhs.x && lhs.y == rhs.y
8 }
9
10 let a = Point(x: 2, y: 2)
11 let b = Point(x: 1, y: 3)
12 print(a == b) // false
13 print(a != b) // true

```

This example declares a struct type called `Point` that conforms to `Equatable`. The `==` operator is implemented using a free function. The code `a == b` is translated by the compiler into the function call `(==)(a, b)`. The parentheses around `==` are required here, because otherwise the parser would parse it as a prefix operator instead of a function name. Also, note that an operator function never has external parameter names which is why we don't have to specify argument labels.

When a type conforms to `Equatable`, it automatically inherits a default implementation of the `!=` operator which simply calls `==` and negates the result. We could add our own implementation of `!=` that overrides the default behaviour (e.g., for performance reasons) but often this is not necessary.

Listing 3.20 shows how to implement the `==` operator using a static method instead of a free function:

Listing 3.20: *Overloading the == operator using a static method*

```

1 struct Point: Equatable {
2   var x: Int
3   var y: Int
4
5   static func ==(lhs: Point, rhs: Point) -> Bool {
6     return lhs.x == rhs.x && lhs.y == rhs.y
7   }
8 }
9
10 let a = Point(x: 2, y: 2)
11 let b = Point(x: 1, y: 3)
12 print(a == b) // false
13 print(a != b) // true

```

Declaring a new prefix / postfix operator

Sometimes we might want to add an operator that doesn't exist yet. For example, Listing 3.21 shows how to declare a new prefix operator called `||`:

Listing 3.21: *Declaring a new prefix operator*

```

1 import Foundation // required for sqrt()
2
3 struct Vec2D {
4     var x: Double
5     var y: Double
6 }
7
8 prefix operator ||
9
10 prefix func ||(vec: Vec2D) -> Double {
11     return sqrt(vec.x * vec.x + vec.y * vec.y)
12 }
13
14 let vec = Vec2D(x: 4, y: 3)
15 print(||vec) // 5.0

```

Since the prefix operator `||` does not exist in the standard library, we declare it with an operator declaration on line 8. Additionally, an operator function is implemented on lines 10–12. It defines the actual behaviour of the operator. In this example, the operator is used to determine the length of two-dimensional vectors (i.e., instances of `Vec2D`). Often there are multiple operator functions provided for the same operator each of which handles a different kind of operand. For example, we could add additional operator functions to handle three-dimensional vectors and so on.

Note that there are now two operators called `||`. One is a prefix operator that is used to compute the length of vectors (this is the operator from the example above) and the other is an infix operator that is used for logical disjunction (this operator is declared in the standard library). Thus, it is possible to have two operators with the same name as long as they have a different notation (i.e., prefix, infix, postfix). This is also the reason why the declarations of prefix and postfix operator functions need to have a corresponding `prefix` or `postfix` modifier.

Declaring a new infix operator

The precedence of prefix and postfix operators is defined through Swift's grammar. Postfix operators have a higher precedence than prefix operators which in turn have a higher precedence than infix operators. However, the precedence and associativity of individual infix operators in relationship to each other cannot be defined by the grammar because the operators are not yet known at parse time.

Thus, the declaration of a new infix operator can also specify the precedence and associativity of that operator. This is done through so-called precedence groups. Each infix operator belongs to a precedence group and each group can specify its associativity as well as its precedence in relation to other groups. Listing 3.22 shows an excerpt from the standard library which defines the operators `&&` and `||` along with the corresponding precedence groups and operator functions:

Listing 3.22: *Declaring a new infix operator*

```

1 precedencegroup LogicalDisjunctionPrecedence {
2     associativity: left
3     higherThan: TernaryPrecedence // not shown in this listing
4 }
5
6 precedencegroup LogicalConjunctionPrecedence {
7     associativity: left
8     higherThan: LogicalDisjunctionPrecedence
9 }
10
11 infix operator && : LogicalConjunctionPrecedence
12 infix operator || : LogicalDisjunctionPrecedence
13
14 extension Bool {
15     public static func && (lhs: Bool, rhs: @autoclosure () -> Bool) -> Bool {
16         return lhs ? rhs() : false
17     }
18
19     public static func || (lhs: Bool, rhs: @autoclosure () -> Bool) -> Bool {
20         return lhs ? true : rhs()
21     }
22 }

```

Both operators are left associative and the `&&` operator has higher precedence than the `||` operator, because the precedence group `LogicalConjunctionPrecedence` is higher than the precedence group `LogicalDisjunctionPrecedence`.

Note that the second operand of the two operator functions doesn't have type `Bool`. Instead, it has a function type that returns a `Bool` and is marked with the `@autoclosure` attribute. This means that the second argument can be any expression of type `Bool`. However, unlike other arguments this expression is not evaluated immediately. Instead, it is automatically wrapped in a closure that returns this expression. This allows for the implementation of short-circuiting operators such as the `&&` operator which doesn't evaluate its second argument if the first argument evaluates to `false`.

Built-in operators

Most operators are declared in the Swift standard library as described in the preceding sections. However, the following list describes a few infix operators that are built into the compiler and cannot be overloaded:

- **Assignment Operator**

The assignment operator `=` assigns the expression on the right hand side to the lvalue on the left hand side. The type checker ensures that the type of the expression is convertible to the type of the lvalue. An example of this is shown in Listing 3.23:

Listing 3.23: *Assignment Operator*

```

1 let x: Int
2 x = 10

```

Note that it is possible to declare a variable without an initializer expression. However, the variable doesn't have a default value and it cannot be used until it has been assigned a value.

- **Type Cast Operator**

There are three different type cast operators: `as`, `as?` and `as!`. The operator `as` is used for upcasting as shown in Listing 3.24:

Listing 3.24: *Type Cast Operator `as`*

```
1 class Animal {}
2 class Dog: Animal {}
3 class Cat: Animal {}
4
5 let cat = Cat()           // cat is of type Cat
6 let animal = cat as Animal // animal is of type Animal
```

The operator `as?` is used for downcasting. Since downcasting may fail at runtime, the result of the `as?` operator is an optional. Thus, if the expression on the left hand side cannot be downcasted to the type on the right hand side, the result is `nil`. An example of this is shown in Listing 3.25:

Listing 3.25: *Type Cast Operator `as?`*

```
1 class Animal {}
2 class Dog: Animal {}
3 class Cat: Animal {}
4
5 let animal: Animal = Cat() // animal has static type Animal
6 let cat = animal as? Cat    // cat is of type Optional<Cat>; cat != nil
7 let dog = animal as? Dog    // dog is of type Optional<Dog>; dog == nil
```

The operator `as!` is used for downcasting as well. However, in contrast to the `as?` operator, its result is not an optional. Instead, the `as!` operator aborts the program with a fatal error, if downcasting fails at runtime. An example of this is shown in Listing 3.26:

Listing 3.26: *Type Cast Operator `as!`*

```
1 class Animal {}
2 class Dog: Animal {}
3 class Cat: Animal {}
4
5 let animal: Animal = Cat() // animal has static type Animal
6 let cat = animal as! Cat    // cat is of type Cat
7 let dog = animal as! Dog    // results in fatal error
```

- **Type Check Operator**

The type check operator `is` can be used to determine whether a value is an instance of a certain subclass. An example of this is shown in Listing 3.27:

Listing 3.27: *Type Check Operator*

```
1 class Animal {}
2 class Dog: Animal {}
3 class Cat: Animal {}
4
5 func describeAnimal(_ animal: Animal) {
6     if animal is Dog {
7         print("This is a dog.")
8     } else if animal is Cat {
9         print("This is a cat.")
10    }
11 }
```

In the above example, the expression `animal is Dog` evaluates to `true`, if the parameter `animal` holds a reference to an instance of the subclass `Dog`.

- **Conditional Expression Operator**

The conditional expression operator `?:` is the only ternary operator in Swift. It takes a boolean condition as well as one expression for the *then clause* and one expression for the *else clause*. If the condition evaluates to `true`, the result of the overall expression is the expression of the *then clause*. Otherwise, the result of the overall expression is the expression of the *else clause*. Thus, the two expressions need to have the same type or they need to be convertible to a common supertype. An example of this is shown in Listing 3.28:

Listing 3.28: *Conditional Expression Operator*

```
1 var x = 42
2 print(x >= 0 ? "positive" : "negative")
```

3.1.12 Pattern Matching

In Swift, pattern matching can be used in `switch`, `if`, `guard`, `while` and `for` statements. There are different kinds of patterns that can be nested within each other to represent the structure of a single value or a composite value. The following sections describe the most common kinds of patterns:

Expression Pattern

An expression pattern consists of a single expression. An example is shown in Listing 3.29:

Listing 3.29: *Expression Pattern Example 1*

```
1 let x = 0
2 switch x {
3 case 42:
4     print("x is 42")
5 default:
6     print("x has some other value")
7 }
```

Here the pattern `42` in the `switch` case is an expression pattern that contains a nested integer literal expression. A value matches an expression pattern, if `pattern ~= value` evaluates to `true`. The pattern matching operator `~=` is a regular infix operator that can be overloaded like any other operator. In the example above, the expression `42 ~= x` is valid, because there is a generic overload of the `~=` operator in the standard library that works with all `Equatable` types and simply returns `pattern == value`.

Listing 3.30 shows a more interesting expression pattern:

Listing 3.30: *Expression Pattern Example 2*

```
1 let x = 25
2 if case 0...50 = x {
3     print("x is in the range 0...50")
4 }
```

In this example, `case 0...50 = x` is a so-called case condition. It matches the value `x` against the expression pattern `0...50`. In the expression `0...50` the `...` operator is a regular infix operator that creates a closed range from 0 to 50. The expression `0...50 ~= x` is valid, because there is an overload of the `~=` operator that takes a range and an element of a range, and returns `true`, if the element lies within the range.

Wildcard Pattern

A wildcard pattern consists of a single `_` and matches any value. An example is shown in Listing 3.31:

Listing 3.31: *Wildcard Pattern Example*

```
1 let x = "test"
2 switch x {
3   case _:
4     break
5 }
```

Since the pattern `_` in the example above always matches `x`, the switch statement doesn't need a `default` case.

Tuple Pattern

A tuple pattern consists of multiple tuple pattern elements. It matches a tuple, if the tuple has the same number of elements and each tuple element matches the corresponding tuple pattern element. An example of this is shown in Listing 3.32:

Listing 3.32: *Tuple Pattern Example*

```
1 let point = (5, 11)
2 switch point {
3   case (0...10, 0...10):
4     print("x and y are in range 0...10")
5   case (0...10, _):
6     print("x is in range 0...10")
7   case (_, 0...10):
8     print("y is in range 0...10")
9   default:
10    print("no element is in range 0...10")
11 }
```

In this example, different tuple patterns are used to check whether both, either or none of the elements of the tuple `point` lie in the range `0...10`. For the tuple `(5, 11)` the output is "x is in range 0...10". Note that the tuple patterns in this example contain nested expression patterns and wildcard patterns.

Value-Binding Pattern

A value-binding pattern consists of the keyword `let` or `var` followed by a nested pattern. Any identifier that occurs within the nested pattern of a value-binding pattern is considered to be an identifier pattern. If pattern matching succeeds, a new local variable

is created for each identifier pattern and initialized with the corresponding sub value of the matched value. An example of this is shown in Listing 3.33:

Listing 3.33: *Value-Binding Pattern Example*

```
1 let point = (1, 1)
2 switch point {
3   case let (x, y) where x == y:
4     print("\(x) == \(y)")
5   case let (x, y):
6     print("\(x) != \(y)")
7 }
```

In the first switch case, the pattern `let (x, y)` binds the two elements of the tuple `point` to the new variables `x` and `y`. The additional `where` clause ensures that the pattern only matches if `x` and `y` are equal. The second switch case matches all remaining cases.

Optional Pattern

An optional pattern matches an optional if it is not equal to `nil` and if the wrapped value matches the nested pattern of the optional pattern. An example of this is shown in Listing 3.34:

Listing 3.34: *Optional Pattern Example*

```
1 let numbers = [1, 2, nil, 4, nil, nil, 7] // numbers is of type Array<Optional<Int>>
2 for case let number? in numbers {
3   print(number) // number is of type Int
4 }
```

In this example, pattern matching is used to loop over an array of optionals while ignoring the elements that are `nil`. The pattern `let number?` first checks whether the current element is not `nil`. If that is the case, it unwraps the optional and uses the wrapped value to initialize a new variable called `number`. Thus, the `for` loop prints the numbers 1, 2, 4 and 7.

Enum Case Pattern

An enum case pattern can be used to match an instance of an enum type. If the corresponding enum case has associated values, nested patterns can be used to match these values as well. An example of this is shown in Listing 3.35:

Listing 3.35: *Enum Case Pattern Example*

```
1 enum Result<T> {
2   case success(T)
3   case error
4 }
5
6 func handleResult<T>(_ result: Result<T>) {
7   switch result {
8     case .success(let value):
9     print("Result is \(value)")
10    case .error:
11    print("Error")
12  }
13 }
```


The pattern `.success(let value)` matches the enum case `Result.success` and binds the associated value to the new variable `value`. Note that we can write `.success` instead of `Result.success`, because the enum type is known from the context (i.e., from the control expression of the `switch` statement).

3.1.13 Protocol-Oriented Programming

Protocol-Oriented Programming is a programming technique that facilitates the writing of reusable code without relying on inheritance. The term was first introduced by Apple's Dave Abrahams in a WWDC session in 2015 [App15]. It uses a feature called protocol extensions and is used heavily in the Swift standard library. To explain how it works, Listing 3.36 shows an example:

Listing 3.36: *Protocol-Oriented Programming in Swift*

```

1  extension Sequence where Iterator.Element: Equatable {
2      func countOccurrencesOfElement(_ element: Iterator.Element) -> Int {
3          var count = 0
4          for x in self {
5              if x == element {
6                  count += 1
7              }
8          }
9          return count
10     }
11 }
12
13 let array = [1, 5, 2, 2, 8, 1, 2, 4, 2]
14 print(type(of: array), array.countOccurrencesOfElement(2))    // prints 'Array<Int> 4'
15
16 let set: Set<Int> = [1, 5, 2, 2, 8, 1, 2, 4, 2]
17 print(type(of: set), set.countOccurrencesOfElement(2))        // prints 'Set<Int> 1'
18
19 let chars = "hello, world!".characters
20 print(type(of: chars), chars.countOccurrencesOfElement("l"))  // prints 'CharacterView 3'

```

In the example, we extend the `Sequence` protocol by adding a new method called `countOccurrencesOfElement()`. To do its job, the method needs to be able to check whether two elements of a sequence are equal. That is why there is a `where`-clause constraint on the protocol extension which makes sure that the extension only applies if the elements of the sequence conform to the `Equatable` protocol. Further down, the example shows that we can now use the method on various sequence types such as `Array`, `Set` and the `CharacterView` of a `String`.

3.1.14 Memory Management

Swift uses automatic reference counting (ARC) to manage memory. ARC keeps track of the number of references that refer to an object. As soon as this reference count goes to 0, the object is destroyed and the corresponding memory deallocated. This means that there are no garbage collector pauses and memory is deallocated at a deterministic point in time. However, it also means that programmers occasionally need to be careful about memory management and e.g., resolve a strong reference cycle by introducing a weak reference [tsp17b].

3.1.15 Interoperability with Objective-C and C

For almost 20 years, Objective-C has been the main programming language used to develop software for Apple’s platforms. It is a superset of the C programming language and can therefore directly call C functions. Since most of Apple’s frameworks are written either in Objective-C or in C, the transition to Swift cannot be done at the flick of a switch. Thus, it is important that they are mostly interoperable in order to make it possible to translate large code bases one file at a time [App17f]. Now that Swift is becoming more and more mature, Apple has begun to make the frameworks “swiftier”. For example, most of the types in the `Foundation` framework are now available in Swift as value types [App16].

3.2 Overview

This section describes the features that are expected from a modern IDE and outlines the high-level components which are required to implement these features.

3.2.1 Features

The following list describes features that are commonly expected from a modern IDE. If implemented well, they can have a tremendous impact on programmer productivity.

- **Code Presentation**

They way code is presented to the programmer can matter a lot. For example, a text editor with a monospace font and support for *syntax highlighting* is usually much more convenient for reading and writing code than a general word processor application. Similarly, many IDEs provide some sort of *outline view* which lists all the types and functions defined in a file in order to give a quick overview of the file content. Another feature that may be worth considering is *code folding*, which allows the programmer to temporarily hide sections of the code.

- **Editing Assistance**

During editing it can be very useful to get some assistance from the IDE. For example, it may automatically reindent code according to configurable formatting rules, insert closing parentheses or display a list of auto-complete suggestions.

- **Error Reporting**

Many IDEs parse the compiler output and display any errors or warnings as problem markers directly in the source code editor. This makes it easier for programmers to quickly grasp the location and the cause of the problem.

Additionally, IDEs may perform some syntactic and semantic analysis already during editing. This can be very important, especially for large projects where compilation may take a while.

- **Code Navigation**

Large software projects may consist of hundreds of source files. Manually navigating through such code bases can be very tedious. Thus, most IDEs have some code navigation features such as “Jump to Definition” or “Open Call Hierarchy”.

- **Code Rewriting**

Another useful feature is the ability to make automatic changes to the code. This may come in the form of a list of predefined refactorings (e.g., *Extract Method*, *Inline Temp*, etc.) [Mar99], but it is also useful to provide quick-fixes for small problems that may occur during editing. For example, the IDE could offer to transform a local variable into a constant if it is not modified anywhere.

- **Program Compilation**

A major difference between a normal text editor and an IDE is that IDEs are capable of building, executing and debugging programs. The programmer also usually expects the option to configure the commands that are used to build a program in order to customize the build process. The output of the builder should be displayed to the user so that it is possible to diagnose build problems.

- **Program Execution**

After a project has been compiled, the user usually wants to launch the executable from within the IDE. The IDE should display the standard output that is printed by the process and the user should be able to provide input, if the process asks for it. Additionally, there should be some options to configure how the program is launched (e.g., setting the program arguments / environment variables, etc.).

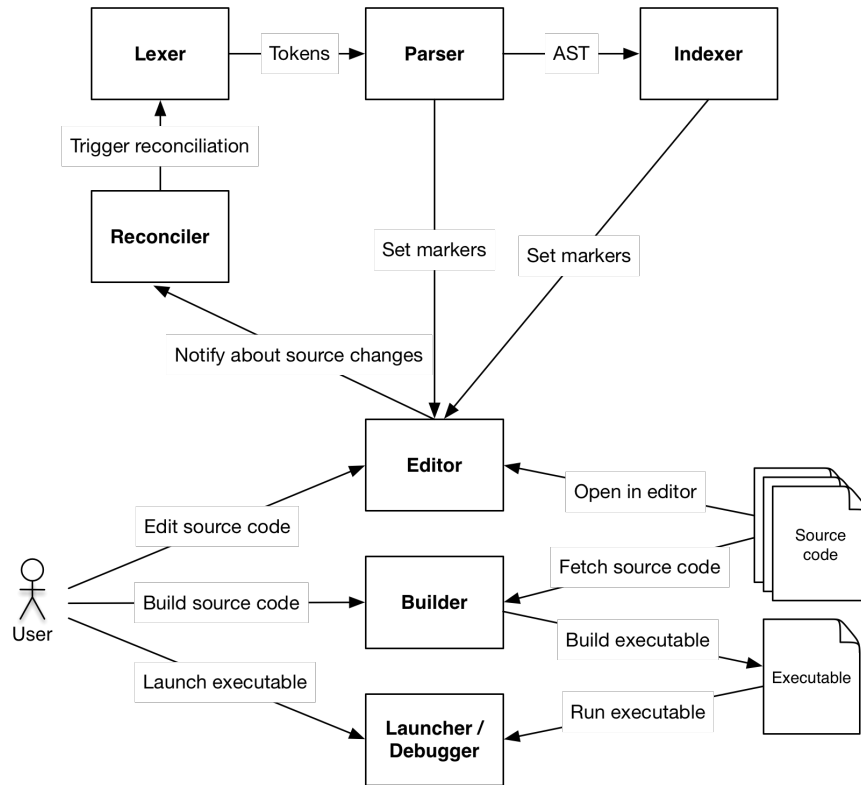
- **Debugging**

The IDE should have the ability to launch the executable with a debugger and allow the user to set breakpoints, step through the statements of the program and inspect the values of the variables that are currently in scope.

3.2.2 Components

Figure 3.1 shows a high-level overview of the most important components that are required to implement the features listed in subsection 3.2.1. In the following subsections, each component is described in more detail. Note that this is a simplified view of the whole system and that each component itself consists of various subcomponents.

Figure 3.1: High-level overview



Editor

The editor is the main interface through which the user interacts with the IDE. It is responsible for displaying the code in a syntax-highlighted form and it needs various other features such as marker annotations (for error reporting) and hyperlinking (for code navigation).

Reconciler

Whenever the user edits the source code, the reconciler is notified about those changes. It waits until there is a short pause in the editing process (e.g., 500ms), in which case it starts the reconciliation in a background thread. During reconciliation, the internal model of the source code (abstract syntax trees and index) is updated and any errors or warnings are displayed in the editor.

Lexer

The lexer is responsible for turning the source code (a stream of characters) into a stream of tokens. It doesn't do any syntactic or semantic analysis but merely groups characters that belong together according to the language's lexical structure.

Parser

The parser takes the stream of tokens emitted by the lexer and builds an abstract syntax tree (AST) according to the language's grammar rules. In the process, it shows problem markers in the editor for any syntax errors that it finds during parsing.

Indexer

The indexer is responsible for the semantic analysis of the code. It traverses the ASTs of the source files in the project and builds an index. The index is a symbol table and can be used to implement features such as auto-completion, "Jump to Definition" and "Open Call Hierarchy".

Builder

The user can trigger the builder to compile the source code into an executable or a library. Most IDEs use an external compiler to build the project. In that case the builder is responsible for the correct invocation of the compiler and for presenting the build output to the user.

Launcher / Debugger

The executable can be launched after the project has been built successfully. The user can also customize the run configuration to set things like the program arguments or the environment variables.

When the user starts the executable in the debugging mode, the launcher launches the executable with a debugger (e.g., LLDB) and lets the user control it during the debugging session.

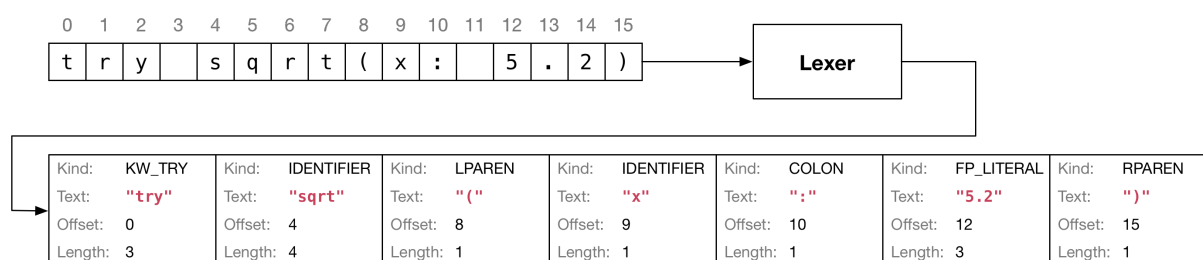
3.3 Conclusion

This chapter has identified the components that are required to implement a modern IDE. The chapters that follow describe how some of these components have been implemented during the project.

4 Lexer

A lexer turns a stream of characters into a stream of tokens. This makes it easier for the parser to check the syntax and to build an AST (abstract syntax tree). Each token stores its kind, text, offset and length. Figure 4.1 illustrates this process:

Figure 4.1: *Lexing Process*



The token kinds of the Swift programming language are defined in its lexical structure [tsp17d]. The following subsections give an overview of the different token kinds.

4.1 Keywords

Keywords are special words that are reserved by the programming language and cannot be used as identifiers.

Swift 3 has 57 keywords such as **if**, **let**, **var** and **struct**. These are reserved in almost all contexts and therefore cannot be used to name a program entity (e.g., a variable or a class). There is one notable exception. All keywords except **inout**, **var** and **let** can be used as parameter names in a function declaration. This allows for more natural function calls, because external parameter names such as **in** and **for** are valid. An example of this is shown in Listing 4.1:

Listing 4.1: *Using keyword `in` as parameter name*

```

1 func find<T: Equatable>(element: T, in xs: [T]) -> Array<T>.Index? {
2     for (i, x) in xs.enumerated() {
3         if x == element {
4             return i
5         }
6     }
7     return nil
8 }
9
10 if let i = find(element: 5, in: [1, 2, 3, 4, 5, 6, 7, 8]) {
11     print("Found number 5 at index \(i)")
12 }

```

Additionally, there are 26 keywords (e.g., `willSet`, `didSet`, etc.) that are only reserved in particular contexts. These are treated as identifiers by the lexer because it doesn't have enough context to decide whether it should be a keyword instead. It is the parser's job to make this distinction later.

Finally, there are 15 keywords that start with a number sign (`#`) such as `#line`, `#function`, `#if`, etc. Note that these are not preprocessor macros, because Swift doesn't have a preprocessor. Instead, they are special compiler directives to do conditional compilation or to log things such as the current file, line or function for debugging purposes.

4.2 Identifiers

Identifiers are used to give names to the entities of a program. This includes variables, properties and functions as well as custom types such as classes, structs, enums and protocols. In many programming languages the characters in an identifier are limited to characters from the English alphabet, the digits 0-9 and the underscore (`_`).

In comparison, identifiers in Swift can contain most characters from Unicode's Basic Multilingual Plane and even some characters from the supplementary planes (e.g., emoji) [uni17]. Listing 4.2 shows an example of valid Swift code that uses the Greek letters α and π as identifiers:

Listing 4.2: *Identifiers in Swift*

```

1 let sectorArea = 9.8125
2 let radius = 5.0
3 let  $\pi$  = 3.14
4 let  $\alpha$  = sectorArea * 360.0 / ( $\pi$  * radius * radius)
5 print(" $\alpha$  = \( $\alpha$ )")

```

Words that are keywords but also match the rules of an identifier are treated as keywords by default. However, in Swift it is possible to use any keyword as an identifier by wrapping it in backticks as shown in Listing 4.3:

Listing 4.3: *Keyword as identifier*

```

1 let `protocol` = "https"
2 print(`protocol`)

```

4.3 Operators

In Swift, most of the basic types and operators are not actually part of the language but are predefined in the standard library. For example, the type `Bool` is a struct type defined in the standard library. Since structs have value semantics in Swift, instances of `Bool` behave as expected. Similarly, the logical operators `&&`, `||` and `!` are also defined in the standard library. A full list of all the operators can be found in the *Swift Standard Library Operators Reference* [App17d]. Programmers can overload these predefined operators for their own types. However, it is even possible to define completely new operators. Listing 4.4 shows how to define a custom power operator:

Listing 4.4: *Custom power operator*

```

1 infix operator **: MultiplicationPrecedence
2
3 func ** (x: Int, y: Int) -> Int {
4     var result = 1
5     for _ in 0..

```

On line 1 of the listing, a new infix operator called `**` is declared. Additionally, the operator is added to the precedence group `MultiplicationPrecedence`. This means that it has the same associativity and precedence as the `*` operator. On line 3 of the listing, an operator function for the power operator `**` is declared.

In order to determine if a given operator application is syntactically correct, the parser needs to know whether the operator is a prefix, infix or postfix operator. Thus, the lexer needs to encode that information into the operator tokens it emits. It differentiates between the three cases based on the characters before and after the operator. Listing 4.5 shows an example:

Listing 4.5: *Lexing / Parsing of Operators*

```

1 prefix operator +++
2 infix operator +++
3 postfix operator +++
4
5 prefix func +++(x: Int) -> String { return "prefix +++" }
6
7 func +++(x: Int, y: Int) -> String { return "infix +++" }
8
9 postfix func +++(x: Int) -> String { return "postfix +++" }
10
11 let x = 0
12 let y = 1
13
14 print(++x)      // 1) prints 'prefix +++'
15 print(x+++y)    // 2) prints 'infix +++'
16 print(x +++ y)  // 3) prints 'infix +++'
17 print(x+++)     // 4) prints 'postfix +++'
18 print(+++ x)    // 5) syntax error
19 print(x ++++)   // 6) syntax error
20 print(x+++ y)   // 7) syntax error
21 print(x ++++y)  // 8) syntax error
```


In this example, the custom operator `+++` is declared to be a prefix, an infix as well as a postfix operator. Each operator has its own operator function which simply returns a `String` that describes the operator. Whether a given operator is interpreted as a prefix, infix or postfix operator, depends on the spacing between the operator and its operand(s). The following list explains each case:

1. The operator `+++` appears before the operand `x` and there is no whitespace in between. Thus, it is treated as a prefix operator.
2. The operator `+++` appears between the two operands `x` and `y` with no spacing. In this case `+++` is treated as an infix operator.
3. The operator `+++` appears between the two operands `x` and `y` with a space on each side of the operator. Again, `+++` is treated as an infix operator.
4. The operator `+++` appears after the operand `x` and there is no whitespace in between. Thus, it is treated as a postfix operator.
5. In this case the lexer generates an infix operator token for `+++` and an identifier token for `x`. That then leads to a syntax error during parsing, because infix operators expect two operands and not just one.
6. The lexer generates an identifier token for `x` and an infix operator token for `+++`. Again, this leads to a syntax error during parsing, because infix operators expect two operands.
7. The lexer first generates an identifier token for `x` and a postfix operator token for `+++`. So far this would be syntactically correct, but since it is followed by an additional identifier token for `y`, the parser reports a syntax error.
8. This is a similar problem where an operator appears between two operands, but the operator isn't treated as an infix operator because of the inconsistent spacing around it.

There is another rule that is worth mentioning. In general, operators cannot contain periods (`.`). This makes it easy to access a member of the result of a postfix operator expression as shown in Listing 4.6:

Listing 4.6: *Accessing member of postfix operator expression*

```

1 postfix operator -!-
2 postfix func -!-(s: String) -> String {
3     return s.lowercased()
4 }
5
6 let greeting = "Hello, World"
7 for c in greeting-!-.characters {
8     print(c)
9 }
```

Otherwise, it would be impossible for the lexer to know, whether this is an infix operator `-!-` or a postfix operator `-!-` followed by a period. However, there is an exception to this rule. If an operator starts with a period, it can also contain periods in the rest of the operator. This allows for operators like Swift's closed range operator (e.g., `0...10`).

4.4 Literals

Swift supports 8 different types of literals. The first 6 are described in this section. The others (array literals and dictionary literals) consist of multiple tokens and therefore don't belong into the lexer section.

4.4.1 Integer Literals

Integer literals are used to express integral numbers and can be written in four different numeral systems (binary, octal, decimal and hexadecimal). Examples are shown in Listing 4.7:

Listing 4.7: *Integer literals in Swift*

```
1 print("Binary 0b10101010 == \"(0b10101010)") // prefix 0b => binary numeral system
2 print("Octal 0o252 == \"(0o252)")           // prefix 0o => octal numeral system
3 print("Decimal 170 == \"(170)")              // no prefix => decimal numeral system
4 print("Hexadecimal 0xAA == \"(0xAA)")        // prefix 0x => hexadecimal numeral system
```

For better readability Swift allows the use of underscores within integer literals (e.g., 200_000, 0xAA_BB_CC, 0o111_222). These underscores do not affect the value of the literal but merely serve as visual separators.

The default inferred type of an integer literal is the Swift standard library type `Int`, which represents a 32-bit or a 64-bit signed integer value type depending on the architecture of the current platform [tsp17e]. If the literal doesn't fit into the inferred type, the compiler emits an error.

4.4.2 Floating-Point Literals

Floating-point literals are used to express floating-point numbers. There are decimal and hexadecimal floating-point literals. Listing 4.8 shows examples for both forms:

Listing 4.8: *Floating-point literals in Swift*

```
1 // decimal floating-point literals
2 print("Decimal fp-literal with fraction 99.5 == \"(99.5)") // 99.5
3 print("Decimal fp-literal with exponent 123e4 == \"(123e4)") // 123 * (10 ^ 4)
4 print("Decimal fp-literal with fraction & exponent 765.4e-3 == \"(765.4e-3)") // 765.4 * (10 ^ -3)
5
6 // hexadecimal floating-point literals
7 print("Hexadecimal fp-literal with exponent 0xFFp2 == \"(0xFFp2)") // 255 * (2 ^ 2)
8 print("Hexadecimal fp-literal with fraction & exponent 0x12.Ap4 == \"(0x12.Ap4)") // 18.625 * (2 ^ 4)
```

Note that it is not possible to leave out the exponent for a hexadecimal floating-point literal. This is because it could lead to confusing code like `print(0x123.beef)`. Does this code print a floating-point number or the result of accessing the property `beef` of the integer literal `0x123`? It could be both, so the compiler reports an error in such ambiguous cases.

The default inferred type of a floating-point literal is the Swift standard library type `Double`, which represents a 64-bit floating-point number. Note that there are also the types `Float` to represent 32-bit floating-point numbers and `Float80` to represent 80-bit floating-point numbers.

4.4.3 String Literals

Swift supports two kinds of string literals. There are *static string literals* which look similar to the string literals of other languages (e.g., "hello"). Additionally, there are so-called *interpolated string literals*. These can contain expressions that are evaluated at run time and concatenated with the rest of the string. An example of this is shown in Listing 4.9:

Listing 4.9: *Interpolated string literal in Swift*

