
Diplomarbeit

Alexander Galkin

Implementing a Multi-Target .Net Compiler for TouchDevelop

29.05.2012

supervised by:

Prof. Dr. Sibylle Schupp

Prof. Dr. rer. nat. habil. Ralf Möller

Technische Universität Hamburg-Harburg
Institute for Software Systems
Schwarzenbergstrasse 95
21073 Hamburg

STS
Software
Technology
Systems

*To my beloved wife Irina and son Kyrill,
who encourage me and believe in me in all times.*

Erklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tage eingereichte Studienarbeit vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Alle Abbildungen in dieser Arbeit sind eigene Darstellungen. Wo Abbildungen übernommen und abgeändert wurden, ist die Quelle der ursprünglichen Abbildung angegeben.

Hamburg,

_____.

.....
(Unterschrift)

Acknowledgments

First of all, I would like to thank to my supervisor Prof. Sibylle Schupp for her patience, generous but prodding attitude to my work, personal supervision of my thesis and last but not least for her immense efforts to help me improve my English writing skills.

I am thankful to Nikolai Tillmann for his topic suggestions, valuable information about TouchDevelop and interesting conversations.

I am very grateful to the the author of the FParsec framework Stephan Tolksdorf who has been very patient and answered my numerous emails almost instantly.

I am indebted to my father-in-law Anatoli Benke who took his days off to come and personally support me during the last week of my work.

Abstract

TouchDevelop is a novel development environment that is specially designed to support quick and easy source code authoring on mobile devices running Windows Phone. TouchDevelop is a common name both for the mobile IDE and the language it uses. As a language, TouchDevelop is a multiparadigm, strongly typed language with static type checks.

This work presents a language specification that was derived from the script corpus downloaded from the TouchDevelop cloud in December 2011 using a reverse-engineering approach. It provides a formal description of the language as well as the implementation of a compiler consisting of a parser, static semantic checker, and code generator.

The base-line grammar was reverse-engineered and successfully tested on a special very small testset of scripts using ANTLR. This non-optimized yet highly human-readable base-line grammar contains two left-recursive rules and requires the look-ahead of two tokens.

Since the parser automatically generated by ANTLR from this grammar is computationally inefficient and did not support the abstract syntax of the language, we also implemented a non-backtracking recursive-descent strong $LL(1)$ parser in F# using the parser combinator library FParsec. This library provides an embedded expression parser, which we customized for correct generation of abstract semantic trees out of TouchDevelop expressions. This parser successfully parses 278 out of 282 sample scripts from the code base requiring less than a second for 35075 lines of code. For the four scripts it fails to parse, it does so correctly since the scripts contain syntax errors.

The static semantics for TouchDevelop was implemented manually in F# using the rules described in the language specification using denotational semantics.

The code generator performs two transformations. The abstract syntax tree is first transformed into a CodeDom tree representation. This tree is then converted into a .Net binary by means of the CodeDom classes. The dynamic semantics of .Net runtime was used for code execution.

The compiler targets .Net client profile and Silverlight 4.0: the console-only code compiles to a full-fledged .Net 4.0 console application whereas the low-trust Silverlight 4.0 dynamically linked library is generated as the target for Silverlight.

The compiler does not yet have industrial strength. In particular, there is no support for Windows Phone as a compilation target, because the compiled package has to be digitally signed to allow its deployment and execution on a mobile device or emulator. Therefore, none of phone-specific aspects of TouchDevelop, including the described event model, are currently supported. Furthermore, only a minimal set of the TouchDevelop library was implemented to test the compiler.

This work contributes to the best practices in reverse engineering, especially to the use of ANTLR in the two-step reverse engineering of language specification, as well as to multi-targeting compiler implementations for the .Net platform.

Zusammenfassung

TouchDevelop ist eine neuartige Entwicklungsumgebung, die speziell für das schnelle und einfache Quellcode-Authoring auf mobilen Geräten mit Windows Phone konzipiert wurde. Das Wort TouchDevelop ist eine gemeinsame Bezeichnung für die mobile IDE und für die Programmiersprache. Sprachlich gesehen ist TouchDevelop eine multiparadigmale, stark typisierte Programmiersprache mit statischen Typüberprüfungen.

Diese Arbeit präsentiert eine Sprachspezifikation, die anhand der im Dezember 2011 heruntergeladenen Skriptsammlung unter Verwendung eines Reverse-Engineering-Ansatzes abgeleitet wurde. Neben einer formalen Beschreibung der Sprache ist die Implementation der Compiler-Architektur, bestehend aus einem Parser, einem statischen Semantik-Checker und einem Code-Generator, ebenfalls ein Teil dieser Diplomarbeit.

Die Rekonstruktion der Baseline-Grammatik und deren anschließendes Testen anhand einer kleinen Menge von Testskripts erfolgte durch das ANTLR. Diese zwar rechnerisch nicht-optimierte dennoch für Menschen höchst lesbare Baseline-Grammatik enthält zwei linksrekursive Regeln und benötigt das Look-Ahead in Höhe von zwei Token. Der automatisch durch ANTLR anhand dieser Grammatik erzeugte Parser war rechnerisch nicht optimal und bot keine Unterstützung für die abstrakte Syntax der Sprache.

Um diese Probleme anzugehen, wurde ein nicht-zurückziehende rekursiv-absteigende starke $LL(1)$ -Parser in F# unter Verwendung der Parser-Kombinator-Bibliothek FParsec händisch umgesetzt. Diese Bibliothek enthält einen eingebetteten Ausdrucksparser, der für die korrekte Generierung von abstrakten syntaktischen Bäumen aus dem TouchDevelop-Quellcode angepasst wurde. 278 von 282 Beispiel-Skripts aus der Skriptsammlung (35.075 Zeilen Code), wurden durch den Parser in weniger als eine Sekunde erfolgreich geparkt. Alle vier Skripte, bei denen das Parsen fehlgeschlagen ist, beinhalteten Syntaxfehler.

Die statische Semantik für TouchDevelop wurde anhand der Regeln der denotationellen Semantik aus der Sprachspezifikation manuell in F# implementiert, beschrieben.

Der Code-Generator führt zwei Transformationen durch. Der abstrakte Syntax-Baum wird zuerst in die CodeDOM-Baum-Darstellung umgewandelt. Dieser Baum wird anschließend in .NET Assembly umgewandelt. Für die Ausführung vom Code wurde daher die dynamische Semantik der .NET-Laufzeitumgebung verwendet.

Der Compiler unterstützt .Net 4.0 Clientprofil und Silverlight 4.0 als Kompilierungsziele. Der Code mit Konsolenausgabe wird in eine vollwertige .Net-4.0-Konsolenanwendung kompiliert, während der Code für low-trust Umgebung in eine dynamische Bibliothek für Silverlight 4.0 umgewandelt wird.

Der Compiler hat noch keine industrielle Stärke. Insbesondere gibt es keine Unterstützung für Windows Phone als Kompilierungsziel, da das Deployment-Paket eine digitale Signatur braucht, um auf einem mobilen Gerät oder Emulator installiert und getestet zu werden. Daher werden derzeit keine Telefon-spezifische Aspekte von TouchDevelop, einschließlich des beschriebenen Event-Modells, unterstützt. Nur ein Minimalanteil der TouchDevelop Standardbibliothek wurde implementiert, um den Compiler zu testen.

Diese Arbeit trägt zur Ausarbeitung von Best Practices für Reverse Engineering und die Implementation eines Multi-Targeting-Compilers bei. Insbesondere wurde gezeigt, wie der Einsatz von ANTLR im Zwei-Schritt-Ansatz das Reverse-Engineering der Sprachspezifikation erleichtern kann.

Contents

1	Introduction	1
1.1	TouchDevelop – a development environment for mobile devices	2
1.2	A programming language or just a game?	2
1.3	A programming language for casual developers	4
2	Syntax and semantics of programming languages	7
2.1	Properties of formal languages	7
2.2	Formal notation of program languages syntax and semantics	18
2.3	Reverse engineering of languages	21
3	Language Specification of TouchDevelop	25
3.1	Reverse engineering of the language specification	25
3.2	Script authoring	27
3.3	Notation	30
3.4	Source code representation	30
3.5	Lexical structure	33
3.6	Types	36
3.7	Declarations	42
3.8	Expressions	50
3.9	Statements	54
3.10	Special types	57
3.11	Program execution	59
3.12	Cloud services	62
4	Implementation aspects	63
4.1	Deriving the language specification	63
4.2	Parser implementation (syntax analysis)	65
4.3	Semantic analysis	67
4.4	Code generation	67
5	Discussion	77
5.1	Threats to validity	77
5.2	Comparison to existing works	80
5.3	Reflection on design decisions	84
6	Conclusion	87
	Bibliography	101

List of Figures

1.1	Sample applications implemented in TouchDevelop	3
2.1	Derivation tree a binary literal.	10
2.2	A metaphor for different grammar power as a silhouette of a rose. Numbers represent the types of grammar according to the Chomsky hierarchy. From Grune and Jacobs [2011].	11
2.3	Grammar hacking vs. grammar engineering, from Klint et al. [2005].	22
2.4	The life-cycle of language reverse engineering.	24
3.1	The TouchDevelop as it is seen in the list of installed applications.	27
3.2	TouchDeveloper Script Manager as WP7 Panorama application.	28
3.3	TouchDeveloper code editor.	29
3.4	Using the „fix-it” feature of code editor to declare a new local variable.	29
3.5	Basic definitions used in language semantics.	30
3.6	The usage of Unicode identifiers in TouchDevelop scripts.	32
3.7	Semantic domains for types (see Fig. 3.8 for selection operation)	37
3.8	Semantic of the Boolean type.	39
3.9	Semantics of type Number and expressions with this type.	41
3.10	Semantics of the String type	41
3.11	Scopes in TouchDevelop.	45
3.12	Wall with prompts for Boolean and numeric values.	58
3.13	Execution model of TouchDevelop.	60
3.14	Screenshot of the TouchDevelop web portal.	61
3.15	Information about a single script on the TouchDevelop web portal.	62
4.1	Roadmap of a compiler implementation (adapted from Aho et al. [1986])	63
4.2	Debugging the TouchDevelop grammar in ANTLR.	64
4.3	Classes of the CodeDom syntax tree.	70
4.4	The class diagram of compiler library attributes.	71
4.5	Attribute decorations in the implementation of the TouchDevelop standard library.	73

List of Tables

2.1	Chomsky’s hierarchy of grammars.	11
2.2	An overview of parsing techniques, from Grune and Jacobs [2011], adapted.	13
2.3	The notation by van Wijngaarden.	19
3.1	Unicode symbols used in TouchDevelop application and scripts.	31
3.2	Visualization properties of different source code views.	34
3.3	TouchDevelop keywords and symbols.	34
3.4	Value types in TouchDevelop (Horspool et al. [March, 2012], page 64).	37
3.5	Properties of the <code>invalid</code> type in TouchDevelop.	38
3.6	Operators supported by the <i>Boolean</i> datatype in TouchDevelop.	39
3.7	Operators supported by the <code>Number</code> datatype in TouchDevelop.	40
3.8	Collection types in TouchDevelop.	43
3.9	Meta declarations in TouchDevelop.	46
3.10	Extrinsic (sensor) events in TouchDevelop.	48
3.11	Intrinsic events in TouchDevelop (“<” and “>” indicate the variable parts of a handler).	49
3.12	Operator expressions in TouchDevelop (Precedence: 10 – highest, 0 – lowest).	52
4.1	Ways to generate executable .Net code.	68
4.2	Transformations performed by the TouchDevelop compiler.	72
5.1	Overview of the specification versioning for TouchDevelop as language.	78
5.2	Possible inaccuracies in Horspool et al. [March, 2012]	82
5.3	Design decisions in TouchDevelop.	85

This page is intentionally left blank.

1

Chapter 1

Introduction

I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone.

Bjarne Stroustrup, the inventor of C++.

The reason to start this work was the lack of a comprehensive language specification for TouchDevelop. The existing implementation available only for Windows Phone was a language, an IDE, an interpreter, and a runtime in one with closed sources under a proprietary (albeit royal-free) license.

To derive the language specification we used the original application for Windows Phone and a set of 282 scripts downloaded from the TouchDevelop webpage in December 2011: these were all scripts available on the webpage at that time. We designed our own script to test for different aspects of language syntax and semantics which we used as input for the TouchDevelop phone application. We never performed a decompilation of the TouchDevelop binaries or used any other techniques to reverse engineer the source code of the original application.

Thus, the main goal of this thesis was to reconstruct the language syntax and semantics of a closed-source project by Microsoft Research called TouchDevelop. The secondary goal was to provide an implementation of a simple TouchDevelop compiler using a re-engineered language specification.

The major outcome of this work is the reverse-engineered version of the language specification for TouchDevelop, presented in the third chapter. The language specification includes both syntax and semantics of the language.

As a proof-of-concept for the reverse-engineered specification we implemented a simple TouchDevelop compiler along with a minimal subset of the TouchDevelop standard library. The compiler consists of the parser, semantic checker, and code generator. The details about the compiler implementation are provided and discussed in the fourth chapter.

The current thesis consists of six chapters. The first chapter provides an introduction to TouchDevelop and discusses the role of TouchDevelop as a platform. The second chapter describes the foundations of formal languages, grammars, and the approaches to reverse-engineering of language specification. The approaches presented in this chapter, like the two-step re-engineering of language grammar, are applied to the existing TouchDevelop application to derive the data necessary for the later chapters.

The third chapter contains the language specification for TouchDevelop, reconstructed from publicly available scripts and the behavior of the mobile application. The specification defines both the grammar and the syntax of TouchDevelop and discusses the problems of specification mining in the case of TouchDevelop.

Chapter four presents the details of the compiler architecture, the second outcome of the current work. The workflow of the compiler implementation and the limitations of the current implementation are discussed here as well.

The fourth chapter speculates on the validity and the lifetime of the derived language specification and reflects on the most important design decisions for TouchDevelop as a programming language.

The last, sixth chapter concludes the work and provides an outlook on the further development of the project.

The ANTLR grammar for TouchDevelop, AST types for the parser, list of error messages of the semantic checker and the sample TouchDevelop script used to derive the specification are available in the Appendix.

1.1 TouchDevelop – a development environment for mobile devices

More than 4000 programming languages have been created during the last fifty years. Besides the well-known general-purpose multi-paradigm languages like C++, Java, or C# there are plenty of domain-specific languages that are designed to satisfy the need of a particular domain.

One of the most rapidly evolving domain for software development are mobile devices. The number of smartphones and tablet devices exceeds that of conventional personal computers. Due to the ubiquity of these devices they have been seen for a long time as a promising market for developing games and business applications. However, nobody really treated these devices as a development platform until recently: the properties of these devices – their small-sized screen, low resolution, short battery life, absence of convenient input device like keyboard – do not seem attractive for software development.

On the other side, the high prevalence of mobile devices among a wide range of social groups, carrying these devices at all times, and the permanent connection to the Internet makes them an attractive platform for hobby developers who may want to implement simple projects that would immediately run on the device while commuting.

TouchDevelop was created with the idea in mind to develop a mobile developing environment and language runtime (interpreter) that would enable the direct input of user scripts on mobile devices using touch-sensitive displays and finger input.

TouchDevelop is a new programming environment and language designed „to make it possible to write applications on mobile devices” Tillmann et al. [2011]. The project was initiated in the beginning of 2011 and is currently developed by a group of researchers from the Research in Software Engineering (RiSE) group at Microsoft Research in Redmond, WA. The project rapidly grew into a full-fledged language with an option-rich IDE and solid cloud support.

1.2 A programming language or just a game?

Before we begin with the formal specification of TouchDevelop we have to decide whether we can and should consider TouchDevelop a programming language, a developer tool, or

1.2 A programming language or just a game?

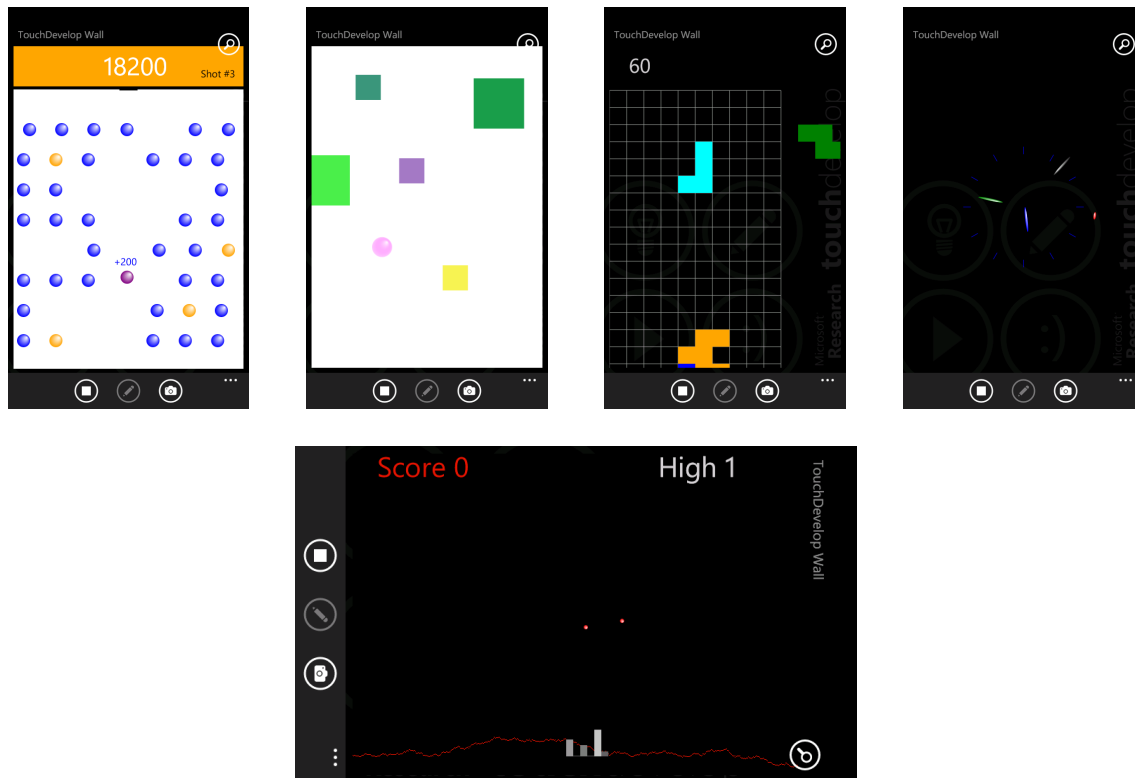


Figure 1.1: Sample applications implemented in TouchDevelop

From left to right, upper row: (a) MegaPegs, a ball-breaking game; (b) SaveTheBubble, dexterity action game; (c) TouchTris, a clone of the famous Tetris game; (d) Analogue clock. Bottom row: Missile Defence - a shooter action game.

just another mobile application for Windows Phone, next to Facebook¹ and Foursquare² applications.

Analyzing the traditional games in his book „A Casual Revolution: Reinventing Video Games and Their Players” Jesper Juul from MIT states that „[there is a] pull [...] of being unable to fit a game into your life. There is a new wave of video games that seem to solve the problem of the missing pull; games that are easy to learn to play, fit well with a large number of player and work in many different situations.” (Juul [2010], page 5). Another common trend in the Internet and social networks is the ever-growing role of casual games in the user’s daily activity known as „gamification” (Deterding [2011]). Casual games, requiring neither prior knowledge of the subject or a thorough study of the game manual (in those rare occasions when one exists) nor special skills, flooded the social networks and contribute to a substantial part of user’s daily activity.

If we look at TouchDevelop from the viewpoint of a player we will see many traits (and design decisions behind them) similar to classical social network games like FarmVille.³ In particular:

- TouchDevelop does not target professional developers. Rather, the focus is set to

¹A free application for Windows Phone to work with the social network Facebook – a platform for internet games, image and video sharing, blogging etc.

²A free client application and a social platform for location-based activities: providing feedback for restaurants, hotels etc. and finding who from your friends is in the vicinity.

³FarmVille is a farming simulation social network game developer by Zynga in 2009 and popular as a game application for the social network Facebook and other platforms

hobby and casual developers and even to people without any developer background.

- TouchDevelop is a cloud-based platform with occasionally connected clients. User activity is stored online and is automatically synced on all connected devices (similar to game progress savings).
- TouchDevelop provides at least two levels of privacy: a private mode for scripts that get synchronized but are not visible to other community members and „publishing” scripts that are then visible for everyone and available for forking.
- Both the mobile client and the web portal feature rich possibilities to review, rate, and comment the scripts submitted by a user. User activity and community contributions can be rated by other community members (similar to „likes” on Facebook or „+1” on Google+).
- Similarly to photo upload in social networks, TouchDevelop provides an one-button-click upload for application screenshots. This similarity is also valid for applications with leaderboards where TouchDevelop provides the transparent submission and storing of user scores.
- Community feedback to user activity is tracked as a user personal rating („hearts” in TouchDevelop), which is directly visible (and shown!) to other community members along with the full list of submitted scripts. This is similar to the achievement systems common for modern casual games.

TouchDevelop is not the first attempt to design a casual game with elements of a programming language. However, those games were usually biased either towards the programming language and algorithmic complexity, like Microsoft Terrarium (Richardson [2003]) or AntMe! (Saumweber [2007, c2008]), which both used C# as a programming language and served as a sample applications to implement artificial intelligence, or towards the visual effects and gaming factor, like Kodu, a visual programming language for children (MacLaurin [2009], Stolee and Fristoe [2011]).

The steady growth of community around TouchDevelop, as seen on its website, might prove the fact that the application has found a good niche between a casual game for a mobile device (phone) and a real-world programming language.

We did not examine the community, but judging from the scripts that get submitted to the webportal and receive high rankings one can tell that TouchDevelop is actively used for developing simple (but not primitive!) casual games (CloudHopper⁴, TapTris⁵) and small utility applications (MyOnlineMeetings⁶, TodoList⁷). The mentioned scripts were all developed (according to the information from personal profiles) by non-professionals: a manager, a technologist, etc.

In this respect we would like to coin a new term to describe an ordinary community member of TouchDevelop website – casual developer.

1.3 A programming language for casual developers

TouchDevelop appeared at the time when programming small but useful or attractive application is no longer a privilege of a professional developer. To address the needs of people with no prior experience in developing applications using traditional, „full size” languages like Java or C#, the platform designers made several crucial decision:

⁴<https://www.touchdevelop.com/wbxsa>

⁵<https://www.touchdevelop.com/vqno>

⁶<https://www.touchdevelop.com/mpuj>

⁷<https://www.touchdevelop.com/qanh>

1.3 A programming language for casual developers

- TouchDevelop is a stand-alone, mostly procedural language with 1 predefined class-like types.
- TouchDevelop supports reactive event-driven programming model with event handlers automatically mapped to preset events.
- TouchDevelop encourages „thumb programming.”
- TouchDevelop runs directly and only on mobile phones — an ubiquitous device today, which everyone normally carries along all the time.
- **GameBoard** leverages programming of 2D sprite-based games and even includes a physics engine for games requiring it.

Therefore, by targeting a broader group of users as developers TouchDevelop was designed with the trade-off between simplicity and powerfulness. It is no wonder that casual developers favor rich built-in capabilities for 2D game programming over the support for object-oriented or functional features.

To enable the implementation of an open-source compiler that would allow the casual developers not only to author scripts, but also to publish them as standalone phone applications, we needed to reverse-engineer the syntax and semantics of TouchDevelop, the programming language of the platform.

2

Chapter 2

Syntax and semantics of programming languages

Before one can start with the reverse-engineering of a language specification, one has to acquire the general knowledge about the syntax and semantics of programming languages. The *minimum minimorum* of this knowledge is summarized in the current chapter.

In the first section we describe the properties of formal languages. We begin with a brief characterization of the most common formalisms, syntax and semantics, followed by the basic definitions from formal language theory, like grammar, language, production etc. Some aspects of formal languages that are required for the fulfillment of the thesis goal, like grammar derivation and parser implementation, are presented with a higher level of details.

The second section deals with other important requirement for language reverse-engineering: the notations for the formalisms discussed in the first section.

The last, third section rests on the definitions from the previous two sections and provides the necessary background information about the principles and best practices for reverse-engineering of language grammars. These will be applied to the TouchDevelop for deriving the language specification in the next chapter.

2.1 Properties of formal languages

2.1.1 Two essential formalisms: syntax and semantics

If we treat a compiler as a tool that receives text input and provides an executable as output, the following data and steps are needed to perform this task:

- a formal, structural and systematic description of the input data (=grammar),
- a program for reading the input data and transforming it to an independent internal format (=parser),
- a formal structural and systematic description of how the output can be generated from input (=semantics),
- a program that would use the previous description to generate the code (=compiler back-end).

So, to successfully develop a compiler we need two different types of formalisms: the structure of the source code, called syntax, and the computational meaning of this code, called semantics.

The description of both semantics and syntax can be either informal or formal. Informal description can be provided in the form of an essay-styled description of the language illustrated with code samples to explain concepts. Formal description, however, has to be written in a notation with precise meaning. To achieve precision the notation has to be standardized and specified using other proven notations or by the means of mathematical logic.

Formalization of the language syntax fulfills the following tasks:

- A formal definition of the language syntax leads to its standardization.. This step is crucial for both language consumers – developers who would like to write syntactically correct programs in this language – and for language implementers, whose task is to provide the infrastructure for correct compiler implementations.
- As soon as the syntax of the language is formally defined one can perform analysis of its (syntax) properties: whether it is possible to derive a context-free grammar for this language, what amount of look-ahead is necessary for the parser, whether the definition is $LL(k)$, $LR(k)$, or ambiguous etc.
- The proper format definition can be directly transformed into a parser by using appropriate tools, like YACC (for LR parsers) or ANTLR (for strong LL parsers), bypassing a manual implementation step.

If syntax concerns with how the program are written, semantics deal with the questions what legal programs mean and describes the behavior these programs produce when executed on some real computer or by a virtual machine. Several alternative semantics are possible for the same syntax. Semantics can not cover the program execution in every tiny detail, but usually this is not necessary: normally only the features that are deemed to be relevant constitute the language semantics. Those features are often the relations between the source code (input) and the program result (output) and whether the execution terminates or not.

Formalization of semantics, in its turn, brings the following benefits:

- A formal definition provides a valid and standard way to interpret the code in the given language. This is important for both developers to understand how the code is executed and for compiler constructors to implement the correct transformation of the language syntax into the executable form.
- A formal definition, along with the syntax, are the prerequisites for static analysis of the language properties, for instance the formal proof of type safety. Besides, it can ensure the validity of certain rules defined as contracts for the code, and the absence of errors.
- Some software available today is capable of generating the back-end compiler code based upon the language semantics, similarly to parser generation tools. These tools are known as „compiler generators” or „compiler compilers”.

The purpose of a grammar, known as *grammar use case*, might be very different depending upon the formalism it tries to describe. In the context of programming languages the following are the most common use cases:

- *Modeling of the source code.* This use case corresponds to the lexer step in compiler construction and helps to define the source code of the language under preservation of rich features like annotations, aspects, scaffoldings or metadata. TouchDevelop relies on the model of source code gained from the tokenized user input so that no syntax error is possible at the script scope.
- *Intermediate program representation.* This is the next step of the source code transformation where the complete program representation is abstracted from its

source code and aligned according to the syntactic rules of the language. A typical example of this use case can be the abstract syntax tree constructed by the parser using the input from lexer.

The above two models can be further used concomitantly or alone as the input to the following tools [Klint et al., 2005]:

- compiler middle- and back-ends for code generation,
- control- or data-flow-based static code analyzers,
- pretty printers,
- documentation generators,
- preprocessors and program specializers,
- debuggers and profilers.

2.1.2 Basic definitions

Before we can discuss the reverse-engineering approach to the grammar we need to precisely define this term. This term originates from linguistics and according to Chomsky represents „the essence of human language” (Chomsky [1976]). Adapting this aphoristic definition for Computer Science we consider grammars as established formalisms and notations to describe a language (Klint et al. [2005]). Possible formalisms can be context-free grammars, algebraic signatures, or regular tree and graph grammars.

Grammars are typically a structural, static descriptions of a system and is seen independent from its interpretation (semantics), which should not necessarily be meaningful; several alternative semantics can exist for one grammar (Klint et al. [2005]). For example, a useful semantics of context-free grammar can be the set of all valid derivation trees, whereas the *de facto* standard semantics for the grammars of this type are their generated languages (Aho et al. [1986]).

A language grammar G is a set of rules for combining entities to a well-formed text. The entities are called terminal symbols if they contain only text and no other entities. The combinations of terminal symbols are described using grammar rules, also known as production rules, and non-terminal symbols. Non-terminal symbols do not appear in the final texts: their only role is to describe the format of the well-formed text.