```
1 import Foundation
2
3 let x = 36.0
4 print("sqrt(x) == \(sqrt(x))")           // string interpolation
5 print("sqrt(x) == " + String(sqrt(x)))    // string concatenation
```

On line 4, string interpolation is used. On line 5, the same result is achieved with string concatenation. Additionally, both kinds of string literals may contain any of the escape sequences listed in Table 4.1:

Table 4.1: *Escape sequences*

Description	Escape Sequence
Null Character	\0
Backslash	\\
Horizontal Tab	\t
Line Feed	\n
Carriage Return	\r
Double Quote	\"
Single Quote	\'
Unicode scalar	\u{...}

The escape sequence for Unicode scalars \u{...} takes between one and eight hexadecimal digits that denote a Unicode code point.

4.4.4 Boolean Literals

The boolean literals consist of the two keywords **true** and **false**. If any of these values is assigned to a variable or a constant, its type is inferred as a `Bool`.

4.4.5 Nil Literal

The keyword `nil` is a literal that is used to denote that an optional does not have a value. An example is shown in Listing 4.10:

Listing 4.10: *nil* literal in Swift

```

1 let x: Int? = nil
2
3 if let x = x {
4     print("x = \(x)")
5 } else {
6     print("x is nil")
7 }

```

Note that the type annotation for the variable `x` is required, because the compiler cannot infer a type from `nil`.

4.4.6 Compiler Literals

There are four keywords that start with a number sign (`#`) and that can be used as literal expressions (`#file`, `#line`, `#column` and `#function`). They are useful for logging and debugging purposes. An example is shown in Listing 4.11:

Listing 4.11: *Compiler literals* in Swift

```

1 func f() {
2     print("This literal appears in file \(#file).")
3     print("This literal appears on line \(#line).")
4     print("This literal appears in column \(#column).")
5     print("This literal appears in function \(#function).")
6 }
7
8 f()

```

4.5 Punctuation

The following tokens are reserved for punctuation and cannot be used as custom operators: `(`, `)`, `{`, `}`, `[`, `]`, `.`, `,`, `:`, `;`, `=`, `@`, `#`, `->`, ``` and `?`.

Additionally, the `&` character cannot be used as a prefix operator and the `!` character cannot be used as a postfix operator.

4.6 Comments

Swift supports comments that extend to the end of the current line (single-line comments) and comments that can spread over several lines (multi-line comments). Comments are ignored by the compiler and could therefore already be discarded by the lexer. However, for advanced IDE functionality such as refactoring, we may want to keep them around, in order to be able to perform valid code transformations. Thus, the lexer defines a separate token kind for comments.

4.6.1 Single-line Comments

Single-line comments begin with a `//` and extend to the end of the current line. Listing 4.12 shows an example:

Listing 4.12: *Single-line comments in Swift*

```
1 print("Hello, ...") // a single-line comment
2 print("... world!")
```

4.6.2 Multi-line Comments

Multi-line comments begin with a `/*` and end with a `*/`. In contrast to some other languages such as Java and C++, multi-line comments in Swift can be nested within each other as shown in Listing 4.13:

Listing 4.13: *Multi-line comments in Swift*

```
1 /*
2   a multi-line
3   comment
4
5   /* a nested multi-line
6       comment */
7 */
8 print("Hello, world!")
```

4.7 Implementation Status

For the most part, the lexer works as expected and there aren't a lot of changes coming in Swift 3 that are relevant to this component. However, there are two known issues that need to be addressed in the future.

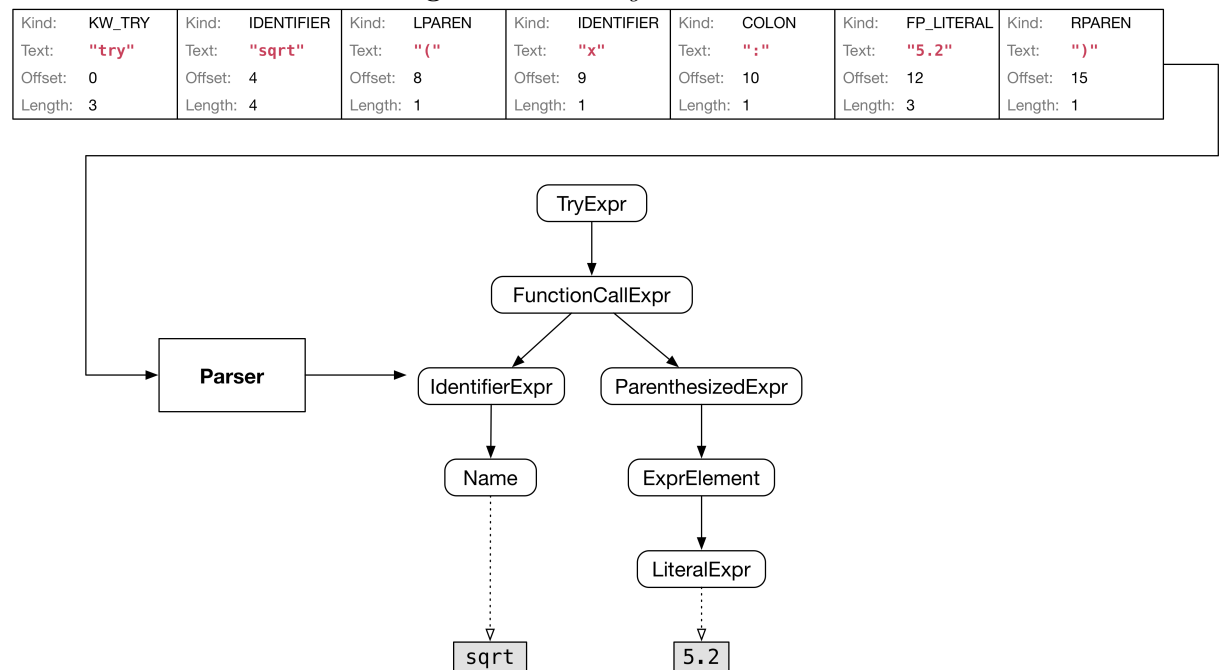
For example, the lexer does not yet support unicode characters that take up more than 1 character in the UTF-16 encoding which Java uses to encode strings. This means for example, that it cannot correctly recognize identifiers that contain emoji.

The second issue is related to interpolated string literals. At the moment, the lexer emits a single string token for the entire interpolated string literal. It would be better to generate individual tokens for the expressions used within the literal. This way the parser and the indexer can analyze that code as well and report syntax and semantic errors if there are any.

5 Parser

The parser consumes the tokens generated by the lexer and uses the rules of the Swift grammar to verify the syntax of the program. At the same time it builds an AST (abstract syntax tree). The AST is an intermediate representation of the code that can later be traversed to perform semantic analysis, type checking, etc. Figure 5.1 shows the AST that results from the tokens for the expression `try sqrt(x: 5.2)`:

Figure 5.1: *Parsing Process*



5.1 Architecture

Tifig uses a recursive-descent parser with arbitrary lookahead and support for backtracking [Alf06]. Its architecture is very much influenced by the patterns in the book *Language Implementation Patterns* by Terence Parr [Ter10]. Most Swift code can be parsed with only one or two tokens of lookahead. However, there are a few situations that require the ability to speculatively parse code and to backtrack if necessary. This section gives an overview of the parser architecture and explains recursive-descent parsing and backtracking in more detail.

5.1.1 Parser Modules

Swift is a general-purpose programming language with relatively many language features. Thus, it is best not to implement the whole parser in one large and complex class. For this reason, the parser has been split up into six modules, each of which is responsible for parsing a group of related language elements. They are the same groups as the ones used in Swift’s official *Language Reference* [tsp17c]: declarations, statements, expressions, patterns, types and attributes. Each group is described in the following list:

- **Declarations**

Declarations introduce new names into a program. They can be used to declare new named objects (e.g., variables, constants, functions, etc.) or new named types (e.g., classes, structs, enums, etc.). Additionally, you can also use a declaration to extend the behaviour of an existing named type (with extension declarations) and to import external symbols into your program (with import declarations). Listing 5.1 shows an example of a struct declaration with two nested property declarations:

Listing 5.1: *Declarations in Swift*

```
1 struct Point {
2     let x: Int
3     let y: Int
4 }
```

- **Statements**

In an imperative programming language like Swift, statements are the instructions that are executed when the program is running. There are simple statements (e.g., declaration statements and expression statements) and there are statements that influence the control flow of the program (e.g., if statements, loop statements, return statements, etc.). Additionally, there are compiler-control statements which can be used to conditionally compile parts of the code. Listing 5.2 shows a for loop statement. The loop’s body is a code block that contains an expression statement in this example:

Listing 5.2: *Statements in Swift*

```
1 for i in 1...10 {
2     print("i = \(i)")
3 }
```

A statement can optionally be terminated with a semicolon (;). However, this is only required if two statements appear on the same line.

- **Expressions**

In Swift, there are four groups of expressions. *Primary expressions* are things like identifier expressions (e.g., referring to a variable / constant) and literal expressions (e.g., numeric literals, array literals, etc.).

Postfix expressions are things like function call expressions, subscript expressions or the application of a postfix operator to a primary expression. Postfix expressions are a superset of primary expressions. Thus, primary expressions by themselves are also considered to be postfix expressions.

Prefix expressions are things like the application of a prefix operator to a postfix expression or inout expressions (used for the argument supplied for an inout parameter). Prefix expressions are a superset of postfix expressions. Thus, postfix expressions by themselves are also considered to be prefix expressions.

Finally, *binary expressions* combine one or more prefix expressions using binary operators. Since a binary expression can consist of a single prefix expression, binary expressions are a superset of prefix expressions.

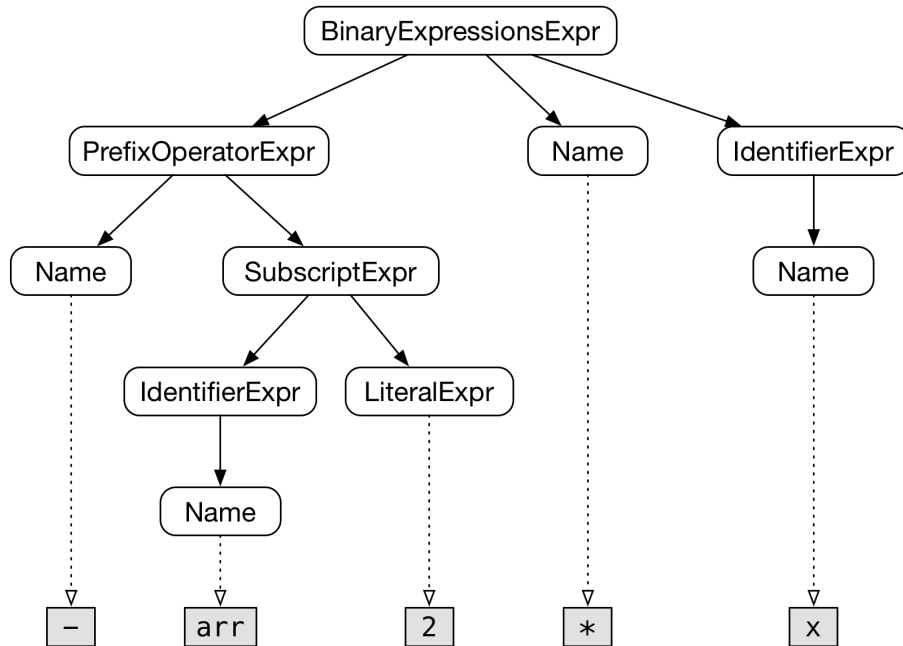
Expressions can be evaluated to a value and have a type. They can be used in various places (e.g., expression statements, variable initializers, if conditions, etc.). As an example, let's look at the expression that is printed out in Listing 5.3:

Listing 5.3: *Expressions in Swift*

```
1 let arr = [1, 2, 3, 4]
2 let x = 10
3 print(-arr[2] * x)
```

A slightly simplified AST for that expression is shown in Figure 5.2:

Figure 5.2: *AST for expression `-arr[2] * x`*



Note that child nodes are evaluated before their parents. Thus, primary expressions (e.g., `arr`, `2`, `x`) are evaluated first. Then the postfix expressions (e.g., `arr[2]`) are evaluated and after that the prefix expressions (e.g., `-arr[2]`). Finally, binary expressions are evaluated (e.g., `-arr[2] * x`).

Therefore, postfix operators have a higher precedence than prefix operators which in turn have a higher precedence than infix operators. If multiple binary operators and operands appear on the same level, the precedence and associativity of the individual operators determines the order of evaluation.

- **Patterns**

Patterns represents the structure of a single value or a composite value. One pattern can be matched with many different values that have the same structure. For example, the tuple pattern `(_, 2)` matches any two-element tuple (pair) whose second element is the integer value 2. Patterns can also be used to extract parts of a composite value. For example the value-binding pattern `let (x, y)` binds the two elements of a pair to the constants `x` and `y`.

Patterns can appear in several places (e.g., variable declarations, switch statements, for loops, etc.). As an example, Listing 5.4 shows how patterns can be used to match certain value structures and to extract information from a composite value:

Listing 5.4: *Patterns in Swift*

```

1 let point = (0, 42)
2
3 switch point {
4     case (0, 0):
5         print("Point is at the origin.")
6     case (let x, 0):
7         print("Point is on x-axis at offset \(x).")
8     case (0, let y):
9         print("Point is on y-axis at offset \(y).")
10    default:
11        print("Point is not on an axis.")
12 }
```

- **Types**

Swift differentiates between *named types* and *compound types*. Named types are introduced through a type declaration (e.g., classes, enums, structs and protocols). Basic types such as `Int` and `Double` are declared in the standard library in the form of named struct types. Struct types cannot be subclassed, but like all named types, they can be extended with extensions.

Compound types don't have a name and they are defined by the language itself. There are two kinds of compound types in Swift: function types and tuple types. Compound types cannot be extended with extensions.

Listing 5.5 shows examples of different kinds of type annotations in Swift:

Listing 5.5: *Types in Swift*

```

1 import Darwin
2
3 let a: Int = 0 // Int is a named type
4 let b: [Int] = [1, 2, 3] // [Int] is syntactic sugar for the named type Array<Int>
5 let c: (Int, String) = (5, "Test") // (Int, String) is a tuple type
6 let d: (Double, Double) -> Double = pow // (Double, Double) -> Double is a function type
```

- **Attributes**

Attributes are used to provide more information about a declaration or a type. Listing 5.6 shows an example of the declaration attribute `@discardableResult` which specifies, that the compiler should not emit a warning if the function `writeToFile()` is called and its return value is discarded:

Listing 5.6: Declaration attribute `@discardableResult`

```

1 @discardableResult
2 func writeToFile(str: String) -> Int {
3     // write to file
4     return bytesWritten
5 }
6
7 writeToFile(str: "File content")    // no compiler warning

```

Additionally, attributes may have arguments as shown in Listing 5.7:

Listing 5.7: Declaration attribute `@available`

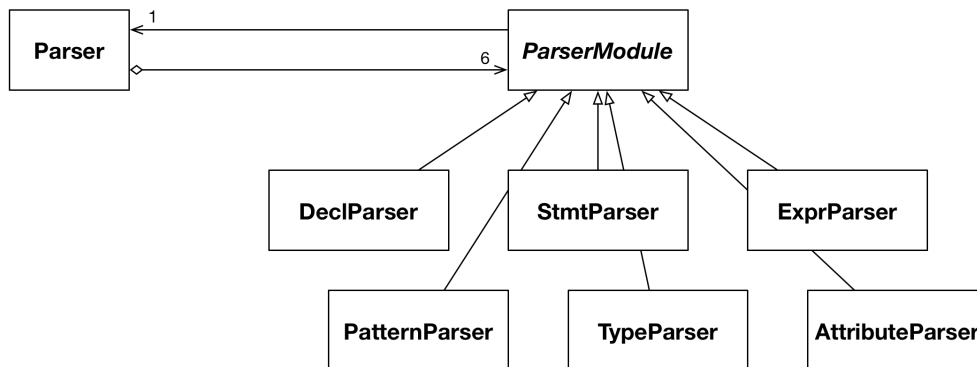
```

1 @available(iOS 9.0, OSX 10.11, *)
2 class MyClass {
3     // class definition
4 }

```

The attribute `@available` has two arguments (iOS 9.0 and OSX 10.11) which specify the operating system versions with which the class was introduced.

Figure 5.3 shows an overview of the parser architecture with its modules. Note that throughout the project the abbreviations `Decl`, `Stmt` and `Expr` are used for the terms declaration, statement and expression, respectively.

Figure 5.3: Parser modules

The individual parser modules inherit from a common, abstract superclass `ParserModule`. An instance of `Parser` aggregates one instance of each of the six parser modules. The individual modules need to be able to talk to each other and to the parser which maintains the parse state. This is why the class `ParserModule` has a reference back to the class `Parser`.

5.1.2 Recursive-Descent Parsing

Recursive-descent parsing is a top-down parsing technique in which each production from the language grammar is implemented with a separate method. For each non-terminal on the right hand side of a production, the corresponding method is called. For each terminal, the parser matches the current token with the expected token and consumes it [Alf06].

In order to illustrate this process, let's look at an example using the production rule for if statements:

$$\begin{aligned} \textit{if-statement} &\rightarrow \mathbf{if} \textit{ condition-list code-block else-clause}_{opt} \\ \textit{else-clause} &\rightarrow \mathbf{else} \textit{ code-block} \mid \mathbf{else} \textit{ if-statement} \end{aligned}$$

Terminals are written in bold and the non-terminal *else-clause* is marked as optional using the *opt* subscript. Note that the two production rules *if-statement* and *else-clause* are mutually recursive, which means that they are defined in terms of each other. This is valid and quite common in language grammars. Listing 5.8 shows the implementation of the *if-statement* production rule in the parser module `StmtParser`:

Listing 5.8: Implementation of production rule for if statements

```

1 private IfStmt ifStmt() throws RecognitionException {
2     match(Kind.KW_IF);
3
4     final ConditionList conditionList = parse(this::conditionList);
5     final CodeBlock thenBlock = parse(this::codeBlock);
6
7     CodeBlock elseBlock = null;
8     IfStmt elseIfStmt = null;
9     if(la(1).is(Kind.KW_ELSE)) {
10         match(Kind.KW_ELSE);
11         if(la(1).is(Kind.KW_IF)) {
12             elseIfStmt = parse(this::ifStmt);
13         } else {
14             elseBlock = parse(this::codeBlock);
15         }
16     }
17     return new IfStmt(conditionList, thenBlock, elseBlock, elseIfStmt);
18 }
```

Note that the name `la` in the method call `la(1)` is an abbreviation for “lookahead”. Thus, the method call `la(1)` returns the next token without consuming it. Similarly, the method call `la(2)` would return the token that comes after the next token. This allows the parser to decide which path to take next. Let's examine the code in detail:

1. First the parser matches the terminal **if**. If the next token is of type `Kind.KW_IF` (i.e., the keyword **if**), the call to `match()` succeeds and the token is consumed. If the next token is something else, `match()` throws a `RecognitionException` which has to be handled somewhere up the call chain.
2. Then the non-terminals *condition-list* and *code-block* are parsed by calling their corresponding production rule methods. These methods are not called directly but by way of the higher-order function `parse()`. Each production rule method returns an AST node and the purpose of the `parse()` method is to abstract away the task of capturing the tokens that make up a specific node.
3. After that, the *else-clause* is parsed. Note that its production rule has been integrated into the production rule for the *if-statement* because it is not used anywhere else. Since the *else-clause* is optional we first check whether the next token is the **else** keyword. If it is, the parser matches the token. The *else-clause* can either be a *code-block* (statements wrapped in braces) or another *if-statement*. Depending on the next token, it either calls the `codeBlock()` or the `ifStmt()` method.

4. Finally, the parser instantiates and returns the `IfStmt` node with the child nodes that were parsed throughout the `ifStmt()` method.

5.1.3 Speculative Parsing and Backtracking

The example in Listing 5.8 never needed more than one token lookahead (`1a(1)`) to fulfill its task. This is the case for most production rule methods. There are a few that require two tokens lookahead (`1a(2)`) but those are very similar to the method shown in Listing 5.8.

However, there are some situations in which a fixed amount of lookahead is not enough to decide which path to take next. The example in Listing 5.9 illustrates this problem. Note that the type annotations are only there to clarify the meaning of the program. They could be inferred by the compiler:

Listing 5.9: *Closures require speculative parsing*

```

1 var x = 1, y = 2, z = 3
2
3 let closure1: () -> [Int] = { [x, y, z] }
4 print(closure1()) // prints '[1, 2, 3]'
5
6 let closure2: () -> Int = { [x, y, z] in x + y + z }
7 print(closure2()) // prints '6'
```

The example defines two closures which both start with the same sequence of tokens `[x, y, z]`. However, they are very different from each other. In `closure1`, `[x, y, z]` is an array literal that is implicitly returned by the closure. In `closure2`, `[x, y, z]` is a capture list which explicitly captures the global variables `x`, `y` and `z` by making a copy of them.

When the parser reaches the opening square bracket (`[`), it cannot know whether the following tokens represent an array literal or a capture list. It is only when it reaches the token following the closing square bracket (`]`) that it knows for certain which path to take. Since array literals and capture lists can be arbitrarily long, a fixed amount of lookahead is not sufficient and the parser needs the ability to do speculative parsing and backtracking. Listing 5.10 shows how the higher-order function `speculate()` can be used to do speculative parsing:

Listing 5.10: *Example of speculative parsing*

```

1 private ClosureExpr closureExpr() throws RecognitionException {
2     match(Kind.LBRACE);
3
4     CaptureList captureList = null;
5     if(speculate(this::captureList)) {
6         captureList = parse(this::captureList);
7     }
8
9     // ...
10 }
```

If `speculate(this::captureList)` returns `true`, it means that the following tokens are indeed a capture list. Internally, `speculate()` calls `captureList()` to see if it can successfully parse a capture list. The method `captureList()` throws an exception if

it is not followed by the keyword `in` (or by a closure signature, but that is not relevant right now). This indicates to the `speculate()` method, that it should backtrack to the previous position and return `false`. Listing 5.11 shows the implementation of `speculate()`:

Listing 5.11: *Implementation of `speculate()`*

```

1 <T extends IASTNode> boolean speculate(ParseFunction<T> parseFunc) {
2     boolean success = true;
3     markers.push(pos);
4
5     try {
6         parseFunc.apply();
7     } catch (final RecognitionException e) {
8         success = false;
9     }
10
11     pos = markers.pop();
12     return success;
13 }
```

First, a boolean variable called `success` is declared and set to `true`. This variable tracks whether the speculation was successful or not. Then, the method stores the current token index by pushing it on the `markers` stack. The reason why the token index is pushed onto a stack and not just stored in a simple instance variable is because there may be nested calls to `speculate()` within the production rule method that was passed in for the parameter `parseFunc`.

Next, the method stored in `parseFunc` is called. In Java 8, function types are expressed by specifying a functional interface with a single method [tjl17b] [tjl17a]. In this case the functional interface is called `ParseFunction` and its only method is `apply()`. If a `RecognitionException` is thrown, `success` is set to `false` to indicate that the speculation failed.

The `speculate()` method then backtracks to its original position by resetting the current token index to the top marker on the `markers` stack. Finally, the `success` variable is returned to indicate to the caller whether the speculation was successful.

5.2 AST

The AST is an intermediate representation of the code. The indexer traverses the AST in order to resolve names, infer expression types and to find semantic errors. A semantic error means that the code is syntactically correct, but there is some other problem with it. For example, a name may be used without it being declared or a value of type A is assigned to a variable of type B where A and B are incompatible.

Once refactoring support is begin added, we also want to be able to analyze the AST to detect problems and to modify and rewrite the AST in order to reflect those changes in the code.

5.2.1 Requirements

As with the parser itself, the requirements for the AST are similar to but not the same as those of an AST that a compiler might generate. Most importantly, the AST should be as abstract as possible but simultaneously as close to the original source as necessary. For example, consider the code examples in Listing 5.12 and Listing 5.13:

Listing 5.12: *Code example A*

```
1 import Darwin
2
3 var x: Int = 42
4 var y = (4 * 5) + 3
5 var z: UInt32 {
6     get {
7         return arc4random() % 100
8     }
9 }
```

Listing 5.13: *Code example B*

```
1 import Darwin
2
3 var x = 42
4 var y = 4 * 5 + 3
5 var z: UInt32 {
6     return arc4random() % 100
7 }
```

Both examples declare 3 variables `x`, `y` and `z`. Each example does it in a syntactically slightly different way but both are semantically equivalent. With the variable `x` the type annotation is redundant, because it can be inferred from the initial value 42. The parentheses in the initializer expression of variable `y` are also unnecessary, because the `*` operator has a higher precedence than the `+` operator. Finally, `z` is a read-only, computed property. If there is no setter, one can leave out the `get` keyword as shown in Listing 5.13 and get the same result.

One could imagine an AST that is the same for both code examples. However, for the purposes of an IDE that would be too abstract. For example, we may want to provide refactorings to transform the declarations to their shorter forms. That would not be possible if the AST does not contain these details. Additionally, since Swift allows the declaration of custom operators, we may not know the precedence of an operator at parse time and therefore cannot tell whether the parentheses are necessary or not.

5.2.2 Structure

The AST consists of nodes that are instances of subclasses of the abstract superclass `ASTNode`. Each language construct (e.g., an if statement, a function call expression, a class declaration) has its own node class. These node classes are grouped in the same way as the parser modules described in section 5.1. There are declarations, statements, expressions, types, patterns and attributes.

Each node has zero or more child nodes (`getChildren()`) and also has a reference to its parent node (`getParent()`). As an example, Listing 5.14 shows how the node class for the `repeat-while` statement is implemented:

Listing 5.14: Node class for *repeat-while* statement

```

1 public class RepeatWhileStmt extends Stmt {
2     private final CodeBlock body;
3     private final IExpr condition;
4
5     public RepeatWhileStmt(CodeBlock body, IExpr condition) {
6         this.body = body;
7         this.condition = condition;
8     }
9
10    public CodeBlock getBody() {
11        return body;
12    }
13
14    public IExpr getConditionExpr() {
15        return condition;
16    }
17
18    @Override
19    public String getTreeStringTagName() {
20        return "repeat_while_stmt";
21    }
22
23    @Override
24    public boolean accept(ASTVisitor visitor) {
25        return acceptVisitor(visitor, body, condition);
26    }
27 }

```

The following list explains the most important aspects of this implementation:

- `RepeatWhileStmt` extends the abstract superclass `Stmt` which in turn extends the abstract superclass `ASTNode`. Thus, it inherits the methods `getChildren()` and `getParent()` which were mentioned above.
- The node class has two child nodes which are stored in the instance variables `body` and `condition`. The `body` has to be a code block whereas the `condition` can be any kind of expression (in a later stage the semantic analyzer will have to make sure that the `condition` expression is of type `Bool`). The node class also defines getter methods for these two instance variables.
- The method `getTreeStringTagName()` returns a short name for the node class `RepeatWhileStmt`. This is used to create a string description of the AST and will be described in more detail in section 5.4.
- The method `accept()` is part of the visitor pattern [E. 94] which is used to traverse an AST. This will be described in the next section.

5.2.3 Visiting an AST

To analyze a program we can traverse its AST using the visitor pattern. There are several variations of this pattern. The one used in this project is very much influenced by the visitor / AST structure of the Eclipse CDT project [ecl17a].

In order to visit an AST, one has to create a subclass of the abstract class `ASTVisitor`. The `ASTVisitor` class defines a `visit()` and `leave()` method for each kind of AST node. The default implementations of these methods do nothing and simply continue

the visitation process. In the `ASTVisitor` subclass one can customize this behaviour by overriding one or more `visit()` / `leave()` methods. An instance of the visitor class is then passed to the `accept()` method of the AST's root node in order to start the visitation process. The example in Listing 5.15 shows how an `ASTVisitor` can be used to collect all `Name` nodes in a source file:

Listing 5.15: *Visiting an AST to collect Name nodes*

```

1 Stream<Name> collectNames(SourceFile ast) {
2     final List<Name> names = new ArrayList<>();
3     ast.accept(new ASTVisitor() {
4         @Override
5         public int visit(Name name) {
6             names.add(name);
7             return PROCESS_CONTINUE;
8         }
9     });
10    return names.stream();
11 }
```

The `visit()` and `leave()` methods return an integer, which can be used to abort the visitation process (by returning `PROCESS_ABORT`) or to skip a subtree of the AST (by returning `PROCESS_SKIP`). The default implementations in `ASTVisitor` return `PROCESS_CONTINUE` which continues the visitation process.

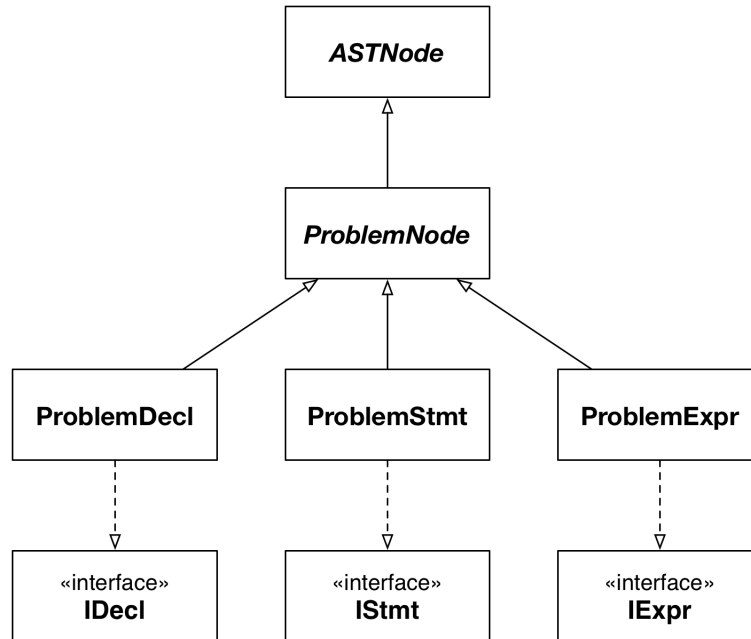
By overriding the `visit()` methods, the tree can be visited in preorder and by overriding the `leave()` methods it can be visited in postorder [Rob11].

5.3 Error handling

When the parser encounters a token that it did not expect, it throws a `RecognitionException`. A method up the call chain will then catch this exception and handle it. To do that, it creates an appropriate `ProblemNode` and inserts it into the AST. This way, the errors can be displayed in the editor simply by visiting the AST and creating a marker for each `ProblemNode` as shown in subsection 7.4.3. Additionally, the indexer, the outline view and other components that rely on the AST can just skip over the `ProblemNodes` and look at the valid parts of the AST.

Since we want to insert the problem nodes in places, where declarations (`IDecl`), statements (`ISmt`) or expressions (`IExpr`) are expected, there are various subclasses of `ProblemNode` that implement the corresponding interfaces. This is shown in Figure 5.4:

Figure 5.4: Problem Nodes



When the parser catches a `RecognitionException` it keeps consuming tokens until it reaches a point, where it can start to parse again. In order to illustrate this process, Listing 5.16 shows an example of how `RecognitionException`s are handled in the production rule method for declarations:

Listing 5.16: Example of error handling

```

1 IDecl decl() {
2   pushStartTokenIndex(getTokenIndex());
3   final int size = getCurrentStartTokenIndexStackSize();
4
5   try {
6     // try to parse a declaration
7   } catch(final RecognitionException e) {
8     reduceStartTokenIndexStackToSize(size);
9     consumeWhile(t -> {
10      return !isStartOfDecl(t);
11    });
12    return addTokens(new ProblemDecl(e.getMessage()));
13  }
14 }

```

First, the current token index is pushed onto a stack. This is required in order to be able to assign the tokens that belong to the resulting AST node at the end of the method. Normally, this is encapsulated in the higher-order function `parse()` which was shown in subsection 5.1.2, but here we need to customize the default behaviour.

The method also stores the current size of the token index stack. If a `RecognitionException` is thrown, the normal control flow is interrupted and some token indices may not be popped off the stack. Thus, the original size is restored in the catch clause with the call to the method `reduceStartTokenIndexStackToSize()`.

Then, additional tokens are consumed until the start of a new declaration is found. Finally, the parser creates and returns a `ProblemDecl` which contains the error message. At this point, the parser resumes the normal parsing process.

5.4 Testing

Tifig uses a set of automated tests to ensure that the quality of the parser does not deteriorate and that existing functionality keeps working after changes are made. The test cases use the parser to parse an input program and compare the resulting AST to an expected value.

In order to compare the resulting ASTs, they are first transformed into a string representation. Listing 5.17 shows an example of a simple function declaration and Figure 5.5 shows the string representation of this program's AST:

Listing 5.17: *Function declaration*

```
1 private func myfunc(x: Int) {
2
3 }
```

Figure 5.5: *String representation of an AST*

```
(function_decl modifiers='private' throwing_behaviour='none'
  (name text='myfunc')
  (parameter_clause
    (parameter variadic='false'
      (name text='x')
      (type_annotation inout='false'
        (type_identifier
          (type_identifier_element
            (name text='Int'))))))
  (code_block))
```

All parser test cases inherit from the superclass `ParserTestCase` which implements a few helper methods. An example of this is shown in Listing 5.18:

Listing 5.18: *Example of a parser test case*