Definition 1. A generative grammar $G = (V_T, V_N, R, S)$ is a 4-tuple containing a set V_T of terminals, a set V_N of non-terminals, a set R of rules, and a starting symbol S such that (1) V_N and V_T are finite sets of symbols, (2) $V_N \cap V_T = \emptyset$, (3) R is a set of pairs (P, Q) such that (3a) $P \in (V_N \cup V_T)^+$ and (3b) $Q \in (V_N \cup V_T)^*$, and (4) $S \in V_N$.¹

Every rule R has the form $A = f_1 | \dots | f_n$, where $A \in V_N$ is a non-terminal symbol, each alternative f_i is a sequence, and $n \geq 1$. Each sequence has the form $e_1 \dots e_m$, where each e_j is a symbol in $T \cup N$ (i.e., either a terminal or a non-terminal), and $m \geq 0$. When $m = 0$, the sequence is empty and is denoted as Λ .

A grammar for binary literals representing one byte (8 bits) can be written using the following rules:

$$B = TTTTTTTT.$$

$$T = "0" \mid "1".$$

We may derive the string "0" or the string "1" from the non-terminal T , by replacing or substituting either "0" or "1" for T . These derivations can be written $T \Rightarrow "0"$ and $T \Rightarrow "1"$ respectively.

¹* and + are Kleene operators defined in the following way: if V is a set of symbols or characters then V^+ is the set of all strings over symbols in V , and $V^* = V^+ \cup \Lambda$ where Λ is an empty string.

In each step of a derivation we replace a non-terminal with one of the alternatives on the right hand side of its rule. A derivation for the binary number „01010110” can be shown as a tree with every tree node representing a non-terminal encoded as T . Each leaf of the tree is labeled by a terminal symbol, such as „0” or „1”. Reading the values on the leaves in sequence from left to right gives the string derived from the symbol at the root of the tree: „01010110” (see Figure 2.1).

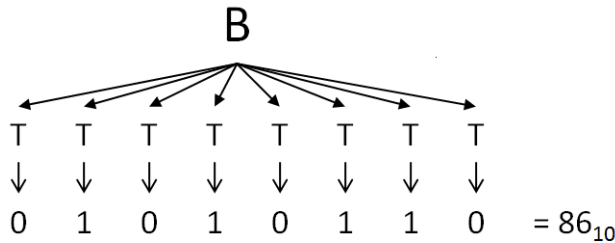


Figure 2.1: Derivation tree a binary literal.

One can imagine a grammar G as a string generator for strings containing only the terminal symbols. Grammars are useful because they are finite and compact descriptions of usually infinite languages.

Definition 2. Let T^* be the set of all strings that can be derived from T , including the empty string Λ . When A is a non-terminal, the set of strings derivable from A is called a language:

$$L(A) = \{w \in T^* \mid A \Rightarrow w\}$$

$L(A)$ denotes a language with A as a starting symbol and hence for the generative grammar G with the starting symbol S the language generated by G or above G is $L(G) = L(S)$.

The class of grammars described above is called the context-free grammars and is just one class in the hierarchy identified by Chomsky, which constitutes

- the unrestricted,
- the context-sensitive,
- the context-free, and
- the regular grammars.

The unrestricted grammars are more powerful than the context-sensitive ones, which are in turn more powerful than the context-free ones, which are again more powerful than the regular grammars [Aho et al., 1986]. The unrestricted grammars cannot be parsed in general; they are mostly of theoretical interest and of little practical use in computing. All context-sensitive grammars can be parsed, but require an excessive amount of time and memory space, and so they are of little practical use either. The context-free grammars are the first class of grammars that are highly useful in computing. The regular grammars can be parsed very efficiently using the constant amount of memory, but they are rather weak; they cannot define parenthesized arithmetic expressions, for instance. Table 2.1 summarizes the hierarchy of grammar classes.

While some text books, as quoted above, state that “Type n grammars are more powerful than Type $n + 1$ grammars” [Aho et al., 1986], other books claim “A regular (Type 3) grammar is not powerful enough to match parentheses” [Grune and Jacobs, 2011]. To address these statements we need to define the power of grammars. One might naively think that the power of a grammar is measured by its ability to generate larger sets of strings, but this is clearly incorrect because the largest possible set of strings is easily generated by the regular expression “ a^* ”, a straightforward Type 3 grammar. The power of a grammar, however, is the richness of the possibilities how we can restrict this set: more powerful grammars can define more sophisticated boundaries between correct and incorrect phrases. Some boundaries are so delicate that they are beyond the capabilities of any grammar.

Chomsky hierarchy	Sample rules	Definition of the grammar
Type 0: Unrestricted grammars	$"a" B "b" \Rightarrow "c"$	Unrestricted phrase structure grammars with ε -rules
Type 1: Context-sensitive (CS) grammars	$"a" B "b" \Rightarrow "a" "c" "b"$	CS grammars without ε -rules
Type 2: Context-free (CF) grammars	$B \Rightarrow "a" B "b"$	Context free ε -free grammars
Type 3: Regular (RE) or finite-state (FS) grammars	$B \Rightarrow "a" \mid "a" B$	Regular grammars (regular expressions)
Type 4: Finite-choice (FC) grammars	$B \Rightarrow "a" \mid "b"$	Finite-choice, no production grammars.

Table 2.1: Chomsky's hierarchy of grammars.

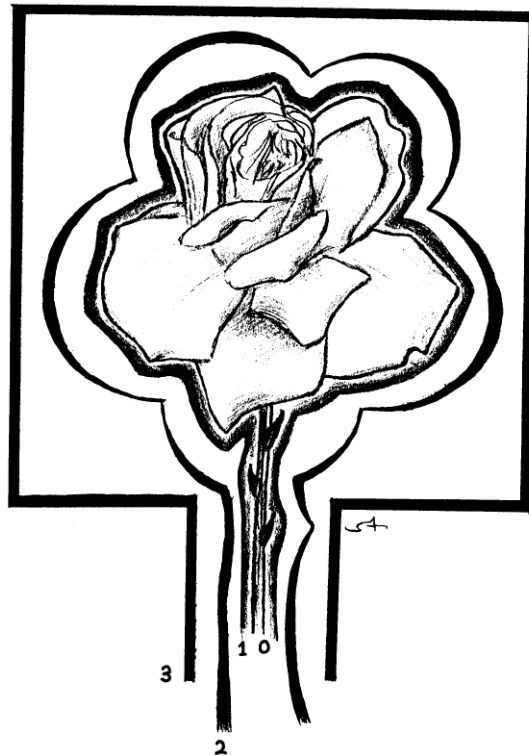


Figure 2.2: A metaphor for different grammar power as a silhouette of a rose. Numbers represent the types of grammar according to the Chomsky hierarchy. From Grune and Jacobs [2011].

This idea can also be depicted metaphorically by a drawing where a rose is approximated by increasingly finer outlines [Grune and Jacobs, 2011]. In this metaphor, the rose corresponds to the language with its petals being the sentences of the language. We apply grammars of different types to approximate its silhouette (Figure 2.2).

- A regular grammar (Type 3) would only allow us to use straight horizontal and straight vertical lines to delineate the flower. A T-shaped form will be already enough (and the only possible here), but the result is very coarse and imprecise.
- A context-free grammar (Type 2) would provide straight lines at any angle and arcs for silhouette approximation. The resulting drawing would resemble a flower, but would not allow us to recognize if it is a rose or camomile.
- A context-sensitive (Type 1) grammar would be a thin, smooth curve that tightly envelops the flower. The smoothness, however, is the constraint here: the line fails to follow all the sharp turns and might miss the silhouette at very abrupt sites. This is a very realistic drawing of a somewhat idealized rose, not the rose that is standing in a vase in front of us.
- An unrestricted phrase structure (Type 0) grammar is the one that can capture the outline perfectly, but only to the extent a human hand can depict.
- The rose itself cannot be caught in any finite description, for its essence remains forever out of our reach.

2.1.3 Syntax

Syntax deals with the structure of source code and refers to the spelling of programs, whether it is „legal” or not. The legality is determined by finding connections and relations between symbols (terminals) and phrases (production rules) that occur in the code. These connections can represent concrete syntax which determines what text strings are accepted as programs, or the abstract syntax that describes the general program structure.

Concrete syntax deals with the string input and is specified by formal grammars via production rules defining the set of valid alternatives for source code. This grammar must contain no ambiguities and each valid program leads to a unique parse tree.

Abstract syntax is generally simpler and more loosely defined as the concrete syntax and is mainly concerned with the „deep structure” of programs [Mosses, 2006]; one operates on the tree representations of the source code. These trees, called abstract syntax trees (AST), represent the operations or transformations described in the source code and each node is created by a particular constructor rule. Every input parameter of this rule is represented as a separate child branch of this node and the result of the node action is propagated towards the tree root as a single parent branch.

The complete syntax description should be a combination of both concrete and abstract syntax. Nonetheless, it has become a usual practice to provide only the description of the concrete syntax and leaving the abstract syntax unrestricted, so that it can be easily defined by the parser or compiler implementors. Some elements of the abstract syntax, however, can be restricted by means of semantics.

2.1.4 Parsing

When a parser reads an input file with source code, its task is not to get all possible derivations for the given grammar. Rather, given the input text and grammar, we need to check if the text follows the grammar, or in other words, if it could have been generated by the grammar. In other words, we have to reconstruct the tree provided on Figure 2.1.

	Top-down	Bottom-up
Non-directional	Unger parser	CYK parser
Directional	Predict/Match automation Depth-first search (with backtrack) Breadth-first search Recursive descent parser Definite Clause grammars	Shift/Reduce automation Depth-first search (with backtrack) Breadth-first search
Linear directional (breadth-first with max 1 breadth)	$LL(k)$	precedence parser bounded-context parser $LR(k)$, $LALR(1)$, $SLR(1)$

Table 2.2: An overview of parsing techniques, from Grune and Jacobs [2011], adapted.

This process is called parsing or syntax analysis.

In practice, parsing is almost exclusively performed with context-free (Type 2) or regular (Type 3) grammars [Aho et al., 1986]. Unrestricted (Type 0) and context-sensitive (Type 1) grammars are user-unfriendly because it is extremely difficult to derive a simple and understandable Type 0 or Type 1 grammar; besides, all known parsers for them have exponential time requirements [Grune and Jacobs, 2011].

Concerning the regular grammars, they are used mainly to describe patterns that have to be found in surrounding text (regular expressions) and can be parsed using a finite-state automates. But in real-world applications regular grammars are often seen as a further restricted context-free grammars and the toolset used for parsing these grammars is directly applied for parsing of regular grammars – sometimes probably not computationally effective. Depending upon the starting point of tree reconstruction, all parsing techniques can be classified by starting at the tree root (top) and going towards the leaves (down), or starting at the leaves (bottom) and ascending up to the root (up). These techniques are called „top-down” and „bottom-up” parsing respectively. These are two principally different techniques with different approaches to parser implementation, each having certain benefits and limitations (see Table 2.2).

After the classification „top-down” or „bottom-up”, the next important criterion for parsing method classification is the directionality: parsers can be directional or non-directional. Non-directional parsers enjoy the random access to the complete input at any time while constructing a parse tree: the complete input has to be known by the time when parsing begins and should remain accessible (that usually means – directly in the operational memory) during complete parsing time.

Non-directional top-down parsing is easy and straightforward, it was first described by Unger [Unger, 1968, Aho et al., 1986]. For the bottom-up approach a non-directional parsing was proposed independently by several people, so that the algorithm was named after its three inventors CYK-parsing. As it is also usual for the bottom-up approaches, the naive implementation of this algorithm is much more efficient than that of Unger [Aho et al., 1986].

The directional methods, conversely, process the input symbol by symbol, from left to right or from right to left (if this is more reasonable), uni-directionally. The main advantage here is that parsing can start as long as the first symbol is at the input and progresses considerably before the last symbol is reached. The comparison between directional and

non-directional approaches is especially illustrative in two classical XML (and HTML) parsers: DOM is a non-directional parser with full document representation whereas SAX is a extremely quick and memory-saving unidirectional parser (for details see Harold [2003]). All directional algorithms can be further classified by the technique used to drive the parsing automation to find all possible derivations. As we discussed in the previous section, parsing the text input is algorithmically a process of creating a tree. Therefore, the algorithms for parsing automations are those used for tree search and traversal: depth-first and breadth-first approaches. These two different techniques, determining the order how the tree nodes are processed, must be combined with backtracking for the means of computational efficacy. One very simple and computationally effective parsing method is called recursive descent parsing and is an example of top-down parsing. This method is especially suitable for manual parser implementation and is often used for this purpose. It works for a class of grammars called $LL(1)$: those that can be parsed by reading the input symbols from the Left, making derivations always from the Leftmost non-terminal, and using a look-ahead of one input symbol for rule matching. $LL(k)$ is the only known linear top-down parser [Grune and Jacobs, 2011], but sometimes it can be also called $RR(k)$ if applied backwards to the input.

One of the problems impeding the development of $LL(1)$ parsers is that the grammar definition is not allowed to contain recursive rules. The recursion can be either left-handed (left recursion), which has the form

$$E = E \dots | T \quad .$$

or right-handed,

$$E = T E \dots | T \quad .$$

where E is a non-terminal, T is a terminal or non-terminal rule with $E \neq T$, and ellipsis stands for any other symbols in the rule. For the recursive definitions the parser cannot choose between the alternatives for E by looking at any bounded number of symbols from the input start.

Several approaches exist to overcome the problem of using recursion in production rules. The most naive one uses the backtracking while matching the grammar rules against the input. In this approach the rules are applied recursively until the successful consumption of the input or until the rule application fails. In case of failure the parser backtracks to the last branching in the rule application and tries to match another branch against the input. This leads to (potentially) undetermined look-ahead of the parser, dramatically impairing its performance.

Another solution is the factorization of the recursive part of the rule. In its general form the factorization is the transformation of the recursive grammar rule. Given in the form:

$$A = A g_1 | \dots | A g_m | f_1 | \dots | f_n \quad .$$

where g_i and f_i stand for sequences of grammar symbols (possibly Λ), $m, n \geq 1$ and no f_j can derive a string beginning with A , so the only left recursion is possible through the first m alternatives. After factorization, this rule is transformed into the following two rules [Aho et al., 1986]:

$$\hat{A} = f_1 A_{opt} | \dots | f_n A_{opt} \quad .$$

$$A_{opt} = g_1 A_{opt} | \dots | g_m A_{opt} | \Lambda \quad .$$

There is a series of linear bottom-up methods, which are generally more powerful than $LL(k)$. One of the typical examples here is the $LR(1)$ parser, which works bottom-up, reading the input symbols from the *Left*, making derivations always from the *Rightmost* non-terminal, and using a look-ahead of one input symbol. The drawback of this approach compared to LL is that the grammars are more difficult to understand and the implementation of a parser is less convenient. Therefore, construction of bottom-up parsers is seldom done by hand. A useful subclass of $LR(1)$ is the class $LALR(1)$ (for „look-ahead LR “), which can be parsed more efficiently, by smaller parsers and can be automatically generated.

The great difference in the number of top-down and bottom-up approaches is easily understood when we examine the choices the corresponding parsers have to face. A top-down parser by its nature has few choices: in case of a terminal symbol it has no choice at all and can only assert that a match is present (or report a parsing error); only if a non-terminal is predicted it has a choice in the production of that non-terminal. Contrarily, a bottom-up parser can always shift the next input symbol, even if a reduction is also possible (and it often has to do so). If, in addition, a reduction is possible, it may have a choice between a number of right-hand sides. In general it has more choice than a top-down parser and more powerful methods are needed to make it deterministic [Grune and Jacobs, 2011].

For the approaches discussed above there are parser generator systems available for $LL(1)$ and $LALR(1)$, as both commercial products and open-source software in the public domain (see next section for more details). Using a parser generator to generate the source code for a parser is always more practical and efficient than writing a parser manually.

In this respect a frequent choice is that between (strong) $LL(1)$ and $LALR(1)$. Both parsers are roughly equal at their performance and memory requirements, so that a resource-saving implementation of either is possible. The main differences between them are the following:

- $LL(1)$ parsers are often easier to read, to understand by a non-professionals (for example project managers, clients), and to modify.
- $LL(1)$ generally requires larger modifications to the grammar than $LALR(1)$ to implement an effective parser.
- $LL(1)$ is capable to provide more user-friendly parser error messages than $LALR(1)$ thanks to the *Follow*(k) set holding the expected alternatives. $LALR(1)$ parsers just report the failure to parse the input.
- $LL(1)$ parser can be directly implemented as a recursive-descent parser, so that the semantic actions can have named variables and attributes, much like in a programming language. $LALR(1)$, as a bottom-up parser, is a table-driven parser and does not provide this possibility.

Choosing a parsing algorithm can be very subjective: for some the requirements made by $LL(1)$ are totally unacceptable, while others consider them a minor inconvenience, largely outnumbered by the advantages of the approach. If one has to design the grammar along with the parser, $LL(1)$ is almost always preferred:

- it is easier and more performant to parse and to perform semantic actions, and
- the well-defined text that conforms to $LL(k)$ grammar is also clearer for a human reader [Aho et al., 1986].

This explains the fact why we chose $LL(k)$ for our reverse-engineering approach, both for the base-line grammar and for the final parser implementation.

2.1.5 Parser combinators

Even though $LL(1)$ parsers can be implemented entirely manually without use of any third-party library, the use of parser combinator libraries allow a very succinct implementations without any impact on the parser performance.

Parser combinator are higher-order functions in functional languages that combine several parsers into one. For example, if a production rule has several alternatives, each of those may consist of a sequence of non-terminals and terminals and a simple parser is available for each of these alternatives, a parser combinator can be used to combine each of these parsers, returning a new parser that can recognize any or all of the alternatives.

Parser combinators employ a recursive descent parsing strategy, so it becomes easier to construct, debug, and text complex parsers. Parsers built using combinators are straightforward to construct, easily readable, modular, well-structured, and easily maintainable; this is why they are used in the prototyping of compilers and processors for domain-specific languages.

Parser combinators are often implemented to look as an infix operator, which is used to combine different parsers to form a complete rule. Thus, parser combinator libraries enable parsers to be defined in an embedded style, using the code that is similar to the rules of the grammar. This implementations can be seen of as a form of executable specification with all of the advantages such a specification brings.

Parser combinators, like all recursive descent parsers, are not limited to context-free grammars and can be used for construction of generalized parsers. Naive implementations of parser combinators might have some shortcomings common for top-down parsing (see previous section). Naive combinatory parsing requires exponential time and space when parsing an ambiguous context-free grammar.

2.1.6 Semantics

There are several classifications of semantics. Depending upon the purpose of the semantics, it can be either used for compiler-time checks of the program (static semantics), to model the run-time behavior (dynamic semantics), or to describe the equivalence relation between the abstract code representation and the outcome [Mosses, 2006].

A compiler check for well-formedness of the program code is similar to the parser check if the source code is well-defined. *Static semantics* are used as formal descriptions for these checks performed before the compiler starts to translate the program into the executable code. They serve the goal to detect potential problems and avoid them. The outcome of these checks is binary and states whether the code passes or fails them. Compiler warnings generated as a by-product of these checks for the rules that do not block the code compilation might be useful for developers.

Dynamic semantics are used to check (and constrain) the program execution and define the well-formed behavior of these programs. The purpose of these semantics is either to ensure certain security aspects of program execution (sandbox environment, isolated storage) or to optimize the observable behavior (for CPU or memory consumptions etc.).

Equivalence bridges the gap between these two types of semantics by providing an abstract model for the relevant features valid for all possible executions of the program. The equivalence can be used to test for program clones: if two different programs have same models, they can be considered clones, even if they differ significantly at the source-code level (using different identifier names, slightly different order that is not important for semantics etc.)

A complete description of a language semantics should include all three types of semantics: all three are required to successfully execute the program.

Depending upon the language properties there are several competing and complimentary approaches to formally describe semantic rules [Schmidt, 2003]:

Operational semantics describes the computational steps needed to process the program's input. Another term used for this formalism is *intensional semantics*, because the sequence of computation steps (the „intension“) is rigorously defined. One of the common approaches here is to define a *term rewriting system*: a set of rules that describes transformations of the source code from input to output. Another set of rules, called *inference rules*, define the context in which a certain term rewriting rule can be applied. Two different programs to check if a number is prime will have a different operational semantics.

Depending upon the size of these steps, operational semantics can be further sub-classified into two approaches: *structural operational semantics* (also known as *small-step semantics*) formally describes every individual step of a computation. Conversely, *natural semantics* (*big-step semantics*) describe steps schematically, focusing on the overall results of the program execution.

Operational semantics is used to publicly expose implementation concepts: heap, stack, storage vectors, registers of CPU and their state. Due to the easiness of its implementation and hardware-near traits this formalism is perfect for describing the semantics of purely imperative and especially stack-based languages. These are many intermediate languages executed by virtual machines (Common Intermediate Language, CIL executed by Common Language Runtime, CLR; Java bytecode run by Java Virtual Machine, JVM etc.) or even independent full-fledged languages (with prominent examples like PostScript, Forth and Cat).

A disadvantage of a operation semantics (especially of the structural semantics) is that the sequence of operations or single re-writing steps have to be explicitly specified even in cases where this is not necessary. This does not mean that this type of semantic is not capable of defining the non-deterministic formalisms: being very close to hardware implementation, this semantic can describe the non-determinism of concurrent operations. But describing this semantics one has to be explicit about the operation order. This is, of course, a very precise and might be necessary for general operations on expressions, but if an expression has several admissible orders (e.g., since it is commutative or non-deterministic), that is inconvenient..

Denotational semantics addresses these drawbacks by emphasizing the meaning the given program has and keeping the necessary implementation steps independent from this meaning. This is only possible by using a *compositional* approach when the meaning of a complex sentence is reconstructed from the meaning of its smaller parts. For this property, to keep the meaning and the implementation apart, this semantics allows simple and elegant definitions of mathematical problems and is also known as „mathematical semantics“ [Scott, 1977].

Denotational semantic defines the meaning of a phrase as the overall meaning of its sub-phrases. Therefore, in order to prove if semantics fulfills certain criteria one can use structural induction. This means, to prove that a property P holds for all programs in the language one must show that the meaning of each construction in the language has the property P . Therefore, one must show that each semantic clause produces a meaning with property P .

Here is an example of the denotational semantics for arithmetic from Schmidt [2003] (for detailed explanation of the syntax for this semantics, refer to the next section):

$$Nat = \mathbb{N}_0, N \in Nat$$

$$\varepsilon : Expression \rightarrow Nat$$

$$\varepsilon [N] = N$$

$$plus : Nat \times Nat \rightarrow Nat$$

$$\varepsilon [E_1 + E_2] = plus(\varepsilon[E_1], \varepsilon[E_2])$$

In this semantics we first define the domain Nat that we will use in our calculations as a set of natural numbers including zero. N is any number from this set. ε is defined as operation that takes an expression and returns its computed value. We define also an operation *plus* that is a binary operation requiring two parameters of type Nat and returning one value of this type. For an expression containing a plus sign we define the meaning of this expression as evaluation of both expressions, adding them using the *plus* function is not explicitly defined here.

Denotational semantics has been often used in teaching of semantics and in research. It was also used to completely define the semantics of the programming language Scheme [Abelson et al., 1998]. Attempts to derive denotational semantics for other programming languages were not so successful [Mosses, 2006].

An *axiomatic semantics* is used to describe properties of programs rather than meanings. In this respect, using axiomatic semantics every program can be (statically) checked not that it does not violate the defined properties (axioms) before the program is executed. An example of axiomatic semantics are code contracts in .Net, which allow one to define certain pre- and post-conditions as well as invariant conditions for every method call that is checked statically or at runtime.

From these types of semantics we decided to use denotational semantics to describe the properties of TouchDevelop. This choice is motivated by the fact that we want to define the meaning of programs in TouchDevelop and the only available options are operational or denotational semantics. Operational semantics requires the exact knowledge of how a compiler works and how program code is transformed to obtain the result. It is not possible to obtain this information by means of the reverse-engineering approach we used (without hacking and disassembling the TouchDevelop phone application). Therefore it is not possible to reverse-engineer the operational semantics.

On the other hand, our approach to reverse engineering by treating the TouchDevelop mobile application as a black box with code input and output in form of the code execution matches the essence captured by denotational semantics: to provide output for known input. This is especially helpful also for the goal of implementing a multi-target compiler, for the concrete implementation on different platform might differ due to external limitations.

2.2 Formal notation of program languages syntax and semantics

As the main goal of this thesis was to derive a language specification for TouchDevelop, the knowledge about different notations for the required language formalisms is essential for this task.

Symbol	Definition	BNF equivalent
:	is defined as $a(n)$	$::=$
;	, or as $a(n)$	
,	followed by $a(n)$	(space)
.	, and as nothing else.	(new line)

Table 2.3: The notation by van Wijngaarden.

2.2.1 Notations for language syntax

A grammar as a structural description of a software system can be represented in many different forms (Klint et al. [2005]):

- Backus-Naur Form (BNF, Extended BNF), which can be visualized in form of railroad diagrams, widely used in this document to describe the grammar of TouchDevelop,
- van Wijngaarden form,
- algebraic data types (=discriminated unions) in functional programming languages (like ML or Haskell) to represent the abstract syntax tree,
- XML Schema Definition (XSD) and Document Type Definitions (DTD),
- Unified Modelling Language (UML) diagrams or other graph-based languages graphically represented as diagrams,
- Syntax Definition Formalisms (SDF), allowing the description of concrete syntax to be divided into modules with explicit dependence, so that a module specifying low-level constructs can be reused in the syntax definition of several languages [Mosses, 2006],
- Abstract Syntax Description Language (ASDL).

From these types of representation the first two are common for formal definitions of grammars. Algebraic data types provide a way to specify the program code as a tree and are used in parser implementations to hold the parsed information in a structured form. The Backus-Naur Form (BNF) was first used for defining ALGOL 60. Here is a sample of this form for a binary literal:

```
<bindigit> ::= 0 | 1
<binsequence> ::= <bindigit> | <binsequence> <bindigit>
<number> ::= 0b<binsequence>
```

Every non-terminal in the original BNF was enclosed in angle brackets and the production arrow was denoted by a special sign ($::=$). Today it is more common to use quotes (single or double) for non-terminals, emphasizing the fact that they are strings.

There are many variants or dialects of this form: an extended BNF (EBNF) was developed by Niklaus Wirth while working on the Pascal specification. This notation supports not only definition and alternation, as basic BNF, but also optional parts in definitions, repetitions, groupings and comments. Today this is a standard for grammar definitions.

Another popular notation is that of van Wijngaarden. Here is an example of the same definition for binary literal in this notation:

```
bindigit: 0 symbol; 1 symbol.
binsequence: bindigit; binsequence, bindigit.
number: 0 symbol, b symbol, binsequence.
```

Every terminal symbol is explicitly denoted by the „symbol” keyword after it. Punctuation is used similarly to its use in natural languages: every rule is terminated with a period, the comma binds tighter than the semicolon (see Table 2.3). The punctuation can be read as text.

Some grammar notations are directly coupled to a specific formalism. For instance, BNF is designed to represent context-free grammars. Other grammar notations, like EBNF, an improved version of BNF, might be more convenient, but not more expressive in the formal sense of the language.

2.2.2 Notations for semantics

Different notations of semantics are very close to notations used in mathematics that is explained by the nature of these formalisms.

Operational semantics traditionally [Plotkin, 1981] uses rules to provide inductive specification for transition relations on states. The rules include the nodes from the abstract syntax tree and the new computed values (the program outcome). Here is an example of the rule for an expression with exception types in the JavaS language from Drossopoulou and Eisenbach [1998]:

$$\frac{\begin{array}{l} \Gamma \vdash e : E, \quad e \neq \iota_1 \\ \Gamma \vdash E \sqsubseteq \text{Exception} \end{array}}{\Gamma \vdash \text{throw } e : \text{void}} \\ C\{\Gamma, \text{throw } e\} = \text{throw } C\{\Gamma, e\}$$

The rule consists of two parts separated by a horizontal line: the upper part describes the condition that has to be met for the rule, the lower part contains the result of the rule application.

The *transition rules* feature the concepts of *bindings* and *stores* [Mosses, 2006].

Bindings are the conventions that bind identifiers to particular values. A binding map shows the actual association between the identifier and its value in the current scope.

Stores are containers for entities used to abstract the computer memory. An assignment statement is treated as a command to change certain locations in the variable storage. Because binding is independent from assignment, the same location can have several bindings that are defined as aliases in the given language.