```

1 public class ClassDeclTests extends ParserTestCase {
2     // class MyClass {
3     //     func f() {}
4     // }
5
6     // (class_decl modifiers=''
7     //     (name text='MyClass')
8     //     (decl_body
9     //         (function_decl modifiers='' throwing_behaviour='none'
10            //         (name text='f')
11            //         (parameter_clause)
12            //         (code_block))))
13     @Test
14     public void testClassDeclWithInstanceMethod() {
15         assertEqualsSourceFileContent();
16     }
17
18     // other test cases
19 }

```

The method `assertEqualsSourceFileContent()` is a helper method that is declared in the superclass `ParserTestCase`. First, it reads the two comments above the test method. The first comment contains the Swift source code and the second comment contains the string representation of the expected AST. The code in the first comment is then parsed and the resulting AST's string representation is compared with the expected value specified in the second comment. In addition to `assertEqualsSourceFileContent()`, there are other helper methods that allow us to test certain language constructs without having to specify a full, valid program each time (e.g., `assertEqualsType()`, `assertEqualsExpr()`, etc).

5.5 Implementation Status

Tifig's parser is fully compatible with the Swift 3 grammar. However, there are still a few remaining issues that should be fixed in the future:

- **Improve Performance**
When the parser needs to speculate, it parses the same code twice. This should be improved in the future (e.g., with a memoizing parser).
- **Interpolated String Literals**
As mentioned in section 4.7, the lexer currently creates only a single token for an interpolated string literal. Once this is fixed, the parser must be updated as well and parse the expressions that are embedded within the string literal.
- **Error handling**
The errors reported by the parser are still too imprecise. This should be improved in the future.

6 Indexer

This chapter describes what the Indexer does and how it is implemented in the Tifig IDE. The code shown in this chapter is in part influenced by what is shown in the book *Language Implementation Patterns* [Ter10].

6.1 The job of an Indexer

The indexer is responsible for semantic analysis of the source code which enables more advanced IDE features such as “Jump to Definition” or “Open Call Hierarchy”. The indexer visits the ASTs of the files in the project in order to create bindings for all the named entities of the program (e.g., variables, functions, classes). A binding is like an entry in a symbol table. Names that refer to the same entity also have the same binding. However, note that two occurrences of the same name do not necessarily refer to the same entity. For example, consider the code in Listing 6.1:

Listing 6.1: *Same name, two different variables*

```
1 let x = 5
2
3 func f() {
4     let x = 10
5     print(x)
6 }
7
8 print(x)
```

In this program, there are two entities called `x`; a global variable and a local variable. Thus, there are two bindings with the name `x`. The name `x` on line 5 refers to the `x` declared on line 4 and is therefore associated with the binding for the local variable. On the other hand, the name `x` on line 8 refers to the `x` declared on line 1 and is therefore associated with the binding for the global variable. The process of finding the correct binding for a specific name is called *binding resolution*.

Because Swift supports function overloading, binding resolution for function calls doesn’t only depend on the scope in which the function call occurs, but also on the type of the arguments (and on the contextual type, but this is explained in subsection 6.5.1).

This means, that the indexer must be able to infer the types of expressions. For example, consider the code in Listing 6.2:

Listing 6.2: *Type Inference and Overload Resolution*

```
1 func log(_ value: Bool) { print("Bool: \(value)") }
2 func log(_ value: Int) { print("Int: \(value)") }
3
4 let x = 2 < 1    // x is of type Bool
5 log(x)          // calls log: (Bool) -> ()
```

In this example, there are two functions called `log()`. The first function takes an argument of type `Bool` and the second function takes an argument of type `Int`. During binding resolution, the indexer needs to figure out, whether the name `log` in the function call `log(x)` refers to the first function or the second function. To do that, it needs to know the type of the variable `x` and since `x` doesn't have an explicit type annotation, it needs to be able to infer the type from the expression `2 < 1`.

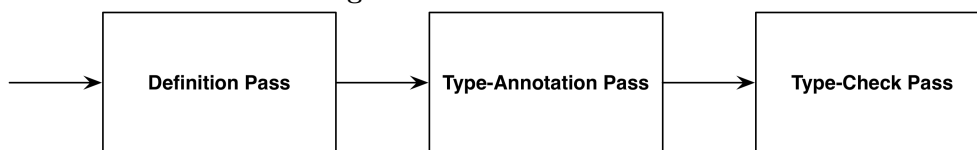
Thus, being able to resolve bindings requires the ability to infer the types of arbitrary Swift expressions. Since type inference in Swift can be quite complicated (see subsection 6.5.1), this is in many ways the most complex task that the indexer has to perform.

6.2 Architecture Overview

The indexer takes a set of ASTs as input and analyzes them in order to obtain the semantic knowledge that is required by the IDE. After indexing, each `Name` node should be backed by a corresponding binding. A binding contains additional information about an entity which might be useful to implement more advanced IDE features. For example, from a class binding we can get to the bindings of its members and from an operator binding we can get to the binding of its precedence group.

The indexer performs three passes to index a Swift project. This is shown in Figure 6.1:

Figure 6.1: *Indexer Overview*



An example of why indexing is a multi-pass process is shown in Listing 6.3:

Listing 6.3: *Indexing is a multi-pass process*

```

1 extension Derived {
2     func f() {
3         g()
4     }
5 }
6 class Derived: Base {}
7 class Base {
8     func g() {}
9 }
  
```

Definition Pass

The definition pass creates scopes and bindings. In the example above, an extension binding and two class type bindings are created in the file scope. The member scope of the extension binding and the member scope of the class type binding `Base` each contain a method binding. Additionally, both method bindings have a parameter scope and a local scope that do not contain any bindings.

Type-Annotation Pass

The type-annotation pass resolves type-annotations as well as other names that are not

part of an expression (e.g., the name of the base class in the type-inheritance clause). This has to be done in a separate pass, because Swift places very few restrictions on the order in which entities are declared. In the example above, the subclass `Derived` is declared before its superclass `Base`. This means that we cannot just perform a single pass from top to bottom because once we reach the type inheritance clause of the class `Derived`, the binding for the class `Base` has not yet been created.

Type-Check Pass

The type-check pass resolves the types of expressions. In the example above, there is only a single expression: the function call `g()` on line 3. We cannot combine the type-annotation pass and the type-check pass into a single pass. This is because when we visit the AST from top to bottom, the expression `g()` is processed before the type-inheritance clause of the class `Derived`. Thus, the class type binding `Derived` doesn't yet know anything about its superclass `Base` and since the method `g()` is a member of `Base`, the indexer cannot find a corresponding binding for the name `g`.

6.3 Definition Pass

During the definition pass, the indexer only looks at declaration nodes. For each declared name, it creates a corresponding binding. There are different kinds of bindings for different declarations. For example, for a variable declaration the indexer creates an instance of `VariableBinding` and for a function declaration it creates an instance of `FunctionBinding`.

Each binding is stored in a lexical scope and each scope can have at most one binding for a particular name. Scopes can have one parent scope and multiple child scopes. Therefore, all scopes together build a tree. The definition pass is responsible for creating the bindings and for building the scope tree.

As an example, Listing 6.4 contains a simple program and Figure 6.2 shows the scopes and bindings that are created for this program during the definition pass:

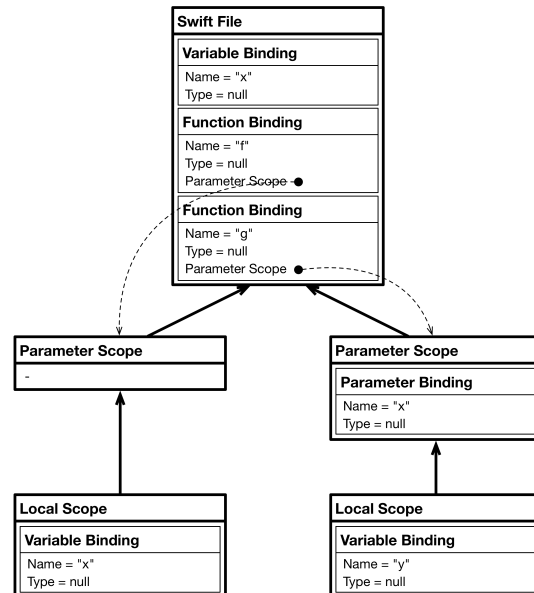
Listing 6.4: *Example Program*

```

1 let x = 5
2 print(x)
3
4 func f() {
5     let x = 10
6     print(x)
7 }
8
9 func g(x: Int) {
10    let y = 15
11    print(x, y)
12 }

```

Figure 6.2: *Scope Tree after Definition Pass*



The figure shows that each scope has a reference to its parent / enclosing scope. The bindings for the global variable `x` and the two global functions `f()` and `g()` are stored in the Swift file scope. The binding for the parameter `x` is stored in the parameter scope of the function `g()` and the bindings for the two local variables `x` and `y` are stored in their corresponding local scopes.

The reason why each function has a separate scope for its parameters is because in Swift it is possible to declare a local variable that has the same name as one of the parameters, in which case the local variable *shadows* the parameter. Each function binding has a reference to its parameter scope which allows us to obtain the parameter bindings from a function binding.

Note that the types of the individual bindings are not yet set after the definition pass. This is the job of the subsequent type-annotation pass and type-check pass.

6.3.1 Bindings

All bindings in Tifig have a type and an access level. The type is used during type-checking and the access level is used to determine which bindings are accessible from a specific location. Apart from the standard Swift access levels `private`, `fileprivate`, `internal`, `public` and `open`, the access level can also be `null`. This is used for entities whose declaration cannot have an access level modifier and it generally means that the access level can be ignored (i.e., the binding is always considered to be accessible). For example, enum case declarations cannot have an access level modifier. Nevertheless, since they are always accessed through their owner, they effectively have the same access level as the corresponding enum type.

Bindings in Tifig can be broadly divided into two groups: declared bindings and implicit bindings.

Declared Bindings

Declared bindings are bindings that have an explicit declaration in the source code. They have a reference to the declaration name which is the `Name` node that appears in the corresponding declaration. If a name is backed by a declared binding, the user can click on the name in order to jump to its definition. Examples of declared bindings are `VariableBinding`, `FunctionBinding` and `ParameterBinding`.

Implicit Bindings

Some names don't have an explicit declaration in the source code. For example, the name `self` is implicitly available within the methods of a named type. While the IDE cannot jump to `self`'s definition (since there is no such definition), we still want to set a binding for that name, because that allows us to set its type. Tifig uses *implicit bindings* for such names. They don't have a reference to a declaration name and the "Jump to Definition" feature doesn't generate hyperlinks for names that are backed by an implicit binding. Other examples of implicit bindings are the `newValue` variable that is implicitly available within the setter of a computed property, and the compiler-generated initializers of named types.

6.3.2 Unavailable Declarations

In Swift a declaration may be marked as "unavailable" which means that compilation will result in an error if you are trying to use such a declaration in your program. This is often used in the Swift standard library for declarations that were previously available and have now been removed or were replaced by something else. Listing 6.5 shows an example:

Listing 6.5: *Example of an unavailable declaration*

```

1 @available(*, unavailable, message: "it has been removed in Swift 3")
2 @discardableResult
3 public prefix func ++ (x: inout Int) -> Int {
4     x = x + 1
5     return x
6 }

```

The prefix increment operator `++` was deprecated in Swift 2 and removed in Swift 3. However, the above declaration is still part of the standard library because it allows the compiler to emit better error messages. The `@available` attribute marks the declaration as “unavailable” and specifies an error message that is displayed if a user compiles a program that tries to use this declaration. Additionally, the function has a `@discardableResult` attribute. This just means that the function has a side-effect and the compiler should not emit a warning if the result of the function is not used.

Note that Tifig ignores unavailable declarations completely. In the future it might be better to create bindings that are marked as unavailable in order to be able to provide better diagnostics.

6.3.3 Conditions

Optional binding conditions and case conditions can define new variables. These variables live in a separate scope from both the enclosing scope as well as the local scope of the corresponding `if`, `guard` or `while` statement. An example of this is shown in Listing 6.6:

Listing 6.6: *Condition Scopes*

```

1 let x: Int? = 42
2 print(x)      // prints 'Optional(42)'
3
4 if let x = x {
5     print(x)  // prints '42'
6     let x = 0
7     print(x)  // prints '0'
8 }

```

The `x` in the first `print()` call refers to the global variable which is of type `Optional<Int>`. In the optional binding condition `let x = x` a new variable `x` is declared in a child scope of the global scope. However, the `x` in the initializer still refers to the global, optional variable.

The `x` in the second `print()` call refers to the variable that has been declared by the optional binding condition, since it shadows the global variable.

On line 6, a local variable is created which shadows the variable `x` from the optional binding condition. Finally, the `x` in the last `print()` call refers to this new local variable.

6.3.4 Extensions

In Swift, extensions allow us to add additional members (e.g., methods, initializers, computed properties) to an existing type. However, as mentioned in section 6.2, Swift places very few restrictions on the order in which entities are declared. Thus, it is possible to extend a type before it is declared. This is shown in Listing 6.7:

Listing 6.7: *Extension can appear before declaration of extended type*

```

1 extension Point: Equatable {
2   static func ==(lhs: Point, rhs: Point) -> Bool {
3     return lhs.x == rhs.x && lhs.y == rhs.y
4   }
5 }
6
7 struct Point {
8   var x: Int
9   var y: Int
10 }
```

For this reason, the indexer cannot add the additional member bindings directly to the extended type. Instead, a new extension binding is created for each extension. After the definition pass, extensions are connected to the corresponding extended type.

Note that the binding of a specific named type doesn't know anything about its extensions. This is because not all extensions are automatically available everywhere. For example, if a user's program adds a method to the type `Int` through an internal extension, this method is only available in the corresponding module and not in other modules or in the standard library. Thus, the extensions are stored in the `SwiftFile` scope and when the type checker performs a member lookup, it searches the corresponding type binding as well as all of the extensions that are visible from the current file.

6.3.5 Implicit Operator Bindings

Most operators in Swift are not part of the language but instead are declared in the standard library. However, there are a few exceptions. The infix operators `=`, `as`, `as?`, `as!`, `is` as well as the ternary operator `?:` are built into the compiler.

However, these operators are still part of a precedence group which is documented in the standard library file `Policy.swift`. During the definition pass Tifig generates implicit operator bindings for these built-in operators. Later, during the type-annotation pass, they are added to the corresponding precedence group. This allows us to treat these built-in operators like regular operators for the most part. They only are treated differently during type-checking where a regular operator results in a call to an operator function whereas a built-in operator gets special treatment (see section 3.1.11).

6.3.6 Implicit Variable Bindings

In some situations, Swift implicitly defines special variables. For example, consider the code in Listing 6.8:

Listing 6.8: *Implicit Variable Bindings*

```

1 import Foundation
2
3 struct Square {
4     var side: Double
5
6     var area: Double {
7         get {
8             return side * side
9         }
10        set {
11            side = sqrt(newValue)
12        }
13    }
14 }
15
16 var square = Square(side: 5.0)
17 print(square.area)      // 25.0
18 square.area = 64.0
19 print(square.side)      // 8.0

```

In this example, `newValue` is an implicitly defined variable that contains the new value that was assigned to the computed property `area`. During the definition pass, Tifig automatically creates an implicit binding for these variables. Later, during the type-annotation pass, it sets the type of these bindings. In the example above, `newValue` has type `Double` (i.e., the same type as the computed property `area`).

It is possible to specify a different name for these variables as shown in Listing 6.9:

Listing 6.9: *Specifying a different name for the setter parameter*

```

1 var area: Double {
2     get {
3         return side * side
4     }
5     set(newArea) {
6         side = sqrt(newArea)
7     }
8 }

```

In this case, Tifig would create a declared binding instead of an implicit binding, because there exists a corresponding declaration name.

Note also that these bindings need to be defined in a separate scope, because they can be shadowed by local variables.

Additionally, there are other situations in which such implicit variable bindings are generated. For example, in the `willSet` and `didSet` clauses of observed variables, there are implicitly defined variables called `newValue` and `oldValue`. Similarly, in the `catch` clause of a `do-catch` statement, there is an implicitly defined variable called `error`.

6.3.7 Implicit Closure Parameters

If a closure uses implicit closure parameters (e.g., `$0`, `$1`, `$2`, ...), the definition pass creates implicit bindings for these parameters. Since there are no explicit type annotations for implicit closure parameters, their types are inferred during the type-check pass.

6.3.8 Imports

Import declarations are different from other declarations. During the definition pass, no new bindings are created for import declarations. Instead, the indexer looks for a module (i.e., an instance of `SwiftModule`) with the corresponding name. If it finds one, this module is added to the list of imported modules of the current `SwiftFile`. During binding resolution, if the indexer doesn't find a suitable binding in the current module, it additionally searches the modules that are imported in the current file.

It is also possible to only import a specific declaration of a module but this is currently not yet supported by Tifig's indexer.

6.3.9 Standard Library

As mentioned previously, Swift relies heavily on declarations from the standard library. Basic types such as `Int`, `Double` and `Bool` as well as arithmetic and logical operators are not part of the language but are instead declared in the standard library. Therefore, the standard library also needs to be indexed in order for the rest of the indexing process to work properly.

Tifig treats the standard library as a separate module called "Swift". This module is indexed once after Tifig has launched and is then implicitly imported in every file. Thus, the public declarations from the standard library are available everywhere. To be able to do this, Tifig's application bundle contains a copy of the Swift files that contain the code for the standard library.

6.4 Type-Annotation Pass

As mentioned previously, every binding in Tifig has a type. Sometimes this type has to be inferred by the type-check pass (e.g., for a variable without an explicit type annotation). The main job of the type-annotation pass is to set the type of the bindings that do have an explicit type annotation. To do that, the type-annotation pass transforms the *AST types* from the type annotations into corresponding *index types*. Subsection 6.4.1 explains the difference between AST types and index types, and gives an overview over the various kinds of index types. Subsection 6.4.2 describes all the tasks that are fulfilled by the type-annotation pass.

6.4.1 Index Types

An AST type is a node in the AST that describes an explicit type annotation. It has a specific location in the source code and is composed of one or more tokens. On the other hand, an index type is a more abstract representation of a type that is used by the indexer for type checking. It doesn't have a specific location in the source code and there may not even be a corresponding AST type, because an index type may be the result of inferring the type of an expression. All index types implement the `IType` interface.

Nominal Types

In Swift, classes, structs, enums and protocols are sometimes called *nominal types*. These types have a name and are declared somewhere in the source code. Additionally, they can be extended, can conform to protocols and can have members. Nominal types are represented by subclasses of the abstract superclass `NominalTypeBinding`. Note that these classes are not just index types but also bindings.

Metatypes

A nominal type is used as the type for instances of the nominal type. In contrast, a metatype is used as the type of a nominal type itself. This is used to distinguish between a member reference to a static member and a member reference to an instance member. Listing 6.10 shows an example:

Listing 6.10: *Nominal Types vs. Metatypes*

```

1 struct S {
2     func f1() {}
3     static func f2() {}
4 }
5
6 let s = S()
7 s.f1()
8 S.f2()
9 s.f2() // error: static member 'f2' cannot be used on instance of type 'S'
```

When the type checker checks an explicit member expression of the form `<owner>.<member>`, it first evaluates the type of the subexpression `<owner>`. Afterwards, it looks for members with the name `<member>` within that owner type.

On line 7, the name `s` resolves to the `VariableBinding` for `s` declared on line 6. The type of this binding is a nominal type (more specifically, a struct type). Thus, when the type checker looks for members with the name `f1`, it only looks for instance members.

On line 8, the name `S` resolves to the `StructTypeBinding` for `S` declared on lines 1-4. The type of this binding is a metatype. Therefore, when the type checker looks for members with the name `f2`, it only looks for static members.

Finally, on line 9, we try to access the static member `f2` through the variable binding `s`. This fails because in Swift we cannot access static members through an instance of a nominal type.

Tuple Types

A tuple type is represented by an instance of the class `TupleType`. Each tuple type consists of multiple tuple type elements. Each tuple type element has an optional name and a type. The names of the individual elements have to be part of the tuple type, because we can later use these names to access individual elements of the tuple. This is shown in Listing 6.11:

Listing 6.11: *Tuple with named elements*

```
1 let tuple = (name: "Toni", age: 26) // tuple is of type (name: String, age: Int)
2 let name1 = tuple.name             // name1 is of type String
3 let name2 = tuple.0                 // name2 is of type String
4 let age1 = tuple.age                // age1 is of type Int
5 let age2 = tuple.1                 // age2 is of type Int
```

The above example shows that it is possible to either access individual elements by name or by index. The element lookup works similar to the member lookup of nominal types. First, the type of the owner is determined and afterwards the type checker looks for an element with the corresponding element name or index within the owner type.

It is also possible to have a tuple type with unnamed elements which means that we can access the individual elements only by index. This is shown in Listing 6.12:

Listing 6.12: *Tuple with unnamed elements*

```
1 let tuple = ("Toni", 26) // tuple is of type (String, Int)
2 let name = tuple.0       // name is of type String
3 let age = tuple.1        // age is of type Int
```

Function Types

Function types are used for entities that can be called with a function call expression. This includes functions, closures, methods, initializers and enum case constructors. A function type is composed of a parameter type and a return type. In Tifig, the parameter type is a tuple type whose elements represent the individual parameters (i.e., external names and parameter types). The return type can be any index type. However, the

return type can never be `null` and even functions that return nothing have a return type of `()` (i.e., empty tuple which in Swift means `Void`).

Listing 6.13 shows the declaration of a variable called `triple` which is initialized with a closure expression. Since the type of this variable is the function type `(Int) -> Int`, it can be called like a regular function.

Listing 6.13: *Function Type Example*

```
1 let triple = { x in x * 3 } // triple has function type (Int) -> Int
2 let three = 3
3 let nine = triple(3)
```

Any Type

The Any type is a special “top type” that is built into the Swift compiler [Ben02]. In Tifig, it is represented by an instance of the class `AnyType`. This class is a singleton, because there exists only one Any type.

Lvalue Reference Types

In Swift it is possible to pass an lvalue to a function *by reference*. To do so, the parameter type has to be preceded by the `inout` keyword and the argument has to be wrapped in an inout expression (e.g., `&arg`). Listing 6.14 shows an example:

Listing 6.14: *Lvalue Reference Types*

```
1 func inc(_ x: inout Int) {
2     x += 1
3 }
4
5 var x = 0
6 print(x)           // prints '0'
7 inc(&x)
8 inc(&x)
9 print(x)           // prints '2'
```

The type of the argument expression `&x` is represented by an instance of the class `LvalueReferenceType`. Internally this class has a reference to the type of the lvalue itself (in the example above, `x` is the lvalue). The type checker makes sure that an argument of lvalue reference type is provided for each `inout` parameter. Additionally, the type wrapped inside the lvalue reference type must exactly match the parameter type. This is different from regular parameters where the argument expression can be of any type as long as it is convertible to the parameter type. An example of this is shown in Listing 6.15:

Listing 6.15: *Regular Parameters vs. Inout Parameters*

```

1 class Base {}
2 class Derived: Base {}
3
4 func f(_ x: Base) {}
5 var d1 = Derived()
6 f(d1) // valid: implicit conversion from Derived to Base
7
8 func f(_ x: inout Base) {}
9 var d2: Base = Derived()
10 f(&d1) // invalid: implicit conversion is not possible with inout parameters
11 f(&d2) // valid: no implicit conversion is necessary

```

At the time of this writing, Tifig does not yet distinguish between lvalues and rvalues. Thus, the indexer will accept expressions like `inc(&5)` even though this is invalid code, because the literal expression 5 is not an lvalue.

Additionally, Swift has support for pointers which can also be initialized with an expression like `&arg`. This is not yet implemented in Tifig either.

Type Aliases

Swift allows the declaration of type aliases in order to introduce a named alias for an existing type in your program. Anywhere in your program, the name of the type alias can be used instead of the existing type. The existing type can be a named type (e.g., a struct or another type alias) or a compound type (e.g., a tuple or a function). A type alias does not create a new type. It just defines another name to refer to an existing type. This is shown in Listing 6.16:

Listing 6.16: *Type Aliases*

```

1 typealias IntPair = (Int, Int)
2
3 func swap(_ p: IntPair) -> IntPair {
4     return (p.1, p.0)
5 }
6
7 var pair = (1, 2) // pair is of type (Int, Int)
8 print(pair)      // prints '(1, 2)'
9 pair = swap(pair)
10 print(pair)      // prints '(2, 1)'

```

In Tifig, type aliases are represented by instances of the class `TypeAliasTypeBinding`. Like with nominal types, this class is both an index type and also a binding. Internally, it stores a reference to the existing type. Therefore, whenever the type checker needs to know the underlying type of a type alias (e.g., in order to perform member lookup) that information can be extracted from the type alias binding.

Protocol Composition Types

A protocol composition type is a compound type that consists of multiple protocol types. Any value whose type conforms to all the protocols listed in a protocol composition type can be assigned to a variable that is of that protocol composition type. An example of this is shown in Listing 6.17:

Listing 6.17: *Protocol Composition Types*

```

1 protocol P1 {
2     func f1()
3 }
4
5 protocol P2 {
6     func f2()
7 }
8
9 struct S: P1, P2 {
10     func f1() {}
11     func f2() {}
12 }
13
14 func f(_ x: P1 & P2) {
15     x.f1()
16     x.f2()
17 }
18
19 let s = S()
20 f(s)

```

In Tifig, protocol composition types are represented by instances of the class `ProtocolCompositionType`.

Generic Type Parameters

In Tifig, generic type parameters are represented by instances of the class `GenericTypeParameterBinding`. Like with nominal types and type aliases, this class is both an index type and a binding. In order to be able to correctly type check the bodies of generic functions, Tifig stores the conformance requirements of a generic type parameter within the corresponding `GenericTypeParameterBinding` instance. Listing 6.18 shows an example:

Listing 6.18: *Generic Type Parameters*

```

1 protocol P {
2     func f()
3 }
4
5 func g<T: P>(x: T) {
6     x.f()
7 }

```

In this example, Tifig creates a `GenericTypeParameterBinding` for the generic type parameter `T`. The requirement that `T` must conform to the protocol `P` is stored within the corresponding `GenericTypeParameterBinding` instance. Later, when the function call expression `x.f()` is type checked, the indexer knows that all instances of type `T` do have a method `f()`, because this is required by the protocol `P`.

Note that Swift also supports different kinds of requirements for generic type parameters (e.g., adding restrictions on associated types of a generic type parameter). However, this is currently not yet supported by Tifig.

Generic Type Instances

A generic type instance is composed of a `NominalTypeBinding` which refers to a generic type as well as a list of type arguments supplied for the generic type parameters. This index type is represented by instances of the class `GenericTypeInstance`. An example is shown in Listing 6.19:

Listing 6.19: *Generic Type Instances*

```

1 struct Pair<T1, T2> {
2     let first: T1
3     let second: T2
4 }
5
6 let pair = Pair(first: 42, second: "hello")
7 let x = pair.first
8 let y = pair.second

```

In this example, the type of the variable `pair` is inferred to be the generic type instance `Pair<Int, String>`. When a member of a generic type instance is accessed, the generic type parameters in the member’s type are replaced by the corresponding type arguments that are included in the generic type instance. Thus, in the example above, the variable `x` is of type `Int` and the variable `y` is of type `String`.

Associated Types

In Tifig, associated types of protocols are represented by instances of the class `Protocol-AssociatedTypeBinding`. Similar to generic type parameters, associated types can have conformance requirements. Additionally, an associated type can have a default type. These properties are stored within the binding so that the type checker can later refer back to them.

Swift Modules

In Tifig, the class `SwiftModule` implements the `IType` interface. This is because we can use a module name as a qualifier to access public declarations of that module. Thus, the module acts like a type and the public declarations are the members of that type. This is sometimes used to access an entity of an imported module that is shadowed by an entity in the current module because the two entities have the same name. An example of this is shown in Listing 6.20:

Listing 6.20: *Using a module name to refer to a shadowed type*

```

1 struct Int {}
2 let x: Int           // Int refers to the Int type declared on line 1
3 let y: Swift.Int     // Swift.Int refers to the Int type declared in the standard library

```

In this example, a struct type called `Int` is declared. This type shadows the `Int` type from the standard library. However, since the standard library is treated like a module with the name “Swift”, we can use the more specific name `Swift.Int` to directly refer to the type `Int` that is contained in the standard library.

Type Variables

Type variables are represented by instances of the class `TypeVariableType`. A type variable is a special kind of index type that is only used during type checking and it acts as a placeholder, if the type of an expression is not yet known. Each type variable has an ID. In this thesis, type variables are referred to by the name `$Tx` where `x` is the ID of the type variable (e.g., a type variable with the ID 1 is called `$T1`).

Additionally, each type variable has a fixed type. In the beginning of the type checking process, the fixed type is `null`. The constraint-based type checker then tries to find a fixed type for each type variable (see section 6.6). After the type checking process is done, the type variables will be replaced by their corresponding fixed types.

Equality of Index Types

During type checking, index types sometimes need to be compared for equality. Some index types (e.g., tuple types, function types, protocol composition types) are composed of other index types. Two instances of those types are equal if their components are equal. For example, there can be two separate instances of `ProtocolCompositionType` that are considered to be equal, if they are composed of the same protocols. In Tifig, this is implemented by overriding Java's `equals()` method.

On the other hand, there are also index types (e.g., nominal types, type aliases, generic type parameters) for which the `equals()` method is not overridden. Two variables of such an index type are only considered to be equal, if they refer to the exact same instance. This is because for example, it is possible to declare a struct type called `S` in one module and another struct type that is also called `S` in a second module. Even though these two struct types have the same name, they are still two distinct types.

6.4.2 Tasks of the Type-Annotation Pass

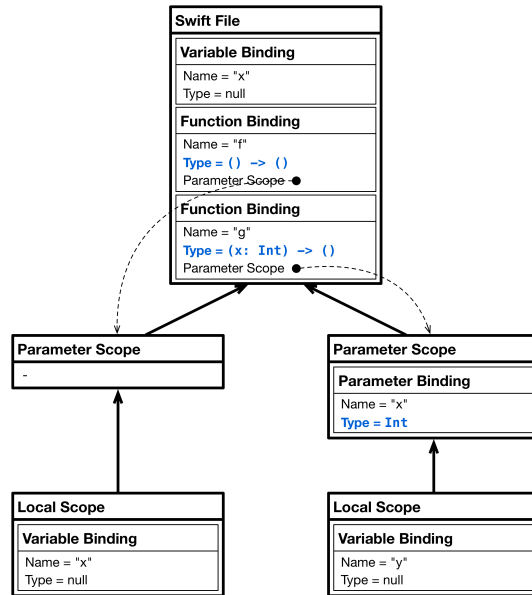
In section 6.3 the scope tree for a small example program was shown. Figure 6.3 shows what this scope tree looks like after the type-annotation pass. As you can see, the types of the two function bindings as well as the type of the parameter binding have been set. The types of the variable bindings have not been resolved yet, because they don't have an explicit type annotation. Thus, their types need to be inferred during the type-check pass.

Figure 6.3: *Scope Tree after Type-Annotation Pass***Listing 6.21:** *Example Program*

```

1 let x = 5
2 print(x)
3
4 func f() {
5     let x = 10
6     print(x)
7 }
8
9 func g(x: Int) {
10     let y = 15
11     print(x, y)
12 }

```



Apart from transforming AST types into index types, the type-annotation pass also resolves all names that are not part of an expression. For example, this includes class and protocol names in type-inheritance clauses as well as the conformance requirements of a generic type parameter. The following list describes all the tasks that are fulfilled by the type-annotation pass:

Variables and Parameters

The type of a variable binding can be resolved during the type-annotation pass, if there is an explicit type annotation. If a variable declaration doesn't have an explicit type annotation, the type must be inferred from its initializer expression during the type-check pass. Note that function parameters always have an explicit type annotation.

Functions and Methods

The type of a function or a method can always be resolved during the type-annotation pass. The type is a function type that consists of the types of the function's parameters and its return type. If a function declaration doesn't have an explicit return type, the return type is implicitly set to `()` (empty tuple type / `Void`) which means that the function returns nothing.

Initializers

Initializers are similar to methods. They also have parameters which must have an explicit type annotation. However, the return type of an initializer is always implicit. For regular initializers the return type is the enclosing nominal type. For failable initializers the return type is an optional containing the enclosing nominal type. An example of this is shown in Listing 6.22:

Listing 6.22: *Types of Initializers*

```

1 struct Rational {
2   let numerator: Int
3   let denominator: Int
4
5   // This is a failable initializer that
6   // has function type (numerator: Int, denominator: Int) -> Rational?
7   init?(numerator: Int, denominator: Int) {
8     guard denominator != 0 else {
9       return nil
10    }
11    self.numerator = numerator
12    self.denominator = denominator
13  }
14
15  // This is a regular initializer that
16  // has function type (integer: Int) -> Rational
17  init(integer: Int) {
18    self.numerator = integer
19    self.denominator = 1
20  }
21 }
```

Subscripts

Tifig uses function types as the types for subscript bindings. These types can be resolved during the type-annotation pass because a subscript declaration always has zero or more parameters and an explicit return type.

Enum Cases

The type of an enum case can also be resolved during the type-annotation pass. There are two kinds of enum cases: those that have associated values and those that don't. The type of an enum case without associated values is simply the enclosing enum type. On the other hand, an enum case with associated values acts like an initializer. Thus, its type is a function type with the return type set to the enclosing enum type. An example of this is shown in Listing 6.23:

Listing 6.23: *Types of Enum Cases*

```

1 enum E {
2   case one
3   case two(String, Int)
4 }
5
6 let x = E.one           // x is of type E
7 let y = E.two           // y is of function type (String, Int) -> E
8 let z = E.two("test", 2) // z is of type E
```

Type Inheritance Clauses

The names that appear in a type inheritance clause are also resolved during the type-annotation pass. An example of this is shown in Listing 6.24:

Listing 6.24: *Resolving names in type inheritance clauses*