In the cases where the semantics of concurrent processes has to be expressed labels on transitions (termed communications) are used to represent the relations between these processes.

Operational semantics can be also seen as term-rewriting systems. Another notation, called reduction semantics, was developed to describe the rules of this semantics [Felleisen and Friedman, 1986]. In this notation the states are purely syntactic and have no mathematical meanings. For example numbers are treated symbolically as decimal numerals and not as numbers in mathematics. A rewriting step is called a *reduction*. Reduction does not necessarily mean that the resulting term is shorter or simpler. The rewritten term is termed as *redex* and the result of the transformation as *reduct*. Reduction is applied literally according to the rules and might be infinite (non-terminating). Some programming languages, like Algol60, were also proposed as formal notations for operational semantics. Concerning denotational semantics, as it follows from its name, this formalism is comprised by semantic rules called denotations.

Denotation is a function that links the information available before its application and the result that represents the information available afterwards. The intermediate steps and states, necessary only for the execution of this function, (usually) deemed non-relevant and are not specified. Denotations are defined inductively.

Semantic functions bind a semantic construct to its denotation. In Section 2.1.6 we defined a semantic function ε for expressions. The definition uses the λ -notation to specify the

function signature, i.e., the arguments and the returning value. The addition expression was then defined inductively as an operation *plus* on two sub-expressions.

Denotational semantics is very powerful at the definition of expressions that are similar to mathematical constructs. For control-flow structures, like loops or jumps, one must be able to provide a well-defined solution $d = F(d)$, where $F(d)$ is a particular composition with denotations of the loop conditions and its body. The solution of this equation is the least fix-point of the continuous function in a Scott-domain (complete partially-ordered set) Mosses [2006].

To achieve a better modularity of the λ -notation one can use auxiliary notation to combine the existing denotations into a new one in a most generic way. One of these notations is monadic semantic. If we need to sequence two different annotations d_1 and d_2 both computing the value of the same particular type, and if the value computed by the first expression must be available to the second one, this is noted as

$$\text{let } x = d_1 \text{ in } d_2 .$$

A set of such denotations represents the mathematical structure called monad. The monadic expressions are often used in the formal definitions of semantics (and syntax) because of its good readability.

Axiomatic semantic is entirely based upon certain branches of mathematical logic. One of the most common logics for axiomatic semantics is the Hoare logic where the rules are represented as tuples of assertions about variable values before and after the execution of every construct. This semantic is usually applied to the statements, denoted as S . The set of assertions that have to be fulfilled before (termed pre-condition, P) and after (post-condition, Q) are combined into a so-called partial correctness formula $P \{S\} Q$. This formula states that if P holds before the statement execution of S and the execution terminates, Q will always hold after the termination.

From the described formal notations, we will use the Extended BNF form for grammar definition, because it is directly supported by the parser generation tool ANTLR. But in order to keep the language specification understandable even for casual developers we will use the railroad diagram representation of EBNF rules in the printed version of specification.

Besides, the monadic syntax is used for the definition of the parser. The semantics is customly implemented in the form of static semantic checker and is provided in an abstract notation form.

2.3 Reverse engineering of languages

There are neither comprehensive books, nor best practices or university courses for grammar development. A usual approach here is to develop a language parser that can safely parse the existing codebase. In this respect, the grammar knowledge existing either explicitly in the form of a language specification or as set of ideas is directly implemented as a proprietary parser, as depicted on the left-hand side of Figure 2.3. A wiser approach is to first derive the base-line grammar in subsequent customization steps without targeting a concrete parser implementation (right-hand side). This grammar is used then to implement an effective parser in the next step. The obtained parsers here are not only tested against the existing code base, but can also be stress-tested to compare the performance of different parser implementations.

There is a number of approaches that can help one to turn grammar hacking into grammar engineering. Some of them are shown on the right-hand side of Figure 2.3:

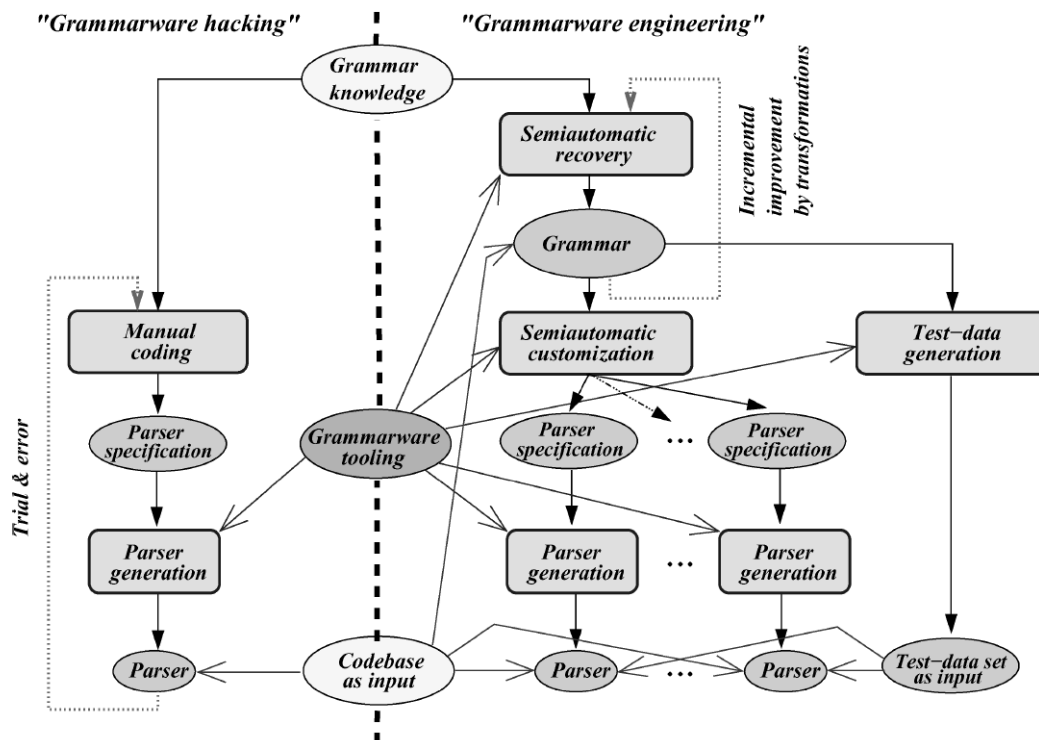


Figure 2.3: Grammar hacking vs. grammar engineering, from Klint et al. [2005].

- If the language specification or any other formal description of the language is available, there is a possibility of semiautomatic recovery of the grammar. In this approach it is possible to incrementally increase the quality of the grammar by adding new transformations that cover omissions and correct the improper rules.
- The yet-to-be-derived grammar can be executed by a prototype parsing framework. At this stage, the quality of the parse tree and the generated syntax tree are not important and the execution serves the goal to merely test the overall validity of the grammar (proof of concept). The existing codebase can be a driving factor for improvements and corrections.
- Parser specification is derived automatically as a part of the grammar using appropriate tools.
- To test the derived parser specification one can test the generated parser against the existing codebase. One can use the metrics known from Software Engineering, like code coverage, to reason about the maturity of the parser.

Reverse engineering of languages is usually confined to the reconstruction of language syntax and their implementation a language parser as a proof of concept for this reconstruction [Lämmel and Verhoef, 2001]. The reconstruction of semantics is only performed for already existing language specification with the aim to make it complete or to find relationships with other languages [Lämmel and Zaytsev, 2011]. The reasons for this confinement are the following [Klint et al., 2005]:

- Lack of best practices,
- Lack of comprehensive foundations,
- Lack of books on grammarware,
- Lack of coverage in curricula.

2.3.1 Principles and life cycles of grammar engineering

The following principles, systematized by Klint et al. [2005], represent the approaches adopted from contemporary common sense in software engineering for reconstruction of language specifications, even if contemporary grammarware development does not adhere to these principles.

1. *Derive the base-line grammars first.* In the attempt to reconstruct the grammar, an early commitment to a concrete use case, specific technology, or other implementational choices should be avoided. The development should start from simple, human-readable grammars: more or less plain structural descriptions using a fundamental notation, preferably paper-based. The first derivations, termed base-line grammars, should be sufficiently structured and annotated to be useful in the potential derivation of concrete syntaxes. If necessary, these grammars can be extended by auxiliary semantics for object models, parse trees, etc.
2. *Derive the grammar use cases and optimize your base-line grammar for them.* Aho et al. [1986] and in particular Grune and Jacobs [2011] describe advanced transformations for the removal of left-recursion in a context-free grammar. These transformations are an important preparatory step for further implementations of a parser using the recursive descent algorithm (see Section 2.1.4).
3. *Divide the specification into smaller parts and re-engineer them separately.* As in programming, the principle of separation of concerns is applied to facilitate reuse and modularity. Possible smaller categories include the basic syntax, the comments and common layout (indentation) for syntax and error handling rules, scopes, and detailed rules for different classes of language constructs (variables, expressions, declarations) for semantic.
4. *Refer to other known grammars, implementations, and specifications.* Many principles are universal and are already realized in other available implementations and documents. Trying to look at the existing grammarware for similar languages (same language family, same or similar paradigm) might be of great help for the deriving of an eventually complete and robust specification.
5. *Repetitively test and assess your grammar.* Besides the code base as a reference point for parser tests, there are several other metrics to assess the grammar: human readability, the general style, special metrics for correctness and completeness, the depth of required look-ahead, backtracking and so on. Formal methods, used for software validations, are in most cases also applicable on grammars and specifications.

The discussed principles are represented on Figure 2.4 as a grammarware life cycle. The most important point here is the separation of the base-line grammar as an intermediate step before working on parser specification. Base-line grammars do not directly commit to the implementation. Instead, they provide solid foundations, which can be customized into other, more specific types of grammarware.

The process of reverse-engineering has several levels of iteration. Several repetitive iteration on one certain issue help to achieve the implementation optimized for performance and memory consumption. The overall feedback and another „global” iteration contributes to the precision and completeness of the derived grammars.

We did not include a separate step with semantics derivation into the discussed workflow. This is because there is no any available information in the form of scientific methodology or best practice how to perform this step. For our specification we relied on the semantics of the .Net framework and Silverlight runtime and

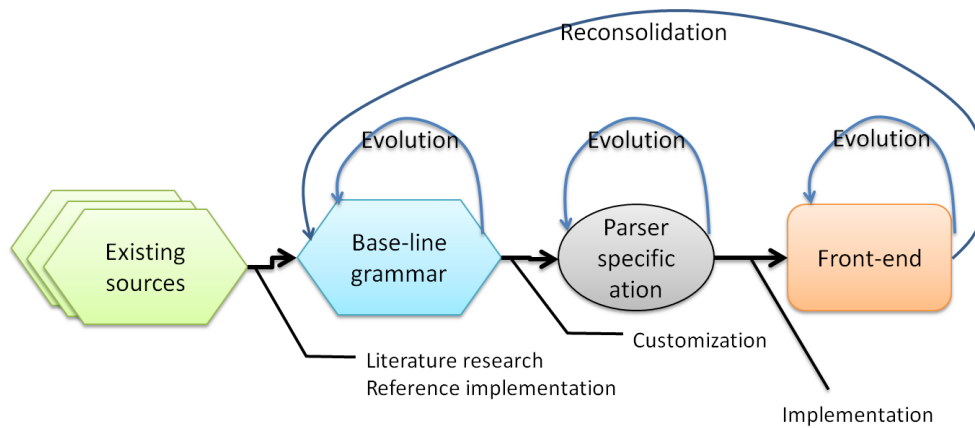


Figure 2.4: The life-cycle of language reverse engineering.

The general workflow for language specification mining, as depicted on Figure 2.4, is the following:

- compilation of the base-line grammar for the object of reverse engineering,
- deriving the grammar use cases (parser, compiler etc.),
- implementation of the grammar in the format dependent upon the selected use case,
- quality assurance and testing of the grammar-dependent software,
- reconsolidation of the derived base-line grammar and return to the begin of the iteration.

In the current work we tried a novel approach to use ANTLR Studio for rapid prototyping of the base-line grammar as the first step in reverse engineering of a language specification. With the support of the tools from ANTLR Studio we were able to derive a human-readable grammar, to debug it and test against the source code and to detect the offending rules preventing us from a computationally effective implementation of the parser. We also used the rich visualizing features of ANTLR studio to prepare the railroad diagrams for this manuscript.

3

Chapter 3

Language Specification of TouchDevelop

The lack of a formal language specification motivated us to start the reverse engineering of the language specification of the TouchDevelop application for Windows Phone OS. This work is based mostly on the original MS Research publication (Tillmann et al. [2011]). At the places where the recently published user manual provides updated or slightly different information we include notes with explanations.

The TouchDevelop project is developing very rapidly with a new minor version every 3-4 months. The changes between different minor versions include changes both in the standard library and in the language. Therefore, we had to constrain our investigations to the sample source code corpus. For this purpose we downloaded 282 sample scripts in December 2011. Our specification therefore does not include any later language features, in particular not those that became available together with the publication of the user manual [Horspool et al., March, 2012]. These changes mostly concern the support for external libraries which we did not exist in the application version we used for mining. Some minor language improvements (like multiple **where** clauses) are mentioned in the note sections. This chapter presents the derived language specification for the TouchDevelop. In the first section we briefly discuss the peculiarities how the best practices, described in the previous chapter, can be applied to the TouchDevelop specification. Since the code editor, language itself and the runtime that are deeply interwoven in TouchDevelop, we provide a short description of the user interface features that are essential for the further description of the language and runtime.

The rest of the chapter, except for the last section, deals with the different aspects of the TouchDevelop language specification, from notation to script execution. The last section describes in a nutshell the cloud services of TouchDevelop, which we see as an important part of the TouchDevelop infrastructure.

3.1 Reverse engineering of the language specification

With the ultimate goal of implementing a multi-target compiler for TouchDevelop we performed the reverse engineering of the language syntax and semantics.

The problems we were facing during our work can be categorized into the following sections:

1. The work on the language parser was impeded by the absence of the formal description of the language grammar. The technical report published internally by Microsoft Research describes TouchDevelop predominantly from the user experience perspective.

The language and the runtime environment are presented in a very schematic form with large omissions, which necessitated the mining for a formal specification. The derived syntax rules were implemented both in the form of an ANTLR grammar and as a custom parser in F# using the FParsec library. You can find both implementations in the Appendix section.