```
1 protocol P {}
2 class Base {}
3 class Derived: Base, P {}
```

After the definition pass, there are three bindings: a protocol type binding called **P** and two class type bindings called **Base** and **Derived**. At this point the indexer has not yet recorded the fact that **Derived** inherits from **Base** and conforms to **P**. This is done during the type-annotation pass. The indexer resolves the names in the type inheritance clause and updates the class type binding **Derived** correspondingly.

In class type declarations the first name in the type inheritance clause can either be the name of a superclass or the name of an adopted protocol. All remaining names in the type inheritance clause must resolve to protocols. The declarations of struct types, enum types and protocol types can only have the names of protocols in their type inheritance clause.

Infix Operators

The declaration of an infix operator can specify which precedence group the operator belongs to. During the type-annotation pass this precedence group name is resolved and the operator binding is updated correspondingly. If no precedence group name is specified, the infix operator belongs to the precedence group **DefaultPrecedence**.

Precedence Groups

All precedence groups together build a partially ordered set. Each precedence group declaration can specify which other precedence groups have higher or lower precedence than the current precedence group. Listing 6.25 shows an excerpt from the standard library which shows how some of the predefined precedence groups are ordered:

Listing 6.25: *Order of precedence groups*

```
1 precedencegroup LogicalDisjunctionPrecedence {
2   associativity: left
3   higherThan: TernaryPrecedence
4 }
5 precedencegroup LogicalConjunctionPrecedence {
6   associativity: left
7   higherThan: LogicalDisjunctionPrecedence
8 }
9 precedencegroup ComparisonPrecedence {
10  higherThan: LogicalConjunctionPrecedence
11 }
```

During the type-annotation pass the indexer resolves the precedence group names after **higherThan:** and records the order relationships within the individual precedence group bindings.

Typealiases

During the type-annotation pass, the right hand side (i.e., the aliased type) of a type alias declaration is transformed into an index type. A reference to this index type is then stored in the corresponding `TypealiasTypeBinding`. This way, the type checker can later obtain the underlying type of a type alias.

Associated Types

During the type-annotation pass, any conformance requirements and default types in associated type declarations are resolved and transformed into index types. A reference to these index types is then stored in the corresponding associated type bindings.

Implicit Operator Bindings

Section 6.3.5 described that the definition pass creates implicit operator bindings for the infix operators `=`, `as`, `as?`, `as!`, `is` as well as for the ternary operator `?:`. The type-annotation pass connects these bindings to their corresponding precedence group (as documented in the standard library file `Policy.swift`).

Implicit Variable Bindings

Section 6.3.6 described that the definition pass creates implicit variable bindings for implicit variables such as `newValue` and `oldValue`. The type-annotation pass assigns a type to each of these bindings. It obtains the type by looking at the context of the corresponding implicit variable. For example, a `newValue` variable in a setter clause has the same type as its enclosing computed property.

6.5 Type-Check Pass

During the type-check pass, the indexer checks whether the expressions in the source code are well-typed. If there are no type errors in an expression, the indexer assigns a type to each of its subexpressions. The indexer also assigns a type to each binding whose type depends on type inference. Additionally, overload resolution happens during the type-check pass. This is because overload resolution usually depends on the types of expressions and those are not yet known before the type-check pass.

In the sections 6.3 and 6.4 the scope tree for a small example program was shown. Figure 6.4 shows what this scope tree looks like after the type-check pass.

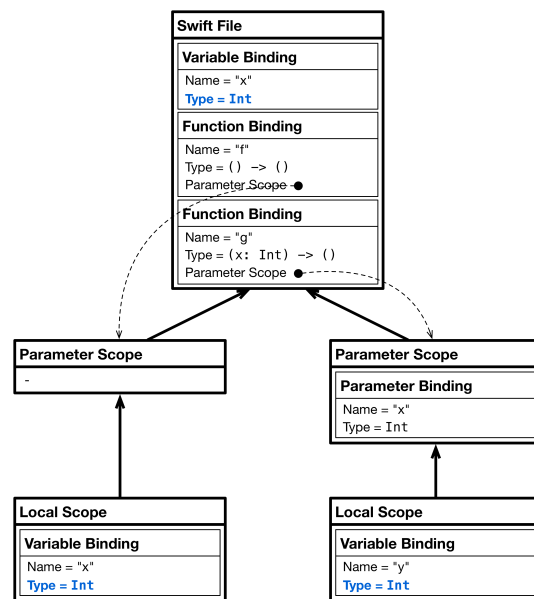
Figure 6.4: *Scope Tree after Type-Check Pass*

Listing 6.26: *Example Program*

```

1 let x = 5
2 print(x)
3
4 func f() {
5     let x = 10
6     print(x)
7 }
8
9 func g(x: Int) {
10    let y = 15
11    print(x, y)
12 }

```



As you can see, the types of the global variable binding and the two local variable bindings have been set. Now, all bindings have a corresponding type. Additionally, all the expressions in this example have been type-checked as well (not shown in figure).

The rest of this section first gives an overview of how type inference works in Swift and then shows how the type-check pass is implemented.

6.5.1 Type Inference in Swift

Section 6.1 described why the indexer needs to be able to infer the types of arbitrary expressions. This section shows some of the main characteristics of type inference in Swift.

In Swift, type annotations can often be omitted, because the type can be inferred by the compiler. Listing 6.27 shows a few examples:

Listing 6.27: *Type Inference Examples*

```

1 func getNameAndAge() -> (name: String, age: Int) {
2   return ("Toni", 26)
3 }
4 let number = 5                // number is of type Int
5 let array = ["Text"]          // array is of type Array<String>
6 let age = getNameAndAge().age  // age is of type Int
7 let fn = { x in x * 2 }        // fn is of function type (Int) -> Int

```

Note that type inference is not always possible. For example, functions, initializers and subscripts always require type annotations for their parameters and return types. Similarly, computed properties as well as stored properties without an initial value always require a type annotation.

Listing 6.27 shows a few examples where type inference does work. The type of the immutable variable `number` is inferred from its initializer expression `5`. Similarly, the type of the closure parameter `x` is inferred from the closure body expression `x * 2`.

Additionally, type inference is limited to a single statement. If a variable or constant is only initialized in a statement that follows the declaration, an explicit type annotation is required. This is shown in Listing 6.28:

Listing 6.28: *Type Inference is limited to a single statement*

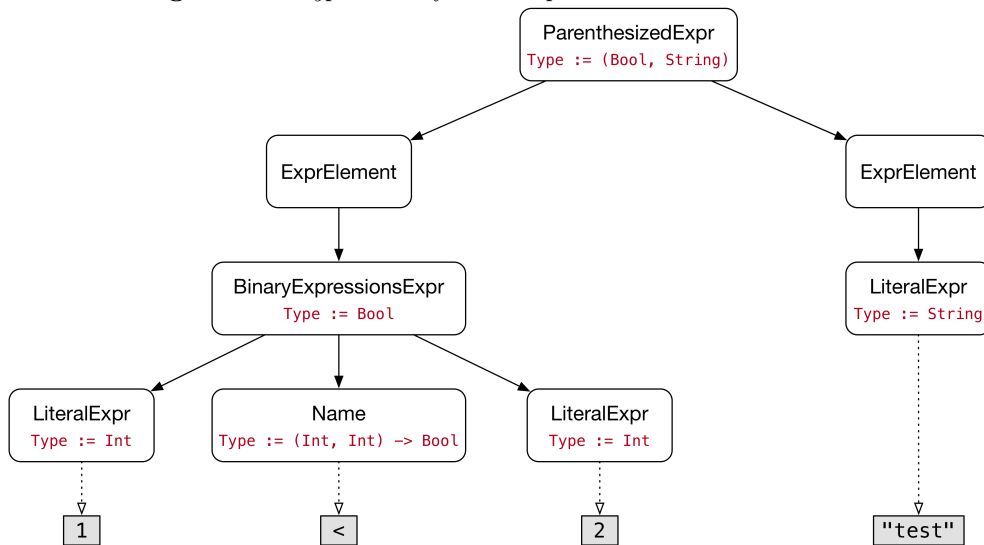
```

1 let x: Int    // type annotation is required, because
2 x = 5         // the constant is initialized in the next statement

```

Bottom-Up Type Inference

In all the examples that we have seen so far, type information flows from the bottom of the AST to the top. For example, Figure 6.5 shows the AST for the expression `(1 < 2, "test")`:

Figure 6.5: *Typed AST for the expression `(1 < 2, "test")`*

Each expression node is annotated with its type. Note that the *ExprElement* nodes are not expression nodes and therefore don't have a type. The *Name* node for the `<` operator is not an expression node either. However, it is backed by a binding which does have the function type `(Int, Int) -> Bool`.

The leaf nodes have an intrinsic type. For example, an integer literal expression defaults to the type `Int` and a string literal expression defaults to the type `String`. Similarly, an identifier expression references some entity (e.g., a variable or a function) that also has a type.

The type of an inner expression node depends on the types of its child nodes. For example, the *BinaryExpressionsExpr* node has type `Bool` which is obtained by applying the operator function of type `(Int, Int) -> Bool` to the two operands of type `Int`. Similarly, the *ParenthesizedExpr* node has type `(Bool, String)` which is obtained by creating a new tuple type with a tuple type element for each expr element.

Bi-directional Type Inference

In addition to this kind of bottom-up type inference, Swift also allows type information to flow from the root of the expression tree down to the leaves. This is called *bi-directional type inference* and is common in languages that use ML-like type systems. However, it is not present in mainstream languages like C++, Java, C#, or Objective-C [App17e]. To better understand how this works, it is useful to look at a few examples:

- **Literals**

The first example shows how bi-directional type inference works with Swift's literals. Consider the code in Listing 6.29:

Listing 6.29: *Int vs. Integer Literal*

```
1 let x = 2           // x is of type Int
2 let y: Double = 2   // OK
3 let z: Double = x   // error: cannot convert value of type 'Int' to specified type 'Double'
```

This code shows that there's a difference between a variable of type `Int` and an integer literal. The variable `x` is initialized with an integer literal and since there is no type annotation, the type of `x` defaults to `Int`. The variable `y` is also initialized with an integer literal but there is an explicit type annotation that specifies that `y` should be of type `Double`. This is valid because it is possible to create a new instance of `Double` from an integer literal. However, when we try to initialize the variable `z` which is of type `Double` with the variable `x` which is of type `Int` we get a compilation error. This happens because there is no implicit coercion from `Int` to `Double`. Instead, one would have to write `let z = Double(x)` in order to create an instance of `Double` from `x`.

Note that you can even write your own type that can be initialized with an integer literal. The way this works is by adopting a special protocol called `ExpressibleByIntegerLiteral`. An example is shown in Listing 6.30:

Listing 6.30: *Conforming to ExpressibleByIntegerLiteral*

```

1 struct EvenNumber: ExpressibleByIntegerLiteral {
2     let value: Int
3
4     init(integerLiteral: Int) {
5         guard integerLiteral % 2 == 0 else {
6             fatalError("\(integerLiteral) is not an even number")
7         }
8         self.value = integerLiteral
9     }
10 }
11
12 let n: EvenNumber = 4

```

The protocol's only requirement is that conforming types need to have an initializer with the signature `init(integerLiteral: Int)`. The types `Int` and `Double` are declared in the standard library and both types conform to the `ExpressibleByIntegerLiteral` protocol. The only thing that is special about `Int` is that it is the default type for integer literals, if the type is not otherwise constrained.

Other literals work in the same vein. For example, an array literal defaults to the type `Array` but it can also be used to create a `Set` because `Set` conforms to the `ExpressibleByArrayLiteral` protocol. An example is shown in Listing 6.31:

Listing 6.31: *Array Literals*

```

1 let arr = [1, 2, 3]           // arr is of type Array<Int>
2 let set: Set<Int> = [1, 2, 3] // set is of type Set<Int>

```

There are other ways by which a literal's type can be constrained from its context. A few examples are shown in Listing 6.32:

Listing 6.32: *Expressions with contextual type constraints*

```

1 // An integer literal is constrained to have type Double,
2 // because of f()'s return type
3 func f() -> Double {
4     return 0
5 }
6
7 // An array literal containing integer literals is constrained
8 // to have type Set<Double>, because of g()'s parameter type
9 func g(_: Set<Double>) {}
10 g([1, 2, 3])
11
12 // An integer literal is constrained to have type Double,
13 // because of the type of the switch statement's control expression
14 var x = 2.5
15 switch x {
16 case 2:
17     print("x == 2")
18 default:
19     print("x != 2")
20 }

```

Finally, it is important to note that this bi-directional type inference works even if the contextual type constraint is not coming directly from an immediate ancestor node of the literal expression. An example of this is shown in Listing 6.33:

Listing 6.33: *Bi-directional type inference over multiple levels*

```

1 func id<T>(_ x: T) -> T {
2     return x
3 }
4 let x = id(id(id(2)))           // x is of type Int (bottom-up, 3 levels)
5 let y: Double = id(id(id(2)))  // y is of type Double (top-down, 3 levels)

```

In the declaration of `x`, the generic type parameter `T` is inferred to be `Int` because of the integer literal. Consequently, `x` is inferred to be of type `Int` as well. On the other hand, in the declaration of `y`, the generic type parameter `T` is inferred to be `Double` because of the explicit type annotation. Therefore, the type of the integer literal is also set to `Double`.

- **Closures**

Closures are another one of Swift’s language features that heavily relies on bi-directional type inference. The parameter types and return types of closures are often not specified explicitly but instead inferred from the closure’s context. Listing 6.34 shows an example:

Listing 6.34: *Type Inference from Closure Context*

```

1 let numbers = [1, 2, 3, 4, 5, 6]
2 let evenNumbers = numbers.filter { n in n % 2 == 0 }
3 print(evenNumbers) // [2, 4, 6]

```

In this example, the `filter()` method expects a closure (or a function) that takes an `Int` and returns a `Bool`. Therefore, the type of the closure that is passed as argument is inferred to be `(Int) -> Bool`.

In some cases, it is even possible to determine the type of a closure from its body. This is shown in Listing 6.35:

Listing 6.35: *Type Inference from Closure Body*

```

1 let inc = { $0 + 1 }
2 print(inc(4)) // 5

```

In this example, `$0` is an implicit closure parameter and the result of the expression `$0 + 1` is implicitly used as the return value for the closure. From the closure body, the type checker can figure out that `$0` should be an `Int` and that the return type of the closure should also be `Int`. Note that this only works with closures whose bodies consist solely of a single expression or return statement.

- **Overload Resolution**

In Swift, functions can be overloaded. While the bindings for most names can be resolved before type checking, this is not the case for function names. This is because overload resolution depends on the types of a function’s arguments which are not known before type checking. In contrast to many other programming languages, overload resolution not only depends on the argument types but also on the contextual type of a function call. Listing 6.36 shows an example:

Listing 6.36: *Overload Resolution in Swift*

```

1 func f() -> Int {
2     return 2
3 }
4
5 func f() -> String {
6     return "test"
7 }
8
9 let x = f()           // error: ambiguous use of 'f()'
10 let y: Int = f()      // Overload Resolution picks f: () -> Int
11 let z: String = f()   // Overload Resolution picks f: () -> String

```

In the declaration of `x`, there is no contextual type constraint, which makes the function call `f()` ambiguous. In the declarations of `y` and `z` there is an explicit type annotation which allows the type checker to choose the correct overload with the matching return type. Note that in many programming languages it is not possible to declare two functions that only differ in their return type.

6.5.2 Implementation Approach

As a first approach, a bottom-up type checker was implemented. This approach assumes that the type of each expression depends solely on the types of its subexpressions. To obtain the type of an expression one can call the `getType()` method of the root expression node. Listing 6.37 shows what the `getType()` method of the `ParenthesizedExpr` node class looked like:

Listing 6.37: *getType() method of the ParenthesizedExpr node class*

```

1 public class ParenthesizedExpr extends ASTNode implements IPrimaryExpr {
2     private final ExprElement[] elements;
3
4     public ParenthesizedExpr(ExprElement[] elements) {
5         this.elements = elements;
6     }
7
8     @Override
9     public IType getType() {
10         if(elements.length == 1) {
11             return elements[0].getExpr().getType();
12         }
13
14         final TupleTypeElement[] typeElements = new TupleTypeElement[elements.length];
15         for(int i = 0; i < elements.length; i++) {
16             final ExprElement exprElement = elements[i];
17             final String elementName = exprElement.getName();
18             final IType elementType = exprElement.getExpr().getType();
19             final TupleTypeElement typeElement = new TupleTypeElement(elementName, elementType);
20             typeElements[i] = typeElement;
21         }
22         return new TupleType(false, typeElements);
23     }
24
25     // other methods
26 }

```

The `getType()` method in the example above creates a tuple type element for each expr element. Each tuple type element is composed of the corresponding expr element's name

and type. Finally, the method returns a new tuple type that is composed of the tuple type elements.

This approach worked fine for simple expressions, but it soon became clear that it is not powerful enough to deal with Swift’s bi-directional type inference. Thus, I decided to translate parts of the Swift compiler’s type checker from C++ to Java in order to integrate it in the Tifig IDE. Apple uses a constraint-based type checker (similar to the Hindley-Milner type inference algorithm [DM82]) in order to deal with bi-directional type inference. This is explained in more detail in the next section.

6.6 Constraint-Based Type Checker

This section gives an overview over the different steps that are performed by the constraint-based type checker. Additionally, it shows various examples that illustrate how different kinds of expressions are type checked.

6.6.1 Overview

The constraint-based type checker performs four steps: Constraint Generation, Constraint Solving, Solution Ranking and Solution Application [App17e].

Constraint Generation

The type checking of an expression starts with constraint generation. During constraint generation, the type checker assigns a type to each subexpression. Since the type of a subexpression is often not yet fully known during constraint generation, type variables are used as placeholders.

Additionally, constraints are generated which impose restrictions on the individual type variables. Before a constraint is added to the constraint system, it is *simplified*. This means that the system checks whether the constraint is already satisfied or whether it can be broken down into smaller constraints.

Constraint Solving

The constraint solver starts by assigning a fixed type to one of the type variables. The fixed type is not chosen at random, but instead represents an educated guess by the constraint solver. For example, if a type variable is used as a placeholder for the type of a literal expression the constraint solver may start by trying to use the default type for the corresponding literal kind (e.g., `Int` for integer literals or `Double` for floating-point literals). Similarly, if a type variable is used as a placeholder for the type of an overloaded name, the constraint solver may start by choosing the type of one of the overload choices.

Next, the constraint solver simplifies all constraints that involved the type variable that was just assigned a fixed type. A constraint is considered to be *solved* by the simplifier, if it is satisfied by the current choice of fixed types. If that is the case, it is removed from the constraint system. A constraint is considered to be *unsolved*, if it still contains type variables that don't have a fixed type. If that is the case, the constraint stays in the constraint system. Finally, if the solver can determine that one of the constraints can never be satisfied with the current choice of fixed types, it backtracks to the previous step in order to try a different fixed type.

If there are no more constraints in the system after the simplification is done, this means that the current choice of fixed types represents a solution to the constraint system. This solution is then stored and the solver backtracks in order to look for additional solutions by trying out different combinations of fixed types.

If there are still unsolved constraints after the simplification is done, the whole process repeats and the solver makes the next guess and assigns a fixed type to a different type variable.

Thus, the solution space explored by the solver can be viewed as a tree. The root node of the tree is the constraint system that directly results from the constraint generation. Each other node is a constraint system that was derived from the root constraint system and the path from the root node to another node represents the guesses that the solver made in order to derive the corresponding constraint system. The leaves are either constraint systems that represent a solution (i.e., all constraints were simplified and all type variables have a fixed type) or they are constraint systems where one or more constraints are not satisfiable anymore.

Solution Ranking

If the constraint solver didn't find any solutions, it means that the constraint system is *unsatisfiable* or in other words, the expression is ill-typed (i.e., it contains a type error).

If there is exactly one solution, the expression is well-typed and the type checker applies this solution to the expression.

Finally, if there are multiple solutions, the solutions are ranked in order to determine whether there is a single solution that is better than all other solutions. If no such solution is found, the expression is considered to be *ambiguous* which is also a type error. Otherwise, the expression is well-typed and the type checker applies the best solution to the expression.

Solution Application

During solution application, type variables that occur in the types of the original expression and its subexpressions are replaced by their corresponding fixed type. Thus, after solution application the expression should have a valid type that doesn't contain any type variables. Additionally, overloaded names are resolved to the corresponding overload choice that was determined by the constraint solver.

6.6.2 Example 1: Literals

The first example shows how the type checker handles bi-directional type inference of literal expressions. The code for this example is shown in Listing 6.38:

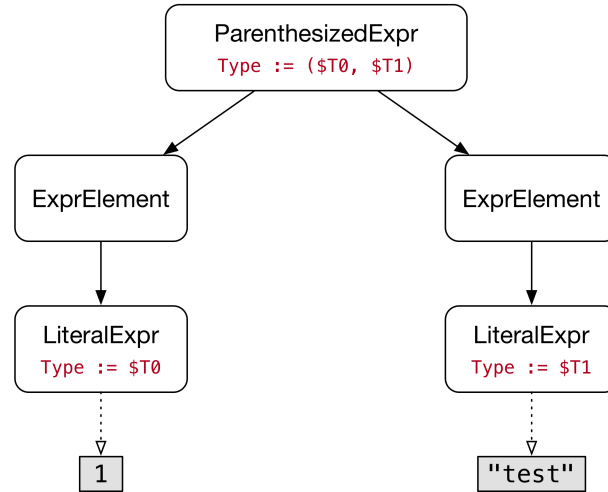
Listing 6.38: Code for Example 1

```
1 let x: (Double, String) = (1, "test")
```

Constraint Generation

The constraint generator walks the AST of the initializer expression (1, "test") in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.6:

Figure 6.6: AST for expression (1, "test") after constraint generation



For literal expressions, the constraint generator creates a fresh type variable. On the other hand, the type of the parenthesized expression is a tuple type that is composed of the types of its elements.

In addition to creating type variables and assigning types to subexpressions, the constraint generator also creates constraints. The following list describes the four constraints that are generated for the example above:

- **\$T0 LiteralConformsTo ExpressibleByIntegerLiteral**
This constraint means that the fixed type of \$T0 (i.e., the type of the literal expression 1) has to conform to the `ExpressibleByIntegerLiteral` protocol.
- **\$T1 LiteralConformsTo ExpressibleByStringLiteral**
This constraint means that the fixed type of \$T1 (i.e., the type of the literal expression "test") has to conform to the `ExpressibleByStringLiteral` protocol.
- **\$T0 Conversion Double**
This constraint means that the fixed type of \$T0 must be convertible to `Double`.

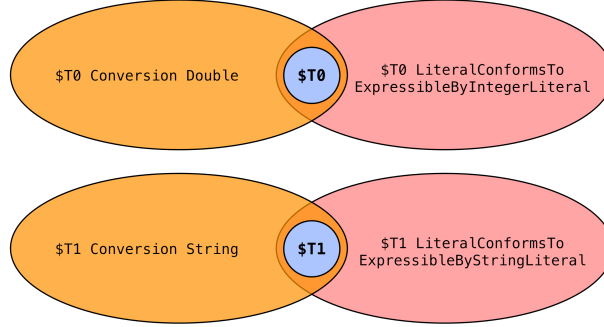
- **\$T1 Conversion String**

This constraint means that the fixed type of **\$T1** must be convertible to **String**.

Note that the constraint generator first actually generates the constraint (**\$T0**, **\$T1**) **Conversion (Double, String)** as a result of the explicit type annotation of the variable **x**. However, this constraint is immediately simplified into the two smaller constraints **\$T0 Conversion Double** and **\$T1 Conversion String**.

Internally, these type variables and constraints are stored in a constraint graph where the type variables are the vertices and the constraints are the edges. The constraint graph is a *hypergraph* which means that edges can join any number of vertices [Sap17]. In other words, an edge is an element of $\mathcal{P}(V) \setminus \{\emptyset\}$ where $\mathcal{P}(V)$ is the power set of V and V is the set of all vertices. Thus, any given constraint connects a type variable to zero or more other type variables. Figure 6.7 shows the constraint graph for the example above:

Figure 6.7: *Constraint Graph*



Each of the two type variables has two edges and each edge is represented as a set that contains only a single type variable.

In addition to the constraint graph, the type variables and the constraints are also tracked by the constraint system. Figure 6.8 shows an overview of the root constraint system after constraint generation:

Figure 6.8: *Root Constraint System*

Constraint System
Type Variables \$T0 := null \$T1 := null
Inactive Constraints <ul style="list-style-type: none"> ▪ \$T0 LiteralConformsTo ExpressibleByIntegerLiteral ▪ \$T1 LiteralConformsTo ExpressibleByStringLiteral ▪ \$T0 Conversion Double ▪ \$T1 Conversion String
Active Constraints <ul style="list-style-type: none"> ▪ -

The type variables **\$T0** and **\$T1** are set to **null** because they don't have a fixed type yet. Note also that the constraints are divided into active and inactive constraints. This is

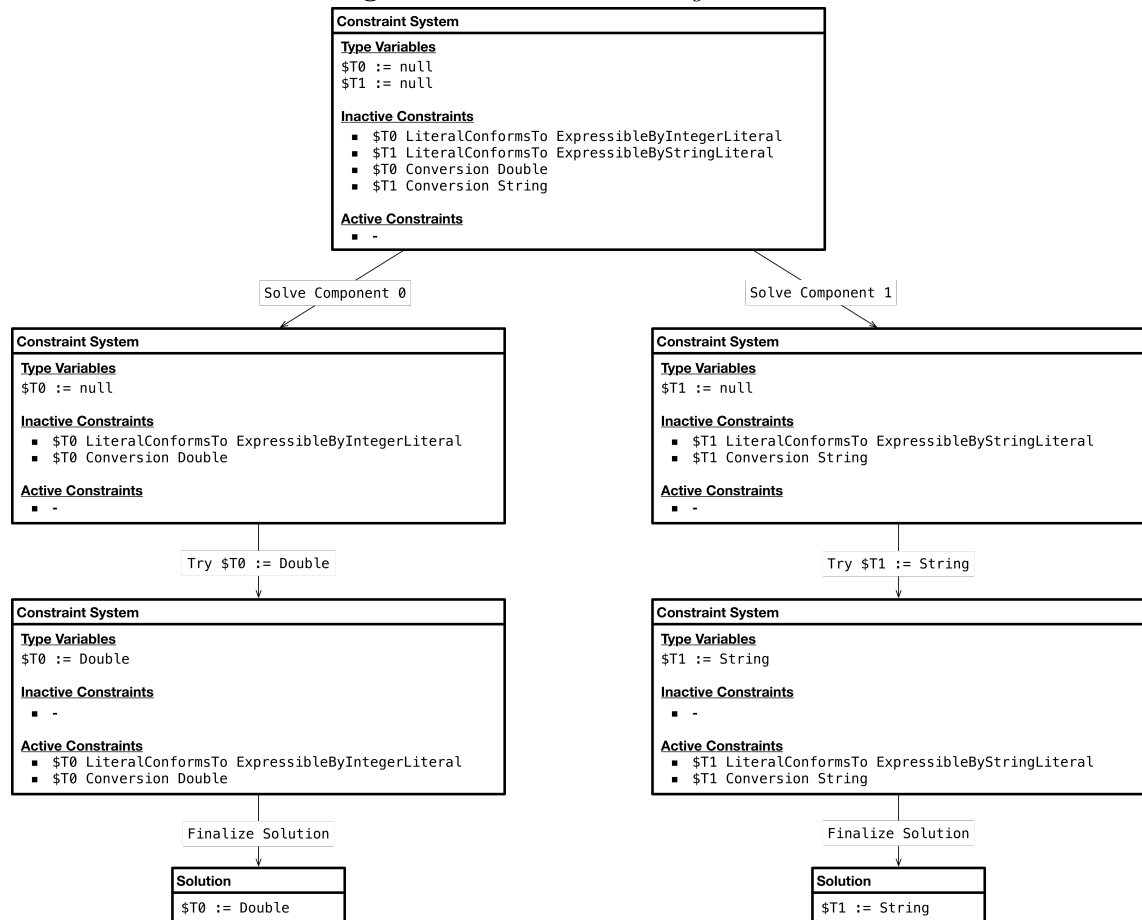
explained in the next section.

Constraint Solving

Figure 6.7 also shows that there are two connected components in the graph [J.A10]. The constraint solver can solve each connected component independent of the other connected components. This can lead to better performance because the solver may have to explore fewer combinations of fixed types. For example, let's assume that there are two type variables `$T0` and `$T1`. For `$T0` the type checker tries 2 different fixed types and for `$T1` the type checker tries 4 different fixed types. If `$T0` and `$T1` are in the same connected component, the solver needs to test all $2 * 4 = 8$ combinations. On the other hand, if the two type variables are in separate connected components, the solver needs to test 2 combinations for the first component and 4 combinations for the second component resulting in a total of only 6 combinations.

For the purposes of the example 1, the component that contains `$T0` is referred to as component 0 and the component that contains `$T1` is referred to as component 1. The constraint solving process for this example is illustrated in Figure 6.9:

Figure 6.9: *Constraint Solving Process*



First, the constraint solver tries to solve component 0. To do this, all type variables and constraints that don't belong to component 0 are temporarily removed from the constraint system.

The constraint solver then tries to find a potential binding for the type variable `$T0` (i.e., a type that could be used as the fixed type of `$T0`). In order to do that, the solver looks at the constraints that reference `$T0` (i.e., the edges of that type variable in the constraint graph). The constraint solver determines that the type `Double` may be a good potential binding for that type variable. Thus, it assigns the fixed type `Double` to the type variable `$T0`. At the same time, constraints that reference `$T0` are activated (i.e., they are moved from the list of inactive constraints to the list of active constraints).

Afterwards, the constraint solver tries to simplify each active constraint. Since `Double` conforms to the `ExpressibleByIntegerLiteral` protocol and is also convertible to `Double` (i.e., convertible to itself), the two constraints are satisfied and therefore removed from the constraint system. Because there are no more constraints in the system, the solver has already found a solution for component 0.

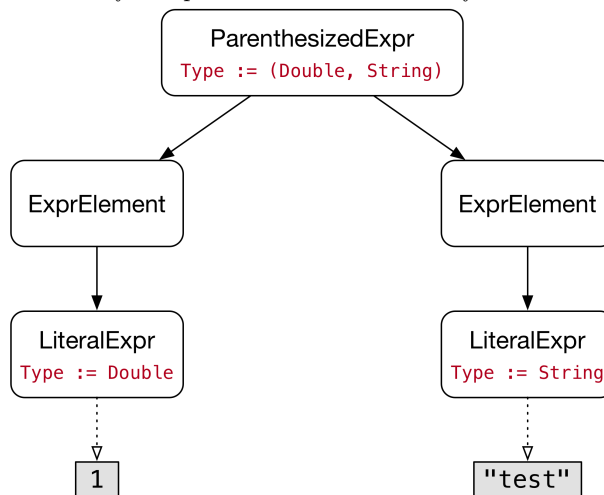
Next, the constraint solver tries to solve component 1. Based on the two constraints that reference `$T1` it tries the type `String` as a potential binding for that type variable. This satisfies both constraints in the system and leads to a solution for component 1.

Finally, the two partial solutions are combined into a single, final solution for the root constraint system. Since there is only one solution, the solution ranking step can be skipped.

Solution Application

The final solution is then applied to the original expression which results in the fully typed AST shown in Figure 6.10:

Figure 6.10: AST for expression `(1, "test")` after solution application



6.6.3 Example 2: Overload Resolution

The second example shows how the type checker resolves calls to overloaded functions. The code for this example is shown in Listing 6.39:

Listing 6.39: Code for Example 2

```

1 func printDescription(_ value: Any) {
2   print("Value: \(value)")
3 }
4
5 func printDescription(_ value: Int) {
6   let parity = value % 2 == 0 ? "even" : "odd"
7   let sign = value >= 0 ? "positive" : "negative"
8   print("Value: \(value)")
9   print("Parity: \(parity)")
10  print("Sign: \sign)")
11 }
12
13 func printDescription(_ value: String) {
14   let count = value.characters.count
15   print("Value: \value)")
16   print("Character Count: \count)")
17 }
18
19 printDescription(5)