2. The reverse engineering of the compiler infrastructure was performed by analyzing the runtime environment and interpreter because a compiler implementation does not exist. An interpreter normally exhibits the dynamic semantic different from that of a compiler, which might require a more accurate and thorough specification and implementation.
3. The only possibility to author, test for syntactic and semantic validity, and run TouchDevelop scripts as of today is to use the TouchDevelop application on a Windows Phone 7.5 (Mango) device. All code snippets used for probing specification details had to be typed using the finger-controlled touchscreen manually. There is no possibility to programmatically generate any script code to (partially) automate specification mining.
4. The only way to obtain TouchDevelop scripts in a plain text format currently is to use a special web API for researchers that allows downloading of scripts from the cloud storage. But we first have to synchronize the phone scripts with the cloud storage. This operation required several manual steps for every script change to get the source code of the authored changes.
5. The use of tokenized input and the mobile application as the only way to author scripts for TouchDevelop account for the fact that there is no standard text representation of the source code. Different „views” on the source code available to the user or developer show considerable differences. We discuss this problem in more detail in the next section.

3.1.1 Reverse engineering of the language syntax

The source code in TouchDevelop is authored and completely edited on mobile devices using the code editor. Phone applications run in a sandboxed environment and hence there is no way to access the temporarily stored scripts from other applications on the phone. However, due to the permanent connection to the cloud services and the complete synchronization of local scripts it is possible to look at the textual representation of the source code using the web interface of the cloud portal or by employing the OData-service. The latter allows for storing the source code in a plain text form.

We took the text-only version as basis for syntax derivations that also included the reverse engineering of the serialization algorithm for TouchDevelop scripts. To accomplish this task we implemented a small console utility for bulk download of TouchDevelop scripts via the API available for researches (at the time of writing these lines this API does not require any authorization for bulk downloads). Using this utility we downloaded 282 sample scripts that were available via TouchDevelop API on December 9th, 2011. This source code corpus was then used to derive the language syntax and the serialization algorithm for Unicode identifiers and for conflicting names.

The Appendix contains the complete list of scripts we used for discovering the language syntax. Aiming at verification and testing of the derived syntax rules we first implemented the grammar for TouchDevelop using the ANTLR parser generator package (Parr and Quong [1994]). The grammar was checked against a selected number of scripts manually and used further to generate the railroad diagrams for this specification. With this grammar as

a basis we implemented a custom language parser in F# using FParsec's parser combinators library and tested the grammar against the above mentioned source code corpus. The reported parsing problems in four scripts turned out to be the syntax errors in the scripts (missed commas, misspelled or erroneously serialized identifier names).

3.1.2 Mining language semantics

Mining of language semantics requires the execution of small test scripts in the TouchDevelop environment.

Our initial approach to download the TouchDevelop package from Windows Phone AppMarket and to install it on the emulator was not successful: the AppMarket does not officially provide any option to download software packages. The free software tool we found to fulfill this task is able to download only the very first version of the application and the manual re-signing of the application to deploy it into the emulation environment does not function reliably. We used the Phone application v2.4 running on a Samsung Focus device to perform all tests we needed to successfully reverse engineer the semantics of the language.

To perform the reverse engineering we developed a sample TouchDevelop script. This script was used to derive the semantics of some complex TouchDevelop behavior, including the paradigms for parameter passing and conformity with IEEE 754-2008 float point arithmetics (see page 96).

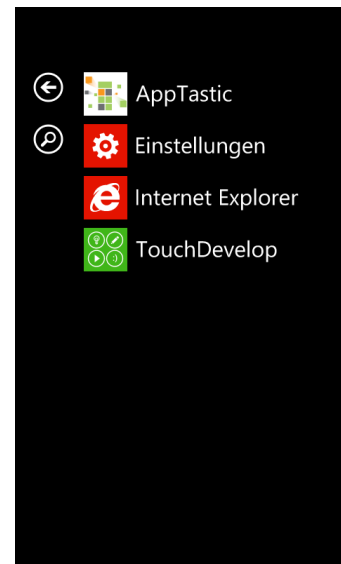


Figure 3.1: The TouchDevelop as it is seen in the list of installed applications.

3.2 Script authoring

As mentioned previously, the only possibility to author user scripts for TouchDevelop is to use the Windows Phone applications that features a script manager, and code editor for creating, editing and sharing scripts by users.

3.2.1 Script manager

The script manager is the core administration tool that enables users to perform a large variety of operations with scripts: creating new scripts, synchronizing the locally edited scripts with the cloud, publishing own scripts (=making them publicly available), as well as the browsing, reviewing and downloading of scripts implemented by other users. Similar to the Game Center functionality in iOS, the script manager also provides native support for application scoreboards and screenshot publishing, thereby promoting competitiveness and facilitating the development of community games.

The script manager can be seen as a mobile client for the web platform known as Bazaar (see Figure 3.2), which serves as a central repository for user scripts, maintains script

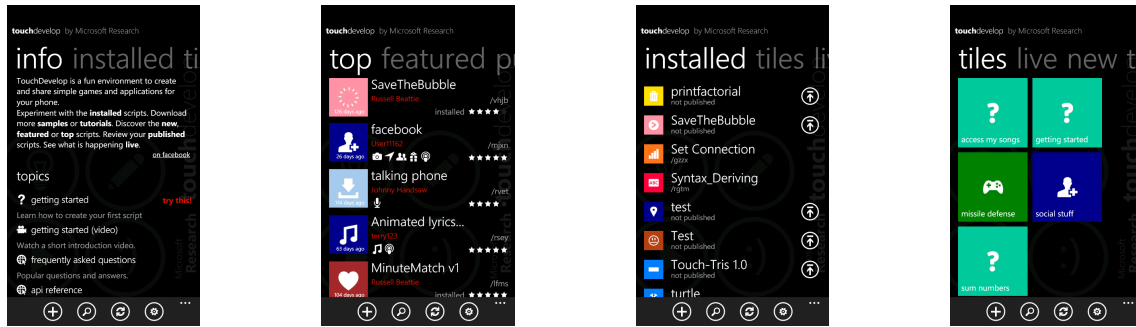


Figure 3.2: TouchDeveloper Script Manager as WP7 Panorama application.

branching, and provides support for the user community (forum for application discussion, leaderboards, evaluations etc.).

Even though the script manager can be seen as a pure browser application, it is implemented with the idea of eventual connectivity in mind: the local script storage is completely synchronized with the cloud in the background if the mobile device is online. This synchronization includes both user changes of the locally installed scripts and updates to installed scripts, published by their authors. The user changes, however, are not seen by other users until the script (or its newer version) is published (see Figure 3.2). Should the cloud not be accessible the complete list of changes is stored locally until reconnection.

3.2.2 Code editor

TouchDevelop as a language is built around the idea of using only a touchscreen as input device to author code. To fulfill this goal, the TouchDevelop IDE features no classical keyboard. Rather, the context-sensitive choice of keywords (tokens) is presented in the form of an on-screen phone keypad, allowing the selection of necessary keywords with your thumb (see Figure 3.3).

The tokenized keypad allows very quick and (syntax) error-free input of the program code but requires some training to get fully mastered. The most tedious part in the code input is the typing of string literals where one has to use the same reduced keyboard and type the letters using repetitive pressing of the keypad buttons, similar to the text input on mobile devices without T9 support.¹

The code editor has two different modes: source view mode and line edit mode. In the source view mode (Figure 3.3, left picture) the complete source code of the given action is shown. Internal TouchDevelop keywords (like `action`) and some augmented syntax information (for instance, the word `var` at the place of local variable declaration) are shown using the accent color of the current Windows Phone theme. This code can be scrolled using the swipe gestures. This mode also shows errors in the code.

Indentation is used to optically mark code blocks. Every new statement starts with a new line. Long statements are wrapped over several lines with additional indentation for every wrapped line.

Tapping any line of code will bring the line selection dialogue (Figure 3.3, middle picture), from here the user can:

1. Extend the selection to several lines to perform any block-based operations using the clipboard.

¹Though the phone standard keyboard is overlaid on top of the phone pad while editing string literals or identifier names in the latest version of the TouchDevelop phone application.

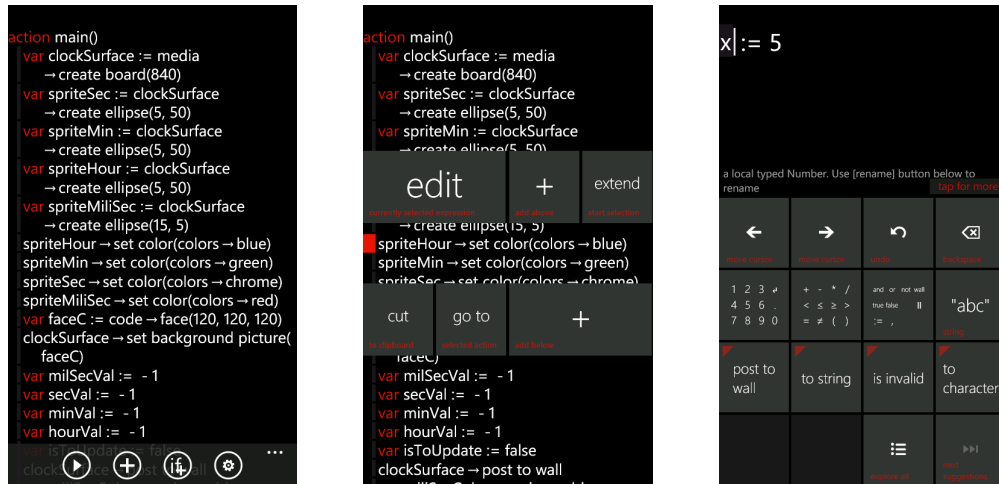


Figure 3.3: TouchDeveloper code editor.

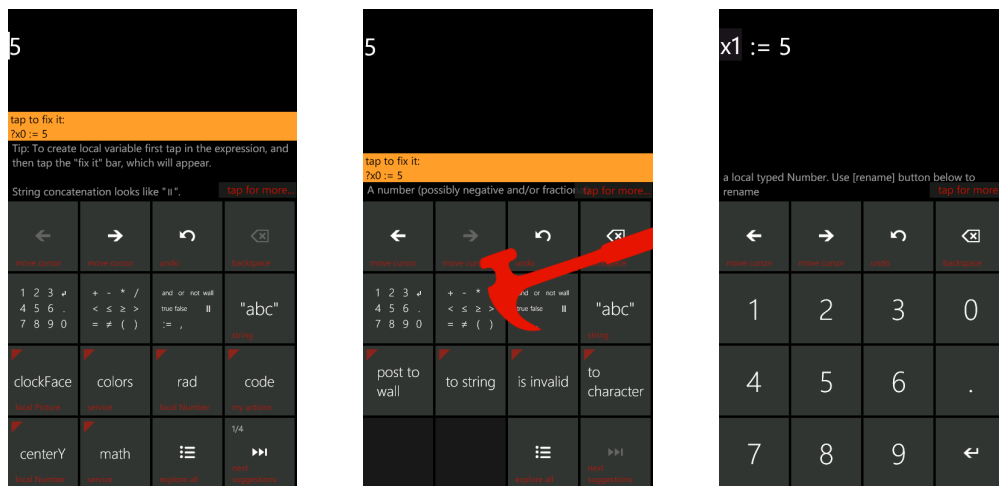


Figure 3.4: Using the „fix-it” feature of code editor to declare a new local variable.

2. Add a new code expression directly before or after this line.
3. Edit the current line in the line editor.

The selection of the last two actions brings us to the line editor mode. In this mode only the current line is visible, the rest of the screen is occupied by the keypad with tokenized actions (Figure 3.3, right picture). These action contain local and global variables, literals, and library functions. Some actions are grouped and the button icon features the miniature picture of the complete group keypad (for example numbers, arithmetic operations).

Code editor applications perform the check for code syntax and (to some extent) semantics „on-the-fly” and suggest quick fixes for the input code (see Figure 3.4). Quick fixes allow the user of the application to quickly author code by writing short fragments and then applying a quick fix to them. For example, the editor has no default support for the introduction of local variables. One way to declare a local variable is to type in some expression and apply a quick fix that translates it into an assignment statement automatically adding a new local variable (appending an appropriate index to its name, if necessary) to the left part of the assignment expression.

The tokenized input allows TouchDevelop to bypass the lexing step while authoring

Selection:

$$\begin{aligned} (_ \rightarrow _ \square _) &: Boolean \times Types \times Types \rightarrow Types \\ (true \rightarrow a \square b) &= a \\ (false \rightarrow a \square b) &= b \end{aligned}$$

Figure 3.5: Basic definitions used in language semantics.

scripts: adding new declarations, adding new statements etc. are performed by tapping an appropriate action on the keypad and eventually setting some properties. This input prevents TouchDevelop scripts from any mistakes at the global (script) and statement scopes, leaving the expressions (authored by the line editor) as the only source of syntax errors.

3.3 Notation

3.3.1 Syntax and semantics

We use railroad diagrams in the text to describe the syntax of TouchDevelop. Railroad diagrams graphically represent the extended Backus-Naur Form (EBNF) as railroads (transitions) and stations (symbols, both terminals and non-terminals).

The railroad diagrams were created from the TouchDevelop grammar implementation in ANTLR. The complete implementation of the grammar can be found in Appendix.

To describe the semantics of TouchDevelop we use predefined selection operation shown on Fig. 3.5.

The selection operation chooses one of the branches depending upon the value of the Boolean expression at the first place.

3.4 Source code representation

Application code is stored as a single text file with default Unicode (UTF-8 without BOM²) encoding. The language itself makes extensive use of Unicode characters: for example the arrow symbol used for referencing type properties and methods (\rightarrow) is represented as a Unicode symbol “rightwards arrow” (U+2192). Table 3.1 provides more details on Unicode symbols in TouchDevelop.

3.4.1 Three views on the source code

Originally targeting exclusively mobile devices, the TouchDevelop environment is available only as a Windows Phone application, which bundles both development and runtime environments. Adapted for touch devices and thumb input, the code editor of TouchDevelop provides the on-screen keypad with context-dependent language tokens. The tokenized input enables rapid and convenient script authoring reducing the need to type long texts. The script entered by the user can be viewed locally using the code editor and this view is the major one. Upon synchronization of the script code with the cloud two additional views

²The byte-order mark (BOM) is a Unicode character used to signal the endianness (byte order) of a text file or stream. Its code point is U+FEFF. BOM use is optional, and, if used, should appear at the start of the text stream. Beyond its specific use as a byte-order indicator, the BOM character may also indicate which of the several Unicode representations the text is encoded in (http://www.unicode.org/faq/utf_bom.html#BOM).