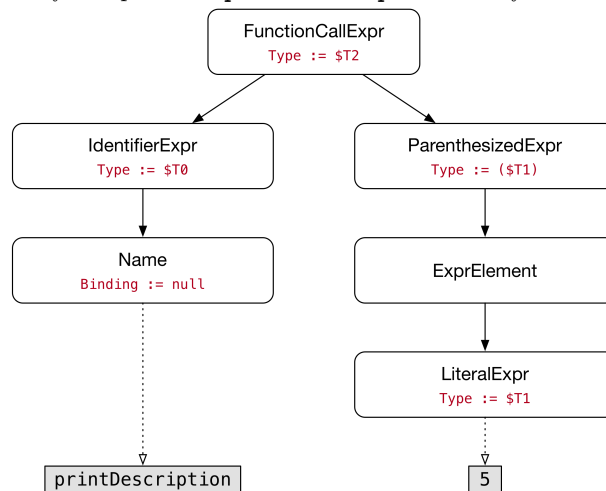
```

In this example, there are three functions called `printDescription()`. The job of these functions is to print a description of their argument. Depending on the argument type, a different function overload is called. This section explains how the type checker resolves and type checks the function call `printDescription(5)`.

Constraint Generation

The constraint generator walks the AST of the expression `printDescription(5)` in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.11:

Figure 6.11: AST for expression `printDescription(5)` after constraint generation



Since the overloaded name `printDescription` is not yet resolved, the constraint generator creates a fresh type variable for the corresponding identifier expression. Like in the first example, the type checker also creates a fresh type variable for the integer literal expression 5. This makes it possible to infer the type of the literal expression based on its context. Finally, since the function name is not yet resolved, the constraint generator also doesn't know the type of the function and therefore cannot determine the type of the overall function call expression. Thus, another fresh type variable is created.

In addition to creating type variables and assigning types to subexpressions, the constraint generator also creates constraints. The following list describes the constraints that are generated for the example above:

- **Disjunction Constraint**

- `$T0 BindOverload printDescription: (Any) -> ()`
- `$T0 BindOverload printDescription: (Int) -> ()`
- `$T0 BindOverload printDescription: (String) -> ()`

A disjunction constraint contains multiple nested constraints and is satisfied, if one of these nested constraints is satisfied. This type of constraint is used for overload resolution. In our example, the constraint generator creates a disjunction constraint that contains three nested constraints, each of which binds `$T0` to one of the three `printDescription` overloads.

- **`$T1 LiteralConformsTo ExpressibleByIntegerLiteral`**

This constraint means that the fixed type of `$T1` (i.e., the type of the literal expression 5) has to conform to the `ExpressibleByIntegerLiteral` protocol.

- **`($T1) -> $T2 ApplicableFunction $T0`**

This constraint means that the fixed type of `$T0` must be a function type which has a single, required parameter and the argument, which is of type `$T1` must be convertible to the type of that parameter. Additionally, the return type of that function type must be equal to `$T2`.

Again, the type variables and the constraints together form a constraint graph which is shown in Figure 6.12. Additionally, Figure 6.13 shows an overview of the root constraint system after constraint generation:

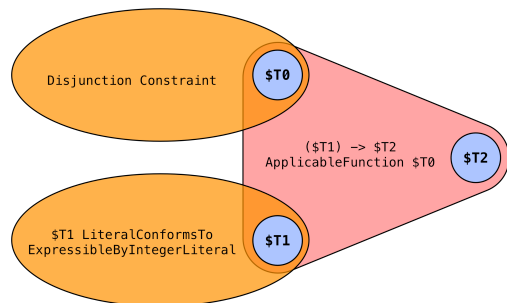


Figure 6.12: *Constraint Graph*

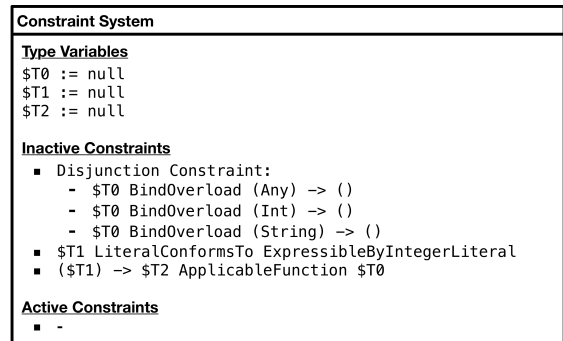


Figure 6.13: *Root Constraint System*

Constraint Solving

Compared to the example 1 there is only one connected component in this example, because all three type variables are connected by the *ApplicableFunction* constraint.

The constraint solving process for this example is illustrated in Figure 6.14. From the root constraint system the solver starts by trying out the different nested constraints of the disjunction constraint. Each nested constraint can immediately be simplified which causes the type variable `$T0` to be bound to the type of the corresponding overload.

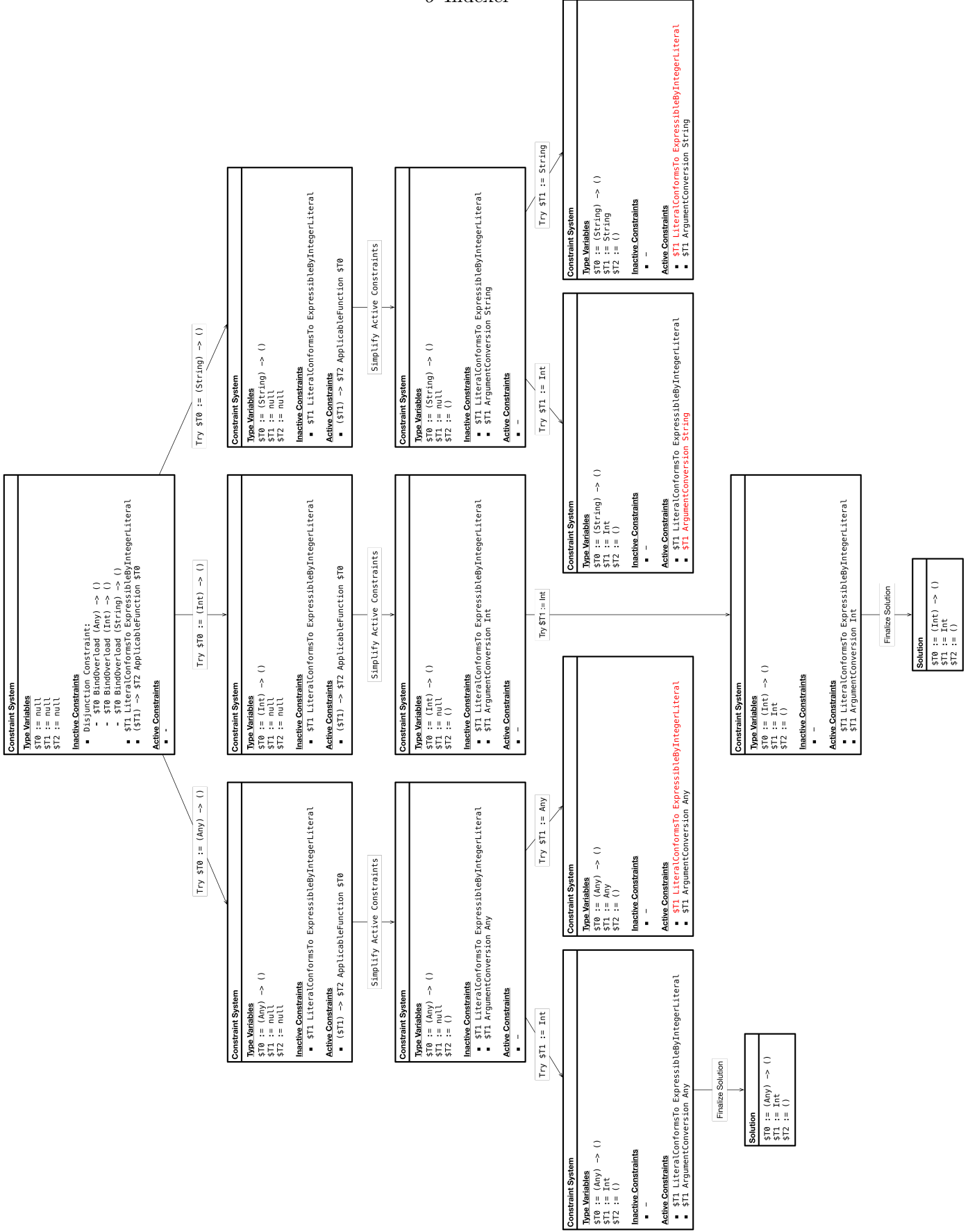
This also activates the *ApplicableFunction* constraint. During simplification this constraint is then replaced by two smaller constraints. The first one is an *ArgumentConversion* constraint which indicates that the argument which is of type `$T1` must be convertible to the parameter type of the corresponding overload. The second one records the fact that the return type of the chosen function must be equal to `$T2` (i.e., the type variable that was used as placeholder for the type of the function call expression). This constraint is immediately simplified which causes the type variable `$T2` to be bound to the return type of the corresponding function (which is the empty tuple type for all three overloads).

Next, the solver tries different potential bindings for the type variable `$T1`. Again, the potential bindings are derived from the constraints that involve `$T1`. For example, in the leftmost branch of the tree, the solver tries the potential binding `$T1 := Int` because of the constraint `$T1 LiteralConformsTo ExpressibleByIntegerLiteral` as well as `$T1 := Any` because of the constraint `$T1 ArgumentConversion Any`. Only the first one leads to a solution because the constraint `$T1 LiteralConformsTo ExpressibleByIntegerLiteral` cannot be satisfied if `$T1` is bound to the type `Any`.

Overall, the solver finds two solutions: one for the function `printDescription: (Any) -> ()` and one for the function `printDescription: (Int) -> ()`. The other leaves of the solver tree contain a constraint that is marked red. This is to indicate the constraint that cannot be satisfied on this path of the tree which causes the solving process to backtrack.

Note that the constraint solver doesn't search these branches of the solver tree simultaneously. Instead, it explores the solution space in a depth-first manner.

Figure 6.14: Constraint Solving Process



Solution Ranking

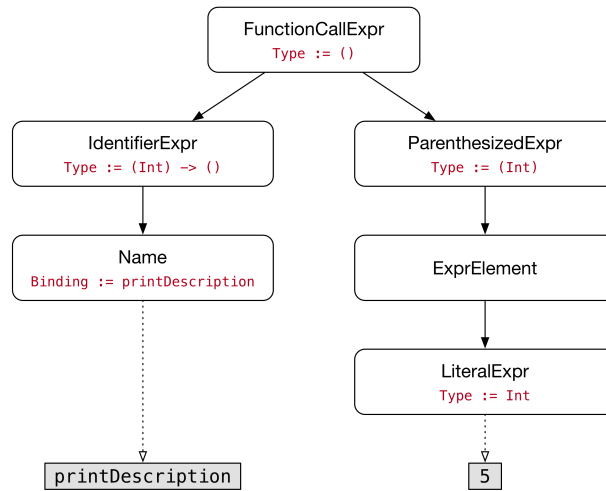
Since the constraint solver found more than one solution for the root constraint system, solution ranking comes into play. First, the ranking algorithm computes the difference between the two solutions. In this example, the two solutions differ only in the overload that was chosen for the function name `printDescription`. The system then checks whether one of the two overloads is more specialized than the other. If this is the case, the system prefers the more specialized overload. Thus, it comes to the conclusion that the solution that picks the overload `printDescription: (Int) -> ()` is better than the solution that picks `printDescription: (Any) -> ()`.

More ranking rules will be explained in subsection 6.6.11.

Solution Application

This final solution is then applied to the original expression which results in the fully typed AST shown in Figure 6.15:

Figure 6.15: *AST for expression `printDescription(5)` after solution application*



6.6.4 Example 3: Binary Expressions

Section 3.1.11 showed how new operators can be declared and how existing operators can be overloaded by declaring additional operator functions. For prefix and postfix operators this means that type checking works just like the type checking of a function call as shown in subsection 6.6.3. For example, listings 6.40 and 6.41 are semantically equivalent. Note that the expression `!!b1` would be a valid expression that performs double negation in other languages like Java and C++. However, in Swift we need to add parentheses around the inner prefix expression. Otherwise, the parser would parse this as a single prefix expression with an operator called `!!`. This is because the operators are not yet known at parse time.

Listing 6.40: *Nested Prefix Expressions*

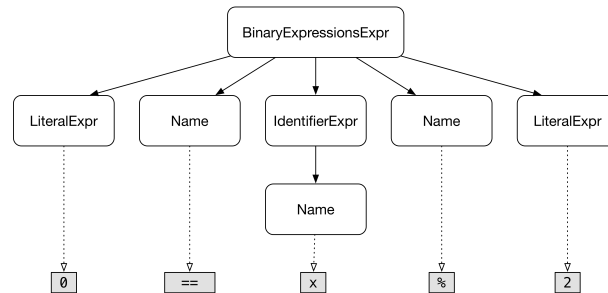
```
1 let b1 = true
2 let b2 = !(b1)
```

Listing 6.41: *Nested Function Calls*

```
1 let negate = (!)
2 let b1 = true
3 let b2 = negate(negate(b1))
```

Type checking binary expressions is a bit more complicated. Since the infix operators are not yet known at parse time, the parser also doesn't know their precedence and associativity. Thus, Tifig parses a series of binary expressions as a flat list. For example, Figure 6.16 shows the AST for the expression `0 == x % 2`:

Figure 6.16: *AST for expression `0 == x % 2`*



However, for type checking we need to know the precedence and associativity of the individual operators. For example, the expression `0 == x % 2` is only valid if the `%` operator has a higher precedence than the `==` operator. To make it easier for the type checker, Tifig's indexer temporarily creates a “shadow tree” for each binary expression which encodes the precedence and associativity of the infix operators in the tree structure. To see how this works let's look at the code for example 3 which is shown in Listing 6.42:

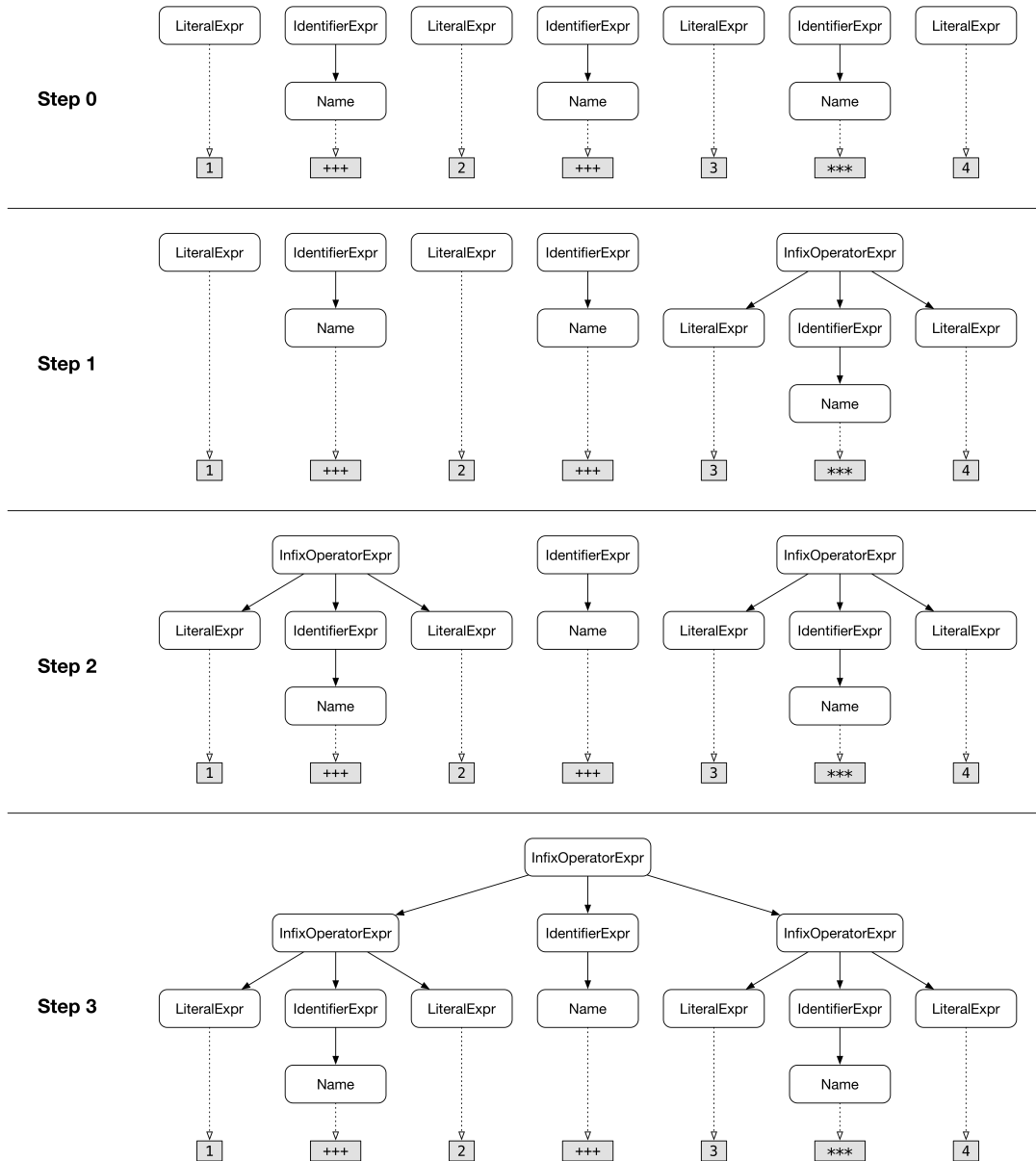
Listing 6.42: *Code for Example 3*

```
1 infix operator ***: MultiplicationPrecedence
2 infix operator +++: AdditionPrecedence
3
4 func ***(lhs: Int, rhs: Int) -> Int { return lhs * rhs }
5 func +++(lhs: Int, rhs: Int) -> Int { return lhs + rhs }
6
7 1 +++ 2 +++ 3 *** 4
```

This example declares two new infix operators `***` and `+++` which behave exactly like the standard library operators `*` and `+`. This makes it easier to explain the constraint solving process, because all operator overloads are shown in the code for this example.

To create the shadow tree, the indexer first looks up the precedence group of each individual operator. These precedence groups are stored in a sorted map. The indexer then repeatedly picks the next operator which belongs to the highest precedence group and collapses the two corresponding operands into an `InfixOperatorExpr` node. Once there are no more operators that belong to the highest precedence group, the process restarts with operators that belong to the second-highest precedence group. This process is illustrated in Figure 6.17:

Figure 6.17: Building a shadow tree for the expression `1 +++ 2 +++ 3 *** 4`

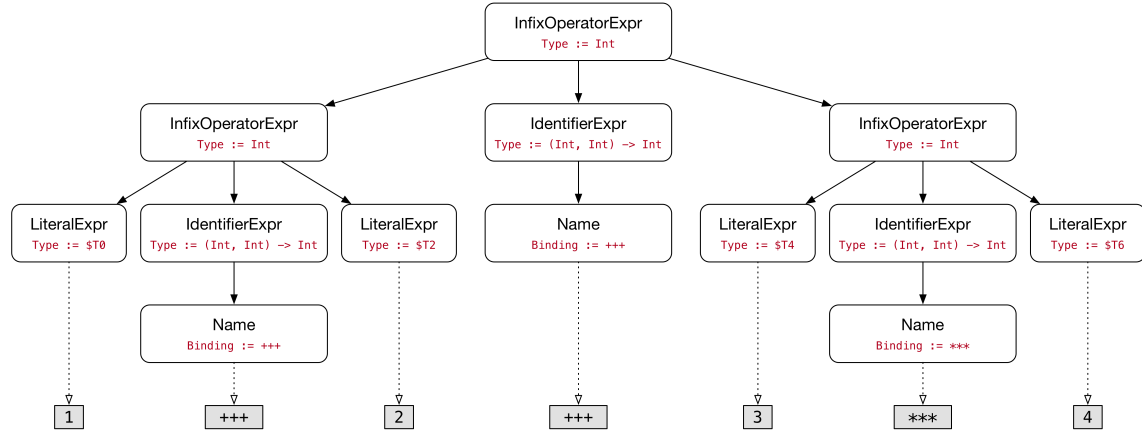


Step 0 shows all the operands and operators of the expression `1 +++ 2 +++ 3 *** 4`. In step 1, the process collapses the two operands of the `***` operator into a new `InfixOperatorExpr` node. This is because the `***` operator has higher precedence than the `+++` operator. Since there are no more `***` operators, the process continues with the `+++` operator. In step 2, the process collapses the two operands of the first `+++` operator into a new `InfixOperatorExpr` node. Note that it starts with the first `+++` operator, because the operator is left-associative. Finally, in step 3 the two operands of the second `+++` operator are collapsed into a new `InfixOperatorExpr` node as well.

Constraint Generation

The constraint generator walks the AST of the expression `1 +++ 2 +++ 3 *** 4` in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.18:

Figure 6.18: AST for expression `1 +++ 2 +++ 3 *** 4` after constraint generation



For literal expressions, the constraint generator creates a fresh type variable. The same thing happens for the identifier expressions of the operators, but in this example, there is only one overload for each of the two operator names `+++` and `***`. Thus, the constraint generator can already fill in the fixed types. This has the effect that the types of the individual infix operator expressions are also already known.

In addition to creating type variables and assigning types to subexpressions, the constraint generator also creates constraints. The following list describes the constraints that are generated for the example above:

- `$T0 LiteralConformsTo ExpressibleByIntegerLiteral`
- `$T2 LiteralConformsTo ExpressibleByIntegerLiteral`
- `$T4 LiteralConformsTo ExpressibleByIntegerLiteral`
- `$T6 LiteralConformsTo ExpressibleByIntegerLiteral`

These four constraints mean that the fixed types of the type variables that were generated for the individual literal expressions must conform to the `ExpressibleByIntegerLiteral` protocol.

- `$T0 OperatorArgumentConversion Int`
- `$T2 OperatorArgumentConversion Int`
- `$T4 OperatorArgumentConversion Int`
- `$T6 OperatorArgumentConversion Int`

These four constraints mean that the fixed types of the type variables that were generated for the individual literal expressions must be convertible to `Int`.

Note that the four *OperatorArgumentConversion* constraints are the result of the simplification of the constraints $(\$T0, \$T2) \rightarrow \text{Int}$ *ApplicableOperatorFunction* $\$T1$ and $(\$T4, \$T6) \rightarrow \text{Int}$ *ApplicableOperatorFunction* $\$T5$. Additionally, the constraint $(\text{Int}, \text{Int}) \rightarrow \text{Int}$ *ApplicableOperatorFunction* $\$T3$ can be simplified away entirely before it is ever added to the constraint system.

The constraint graph and the root constraint system for this example are shown in Figure 6.19 and Figure 6.20, respectively:

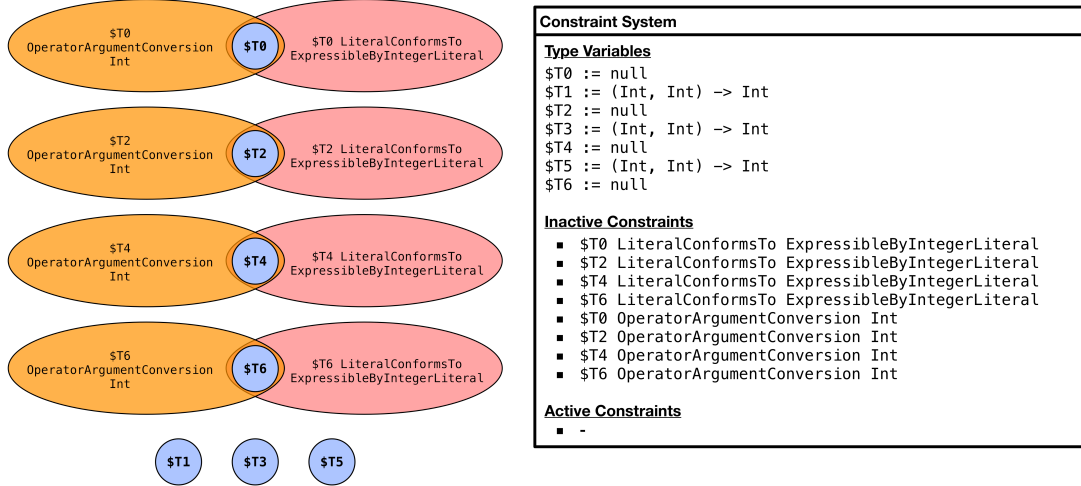


Figure 6.19: Constraint Graph

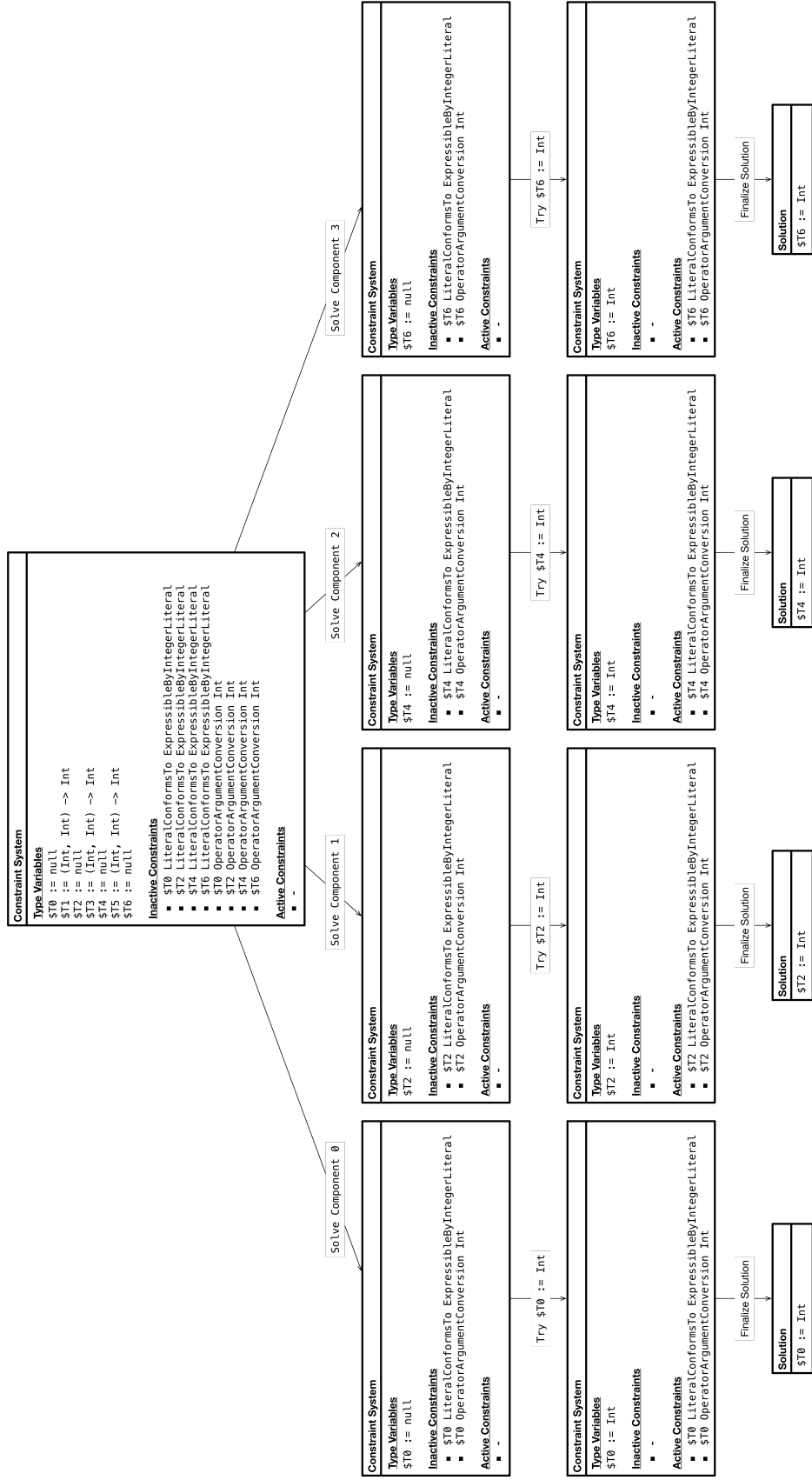
Figure 6.20: Root Constraint System

Constraint Solving

Figure 6.19 shows that there are seven connected components in the graph. The constraint solver ignores the connected components containing the type variables $\$T1$, $\$T3$ and $\$T5$ because these type variables already have a fixed type and there are no constraints in these components.

The constraint solving process for this example is pretty straightforward. The connected components look very similar since each of them contains only a single type variable, one *LiteralConformsTo* constraint and one *OperatorArgumentConversion* constraint. The constraint solver solves each of these components in turn as shown in Figure 6.21:

Figure 6.21: Constraint Solving Process

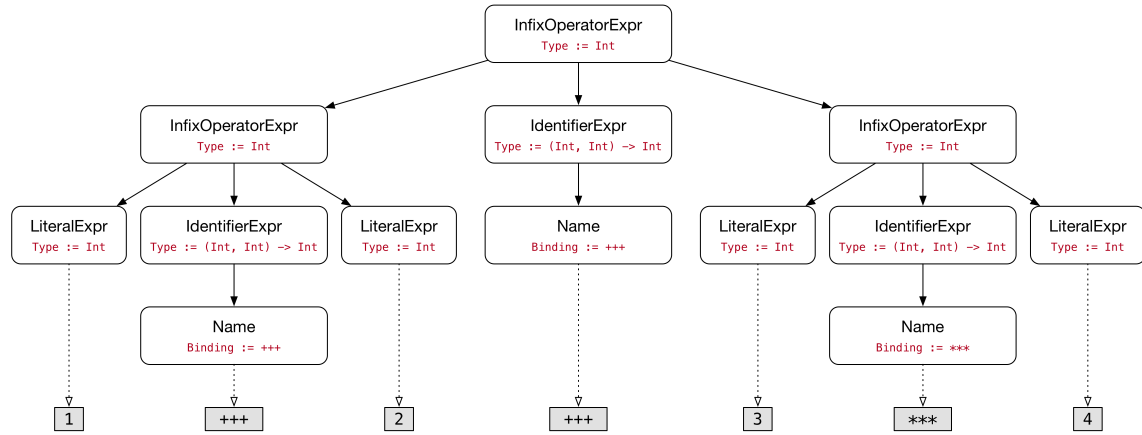


It starts by trying out the fixed type `Int` for the corresponding type variable. This satisfies both constraints and leads to a solution for the corresponding component. Finally, the four partial solutions are combined into a single, final solution for the root constraint system.

Solution Application

The final solution is then applied to the original expression which results in the fully typed AST shown in Figure 6.22:

Figure 6.22: *AST for expression `1 +++ 2 +++ 3 *** 4` after solution application*



6.6.5 Example 4: Explicit Member Expression

This example shows how explicit member expressions are type checked. The code for this example is shown in Listing 6.43:

Listing 6.43: Code for Example 4