Operations	Symbol ^a	Unicode hexadecimal	Unicode description	Where used
Global variable reference	\boxminus /data	U+25F3	WHITE SQUARE WITH UPPER RIGHT QUADRANT	phone only
Assets reference	\clubsuit /data	U+273F	BLACK FLORETTE	phone only
Code reference	\triangleright /code	U+25B7	WHITE RIGHT-POINTING TRIANGLE	phone only
Library reference ^b	\clubsuit /lib	U+267B	BLACK UNIVERSAL RECYCLING SYMBOL	phone only
Property access	\rightarrow	U+2192	RIGHTWARDS ARROW	everywhere
Arithmetic less or equal	\leq	U+2264	LESS-THAN OR EQUAL TO	everywhere
Arithmetic greater or equal	\geq	U+2265	GREATER-THAN OR EQUAL TO	everywhere
Arithmetic not equal	\neq	U+2260	NOT EQUAL TO	everywhere
String concatenation	\parallel	U+2225	PARALLEL TO	everywhere

Table 3.1: Unicode symbols used in TouchDevelop application and scripts.

^aAlternative representations of Unicode symbols are given after the slash.^bOnly for information. Library references are not covered in this document. See Discussion for details.

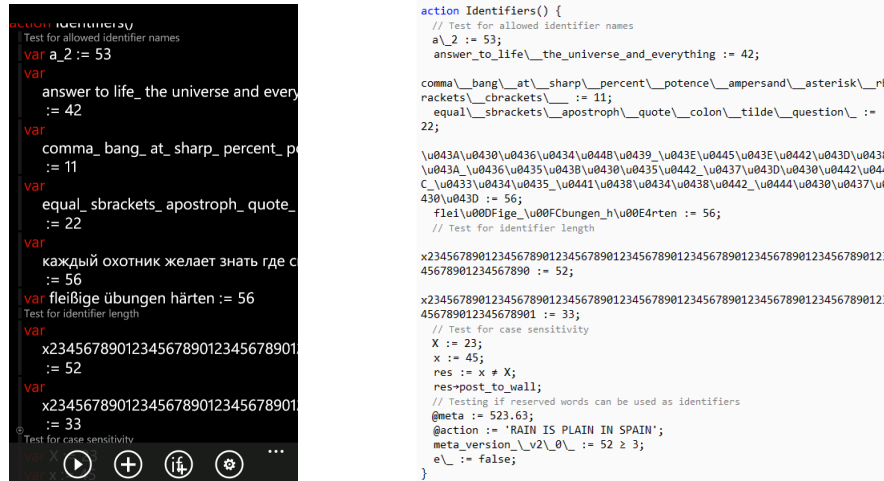


Figure 3.6: The usage of Unicode identifiers in TouchDevelop scripts.

Left: The sample script in TouchDevelop phone applications (phone view) featuring identifiers in Russian and German. Right: The same script as it is seen on the webpage (web view).

come into play: the web view using the main TouchDevelop website and the text-only view via the TouchDevelop RESTful API. In order to be able to reference different visualizations of the scripts we use the following notation:

- **Phone view** is the source code as it is seen directly on the mobile device in the TouchDevelop phone app. Despite substantial differences between different versions of the phone application, we consider it to be a single code representation (see Figure 3.6 left).
- **Web view** is the version of the source code as it is shown to the users who are logged in on the TouchDevelop main webpage (see Figure 3.6 right).
- **API view** is the plaintext-only version of the script, which is available via the RESTful cloud API. The returned text contains the complete script and meta information placed in meta declarations (see Section 3.7.4).

A major problem originates from the fact that these three views on the source code of a TouchDevelop script are not consistent and the differences between different views go far beyond differences in syntax highlighting. The phone application, for example, uses the `var` keyword to indicate variable declarations. The web view does not have this feature, instead, all the variables are shown with prepended dollar sign (\$). The text-only version, finally, obtained via the API for researchers, contained none of these, at least, not until recently.³

Since the only way to get a text-only representation of a TouchDevelop script’s source code is to use the API (the web-view is pre-rendered HTML with applied code highlighting and other visualization techniques), we used this last view for reverse engineering of the language syntax. For this reason there can be substantial differences between our specification and the specification recently published by the TouchDevelop creators (Horspool et al. [March,

³Microsoft Research adopted a new format for script serialization in API view on April 15th, 2012. The old format is available as well, but requires providing a special parameter to the webservice. Since we downloaded the scripts for TouchDevelop much earlier, we do not use this syntax for any variable definitions: the dollar sign is treated, however, as a normal character allowed at the beginning of a variable name.

2012]), for the latter tries to mimic the syntax as it is shown on the phone screen and considers all other types of views mere “serialized text forms” (Horspool et al. [March, 2012], page 109). We discuss these differences in details in Chapter 5.

3.5 Lexical structure

3.5.1 Identifiers

Due to the use of Unicode the valid name for any identifier used in the program can contain any Unicode character that can be typed on the Windows Phone⁴, covering all Latin- and Cyrillic-based European languages (see Fig. 3.6). Every character that can satisfy the `System.Char.IsLetterOrDigit()` predicate in .Net can be used here. Additionally, space and underscore are both considered valid symbols. The attempt to use any other symbols, like quotes, bang, asterisk, bracket, ampersands leads to their substitution with an underscore; multiple underscores are collapsed into a single char.

Identifiers can contain spaces. Every space in an identifier name is escaped by an underscore in the text-only view, nonetheless shown as space in the other views. The underscore itself is escaped using a preceding backslash (`_`).

Identifiers can clash with any of the reserved words (see Table 3.3 for the complete list of reserved words): in this case the local variable name is internally prefixed with an at-sign (`@`) similar to keyword escaping in C# (Marshall [2009]). As in case with the space, the prefix is not shown in the phone view and can be seen only in the text version.

For names of global variables and actions as well as for script names there is no prefix escaping. To avoid name collision with any of the reserved words the TouchDevelop application automatically appends a running index to an offending name. This index is automatically incremented on every subsequent trial to declare the entity with clashing name.

Identifiers are case-sensitive and the complete identifier name is used to distinguish between entities. We could not find any sensible boundary for the identifier length: two different identifiers, 91 and 90 symbols long, sharing the first 90 symbols, were correctly distinguished by both the editor and the runtime system (see Fig. 3.6).

3.5.2 Keywords

The TouchDevelop language consists of a relatively compact set of keywords (see Table 3.3). Keywords are reserved words: global variables and action names are not allowed to coincide with them. It is nonetheless possible to use them for local variable names, the TouchDevelop environment automatically escapes them in this case (see Section 3.5.1).

TouchDevelop uses 22 symbols and has 16 reserved words. Action declarations are not allowed to clash with reserved words, whereas local variables are escaped internally if they clash (see previous section for details on escaping).

Out of 22 symbols only the assignment operator (`:=`) and commentary sign (`//`) are compound: the rest of the symbols is represented as single characters in Unicode.

⁴Windows Phone OS in its current version 7.5 does not support languages with right-to-left writing direction as input languages (despite the full support for rendering this type of writing in the built-in browser and in managed Silverlight applications).

Criteria	Phone view	Web view
Script overview	Script is pre-parsed and different script entities (actions, data, events) are shown in separate sections	Script is shown as a single text file.
Meta information	Meta information is used to render the correct script icon and visibility for actions. Otherwise, no meta information is visible.	Meta information is fully visible as plain text. Icon rendering takes meta information into account.
Unicode	Full Unicode support for all types of identifiers (action and event names, local variables), string literals and comments.	Unicode symbols in identifiers are escaped (\uXXXX). Unicode symbols in comments and string literals are in canonical UTF-8 format.
Spaces in identifiers	Identifier names are shown with spaces and Unicode symbols	Spaces in identifiers are escaped by underscore. Underscore signs are escaped with “_” symbol.
Statements and statement blocks	Statements are shown one per line without semicolon at the line end. Blocks of the same nesting level have same indentation. Scopes are indicated by indentation, curly brackets are missing. Else-branches are always shown. Empty else branches contain only the “do nothing” statement.	Statements are separated using semicolons, except for the statements with subsopes: no semicolon after the closing curly bracket. Blocks of the same nesting level have same indentation and are marked with curly brackets in Indian Hill style. Empty else-branches are now shown.
Local variables	Variable declaration with initialization are denoted with the keyword var .	Variable declaration with initialization are not designated. Every local variable is prepended with a dollar sign.

Table 3.2: Visualization properties of different source code views.

() { } , * / → ; + ”
 || < ≤ ≥ > = := ≠ : - //
 or not and meta returns while event do
 if then else for foreach action private ...

Table 3.3: TouchDevelop keywords and symbols.

3.5.3 Operators

Touch develop supports 2 prefix, one postfix, and 12 infix operators on different data types. These operators are all disjoint and cannot be overloaded, meaning that no two types share the same operator: for example string concatenation is indicated not with the plus sign, but a double pipe (`||`).

For the complete list of operators with their precedence and associativity please refer to Table 3.12).

3.5.4 Literals

TouchDevelop provides literals for three predefined value types: `Boolean`, `Number`, and `String`.

3.5.4.1 Boolean literals

Boolean literals can be either “true” or “false”. The literals are written without quotes. There is no conversion or mapping between other types (like `Number` or `String`) and Boolean literals.

3.5.4.2 Number literals

A number literal is a decimal number, with or without fractional part. The number is always treated as a real number, even if the fractional part is omitted. For bigger numbers TouchDevelop supports the scientific format with base and exponential part separated by an “E” character for output. This syntax is, however, is not supported in literals that contain only numbers and a decimal separator.

3.5.4.3 String literals

String literals are any Unicode sequences between two quotes (“example”). There are no limitations for these literals: both non-Latin based and right-to-left Unicode character sets are supported.

3.5.5 Comments

Only single-line comments are supported. Comments can come at every place (within and between declarations) and begin with two slashes. The rest of the line until the newline character is skipped, providing no way to limit commentaries to only a part of a line. The length of a single line of code is not limited. Therefore one can write as lengthy comment as one wishes: it will be wrapped into several lines in phone and web views.

The content of the comment can be any Unicode sequence, including the non-Latin based and right-to-left character sets.

3.5.6 Delimiters

The point is used as a decimal delimiter in all numeric values irrespective of the locale settings on Phone device.

Blocks are embraced with parentheses. Single statements within blocks are separated with semicolons.

Semicolons are also used to separate single meta declarations in script scope.

3.6 Types

TouchDevelop is a statically typed language (Tillmann et al. [2011], see Section 5.3 for more details), albeit the explicit type indication is not always necessary. Type information is inferred and used by the phone application to compile a list with context-available tokens and to present them on the keys of the phone pad.

The complete type system of TouchDevelop consists of predefined types. The language does not support any mechanism for creating new types, like composition of existing types or sub-typing.

All types in TouchDevelop are simple types: they can be only referenced using their names, which nonetheless do not count as reserved words. No complex types, like collections, are possible at the language level. Instead, predefined collection types for most common types (**Number**, **String** etc.) exist. These collections provide properties similar to indexers for collections in other languages.

3.6.1 Instantiatable and singleton types

There are two main classes of types in TouchDevelop. The first class constitutes the types that can be instantiated as local variables or parameters by the user. The second group are the singleton types, which do not support instantiation. To clearly designate the difference between these two types, the singleton types are written fully in lowercase, whereas the types supporting instances have their first letter capitalized.

Singleton types can be seen as bundled collections of properties and methods. Examples of these types are **bazaar** (for cloud services), **code** (for predefined actions), **data** (predefined global variables), or **phone** (phone services like SMS, calling) etc. These types can be accessed directly without prior instantiation, which makes them similar to static types in object-oriented languages.

3.6.2 Value and reference types

Another important classification of types in TouchDevelop, which is imminent to .Net platform and is present in most .Net languages, is that of reference and value types.

Variables of value types directly contain their data in contrast to reference types, where variables store only a reference to data, but not the actual value. The reason for this is because these data represent entities of complex structure, known in .Net as objects. It is therefore possible for two variables of reference type to point to the same entity so that operations on one variable can affect the object referenced by the other variable. These variables are called aliases. In case of value types, every variable keeps its own copy of the data and therefore it is not possible that any operation on one variable affects the value of another one.

Instances of value types are copied using a shallow (bitwise) copy. Therefore, an assignment of a value type variable in TouchDevelop leads to appearance of a new variable holding the copy of the r-value of the assignment statement. Conversely, for reference types only a reference to the original data is copied, so that an assignment on reference types leads to aliasing when two different variables point to the same or different, but overlapping entities.

All value types in TouchDevelop are predefined simple types, the complete list of value types is given in Table 3.4.

3.6 Types

Type name	Short description
Vector3	Vector in 3D space with x, y and z coordinates.
DateTime	Date and time with millisecond precision.
Number	Real value of IEEE double precision.
Boolean	Boolean value
String	Line of text

Table 3.4: Value types in TouchDevelop (Horspool et al. [March, 2012], page 64).

Domains for reference and value types:

$Val \subseteq Types, Ref \subseteq Types, Ref \cup Val = Types, Ref \cap Val = \emptyset$

$Bool \subseteq Val, String \subseteq Val, Num \subseteq Val$

Operations:

$typeof : Types \rightarrow String$

$typeof = \lambda t.(t \in Val \rightarrow (t \in Bool \rightarrow "Bool" \sqcap (t \in String \rightarrow "String" \sqcap (t \in Num \rightarrow "Num" \sqcap "Unknown")) \sqcap accessproperty(t, "type"))$

Figure 3.7: Semantic domains for types (see Fig. 3.8 for selection operation)

3.6.3 Predefined value types

3.6.3.1 Nothing, invalid and is_invalid()

The **Nothing** type is used to designate the cases where no variable or entity is expected. For example, for properties without any return values **Nothing** is used as the type for return parameters, telling the TouchDevelop interpreter that no value is returned from the code. This type supports no operations or methods and is similar to the void construct in other languages (like C# or Java).

Every type in TouchDevelop supports the property **is_invalid()**. This property is used to check whether the variable contains any valid value or not. This property returns **true** if the current variable state is invalid, meaning that it has not been assigned. This is true for both reference and value types in TouchDevelop, therefore every type in TouchDevelop can be seen as a type supporting the Null state (similar to nullable value types in C#: **Nullable<T>**).

A variable can contain an invalid value only if it has not been assigned. This is possible in two different scenarios.

For return parameters of an action one can read from these parameters prior to their assignment in the code block and the value of this parameter is considered invalid.