```

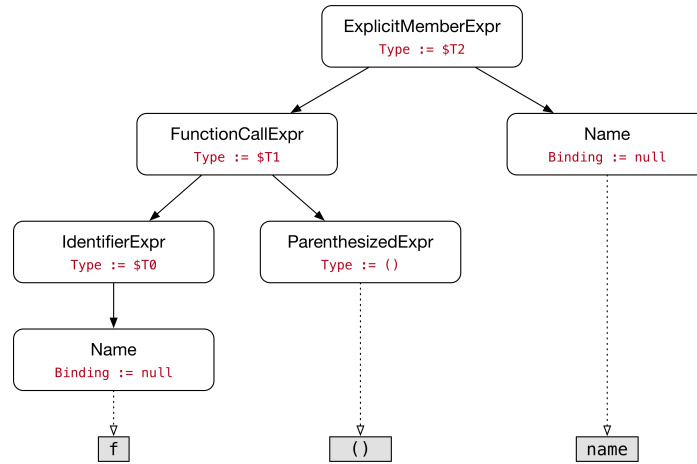
1 struct Person {
2   let name: String
3 }
4
5 func f() -> Person {
6   return Person(name: "Steve")
7 }
8
9 func f() -> Int {
10  return 0
11 }
12
13 f().name

```

Constraint Generation

The constraint generator walks the AST of the explicit member expression `f().name` in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.23:

Figure 6.23: AST for expression `f().name` after constraint generation



Since the overloaded name `f` is not yet resolved, the constraint generator creates a fresh type variable for the corresponding identifier expression. This also means that we don't yet know the type of the function call expression as well as the type of the explicit member expression. Thus, two additional type variables are created for these expressions.

The following list describes the constraints that are generated for the example above:

- **Disjunction Constraint**
 - `$T0 BindOverload f: () -> Person`
 - `$T0 BindOverload f: () -> Int`

This disjunction constraint contains two nested constraints, each of which binds `$T0` to one of the two `f` overloads. One of these nested constraints must be satisfied in order for the disjunction constraint itself to be satisfied.

- **`() -> $T1 ApplicableFunction $T0`**
This constraint means that the fixed type of `$T0` should be a function type which has no required parameters. Additionally, the return type of that function type should be equal to `$T1`.
- **`$T1.name ValueMember $T2`**
This constraint means that the fixed type of `$T1` must have a member that is called “name” and is of type `$T2`.

The constraint graph and the root constraint system for this example are shown in Figure 6.24 and Figure 6.25, respectively:

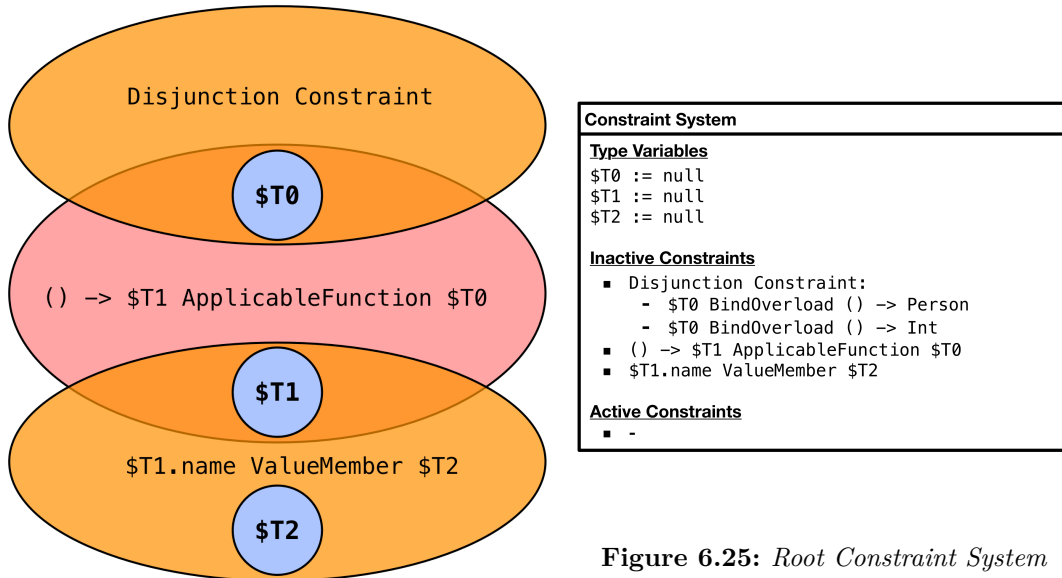


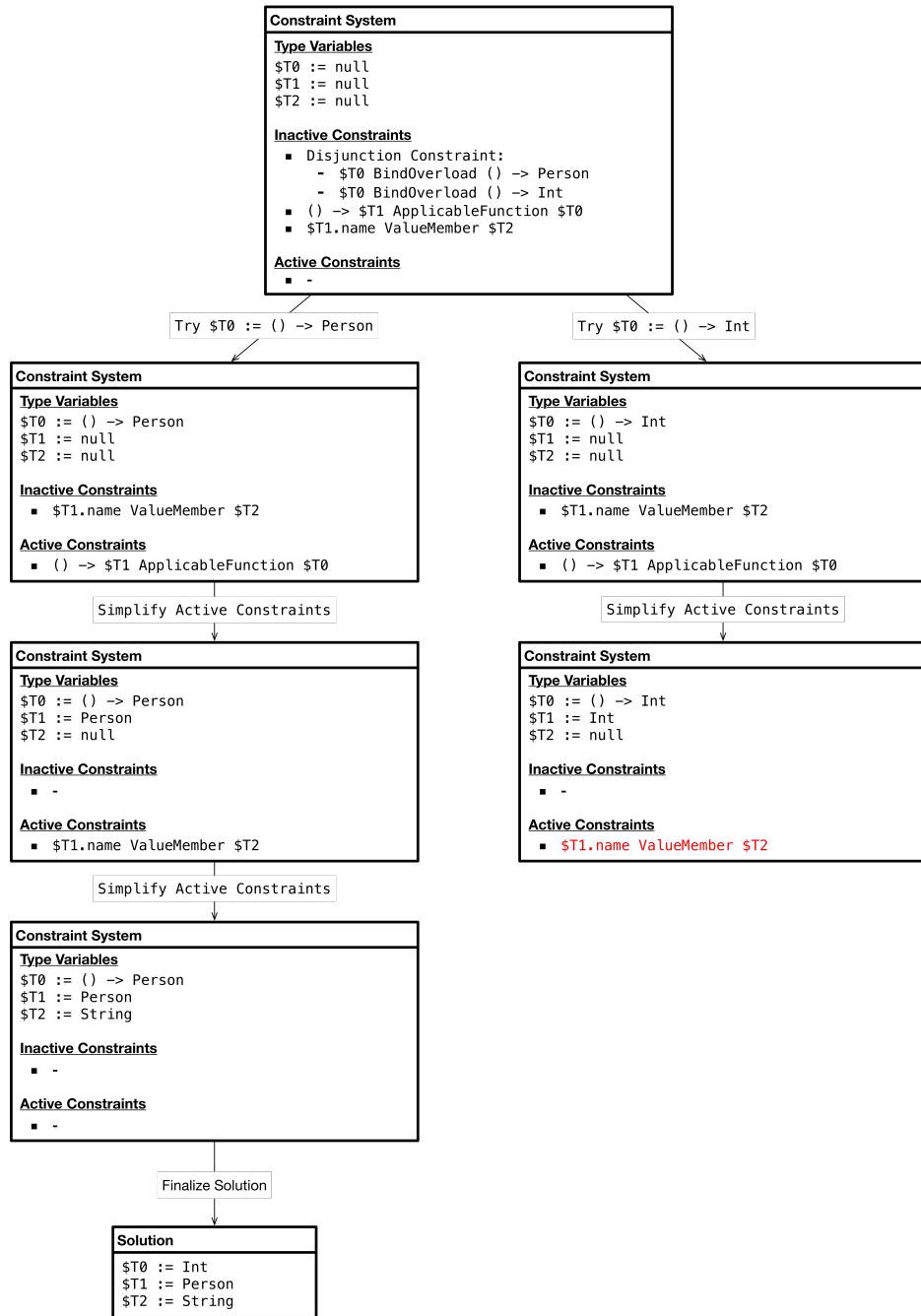
Figure 6.24: *Constraint Graph*

Figure 6.25: *Root Constraint System*

Constraint Solving

The constraint solving process for this example is illustrated in Figure 6.26:

Figure 6.26: Constraint Solving Process



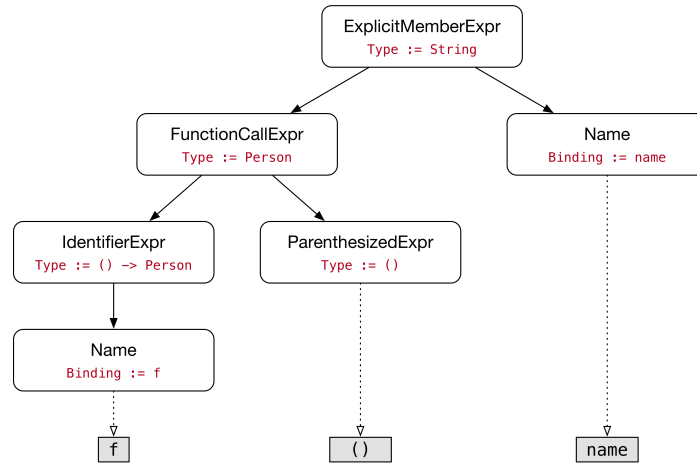
The solver starts by picking the first overload and therefore sets the fixed type of $\$T0$ to $() \rightarrow \text{Person}$. The simplification of the constraint $() \rightarrow \$T1 \text{ ApplicableFunction } \$T0$ then sets the fixed type of $\$T1$ to Person . Finally, the constraint $\$T1.\text{name ValueMember } \$T2$ is simplified. In the process, the simplifier looks for instance members called “name” in the struct type Person . It finds a **name** property which is of type String . Thus, the fixed type of $\$T2$ is set to String and all constraints are satisfied. The solver creates a solution that contains the corresponding type bindings for the different type variables.

Next, the solver backtracks in order to look for additional solutions. It picks the second overload and therefore sets the fixed type of $\$T0$ to $() \rightarrow \text{Int}$. The *ApplicableFunction* constraint can be simplified and the fixed type of $\$T1$ is set to Int . However, since the Int type doesn’t have a member called “name”, the simplification of the constraint $\$T1.\text{name ValueMember } \$T2$ fails. Thus, there is only a single solution for this constraint system.

Solution Application

The final solution is then applied to the original expression which results in the fully typed AST shown in Figure 6.27:

Figure 6.27: *AST for expression $f().\text{name}$ after solution application*



6.6.6 Example 5: Implicit Member Expression

This example shows how implicit member expressions are type checked. The code for this example is shown in Listing 6.44:

Listing 6.44: *Code for Example 5*

```

1 enum DayOfTheWeek {
2   case monday, tuesday, wednesday, thursday, friday, saturday, sunday
3 }
4
5 var day = DayOfTheWeek.monday
6 day = .friday

```

On line 5 of this example, the explicit member expression `DayOfTheWeek.monday` is used to initialize the variable `day`. This works because enum cases are considered to be static members of their enclosing enum type. The type of the variable `day` is inferred to be `DayOfTheWeek` because enum cases are instances of their enclosing enum type.

On line 6, the value of the variable `day` is changed. This time, an implicit member expression is used, because the type of `day` is already known and the type checker can therefore infer that `.friday` refers to `DayOfTheWeek.friday`.

Note that this works not just with enum cases but with any kind of static member that is an instance of its enclosing type. For example, Listing 6.45 shows how this can be used in Apple’s UI framework `UIKit`:

Listing 6.45: *UIKit Example*

```

1 import UIKit
2
3 let errorLabel = UILabel()
4 errorLabel.text = "An error occurred!"
5 errorLabel.textColor = .red

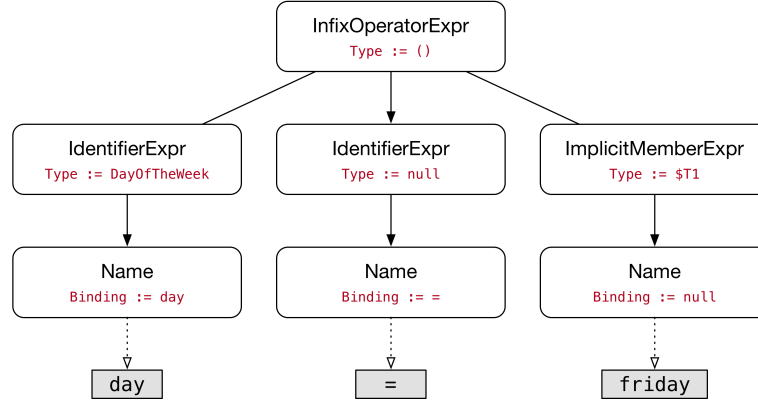
```

In this example, the `textColor` property of `UILabel` is of type `UIColor`. This type is a class type and not an enum type. It defines several static properties to quickly access well-known colors (e.g., `UIColor.blue`, `UIColor.green`, `UIColor.red`). Since the type is already known from the context (i.e., the assignment to the `textColor` property), we can use `.red` instead of `UIColor.red`.

Constraint Generation

The constraint generator walks the AST of the expression `day = .friday` in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.28:

Figure 6.28: *AST for expression `day = .friday` after constraint generation*



The constraint generator creates a fresh type variable `$T0` for the identifier expression `day`. Since the name `day` is not overloaded, it can be resolved immediately and the fixed type of `$T0` is set to `DayOfTheWeek`.

The assignment operator is resolved to the implicit operator binding `=` that was generated during the definition pass. The type of this operator binding is set to `null`, but it doesn't matter, because the operator is not treated like a normal operator and the built-in behaviour doesn't rely on the type of the operator (see below).

For the implicit member expression `.friday`, the constraint generator creates two type variables: `$T1` for the base type and `$T2` for the type of the member `friday`.

Finally, the type of the assignment expression is set to the empty tuple `()`. This is different from other languages like Java or C++ where assignment expressions have the same type as the variable that is assigned a new value.

The following list describes the constraints that are generated for the example above:

- **Metatype(`$T1`).friday UnresolvedValueMember `$T2`**
This constraint means that `$T1` must have a static member that is called “friday” and is of type `$T2`.
- **`$T2` Conversion `$T1`**
This constraint means that the type of the static member “friday” (i.e., `$T2`) must be convertible to its enclosing type (i.e., `$T1`).
- **`$T1` Conversion `DayOfTheWeek`**
This constraint results from the assignment expression and it means that the type of the right hand side (i.e., `$T1`) must be convertible to the type of the left hand side (i.e., `DayOfTheWeek`).

Figure 6.29 shows the root constraint system for this example:

Figure 6.29: *Root Constraint System*

Constraint System
Type Variables \$T0 := DayOfTheWeek \$T1 := null \$T2 := null
Inactive Constraints <ul style="list-style-type: none"> Metatype(\$T1).friday UnresolvedValueMember \$T2 \$T2 Conversion \$T1 \$T1 Conversion DayOfTheWeek
Active Constraints <ul style="list-style-type: none"> -

Constraint Solving

The constraint solver starts by trying the potential binding `$T1 := DayOfTheWeek` based on the constraint `$T1 Conversion DayOfTheWeek`. During the simplification of the *UnresolvedValueMember* constraint, the simplifier looks for static members called “friday” in the enum type `DayOfTheWeek`. It finds the enum case `friday` and therefore sets the fixed type of `$T2` to `DayOfTheWeek`. Now, all constraints are satisfied. The final solution is shown in Figure 6.30:

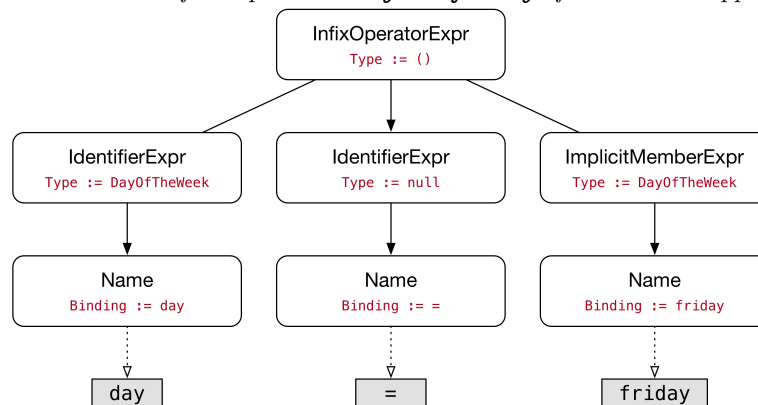
Figure 6.30: *Final Solution*

Solution
\$T0 := DayOfTheWeek \$T1 := DayOfTheWeek \$T2 := DayOfTheWeek

Solution Application

This final solution is applied to the original expression which results in the fully typed AST shown in Figure 6.31:

Figure 6.31: *AST for expression `day = .friday` after solution application*



6.6.7 Example 6: Optionals

Swift has several language constructs that are related to the handling of optionals (see subsection 3.1.9). This example shows how a forced-value expression is type checked. Listing 6.46 contains the code for this example:

Listing 6.46: Code for Example 6

```

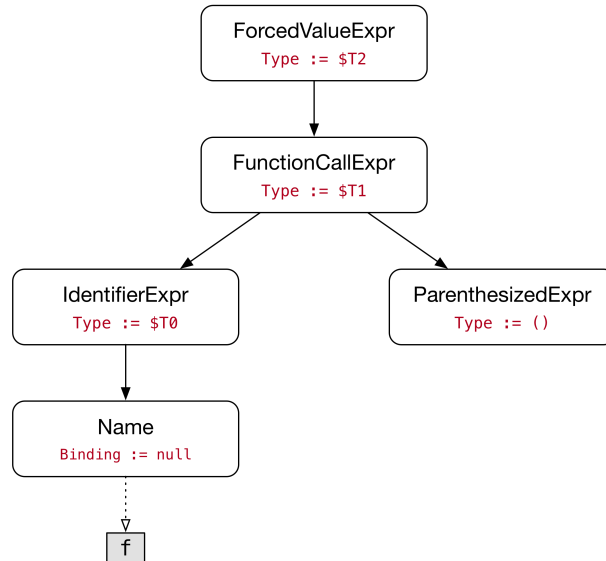
1 func f() -> Int { return 0 }
2 func f() -> Int? { return 0 }
3
4 let x = f()!
```

In this example there are two functions called `f()`. One returns an `Int` and the other returns an `Int?`. The variable `x` is initialized with a call to `f()` that is immediately force-unwrapped. Since only optionals can be unwrapped like this, the type checker chooses the `f()` overload that returns an `Int?`. The following subsections show how this is achieved.

Constraint Generation

The constraint generator walks the AST of the expression `f()!` in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.32:

Figure 6.32: AST for expression `f()!` after constraint generation



The constraint generator first creates a fresh type variable `$T0` for the identifier expression `f`. Since there are multiple functions called `f`, it cannot yet resolve the overload and therefore has to create placeholder type variables for the function call expression as well as the forced-value expression.

Additionally, the system creates the type variable `$T3` which represents the type of the variable `x`. This is not shown in the AST of the initializer expression.

The following list describes the constraints that are generated for the example above:

- **Disjunction Constraint**
 - `$T0 BindOverload f: () -> Int`
 - `$T0 BindOverload f: () -> Int?`

The disjunction constraint that is generated for the identifier expression `f` contains two nested constraints, each of which binds `$T0` to one of the two `f` overloads.

- **`() -> $T1 ApplicableFunction $T0`**
This constraint means that the fixed type of `$T0` must be a function type which has no required parameters and a return type that is equal to `$T1`.
- **`$T1 OptionalObject $T2`**
This constraint means that `$T1` (i.e., the return type of the function `f()`) must be an optional type which results in `$T2` (i.e., the type of the forced-value expression) when unwrapped.
- **`$T2 Conversion $T3`**
This constraint means that the initializer expression which has the type `$T2` must be convertible to the type `$T3` (i.e., the type of the variable `x`).

Figure 6.33 shows the root constraint system for this example:

Figure 6.33: *Root Constraint System*

Constraint System
Type Variables <code>\$T0 := null</code> <code>\$T1 := null</code> <code>\$T2 := null</code> <code>\$T3 := null</code>
Inactive Constraints <ul style="list-style-type: none"> ▪ Disjunction Constraint: <ul style="list-style-type: none"> – <code>\$T0 BindOverload () -> Int</code> – <code>\$T0 BindOverload () -> Int?</code> ▪ <code>() -> \$T1 ApplicableFunction \$T0</code> ▪ <code>\$T1 OptionalObject \$T2</code> ▪ <code>\$T2 Conversion \$T3</code>
Active Constraints <ul style="list-style-type: none"> ▪ -

Constraint Solving

The constraint solver first tries to choose the overload `f: () -> Int` and therefore sets the fixed type of `$T0` to `() -> Int`. During the simplification of the constraint `() -> $T1 ApplicableFunction $T0`, the fixed type of `$T1` is then set to `Int`. However, this means that the simplification of the constraint `$T1 OptionalObject $T2` fails because the fixed type of `$T1` (i.e., `Int`) is not an optional.

Therefore, the solver backtracks and now chooses the overload `f: () -> Int?`. During the simplification of the *ApplicableFunction* constraint, `$T1` is set to `Int?` and during

the simplification of the *OptionalObject* constraint, $\$T2$ is set to `Int`. Finally, the solver tries the potential binding $\$T3 := \text{Int}$ based on the constraint $\$T2 \text{ Conversion } \$T3$.

Now, all constraints are satisfied. The final solution is shown in Figure 6.34:

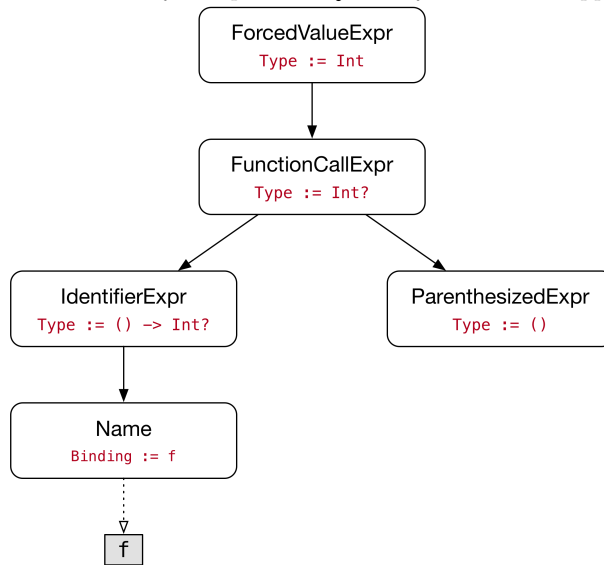
Figure 6.34: *Final Solution*

Solution
$\$T0 := () \rightarrow \text{Int?}$
$\$T1 := \text{Int?}$
$\$T2 := \text{Int}$
$\$T3 := \text{Int}$

Solution Application

This final solution is applied to the original expression which results in the fully typed AST shown in Figure 6.35:

Figure 6.35: *AST for expression $f()!$ after solution application*



6.6.8 Example 7: Initializer Call

This example shows how an initializer call is type checked. The code for this example is shown in Listing 6.47:

Listing 6.47: *Code for Example 7*

```

1 struct Circle {
2   let radius: Double
3 }
4
5 let circle = Circle(radius: 2.5)

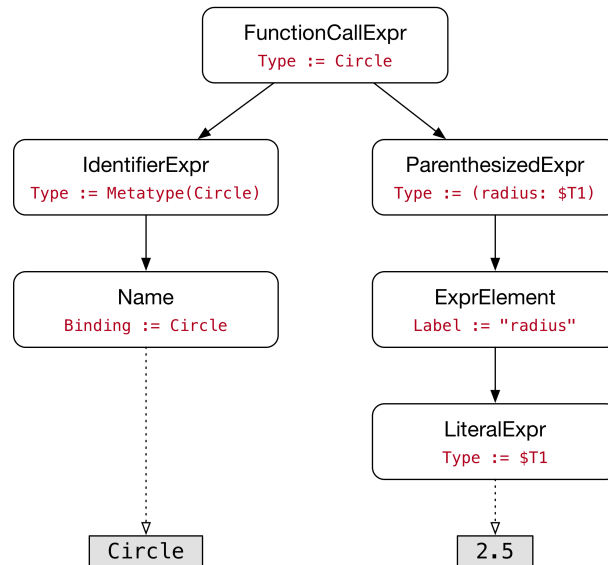
```

The expression `Circle(radius: 2.5)` calls the compiler-generated, memberwise initializer which in turn constructs a new instance of the struct type `Circle`.

Constraint Generation

The constraint generator walks the AST of the expression `Circle(radius: 2.5)` in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.36:

Figure 6.36: *AST for expression `Circle(radius: 2.5)` after constraint generation*



In this example, much of the work happens already during constraint generation. The constraint generator first creates the type variable `$T0` for the identifier expression `Circle`. Since there is only one entity (i.e., a struct type) with this name, the overload is immediately resolved during constraint generation and the fixed type of `$T0` is set to `Metatype(Circle)`.

Next, the constraint generator creates a fresh type variable `$T1` for the floating-point literal `2.5`.

Finally, the constraint generator visits the `FunctionCallExpr` node. It generates the constraint `(radius: $T1) -> $T2 ApplicableFunction Metatype(Circle)` where `$T2` is a fresh type variable that represents the type of the overall function call expression. However, since the second type of this constraint is a metatype, the constraint simplifier recognizes that this is a call to an initializer. Thus, this constraint is not added to the constraint system, but is instead simplified.

The constraint simplifier creates a fresh type variable `$T3` which represents the type of the initializer's parameter list. The function type `$T3 -> $T2` is therefore considered to be a placeholder for the type of the initializer. The simplifier then looks for initializers in the struct type `Circle`. If there were multiple initializers, a corresponding disjunction constraint would be generated. In this case there is only the compiler-generated, memberwise initializer. Since the type of this initializer is `(radius: Double) -> Circle`, the simplifier binds `$T3` to `(radius: Double)` and `$T2` to `Circle`. Finally, the simplifier adds the constraint `(radius: $T1) ArgumentTupleConversion $T3` which is immediately simplified to `$T1 ArgumentConversion Double`.

Additionally, the system creates the type variable `$T4` which represents the type of the variable `circle`. This is not shown in the AST of the initializer expression.

The following list describes the constraints that are generated for the example above:

- **\$T1 LiteralConformsTo ExpressibleByFloatLiteral**
This constraint means that the fixed type of `$T1` (i.e., the type of the literal expression `2.5`) must conform to the `ExpressibleByFloatLiteral` protocol.
- **\$T1 ArgumentConversion Double**
This constraint means that the fixed type of `$T1` must be convertible to `Double`.
- **Circle Conversion \$T4**
This constraint means that the initializer expression which has the type `Circle` must be convertible to the type `$T4` (i.e., the type of the variable `circle`).

Figure 6.37 shows the root constraint system for this example:

Figure 6.37: *Root Constraint System*

Constraint System
Type Variables <code>\$T0 := Metatype(Circle)</code> <code>\$T1 := null</code> <code>\$T2 := Circle</code> <code>\$T3 := (radius: Double)</code> <code>\$T4 := null</code>
Inactive Constraints <ul style="list-style-type: none"> ▪ <code>\$T1 LiteralConformsTo ExpressibleByFloatLiteral</code> ▪ <code>\$T1 ArgumentConversion Double</code> ▪ <code>Circle Conversion \$T4</code>
Active Constraints <ul style="list-style-type: none"> ▪ -

Constraint Solving

There are two connected components in this constraint graph (note that components that contain only type variables which already have a fixed type are ignored). One component contains the type variable `$T1` and the other contains the type variable `$T4`.

The constraint solver starts with the first component and tries the potential binding `$T1 := Double`. This satisfies both the constraint `$T1 LiteralConformsTo ExpressibleByFloatLiteral` as well as the constraint `$T1 ArgumentConversion Double`. Thus, this component is already solved.

For the second component, the constraint solver tries the potential binding `$T4 := Circle`. This satisfies the constraint `Circle Conversion $T4` which means that this component is also solved.

Finally, the solver combines the two partial solutions into the final solution shown in Figure 6.38:

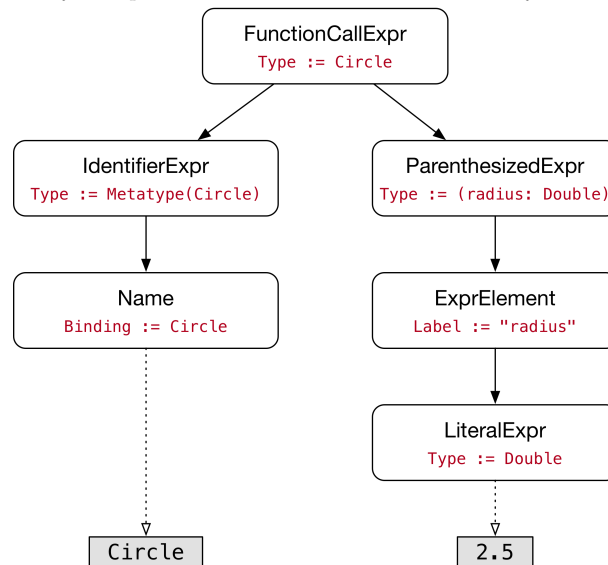
Figure 6.38: *Final Solution*

Solution
<code>\$T0 := Metatype(Circle)</code>
<code>\$T1 := Double</code>
<code>\$T2 := Circle</code>
<code>\$T3 := (radius: Double)</code>
<code>\$T4 := Circle</code>

Solution Application

This final solution is applied to the original expression which results in the fully typed AST shown in Figure 6.39:

Figure 6.39: *AST for expression `Circle(radius: 2.5)` after solution application*



6.6.9 Example 8: Generic Function

This example shows how a call to a generic function is type checked. The code for this example is shown in Listing 6.48:

Listing 6.48: Code for Example 8

```

1 func _max<T: Comparable>(_ x: T, _ y: T) -> T {
2   return y >= x ? y : x
3 }
4
5 let result = _max(99, 42)

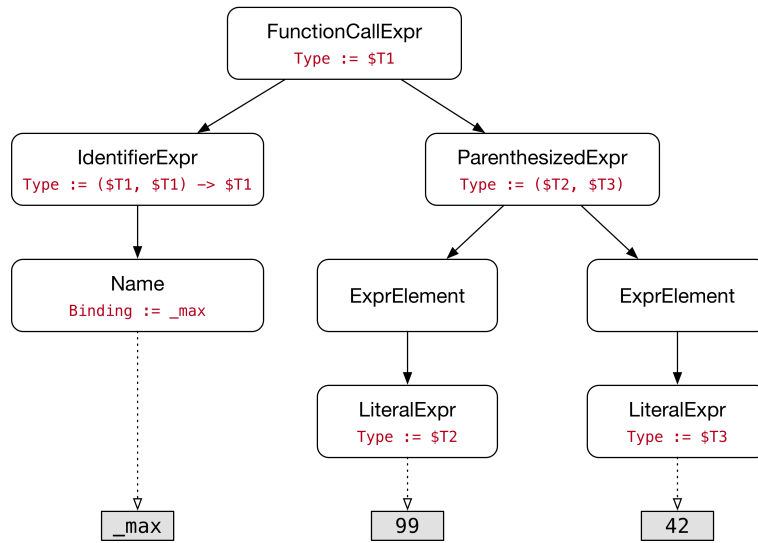
```

The generic function in the example above works just like the `max()` function from the standard library. However, in order to avoid ambiguity, it is called `_max()`.

Constraint Generation

The constraint generator walks the AST of the expression `_max(99, 42)` in postorder and assigns a type to each subexpression. The resulting AST is shown in Figure 6.40:

Figure 6.40: AST for expression `_max(99, 42)` after constraint generation



The constraint generator first creates the type variable `$T0` for the identifier expression `_max`. However, since there is only one function with this name, the overload is immediately resolved during constraint generation. In the process, the generic type parameter `T` in the type of the `_max()` function is replaced with a fresh type variable `$T1`. Thus, the fixed type of `$T0` is set to the function type `($T1, $T1) -> $T1`.

The type variables `$T2` and `$T3` are created for the two integer literals. Additionally, the system creates the type variable `$T4` which represents the type of the `result` variable. This is not shown in the AST of the initializer expression.

The following list describes the constraints that are generated for the example above:

- **\$T1 ConformsTo Comparable**
This constraint is generated due to the conformance requirement `T: Comparable` in the signature of the function `_max()`. It means that the fixed type of `$T1` must conform to the protocol `Comparable`.
- **\$T2 LiteralConformsTo ExpressibleByIntegerLiteral**
\$T3 LiteralConformsTo ExpressibleByIntegerLiteral
These two constraints mean that the fixed types of the type variables that were generated for the individual literal expressions must conform to the `ExpressibleByIntegerLiteral` protocol.
- **\$T2 ArgumentConversion \$T1**
\$T3 ArgumentConversion \$T1
These two constraints mean that the fixed types of the type variables that were generated for the individual literal expressions must be convertible to the fixed type of `$T1`.
- **\$T1 Conversion \$T4**
This constraint means that the initializer expression which has the type `$T1` must be convertible to the type `$T4` (i.e., the type of the variable `result`).

Note that the two *ArgumentConversion* constraints are the result of the simplification of the constraint `($T2, $T3) -> $T1 ApplicableFunction ($T1, $T1) -> $T1`.

Figure 6.41 shows the root constraint system for this example:

Figure 6.41: Root Constraint System

Constraint System
Type Variables <code>\$T0 := (\$T1, \$T1) -> \$T1</code> <code>\$T1 := null</code> <code>\$T2 := null</code> <code>\$T3 := null</code> <code>\$T4 := null</code>
Inactive Constraints <ul style="list-style-type: none"> ▪ <code>\$T1 ConformsTo Comparable</code> ▪ <code>\$T2 LiteralConformsTo ExpressibleByIntegerLiteral</code> ▪ <code>\$T3 LiteralConformsTo ExpressibleByIntegerLiteral</code> ▪ <code>\$T2 ArgumentConversion \$T1</code> ▪ <code>\$T3 ArgumentConversion \$T1</code> ▪ <code>\$T1 Conversion \$T4</code>
Active Constraints <ul style="list-style-type: none"> ▪ -

Constraint Solving

The constraint solving process for this example is straightforward. Based on the constraint `$T2 LiteralConformsTo ExpressibleByIntegerLiteral`, the solver tries the potential binding `$T2 := Int`. From there, the type `Int` propagates to the type variable `$T1` because of the constraint `$T2 ArgumentConversion $T1`. Since `Int` conforms to the protocol `Comparable`, the solver then continues by trying the potential binding `$T4 := Int` based on the constraint `$T1 Conversion $T4`. At this point, all constraints except for the ones that involve `$T3` have been simplified and removed. Finally, the solver tries

the potential binding `$T3 := Int` based on the constraint `$T3 LiteralConformsTo ExpressibleByIntegerLiteral`. This satisfies the remaining two constraints and the system is solved. Figure 6.42 shows the final solution of this constraint system:

Figure 6.42: *Final Solution*

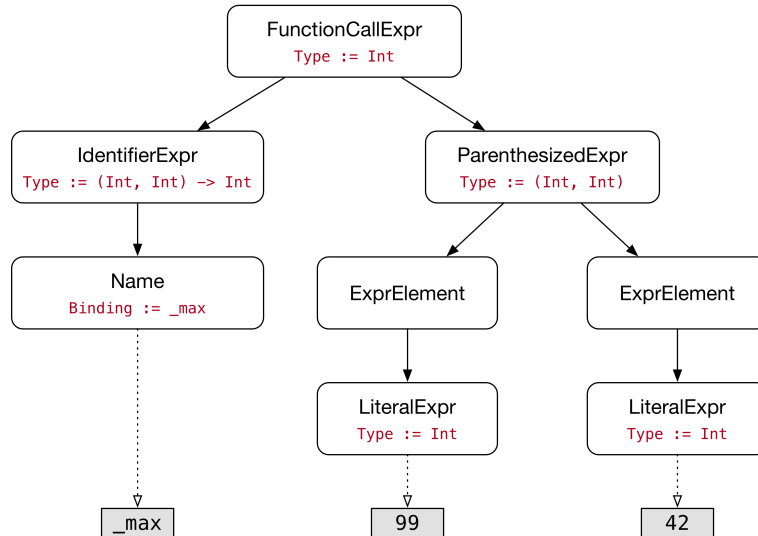
Solution
<code>\$T0 := (Int, Int) -> Int</code>
<code>\$T1 := Int</code>
<code>\$T2 := Int</code>
<code>\$T3 := Int</code>
<code>\$T4 := Int</code>

Note that the fixed type of `$T0` is simplified from `($T1, $T1) -> $T1` to `(Int, Int) -> Int` when the solution is finalized.

Solution Application

This final solution is applied to the original expression which results in the fully typed AST shown in Figure 6.43:

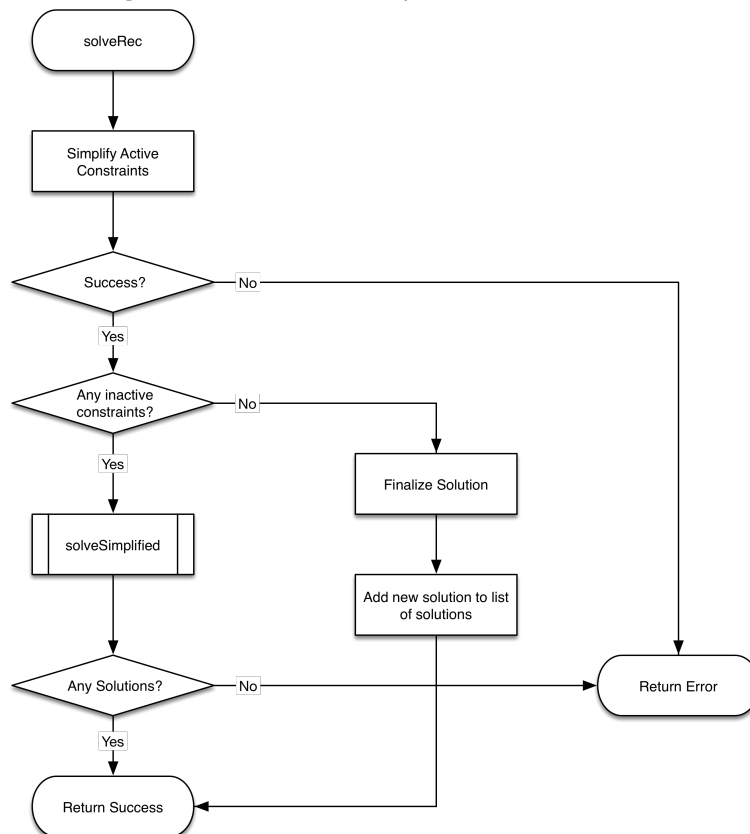
Figure 6.43: *AST for expression `_max(99, 42)` after solution application*



6.6.10 Solver Algorithm

The preceding examples showed how the constraint solver type checks different kinds of expressions. This section describes how the solver is implemented. The two mutually recursive methods `solveRec()` and `solveSimplified()` are the central components of the solver algorithm. Figure 6.44 shows a flow chart of the `solveRec()` method. For simplicity reasons, the flow chart doesn't show the extra logic that is used, if there are multiple connected components in the constraint graph:

Figure 6.44: Flow Chart for `solveRec()` method



The method `solveRec()` is called after constraint generation. Additionally, it is called every time after the solver applies a new potential binding as well as every time after the solver picks an overload from a disjunction constraint. The method returns a boolean to indicate success or failure as well as a list of solutions. It starts by simplifying active constraints. If an active constraint can be simplified, it is removed from the constraint system. Note that the simplification may lead to the generation of smaller constraints which was shown in some of the preceding examples. If an active constraint cannot be solved yet, it is added back to the list of inactive constraints. Finally, if a constraint is unsatisfiable with the current choice of fixed types, the simplifier returns an error.

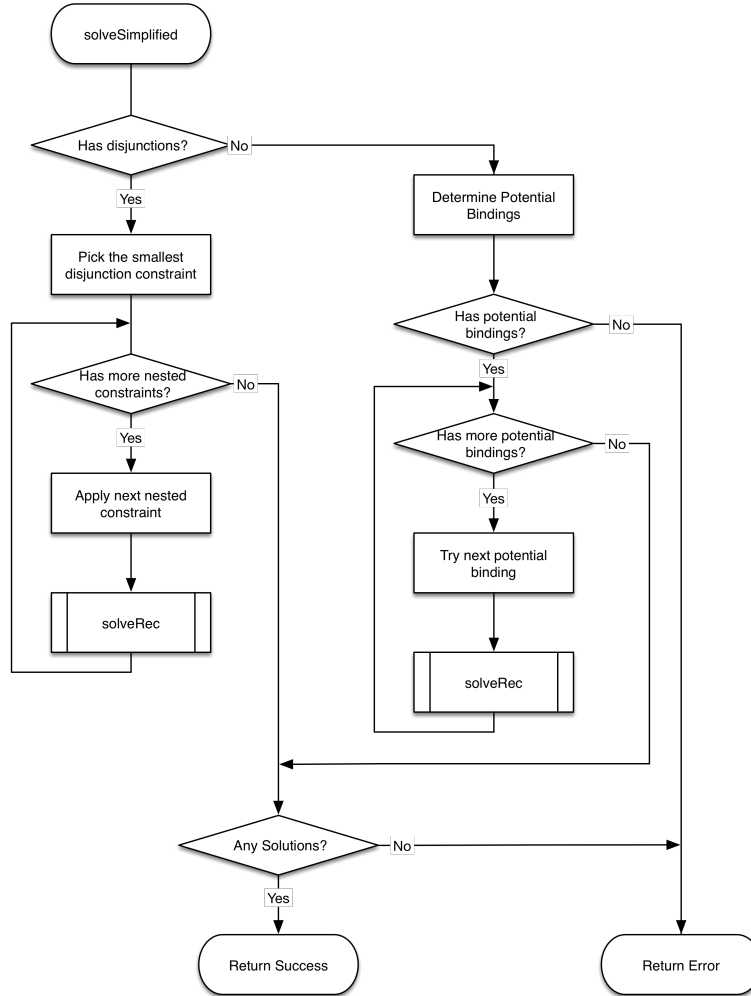
If there are any unsatisfiable constraints, `solveRec()` returns `true` to indicate that there was an error. Otherwise, it proceeds by checking whether there are any inactive constraints.

If there are no more inactive constraints, it means that the solver has found a solution. It finalizes the solution and adds it to the list of solutions that is returned from `solveRec()` through an out parameter. Finally, `solveRec()` returns `false` which indicates success.

If there are still inactive constraints, the solver calls the `solveSimplified()` method which will be explained below. If there are any solutions after `solveSimplified()` returns, `solveRec()` returns `false` (i.e., success). Otherwise, it returns `true` (i.e., error).

Figure 6.45 shows a flow chart of the `solveSimplified()` method:

Figure 6.45: Flow Chart for `solveSimplified()` method



The `solveSimplified()` first looks for disjunction constraints. If there are disjunction constraints, it picks the smallest (i.e., the disjunction constraint with the fewest nested constraints). Then, it applies each nested constraint one after the other and calls `solveRec()` each time in order to explore whether the overload choice leads to a solution.

If there are no disjunction constraints, the system determines potential bindings for the type variables in the constraint system. If there are potential bindings, it applies one

after the other and calls `solveRec()` each time in order to explore whether the potential binding leads to a solution.

If there are no potential bindings *and* no disjunction constraints, `solveSimplified()` returns `true` to indicate that there was an error.

Otherwise, if there are any solutions in the end, `solveSimplified()` returns `false` to indicate success. If there are no solutions, it returns `true` to indicate that there was an error.

When the solver is stuck, either because there is an unsatisfiable constraint or because it found a solution, it backtracks to a point where other potential bindings or nested constraints can be tried out. During backtracking the solver also needs to revert any changes that were made to the constraint system. This includes fixed types that were set on type variables as well as constraints that were added or removed. To do that, the solver uses so-called solver scopes. An example of this is shown in Listing 6.49:

Listing 6.49: *Trying Potential Bindings*

```

1  boolean tryTypeVariableBindings(ConstraintSystem cs, TypeVariableType typeVar,
2      List<PotentialBinding> bindings, List<Solution> solutions) {
3      boolean anySolved = false;
4
5      for(final PotentialBinding binding : bindings) {
6          IType type = binding.getBindingType();
7
8          // Try to solve the system with typeVar := type
9          try(final SolverScope scope = new SolverScope(cs)) {
10             cs.simplifier().addConstraint(ConstraintKind.Bind, typeVar, type);
11
12             if(!cs.solver().solveRec(solutions)) {
13                 anySolved = true;
14             }
15         }
16     }
17
18     return !anySolved;
19 }
```

The `tryTypeVariableBindings()` method is used to try out different potential bindings for a specific type variable `typeVar`. To apply a potential binding, the constraint `typeVar Bind type` is added to the system. This is immediately simplified by setting the fixed type of `typeVar` to `type`. Afterwards, it calls `solveRec()` to recursively solve the system.

Note that this happens within a `try` statement. In the beginning of that `try` statement, a new `SolverScope` is created. This class records the current state of the constraint system and reverts the system back to that state once execution leaves the scope of the `try` statement. In the original implementation in the Swift compiler, `SolverScope` is a C++ RAII class [rai17]. Tifig uses Java’s try-with-resources mechanism to achieve the same effect [try17].

6.6.11 Ranking Rules

This section looks at the ranking rules that are used to determine the “best” solution, if there are multiple solutions.

Prefer more specialized overloads

If there are two or more solutions that differ in a specific overload choice, the solution ranking algorithm favors the solution which chooses the overload that is more specialized than the other overloads. This was shown in example 2 (see subsection 6.6.3). Another example is shown in Listing 6.50:

Listing 6.50: *Prefer more specialized overloads*

```
1 func f(_ x: Int?) {}
2 func f(_ x: Int) {}
3
4 f(42) // picks f: (Int) -> ()
```

In this example, the overload `f: (Int) -> ()` is considered to be more specialized than `f: (Int?) -> ()`, because `Int` is convertible to `Int?` (see subsection 6.6.14).

Prefer overloads with fewer ignored parameters

The solution ranking algorithm also prefers overloads for which fewer parameters have been ignored in the function call. A parameter can be ignored (i.e., no argument needs to be provided) if it is either variadic or has a default value. An example of this is shown in Listing 6.51:

Listing 6.51: *Prefer overloads with fewer ignored parameters*

```
1 func f(_ x: Int, _ y: String = "") {}
2 func f(_ x: Int) {}
3
4 f(42) // picks f: (Int) -> ()
```

In this example, the overload `f: (Int) -> ()` is considered to be “better”, because none of its parameters have been ignored whereas with the overload `f: (Int, String) -> ()` the parameter `y` is ignored.

Prefer regular methods over protocol extension methods

The solution ranking algorithm prefers regular methods over methods that are inherited from a protocol extension. An example of this is shown in Listing 6.52:

Listing 6.52: *Prefer regular methods over protocol extension methods*

```

1 protocol P {}
2 extension P {
3     func f() {}
4 }
5
6 struct S: P {
7     func f(_ x: Int = 0) {}
8 }
9
10 let s = S()
11 s.f() // picks f: (Int) -> ()

```

In this example, the ranking algorithm chooses the regular method over the protocol extension method even though the protocol extension method has fewer ignored parameters. Thus, this ranking rule takes precedence over the other rules.

Prefer protocol extension methods from derived protocols

The solution ranking algorithm prefers one protocol extension method over another, if the first one belongs to a protocol that is derived from the protocol of the second protocol extension method. An example of this is shown in Listing 6.53:

Listing 6.53: *Prefer protocol extension methods from derived protocols*

```

1 protocol P2 {}
2 extension P2 {
3     func f() {}
4 }
5
6 protocol P1: P2 {}
7 extension P1 {
8     func f(_ x: Int = 0) {}
9 }
10
11 struct S: P1 {}
12
13 let s = S()
14 s.f() // picks f: (Int) -> ()

```

In this example, the ranking algorithm prefers the method `f: (Int) -> ()` from protocol P1 over the method `f: () -> ()` from protocol P2, because P1 inherits from P2. Again, this ranking rule takes precedence over the other rules.

6.6.12 Contextual Type Constraints

Some expressions in Swift have a contextual type from their enclosing statement or declaration. These contextual types can influence type checking and overload resolution, because the type-check pass creates an additional *Conversion* constraint which ensures that the type of the expression is convertible to the corresponding contextual type. The following subsections show a few examples.

Variable Declarations

If a variable declaration has an explicit type annotation *and* an initializer expression, the type in the type annotation is considered to be the contextual type of the initializer expression. An example of this is shown in Listing 6.54:

Listing 6.54: *Contextual Type in Variable Declaration*

```
1 func f() → Int { return 42 }
2 func f() → String { return "test" }
3
4 let x: Int = f() // picks f: () → Int due to contextual type constraint
```

Return Statements

The contextual type of the expression in a return statement is the return type of the enclosing function. Note that a return statement can also occur in a computed property or an observed property. In that case, the type of the corresponding property is considered to be the contextual type. An example is shown in Listing 6.55:

Listing 6.55: *Contextual Type in Return Statement*

```
1 func f() → Int { return 42 }
2 func f() → String { return "test" }
3
4 func g() → String {
5   return f() // picks f: () → String due to contextual type constraint
6 }
```

Boolean Conditions

The expression in a boolean condition has a contextual type of `Bool`. Note that boolean conditions can occur in `if`, `guard`, `while` and `repeat-while` statements. An example is shown in Listing 6.56:

Listing 6.56: *Contextual Type in Boolean Condition*

```
1 func f() → Int { return 42 }
2 func f() → Bool { return true }
3
4 // picks f: () → Bool due to contextual type constraint
5 if f() {
6   print("is here")
7 }
```

Where Clauses

The expression in a where clause has a contextual type of `Bool`. Note that where clauses can occur in `switch`, `for` and `do` statements. An example is shown in Listing 6.57:

Listing 6.57: *Contextual Type in Where Clause*

```

1 func f() → Int { return 42 }
2 func f() → Bool { return true }
3
4 switch (1, 2) {
5 case let (x, y) where f(): // picks f: () → Bool due to contextual type constraint
6     break
7 default:
8     break
9 }

```

Throw Statements

The expression in a `throw` statement needs to conform to the standard library protocol `Error`. Thus, the expression has a contextual type of `Error`. An example is shown in Listing 6.58:

Listing 6.58: *Contextual Type in Throw Statement*

```

1 enum MyError: Error {
2     case error1
3     case error2
4 }
5
6 func f() → Int { return 42 }
7 func f() → MyError { return .error1 }
8
9 func g() throws {
10     throw f() // picks f: () → MyError due to contextual type constraint
11 }

```

6.6.13 Pattern Matching

Section 3.1.12 showed how pattern matching works in Swift. The type checker needs to verify whether a pattern is valid for the type of the given value. For example, a tuple pattern is only valid for a tuple value that has the same number of elements. After the type checker has determined that a pattern is valid for a type, it also needs to recursively verify the nested patterns.

For identifier patterns, the type checker sets the type of the identifier name’s binding. For expression patterns, it type checks the expression `pattern ~= type`. This way, the pattern matching mechanism is extensible since the user can define a custom kind of pattern by providing a corresponding overload of the `~=` operator. An example of this is shown in Listing 6.59:

Listing 6.59: *Custom ~= operator overload*

```

1 func ~=<T>(pattern: (T) -> Bool, value: T) -> Bool {
2     return pattern(value)
3 }
4
5 func greaterThan<T : Comparable>(_ a: T) -> (T) -> Bool {
6     return { $0 > a }
7 }
8
9 let x = 11
10 switch x {
11 case greaterThan(10):
12     print("x > 10")
13 default:
14     print("x <= 10")
15 }

```

This example defines an overload of the `~=` operator which takes a pattern that is a predicate function and applies the predicate to the given value. Additionally, it defines the higher-order function `greaterThan()`. This function takes a parameter `a` and returns a predicate that returns `true`, if its parameter is bigger than `a`. This way we can use the expression `greaterThan(10)` as a pattern [Beg15].

6.6.14 Conversions

The various type-checking examples that were shown in this chapter contained a lot of conversion constraints (e.g., *Conversion*, *ArgumentConversion*, *OperatorArgumentConversion*). These constraints consist of two types and convey to the constraint system that the first type must be convertible to the second type. The conversions are implicit which means that there is no explicit casting / coercion syntax necessary. The following examples show various kinds of implicit conversions that are valid in Swift:

- A type `T` is convertible to itself. An example of this is shown in Listing 6.60:

Listing 6.60: *Conversion from Int to Int*

```

1 let x = 2           // x is of type Int
2 let y: Int = x

```

- A type `T` is convertible to the existential type `Any`. An example of this is shown in Listing 6.61:

Listing 6.61: *Conversion from Int to Any*

```

1 let x = 2           // x is of type Int
2 let y: Any = x

```

- A class type `T` is convertible to the existential type `AnyObject`. An example of this is shown in Listing 6.62:

Listing 6.62: *Conversion from class type C to AnyObject*

```

1 class C {}
2 let x = C()         // x is of type C
3 let y: AnyObject = x

```

- A type `T` that conforms to the `Hashable` protocol is convertible to the existential type `AnyHashable`. An example of this is shown in Listing 6.63:

Listing 6.63: *Conversion from `Int` to `AnyHashable`*

```
1 let x = 2           // x is of type Int
2 let y: AnyHashable = x
```

- A nominal type `T` is convertible to a protocol type that it conforms to. An example of this is shown in Listing 6.64:

Listing 6.64: *Conversion from nominal type to protocol type*

```
1 protocol P {}
2 struct S: P {}
3 let x = S()       // x is of type S
4 let y: P = x
```

- A nominal type `T` is convertible to a protocol composition type if it conforms to all the protocols in the protocol composition type. An example of this is shown in Listing 6.65:

Listing 6.65: *Conversion from nominal type to protocol composition type*

```
1 protocol P1 {}
2 protocol P2 {}
3 struct S: P1, P2 {}
4 let x = S()       // x is of type S
5 let y: P1 & P2 = x
```

- A class type `T` is convertible to a class type that it directly or indirectly inherits from. An example of this is shown in Listing 6.66:

Listing 6.66: *Conversion from class type to base class type*

```
1 class B {}
2 class C: B {}
3 let x = C()       // x is of type C
4 let y: B = x
```

- A function type `In1 -> Out1` is convertible to a function type `In2 -> Out2` if `In2` is convertible to `In1` (contravariance) and `Out1` is convertible to `Out2` (covariance). An example of this is shown in Listing 6.67:

Listing 6.67: *Conversion from one function type to another*

```
1 let x = { (_, Any) in 2 } // x is of type (Any) -> Int
2 let y: (Int) -> Any = x
```

- A type `T1` is convertible to a function type `@autoclosure () -> T2` if `T1` is convertible to `T2`. Note that the `@autoclosure` attribute is only allowed in parameter types. An example of this is shown in Listing 6.68:

Listing 6.68: *Conversion from `Int` to `@autoclosure () -> Any`*

```
1 func f(_ x: @autoclosure () -> Any) {}
2 let x = 42           // x is of type Int
3 f(x)
```

- A type `T1` is convertible to `T2`? if `T1` is convertible to `T2`. An example of this is shown in Listing 6.69:

Listing 6.69: *Conversion from Int to Any?*

```
1 let x = 2           // x is of type Int
2 let y: Any? = x
```

- A type `T1?` is convertible to `T2?` if `T1` is convertible to `T2`. An example of this is shown in Listing 6.70:

Listing 6.70: *Conversion from Int? to Any?*

```
1 let x = Optional(2) // x is of type Int?
2 let y: Any? = x
```

- A type `Array<T1>` is convertible to `Array<T2>` if `T1` is convertible to `T2`. An example of this is shown in Listing 6.71:

Listing 6.71: *Conversion from Array<Int> to Array<Any>*

```
1 let x = [1, 2, 3] // x is of type Array<Int>
2 let y: Array<Any> = x
```

Note that this kind of *covariance* also works with `Set` and `Dictionary`. But these types are special cases and other generic types in Swift are *invariant*.

6.7 Testing

Like the parser, the indexer is tested with a comprehensive set of automated tests and the Swift code that is supposed to be tested is provided in the form of comments above the individual JUnit test methods [jun17].

The standard library is parsed and indexed before the first test case is executed. Afterwards, its public members are available in every test case. This is important because core types (e.g., `Int`, `Bool`) and operators (e.g., `+`, `&&`) are declared in the standard library.

There are three different kinds of indexer test cases: single-file test cases, multi-file test cases and multi-module test cases.

6.7.1 Single-File Test Cases

For single-file test cases there is only one comment above each test method which contains the contents of a standalone Swift file. An example of such a test case is shown in Listing 6.72:

Listing 6.72: *Example of a single-file indexer test case*

```

1 public class FunctionBindingTests extends SingleFileIndexerTestCase {
2     // func f(x: Int) {}
3     // func f(y: Int) {}
4     // f(x: 0)
5     @Test
6     public void testOverloadResolution() {
7         final IBinding fBinding = getLastOccurrence("f").getBinding();
8         assertBindingProperties(AccessLevel.Internal, 0, fBinding);
9     }
10
11     // other test cases
12 }
```

All single-file indexer test cases inherit from the superclass `SingleFileIndexerTestCase` which implements a few helper methods. In the example above, the test code defines two free functions called `f()` that only differ in their parameter names. Additionally, there is a function call `f(x: 0)`. Before each test method is executed, the corresponding test code is parsed and indexed. Note that we don't care about the AST in this test, because the parser is already tested with corresponding parser tests as described in section 5.4. Instead, the goal of this test is to ensure that the name `f` is resolved to the correct function binding based on the argument label that is provided in the function call. With the call to the helper method `getLastOccurrence()`, we first obtain a reference to the last `Name` node with the name `f`. Note that this corresponds to the `f` in the function call `f(x: 0)`. Then, we get the binding that this name was resolved to. Finally, we test a few properties of the binding with a call to the helper method `assertBindingProperties()`. We assert that the access level of the binding is `internal` and that the definition name of the binding is located at marker position 0. A pair of `£` signs in the test code comment indicates a marker. The test code can contain any number of markers each of which has an index starting at 0. Note that these markers are removed from the test code before it is parsed.

Listing 6.73 shows a more complex test case which introduces two new helper methods:

Listing 6.73: *Example of a single-file indexer test case*

```

1 public class EnumTypeBindingTests extends SingleFileIndexerTestCase {
2     // enum E<T> {
3     //     case fonef(T)
4     // }
5     // let fe1f = E.fonef(2)
6     // let fe2f = E.fonef("test")
7     @Test
8     public void testGenericEnumType() {
9         final IBinding oneBinding1 = getNameAtMarkerIndex(2).getBinding();
10        assertBindingProperties(null, 0, oneBinding1);
11
12        final IBinding e1Binding = getNameAtMarkerIndex(1).getBinding();
13        assertBindingProperties(AccessLevel.Internal, 1, e1Binding);
14        assertEquals("E<Int>", e1Binding.getType());
15
16        final IBinding oneBinding2 = getNameAtMarkerIndex(4).getBinding();
17        assertBindingProperties(null, 0, oneBinding2);
18
19        final IBinding e2Binding = getNameAtMarkerIndex(3).getBinding();
20        assertBindingProperties(AccessLevel.Internal, 3, e2Binding);
21        assertEquals("E<String>", e2Binding.getType());
22    }
23
24    // other test cases
25 }

```

Firstly, there is the method `getNameAtMarkerIndex()` which allows us to obtain any `Name` node that is marked. Secondly, the `assertEquals()` method provides a convenient way to compare an index type (i.e., a type that implements the `IType` interface) to an expected type supplied in the form of a `String`.

Note that the access level of enum cases is set to `null`, because they cannot have an explicit access level modifier and they are accessible anywhere the enclosing enum type is accessible.

6.7.2 Multi-File Test Cases

For multi-file test cases there are two comments above every test method each of which contains the contents of an individual Swift file. An example of such a test case is shown in Listing 6.74:

Listing 6.74: *Example of a multi-file indexer test case*

```

1 public class FunctionBindingTests extends MultiFileIndexerTestCase {
2     // fileprivate func f() {}
3
4     // func fff(x: Int = 0) {}
5     // f()
6     @Test
7     public void testAccessLevelFileprivate() {
8         final IBinding fBinding = getLastOccurrence("f", 1).getBinding();
9         assertBindingProperties(AccessLevel.Internal, 0, 1, fBinding);
10    }
11
12    // other test cases
13 }

```

All multi-file indexer test cases inherit from the superclass `MultiFileIndexerTestCase` which implements a few helper methods. These are the same helper methods that were shown in subsection 6.7.1 but some of them take an additional argument which indicates the file index. The upper comment has file index 0 and the comment below has file index 1.

The example above tests function overload resolution across two files. Normally, the function in file 0 would be preferred over the function in file 1 because it is a better match for the function call `f()`. However, in this case the function in file 0 is not accessible from file 1 because it has access level `fileprivate`. Thus, the function call `f()` resolves to the function in file 1.

6.7.3 Multi-Module Test Cases

For multi-module test cases there are two comments above every test method each of which contains the contents of a Swift file that belongs to a separate module. An example of such a test case is shown in Listing 6.75:

Listing 6.75: *Example of a multi-module indexer test case*

```

1 public class StructTypeBindingTests extends MultiModuleIndexerTestCase {
2     // public struct S {
3     //     public init() {}
4     //     public func fff() {}
5     // }
6
7     // import Module0
8     // let fsf = S()
9     // s.f()
10    @Test
11    public void testMethodCallOnStructType() {
12        final IBinding fBinding = getLastOccurrence("f", 1).getBinding();
13        assertBindingProperties(AccessLevel.Public, 0, 0, fBinding);
14        final IBinding sBinding = getNameAtMarkerIndex(0, 1).getBinding();
15        assertEquals("S", sBinding.getType());
16    }
17
18    // other test cases
19 }
```

All multi-module indexer test cases inherit from the superclass `MultiModuleIndexerTestCase` which implements a few helper methods. These are the same helper methods that were shown in subsection 6.7.1 but some of them take an additional argument which indicates the module index. The upper comment has module index 0 and the comment below has module index 1.

The example above tests the ability to use a struct type that is declared in a different module. To do that, we need to use the access level modifier `public` for the struct type as well as for the members that we want to use. Additionally, we need to import the module that contains the corresponding type. In a multi-module test case, the two modules are called `Module0` and `Module1`, respectively.

6.8 Implementation Status

While the indexer already works quite well for simple programs, there are still a few important pieces that are currently incomplete or missing:

- **Improve Generics Support**

Tifig supports indexing of simple generic types and functions. However, especially protocols with associated types are not yet fully supported. Unfortunately, the standard library contains a lot of code that makes extensive use of this feature. Thus, a lot of the code that often occurs in regular Swift programs cannot be indexed yet.

- **For Loops**

Currently, `for` loops cannot be correctly indexed yet. This is because the indexer needs to extract an associated type from the given sequence in order to determine the element type of the sequence. Unfortunately, this is not yet supported by the indexer.

- **Partial Imports**

Instead of importing an entire module, it is also possible to only import a specific declaration of the module. This is not yet supported by Tifig's indexer.

- **Lvalue vs. Rvalue**

The indexer should distinguish between *lvalues* and *rvalues*. In Swift, an lvalue is an expression that can be assigned to or passed to an `inout` parameter. Every other expression is considered to be a rvalue [Swi17b].

- **Add support for Pointers**

Swift has support for pointers, but in Tifig's indexer this is not yet implemented.

- **Error Handling**

The errors reported by the indexer are still too imprecise. Additionally, many kinds of semantic errors are not reported at all. While they are reported by the compiler, it would be nicer if Tifig could display these errors directly in the editor.

- **Persistent index**

In the future, it might worth considering to persist the index. This way there would be no need to reindex all projects every time a workspace is opened.

7 User Interface

This chapter describes the various UI elements that were developed for Tifig.

7.1 Wizards

The Tifig IDE contains wizards to create new Swift projects and new files.

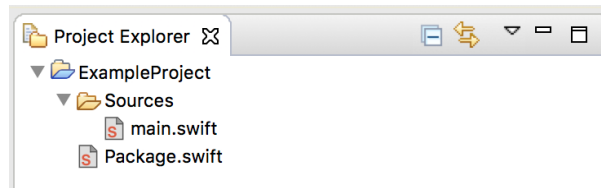
7.1.1 Project Wizard

The project wizard lets you create a new Swift project. At the moment, it is still very basic. One can only choose the name of the project and its location in the file system. In the future, it may be extended with more configuration options.

The project wizard sets up the initial file structure of the project and opens `main.swift` in the Swift editor. Additionally, it configures the project with the Swift project nature and switches the workbench to the Swift perspective. This is explained in sections 7.2 and 7.3.

The initial file structure is dictated by the Swift Package Manager [App17c] which is used to build projects in Tifig. Figure 7.1 shows a newly created project with its initial files and folders:

Figure 7.1: *Initial file structure of a new Swift project*

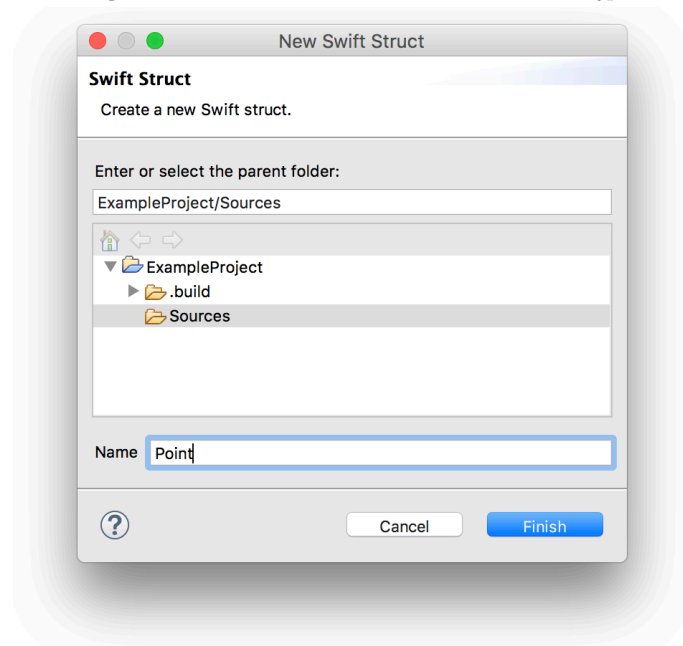


`Package.swift` is the so-called *manifest file*. It defines the package's name and its contents. By convention, source files are located in the `Sources` directory. The `main.swift` file is special, because it is the only file in the module which can contain top-level statements (all other Swift files can only contain declarations). It is the entry-point for Swift packages with an executable target.

7.1.2 File Wizards

In addition to the project wizard, Tifig also has five file wizards. Four of them create a new Swift file with a custom type (a class, a struct, an enum or a protocol). The fifth file wizard creates an empty Swift file. Like with the project wizard, the file wizards only have configuration options to specify the name and the location of the file. Figure 7.2 shows the wizard to create a new struct type:

Figure 7.2: Wizard to create a new struct type

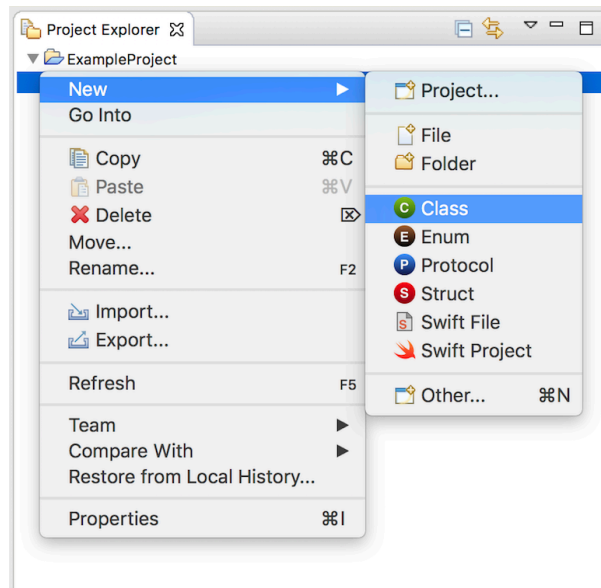


When the user clicks *Finish*, a new file called `Point.swift` is created in the project's `Sources` folder. This file contains an empty struct type called `Point` as shown in Listing 7.1:

Listing 7.1: Struct type *Point*

```
1 struct Point {
2
3 }
```

Plug-in extensions are used to make the wizards available in the usual locations within the workbench (e.g., in the `File -> New` menu or in the context menu). Figure 7.3 shows the Swift wizards in the context menu:

Figure 7.3: *Wizards in context menu*

Note that these wizards only appear in the context menu, if the current project is configured with the Swift project nature or if the active perspective happens to be the Swift perspective.

7.2 Project Nature

Project natures allow a plug-in to tag a project as a specific kind of project [ecl17e]. The Tifig IDE uses the Swift project nature to add Swift-specific behaviour to projects. When a new project is created with the Swift project wizard, it is automatically configured with the Swift project nature.

The nature adds the Swift builder to the project's build spec. This means that whenever the user triggers a build, Tifig will use the Swift builder to build the project.

7.3 Swift Perspective

When a new Swift project is created, the workbench is automatically set to use the Swift perspective. A perspective can configure the layout of the current workbench page. This means that it can set the views that are shown by default as well as configure the action sets that are displayed in the toolbar. Additionally, it can add shortcuts for wizards and views that are often used in this perspective [ecl17d].

By default, the Swift perspective displays the outline view, the problems view and the console view. It also adds shortcuts for the Swift-specific wizards.

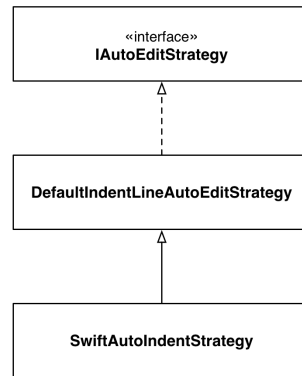
7.4 Editor

When the user clicks on a Swift source file (file with `.swift` extension) in the *Project Explorer*, Tifig opens the file in the Swift editor. The class `SwiftEditor` is a subclass of `TextEditor` which is provided by the Eclipse platform. `SwiftEditor` itself is not very interesting, but it sets up a few other components that implement features which facilitate the editing of Swift source code. These components are described in the following sections.

7.4.1 Auto Indenting

During editing, the Swift editor assists the user by automatically indenting the cursor to the correct position based on the code that is being written. Figure 7.4 shows the classes that are involved in this process:

Figure 7.4: *Classes involved in Auto Indenting*



The class `DefaultIndentLineAutoEditStrategy` implements the most basic auto edit behaviour for source code editors and is provided by the Eclipse platform. Every time the user enters a line break, it copies the level of indentation that was used on the previous line. That is all it does.

The class `SwiftAutoIndentStrategy` extends this behaviour in two ways. Firstly, if a line break is entered after an opening curly brace (`{`), it increases the level of indentation, because the user usually wants to indent the statements in a code-block or the members of a type declaration (e.g., a class, a struct, an enum). Secondly, once the user types a closing curly brace (`}`), it automatically reduces the level of indentation to match the level of the corresponding opening curly brace. Note that this implementation has been mostly copied from the Java Editor Example project [jav17].

Whenever the user edits the code, the method `customizeDocumentCommand()` is called on the `SwiftAutoIndentStrategy` passing it a reference to the current `IDocument` and an instance of the class `DocumentCommand`. The properties `offset`, `length` and `text` of the document command describe the change that is about to happen. The auto indent strategy can then look at the current document and at the document command

and decide to change some of the command’s properties in order to customize the code change.

It goes without saying, that the implementation of the `customizeDocumentCommand()` method must be very fast. Otherwise the user could experience a lot of lagging in the editing process.

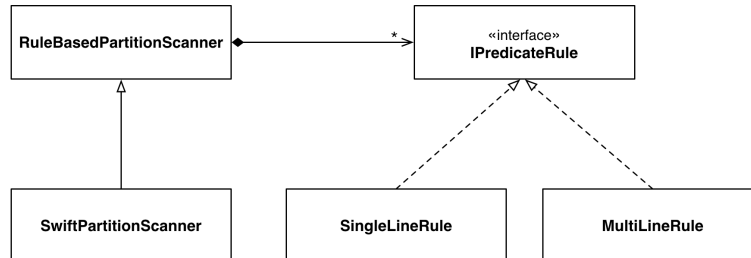
7.4.2 Syntax Highlighting

The syntax highlighting process happens in two phases. First the code is divided into several partitions (*Partitioning Phase*). Then, each partition is split up into tokens, each of which can specify a set of text attributes such as the text color and the font weight (*Presentation Reconciliation Phase*). A lot of the code described in this section has been directly adopted from a series of articles called “Create a commercial-quality Eclipse IDE” [Dev06].

Partitioning Phase

When a Swift source file is opened in Tifig, the `SwiftPartitionScanner` divides the code into several partitions based on a set of `IPredicateRules`. Figure 7.5 shows the classes that are involved in this process:

Figure 7.5: *Classes involved in Partitioning Phase*



In the Tifig IDE, there are four kinds of partitions: single-line comment, multi-line comment, string literal and default. The class `SingleLineRule` can be used to recognize partitions that cannot span across multiple lines (e.g., single-line comments and string literals). The class `MultiLineRule` is used to recognize multi-line comment partitions. The partition type “default” doesn’t require a rule. Instead, everything that is not caught by any of the rules is part of a default partition.

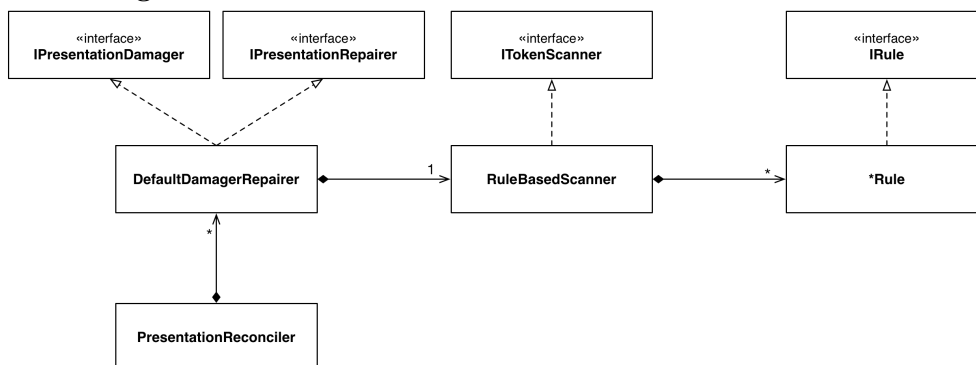
The main reason for doing this extra step is to be able to differentiate between code sections during syntax highlighting. For example, we probably don’t want to highlight Swift keywords in a multi-line comment. However, we may want to emphasize Swift documentation markup [App17a] within comments. Thus, it makes sense to partition the code, so that different syntax highlighting rules can be applied for each kind of partition.

Another reason is that we don't want to recompute the syntax highlighting for the whole file each time a character is added or removed. By dividing the code into partitions, the system can efficiently update the syntax highlighting only for the affected regions.

Presentation Reconciliation Phase

The presentation reconciliation phase is responsible for updating the syntax highlighting in the Swift editor every time the code changes. Figure 7.6 shows the classes that are involved in this process:

Figure 7.6: *Classes involved in Presentation Reconciliation Phase*



The `PresentationReconciler` has a `DefaultDamagerRepairer` for each type of partition. When code changes in a certain partition, the corresponding `DefaultDamagerRepairer` “computes the damage” which is Eclipse parlance for determining the code regions that are affected by the change. It then “repairs” these regions by updating the syntax highlighting. In order to do that, each `DefaultDamagerRepairer` has a `RuleBasedScanner`. The scanner maintains a set of rules that describe how a certain type of partition should be divided into tokens. Each token can have text attributes that specify its text color, font weight, etc. [ecl17f].

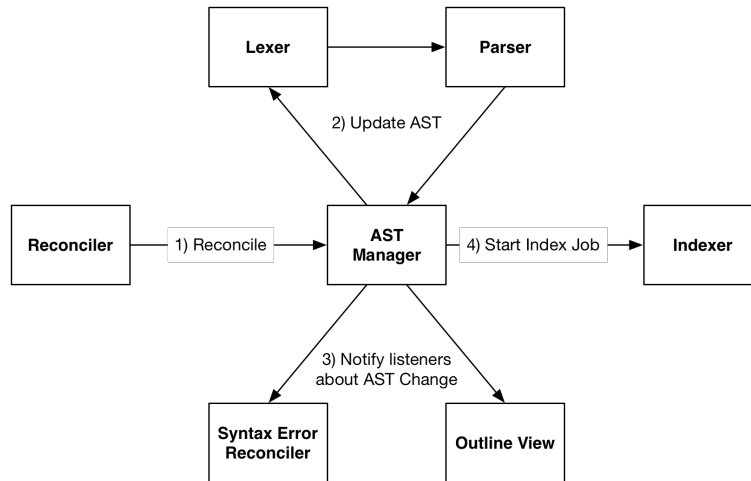
For example, the `RuleBasedScanner` for the default partition type has a rule to recognize Swift keywords. Every time a keyword is recognized, the scanner emits a token with a dark blue, bold font.

7.4.3 Reconciler

As was shown in subsection 7.4.2, the `PresentationReconciler` takes care of updating the syntax highlighting after every code change. In contrast, the `Reconciler` is responsible for updating the AST and the index, which represent the internal model of the source code. Since parsing Swift source code may take significantly longer than repairing damaged code regions, it is not feasible to run the `Reconciler` after every code change. Thus, a slightly different approach is necessary. The `Reconciler` has a 500ms timer that is restarted after every code change. When the timer reaches 0 it means that the programmer has paused for half a second, which is a good opportunity to start the reconciliation process in a background thread.

To update the internal model, the **Reconciler** invokes the **AST Manager** which maintains an AST for each source file. The **AST Manager** uses the **Lexer** and the **Parser** to create an updated AST for the changed source file. After that is done, it notifies the **Syntax Error Reconciler** and the **Outline View** about the change which will in turn update themselves. Finally, the **AST Manager** invokes the **Indexer** in order to update the index as well. Figure 7.7 shows the components that are involved in this reconciliation process:

Figure 7.7: *Components involved in Reconciliation Process*



Syntax Error Reconciler

The **Syntax Error Reconciler** is the component that is responsible for updating the syntax error markers in the Swift editor every time the AST changes. First, it deletes all existing markers that indicate a Swift syntax error. Then it visits the AST and adds a new marker for every **ProblemNode**. This is shown in Listing 7.2:

Listing 7.2: *Excerpt from Syntax Error Reconciler*

```

1 private void updateMarkers(IFile file, SourceFile ast) {
2     try {
3         file.deleteMarkers(SWIFT_PARSER_PROBLEM_MARKER, true, IResource.DEPTH_INFINITE);
4     } catch (final CoreException e) {
5         SwiftUIPlugin.logError("Failed to delete syntax error markers.", e);
6     }
7
8     ast.accept(new ASTVisitor() {
9         @Override
10        public int visit(ProblemNode problemNode) {
11            try {
12                final IMarker marker = file.createMarker(SWIFT_PARSER_PROBLEM_MARKER);
13                marker.setAttribute(IMarker.MESSAGE, problemNode.getMessage());
14                marker.setAttribute(IMarker.CHAR_START, problemNode.getOffset());
15                marker.setAttribute(IMarker.CHAR_END, problemNode.getOffset() + problemNode.getLength());
16                marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
17            } catch (final CoreException e) {
18                SwiftUIPlugin.logError("Failed to add syntax error marker.", e);
19            }
20            return PROCESS_CONTINUE;
21        }
22    });
23 }

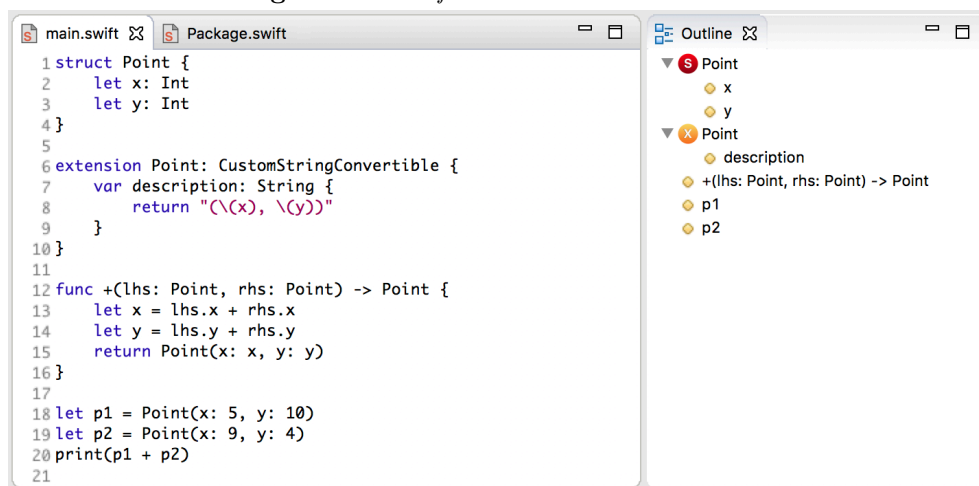
```

Note that the `updateMarkers()` method needs to be called on the main thread since background threads are not allowed to update the UI [thr17].

Outline View

The outline view is a simple tree view that shows an outline of the source file that is currently being edited. It gives the user a convenient overview of the functions, properties and types declared in the file. By clicking on a name in the outline view, the user can jump directly to the declaration of that program entity. Figure 7.8 shows a screenshot of the Swift editor with the outline view on its right hand side:

Figure 7.8: *Swift Editor with Outline View*



The outline view looks for `IDecl` and `IDeclContainer` nodes in the AST. `IDecl` nodes are the leafs in the outline tree view (e.g., properties, functions, methods). `IDeclContainer` nodes are custom types (e.g., classes, structs, etc.) and type extensions. Note that `SourceFile`, the node type of the AST's root node, is also an `IDeclContainer`. Thus, not every `IDeclContainer` is also an `IDecl`.

Hyperlinking

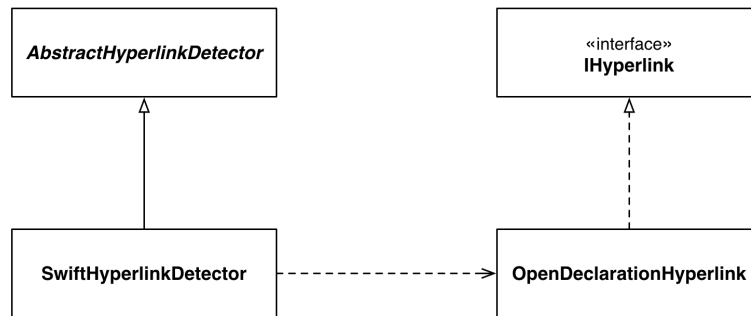
Hyperlinking is a convenient navigation feature that is supported by text editors in the Eclipse platform. It works by registering a hyperlink detector for a specific editor. When the user holds down the Command key (on macOS) or the Control key (on Linux) and hovers the mouse pointer over some text in the editor, the hyperlink detector's `detectHyperlinks()` method is called. This method may return zero or more hyperlinks which are then presented to the user in the UI. When the user clicks on a hyperlink, its `open()` method is called, which performs some action (e.g., jumping to a specific place in the editor).

Tifig registers a hyperlink detector for the Swift editor, which allows users to quickly jump to the declaration of a specific identifier. To find out whether the mouse is hovering

over an identifier, it uses the index. For each file the index maintains a list of all **Name** nodes. The hyperlink detector determines whether the cursor lies within the source region of one of these **Name** nodes and checks whether the corresponding node is backed by a valid binding. If this is case, it returns an instance of **OpenDeclarationHyperlink**.

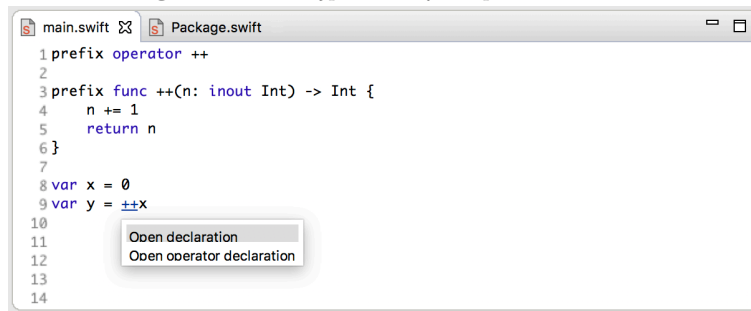
When the user clicks on such a hyperlink, it jumps to the identifier's declaration. Note that the declaration may be located in a different file, in which case Tifig opens the editor for that file. Figure 7.9 shows the classes that are involved in this process:

Figure 7.9: *Classes involved in Hyperlinking*



For operator names, the hyperlink detector returns two hyperlinks: one that points to the declaration of the operator and one that points to the declaration of the operator function. In this case, Eclipse automatically shows a popup which lets the user select the corresponding hyperlink. This is shown in Figure 7.10:

Figure 7.10: *Hyperlinks for operator names*



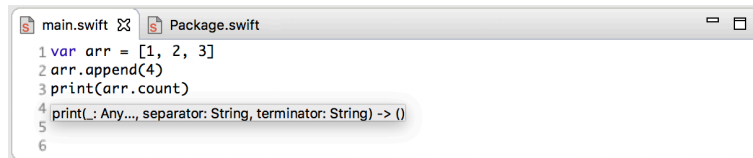
If the user clicks on the first hyperlink (*Open declaration*), Tifig jumps to the declaration of the corresponding operator function on line 3. If the user clicks on the second hyperlink (*Open operator declaration*), Tifig jumps to the declaration of the `++` operator on line 1.

7.5 Type Information Hover

Type inference can be very convenient because it avoids cluttering up the code with redundant type annotations. However, sometimes the user might not be entirely sure which type is inferred by the compiler. To help with this problem, the Swift editor in Tifig has another feature which allows users to quickly find out the inferred type of an entity simply by hovering over the corresponding name in the source code.

It can also be useful to discover the capabilities of an API without having to consult the documentation. For example, Figure 7.11 shows that the `print()` function which is provided by the standard library has additional optional parameters that may be useful:

Figure 7.11: *Type Information Hover*

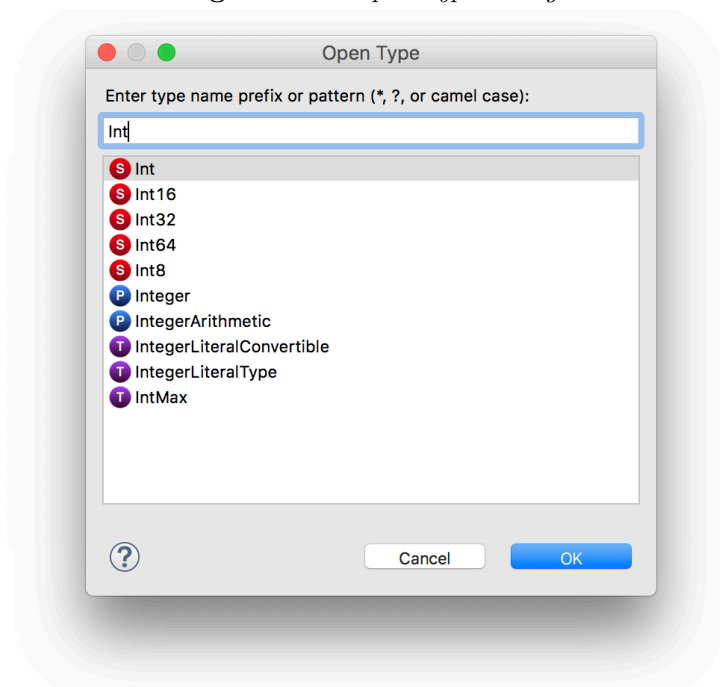


Note that this type information is obtained from the binding of the corresponding `Name` node.

7.6 Open Type Dialog

Tifig also provides an *Open Type Dialog* which allows users to quickly jump to the declaration of a specific top-level type. The feature uses the class `ElementListSelectionDialog` which is provided by the Eclipse platform. The necessary data about the individual types is obtained from the index and it includes types from the standard library. The *Open Type Dialog* can be triggered with the keyboard shortcut Command-Shift-T on macOS or Control-Shift-T on Linux. This opens a small dialog that contains a text field and an alphabetically ordered list of all top-level types. The user can type into the text field to filter the list. The user can then select a type using either the arrow keys or the mouse. When the user presses Enter or clicks on the OK button, the dialog is closed and Tifig jumps to the corresponding type declaration. Figure 7.12 shows a screenshot of the *Open Type Dialog*:

Figure 7.12: *Open Type Dialog*



7.7 Builder

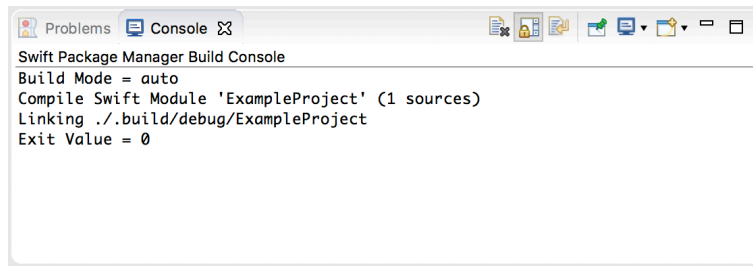
The `SwiftPackageManagerBuilder` is responsible for building Swift projects in the Tifig IDE. Tifig knows to use this particular builder, because it is added to the project’s build spec, when the `SwiftProjectNature` is configured (see section 7.2).

As its name implies, the builder uses the Swift Package Manager to build projects. Provided that the project is structured according to the conventions of the Swift Package Manager (see subsection 7.1.1), one can simply execute the `swift build` command in order to build the project. This is exactly what the `SwiftPackageManagerBuilder` does.

By default, the builder is set to “Build Automatically” which means that the project is compiled every time the user saves a file. This can be turned off in which case the user has to trigger builds manually.

In order to communicate the build result to the user, Tifig prints the output of the Swift Package Manager to a message console in the console view. An example of this is shown in Figure 7.13:

Figure 7.13: *Swift Package Manager Build Console*

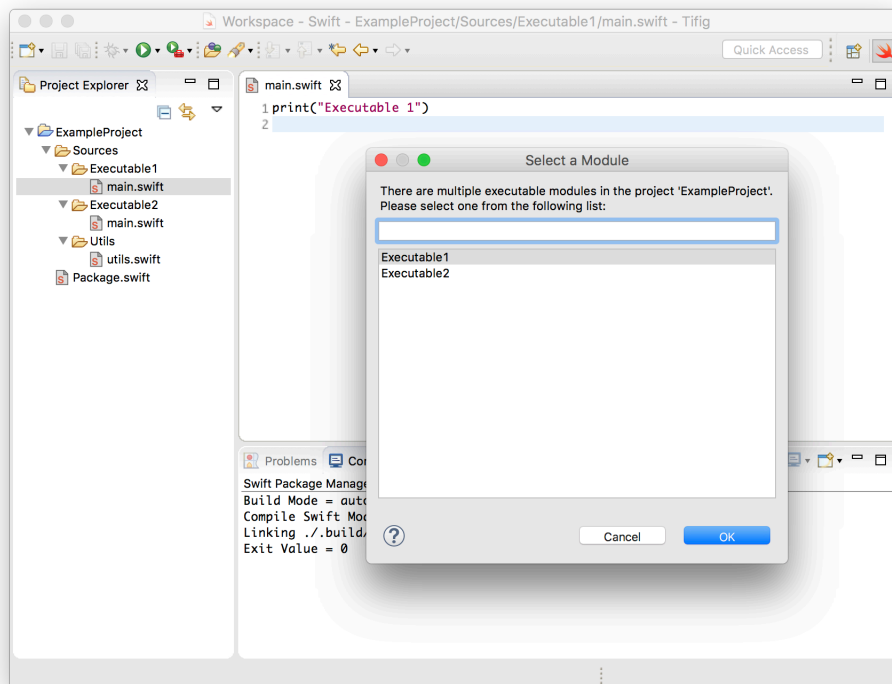


7.8 Launcher

When a user runs a project in Tifig, the following steps are performed under the hood:

1. Tifig looks for an existing launch configuration for the current project. The launch configuration must be of type `SwiftApplicationLaunchConfigurationType` and its `PROJECT_NAME` attribute must be equal to the name of the project.
 - If it finds a valid launch configuration, it proceeds to step 2.
 - If no valid launch configuration is found, Tifig looks for executable modules in the current project. By default, a new Swift project in Tifig has a single executable module. The user can create multiple modules by adding subfolders to the `Sources` folder. Each subfolder that contains a `main.swift` file is considered to be an executable module and each subfolder without such a file is considered to be a library module. If there are multiple executable modules, Tifig will present a dialog that lets the user select which executable should be launched. This is shown in Figure 7.14:

Figure 7.14: *Module Selection Dialog*



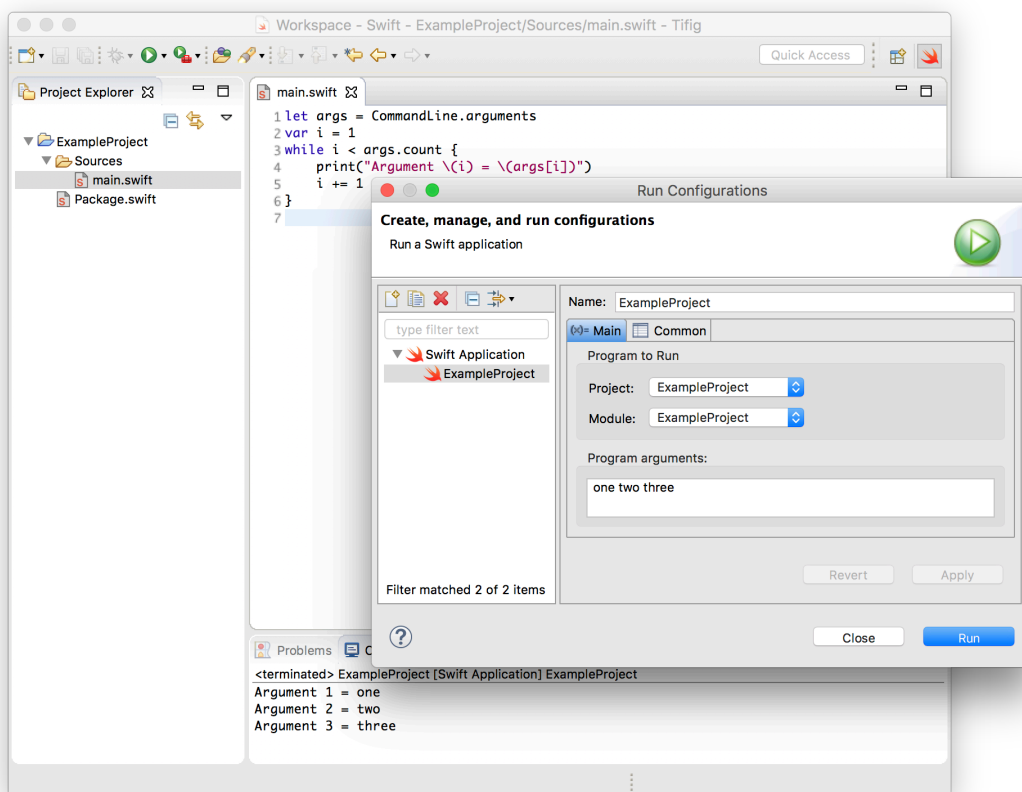
In this example there are three modules because there are three subfolders in the `Sources` folder. The first two modules are executable modules because they contain a `main.swift` file which represents the entry point for execution. The third module called `Utils` is a library module and can therefore not be launched.

Once the user selects an executable module from the list (or if there is only one executable module), a new launch configuration is instantiated from the `SwiftApplicationLaunchConfigurationType`. Its `PROJECT_NAME` attribute is set to the project name and its `MODULE_NAME` attribute is set to the name of the corresponding module. Tifig then saves the new launch configuration and proceeds to step 2.

2. The launch configuration's `launch()` method is called to initiate the launch. The actual work is done by the launch configuration type's delegate (`ILaunchConfigurationDelegate`). It executes the binary in the `/.build/debug` folder and connects the process to a new process console in the console view. This way the user can see the process' output and the process can get input from stdin.

Additionally, users can customize launch configurations by selecting `Run -> Run Configurations...` from the menu bar. At the moment there aren't a lot of customization options but this may be extended in the future. Figure 7.15 shows how to specify the arguments that are passed to the program when it is launched:

Figure 7.15: *Customizing a Run Configuration*



7.9 Implementation Status

The following list describes a few UI features that are currently not yet supported by Tifig and should be added in the future:

- **Auto Completion**

Auto completion is an important feature that users expect from a modern IDE. Thus, this feature should be added to Tifig as well.

- **Code Navigation**

Tifig currently supports the code navigation features *Open Type* and *Jump to Definition*. Two other code navigation features that are commonly used in Eclipse-based IDEs are *Open Call Hierarchy* and *Open Type Hierarchy*. These features should be added to Tifig as well.

- **Parse Compiler Output**

The warnings and errors that are emitted by the compiler are currently only displayed in the console. In the future, it would be more convenient if Tifig were able to parse the compiler output and display the errors in the form of markers in the editor.

8 Conclusion

This chapter evaluates the project results and mentions further work that could be done in the future to improve the Tifig IDE.

8.1 Results

The following list gives an overview over the main features that were implemented during the term project and the subsequent master thesis:

- **Perspective & Wizards**
A simple Swift perspective and a few wizards to create new Swift projects and files have been developed.
- **Parser**
Code that is entered by the user is automatically parsed by a custom Swift parser and syntax errors are reported in the form of markers in the editor.
- **Indexer**
After the Swift code is parsed, it is indexed by a custom indexer. The indexer features a constraint-based type checker that is similar to the type checker in Apple's official Swift compiler. In addition to the user's own code, the indexer also indexes the standard library and makes its public symbols available in every project.
- **Editor**
The Swift Editor consists of several smaller components that implement editor features such as auto indenting and syntax highlighting.
- **Code Navigation**
The semantic knowledge that is obtained by the indexer allowed for the development of the code navigation features *Open Type* and *Jump to Definition*.
- **Builder**
The builder is still very basic and it is not yet possible to specify build settings. Thus, there is certainly room for improvement in the future. However, I think that using the Swift Package Manager was the right decision, because it is a simple solution that probably suits most people's needs.
- **Launcher**
A simple launcher has been developed as well. One can specify the program arguments in the run configuration and select which executable module that should be launched. Other than that, there aren't a lot of customization options yet.

In addition to the development of these components, I made a public website for the project (<https://www.tifig.net>) where I released several alpha versions of the Tifig IDE.

8.2 Outlook

Overall, I think that my term project and the subsequent master thesis were a success. However, there is still a lot that can be done to improve the existing components as described in the *Implementation Status* sections in the chapters 4, 5, 6 and 7. Also, there are additional features that are still missing and need to be implemented in the future (e.g., debugging support, refactoring support, etc.).

8.3 Acknowledgements

I would like to express my gratitude to my supervisor Prof. Peter Sommerlad for the useful comments and assistance during the weekly meetings throughout both the term project and the master thesis.

I would also like to thank Silvano Brugnoli for allowing me to include a version of his Eclipse plug-in pASTa (Painless AST Analysis) in the Tifig IDE.

Furthermore, the product- and the branding-plugin were adopted from the Cvelop IDE [fS17]. Thanks to the *Institute for Software* for giving me access to the corresponding source code.

Bibliography

- [Alf06] Alfred V. Aho and Monica S. Lam and Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2006.
- [App14] Apple. WWDC 2014 Keynote, June 2014. <https://developer.apple.com/videos/play/wwdc2014/101/>.
- [App15] Apple. Protocol-Oriented Programming in Swift, June 2015. <https://developer.apple.com/videos/play/wwdc2015/408/>.
- [App16] Apple. What's New in Foundation for Swift, June 2016. <https://developer.apple.com/videos/play/wwdc2016/207/>.
- [App17a] Apple. Markup Formatting Reference, March 2017. https://developer.apple.com/library/content/documentation/Xcode/Reference/xcode_markup_formatting_ref/.
- [App17b] Apple. Swift, March 2017. <https://swift.org/>.
- [App17c] Apple. Swift Package Manager, March 2017. <https://swift.org/package-manager/>.
- [App17d] Apple. Swift Standard Library Operators Reference, March 2017. https://developer.apple.com/reference/swift/swift_standard_library_operators.
- [App17e] Apple. Type Checker Design and Implementation, March 2017. <https://github.com/apple/swift/blob/master/docs/TypeChecker.rst>.
- [App17f] Apple. *Using Swift with Cocoa and Objective-C*. 2017.
- [App17g] Apple. Xcode, March 2017. <https://developer.apple.com/xcode/>.
- [Beg15] Ole Begemann. Pattern Matching in Swift, September 2015. <https://oleb.net/blog/2015/09/swift-pattern-matching/>.
- [Ben02] Benjamin C. Pierce. *Types and Programming Languages*. 2002.
- [Dev06] Prashant Deva. Create a commercial-quality Eclipse IDE, September 2006. <https://www.ibm.com/developerworks/opensource/tutorials/os-ecl-commplgin1/>.
- [DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs, 1982.
- [E. 94] E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. 1994.

Bibliography

- [ecl17a] Eclipse CDT (C/C++ Development Tooling), March 2017. <https://eclipse.org/cdt/>.
- [ecl17b] Eclipse Java development tools (JDT), March 2017. <https://eclipse.org/jdt/>.
- [ecl17c] Eclipse Project, March 2017. <https://eclipse.org/eclipse/>.
- [ecl17d] Perspectives, March 2017. http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fworkbench_perspectives.htm.
- [ecl17e] Project natures, March 2017. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_natures.htm.
- [ecl17f] Syntax coloring, March 2017. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Feditors_highlighting.htm.
- [Fou04] The Apache Software Foundation. Apache License, January 2004. <http://www.apache.org/licenses/LICENSE-2.0>.
- [fS17] Institute for Software. Cevelop - The C++ IDE for professional developers, March 2017. <https://www.cevelop.com/>.
- [Gal16] Alexis Gallagher. A recipe for Value Semantics (not value types!), December 2016. <https://realm.io/news/swift-gallagher-value-semantics/>.
- [Har96] Harold Abelson and Gerald Jay Sussman and Julie Sussman. *Structure and Interpretation of Computer Programs*. 1996.
- [J.A10] J.A. Bondy and U.S.R. Murty. *Graph Theory*. 2010.
- [jav17] Example - Java Editor, March 2017. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fsamples%2Forg.eclipse.ui.examples.javaeditor%2Fdoc-html%2Fui_javaeditor_ex.html.
- [jun17] JUnit, March 2017. <http://junit.org/junit4/>.
- [Kre16] Ted Kremenek. Swift 3.0 Released!, September 2016. <https://swift.org/blog/swift-3-0-released/>.
- [Kre17] Ted Kremenek. Swift 4 Release Process, February 2017. <https://swift.org/blog/swift-4-0-release-process/>.
- [Lat10] Chris Lattner. Initial Swift Commit, July 2010. <https://github.com/apple/swift/commit/18844bc65229786b96b89a9fc7739c0fc897905e>.
- [Lat17a] Chris Lattner. Chris Lattner's Résumé, March 2017. <http://www.nondot.org/sabre/Resume.html>.
- [Lat17b] Chris Lattner. Update on the Swift Project Lead, January 2017. <https://lists.swift.org/pipermail/swift-evolution/Week-of-Mon-20170109/030063.html>.

Bibliography

- [Mar99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.
- [rai17] RAII, March 2017. <http://en.cppreference.com/w/cpp/language/raii>.
- [Rob11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 2011.
- [Sap17] A.A. Sapozhenko. Hypergraph, March 2017. <https://www.encyclopediaofmath.org/index.php/Hypergraph>.
- [swi15] The Swift.org Blog, December 2015. <https://swift.org/blog/welcome/>.
- [swi17a] Community Guidelines, March 2017. <https://swift.org/community/>.
- [Swi17b] Lexicon, March 2017. <https://raw.githubusercontent.com/apple/swift/master/docs/Lexicon.rst>.
- [Ter10] Terence Parr. *Language Implementation Patterns*. 2010.
- [thr17] Threading issues, March 2017. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fswt_threading.htm.
- [tjl17a] Function Types. In *The Java Language Specification*. Oracle, 2017.
- [tjl17b] Functional Interfaces. In *The Java Language Specification*. Oracle, 2017.
- [try17] The try-with-resources Statement, March 2017. <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>.
- [tsp17a] About Swift. In *The Swift Programming Language*. Apple, 2017.
- [tsp17b] Automatic Reference Counting. In *The Swift Programming Language*. Apple, 2017.
- [tsp17c] Language Reference. In *The Swift Programming Language*. Apple, 2017.
- [tsp17d] Lexical Structure. In *The Swift Programming Language*. Apple, 2017.
- [tsp17e] The Basics. In *The Swift Programming Language*. Apple, 2017.
- [uni17] Unicode, March 2017. <http://www.unicode.org/>.