The second possibility is to directly create a variable using the **invalid** (singleton) type. This type provides properties for direct creation of variable with invalid flag set to true for many TouchDevelop types (see Table 3.5). As long as this variable is not assigned any value after creation, its value is considered as invalid. This approach is used in TouchDevelop libraries to signal exceptional situations where the assignment was not possible. For example, if requesting a current geo-position fails, the returned position variable contains an invalid value and one can check if the position is a valid position by calling **is_invalid()** before processing this value.

The invalid type can be, on the other side, seen as a default constructor for the types mentioned in Table 3.5. The default value for the newly created instances is not defined and any attempt to access the value leads to the immediate halt of script execution.

type	collection	type	collection
appointment	appointment collection	picture	pictures
board		picture album	picture albums
boolean		place	place collection
camera		playlist	playlists
color		printer	printer collection
contact	contact collection	song	songs
datetime		song album	song albums
device	device collections	sound	
json object		sprite	sprite set
link	link collection	string	string collection
location	location collection	textbox	string map
map		tile	
media link	media link collection	vector3	
media player	media player collection	web request	
media server	media server collection	web response	
message	message collection	xml object	
motion			
number	number map		

Table 3.5: Properties of the invalid type in TouchDevelop.

3.6 Types

Description	Character	Associativity
Negation	not	right
Boolean operators	and, or	left

Table 3.6: Operators supported by the *Boolean* datatype in TouchDevelop.

Domain definition:

$b \in Bool = Boolean = \{true, false\}$

Operations:

$not : Bool \rightarrow Bool$

$not = \lambda b. \neg b$

$and : Bool \times Bool \rightarrow Bool$

$and = \lambda b_1. \lambda b_2. b_1 \wedge b_2$

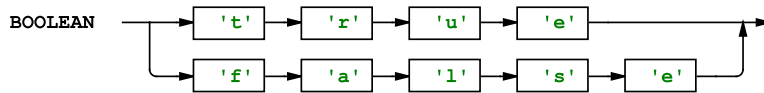
$or : Bool \times Bool \rightarrow Bool$

$or = \lambda b_1. \lambda b_2. b_1 \vee b_2$

Figure 3.8: Semantic of the Boolean type.

3.6.3.2 Boolean

The **Boolean** datatype holds the result of a comparison between two numeric values and can be only **true** or **false**. It is the only datatype that can be used in conditional expressions in the branching operator **if** or in **while** loops. There is no implicit conversion between numeric values and Boolean ones.

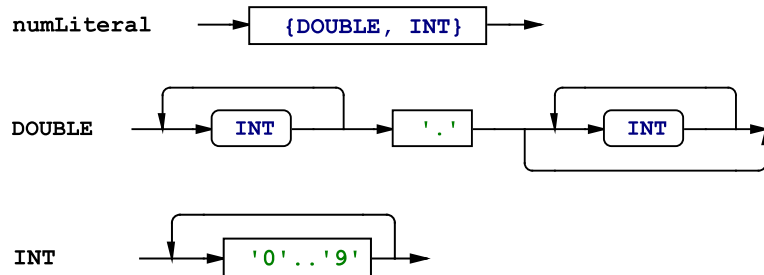


The **Boolean** type supports very few operations only (see Table 3.6), in particular there is no way to compare two different Boolean variables for equality. Operations on Boolean types are all commutative, associative, and distributive.

Boolean expressions are always evaluated eagerly, so every partial expression in a complex Boolean expression must be a valid one and is evaluated every time without shortcuts.

3.6.3.3 Number

Number is the only type to represent numerical values in TouchDevelop. The **Number** type can be directly assigned using a numerical literal. Number literal can be any real number. It has the internal precision of the IEEE type **double** and can be used to store both integer and fractional values. Longer literals are rounded down to the precision of **double**.



Numeric literals always start with a digit optionally followed by a decimal point and the

Description	Operator	Associativity
Parentheses	(,)	left/right
Multiplication	*	left
Division	/	left
Addition	+	left
Subtraction	-	left
Arithmetic comparison	<, ≤, =, ≥, ≠	right

Table 3.7: Operators supported by the **Number** datatype in TouchDevelop.

fractional part.

For negative values the value is preceded by the „minus” sign. In current specification we treat the unary minus not as a part of **Number** literal, but as a separate operation (because it is possible to negate expressions, not only numbers) on **Number**. Positive numbers do not allow any unary „plus” sign and must contain only numbers.

Should the number exceed the value of 10^6 it is automatically converted into a scientific floating-point representation with exponent part (for example $1.345E+12$). This conversion, however, is only working for outputting the value to the wall. Trying to assign a variable a value greater than 10^6 leads to the TouchDevelop suggested “quick fix” that converts the number to the scientific format, but the letter E is not recognized as a valid identifier for numeric literals and after the next automatic fix the value $1.345E + 12$ is converted to $1.345 + 12$.

Numeric literals are all decimals (even if starting with zero), there is no support for any other bases.

Following the IEEE 754 specification for floating-point arithmetic the **Number** datatype supports the NaN and infinity values as well as the methods to check for these values.

Number supports all common arithmetic operations as well as the comparison operations (see Table 3.7). Because **Number** is a fixed-length datatype (\mathbb{R}_{IEEE} , see Figure 3.9 for definition) that represents a finite set of values, the arithmetic operations on the variables of these types have less properties compared to \mathbb{R} that is infinite. In particular, commutative, associative and distributive properties do not hold for these operations due to the fact these operations are inexact and eventually include rounding to the next element of the set. The rules for “**roundToNext()**” are defined by IEEE 754-2008 (IEE [2008]).

All comparison operations are predicates that return the value of type **Boolean**.

3.6.3.4 String

The **String** datatype is used to store Unicode character sequences. There is no special type for a single character. Single quotes are used to indicate a string literal. The size of the string literal is not constrained by the language and most likely has the same limitation as in Silverlight: 32764 characters.

Every instantiatable datatype can be implicitly converted to **String**.

String supports only one operator, the infix string concatenation written as `||`. If any of the operands is not **String** it is implicitly converted to this type. The return value of this operation is always of type **String**.

Domain definition:

$n \in \text{Number}$, $\text{Number} = \mathbb{R}_{IEEE}$

here \mathbb{R}_{IEEE} is the domain of float point numbers as defined in IEEE 754-2008 for **binary32** (IEE [2008] Table 3.5, page 13), including the values $NaN, \infty = \{+\infty, -\infty\}$

Arithmetic operations:

$\text{op} = \{\text{addition} ::= '+', \text{subtraction} ::= '-', \text{multiplication} ::= '*', \text{division} ::= '/'\}$

here the names for operations refer to the arithmetic operations defined in IEEE 754-2008 (IEE [2008] Section 5.4.1, page 31)

Comparison predicates:

$\text{pred} = \{\text{EQ} ::= '=', \neg\text{EQ} ::= '\neq', \text{LT} ::= '<', \text{LT EQ} ::= '\leq', \text{GT} ::= '>', \text{GT EQ} ::= '\geq'\}$

here the names for predicates refer to the predicated defined in IEEE 754-2008 (IEE [2008] Tables 5.1-5.3, page 29).

Infinity arithmetics (in concordance with IEEE 754-2008, IEE [2008] p. 34):

for any finite n

- $\text{addition}(\infty, n) = \text{addition}(n, \infty) = \infty$
- $\text{subtraction}(\infty, n) = \text{subtraction}(\infty, n) = \infty$

for any finite n , $n \neq 0$

- $\text{multiplication}(\infty, n) = \text{multiplication}(n, \infty) = \infty$
- $\text{division}(\infty, n) = \text{division}(n, \infty) = \infty$
- $\text{division}(n, 0) = \infty$

NaN arithmetics:

- $\text{multiplication}(\infty, 0) = \text{multiplication}(0, \infty) = NaN$
- $\text{addition}(-\infty, +\infty) = NaN$
- $\text{division}(0, 0) = \text{division}(\infty, \infty) = NaN$

Figure 3.9: Semantics of type **Number** and expressions with this type.

Domain definition:

$'a' \dots 'z': \text{Char}$

$s \in \text{String} = \text{Char}^*$

Operations:

$\text{concat} ::= '||'$

Operations:

$\text{tostring} = \lambda a. (\text{typeof}(a) = \text{"String"} \rightarrow a \sqcap (\text{typeof}(a) = \text{"Num"} \rightarrow \text{conv}(a) \sqcap (\text{typeof}(a) = \text{"Bool"} \rightarrow (a \rightarrow \text{"true"} \sqcap \text{"false"}) \sqcap \text{""}))$

$\text{concat} : \text{Types} \times \text{Types} \rightarrow \text{String}$

$\text{concat} = \lambda s_1. \lambda s_2. (s_1 s_2)$

Figure 3.10: Semantics of the **String** type

3.6.4 Predefined collection types

TouchDevelop does not provide support for user-defined collection type, but has some frequently used collections built-in (see Table 3.8).

User collections can be created using the singleton class **collections** (for value types) or **invalid** (for system types) which provides parameterless constructors for every collection type labeled with C in the Table 3.8. The only exception is the **Sprite Set** collection, which is bound to a certain **Board** instance and created using the appropriate method of the **Board** instance.

There is no possibility to specify the number of dimensions or dimension size for collections: every newly created collection is one-dimensional and has zero elements (if it contains items of a value type) or is invalid (for items of system types) until at least one element is added. Adding a new element automatically increased the dimension size by one and deleting an element decreased it.

Most of the system collections (like **Songs**) are read-only and can not be modified from the script. They are available via global singleton classes.

The members of every collection can be accessed via an access property **at(elemnum: Number)** or using a collection iterator in a **foreach** loop. If the index lies outside of the collection boundaries, an item instance is returned with **is invalid** property set to true. Two special collections, called **String Map** and **Number Map**, implement the dictionary collections with keys represented by **Number** and **String** respectively.

3.6.5 Type hierarchy

TouchDevelop features only a very constrained support for a type hierarchy: there is only one implicit type conversion and due to lack of sub-typing or inheritance the type hierarchy is flat.

Every instantiatable type can be implicitly converted to **String**. This conversion takes place before any operation that requires **String** as one of its parameters: these include custom actions with **String** parameters (both as input and as return) as well as the string concatenation operator **||**.

String and all types below in the type hierarchy implement the **is valid()** property that returns **false** if the value of the current variable is not a valid one. Invalid variable values are possible if the called method fails to return the expected value. While this is not possible within user-defined scripts, this situation might occur if the user cancels the string input or some phone sensor fails to provide its readings. In this respect the **is valid** property is the counterpart to the optional value **Some** in functional languages or to nullable types in object-oriented languages (C#).

3.7 Declarations

Declarations are the only constructs that are allowed in the topmost scope of TouchDevelop scripts (script scope) and this is the only scope where they are allowed. No nested declarations (like action-level global variables or nested actions) are possible.

3.7.1 Program structure

The complete program code is stored in a single text file.

Collection name	Item type	Type(s) with constructor	Type of collection	Supported methods
Appointment Collection	Appointment	invalid	Indexed List	R
Contact Collection	Contact	invalid	Indexed List	R
Device Collection	Device	invalid	Indexed List	R
Link Collection	Link	collections/invalid	Indexed List	CAMRUDEF
Location Collection	Location	collections/invalid	Indexed List	CAMRUDEF
Media Link Collection	Media Link	invalid	Indexed List	R
Media Player Collection	Media Player	invalid	Indexed List	ARE
Media Server Collection	Media Server	invalid	Indexed List	ARE
Message Collection	Message	collections/invalid	Indexed List	CAMRUDEF
Number Map	Number	collections	Map (Number \rightarrow Number)	CAMRU + Agg
Pictures	Picture	media/invalid	Indexed List	RF + Rand
Picture Albums	Picture Album	media/invalid	Indexed List	R + Rand
Place Collection	Place	collections/invalid	Indexed List	CAMRUDEF
Playlists	Playlist	media/invalid	Indexed List	R
Printer Collection	Printer	invalid	Indexed List	ARE
Songs	Song	media/invalid	Indexed List	R + Rand
Song Albums	Song Album	media/invalid	Indexed List	R + Rand
Sprite Set	Sprite	Board/invalid	Indexed List	CAMRDF
String Collection	String	collections/invalid	Indexed List	CAMRUDEF
String Map	String	collections/invalid	Map (String \rightarrow String)	CAMRU

Table 3.8: Collection types in TouchDevelop.

Abbreviation used:

C - Create, A - Add, M - add Many, R - Read, U - Update, D - Delete, E - clEar or Empty, F - Find, Agg - aggregate functions, Rand - random access.

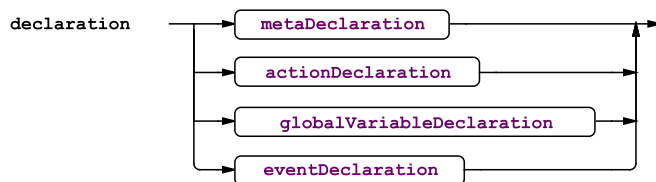


The module contains one or several declarations. The order of declarations is not restricted. Creating a new script in the TouchDevelop environment leads to the simplest program containing meta information and one parameterless action “main()” with the following content:

```

action main()
"TouchDevelop is" -> post to wall
  
```

There are four different types of declarations: meta-declarations, declaration of global variables, action declarations, and event declarations.



Meta declarations as well as single statements in action and event bodies are terminated using a semicolon and a line break. Every statement or declaration occupies the complete line: there is no way to have two statements or declaration within a single line. In case of the declarations that use curly brackets to group expressions (action, global declaration, event declaration), the semicolon is not used after the last closing curly bracket and it is only the line break that terminates the declaration in this case.

3.7.2 Scopes

TouchDevelop has four scopes, namely script scope, action scope, event scope, and block scope, indicated by curly parentheses (see Figure 3.11).

Script scope is the single, top scope. It begins with the first declaration and ends after the last declaration and allows only declarations (see Section 3.7).

Action and event handler declarations introduce their own new scopes. Actions and events can contain only statements or statement blocks. Block scopes are introduced only by flow control structures: the conditional operator **if...then...else** and the loop operators **for**, **foreach**, and **while**. Block scopes, similar to action or event scopes, contain only statements of statement blocks. Block scopes can be nested to any degree.

3.7.3 Nesting and hiding

All predefined standard types, operations and actions are available in all scopes. They cannot be hidden by local declarations, because TouchDevelop does not support declarations for types or operations. Standard action are defined on types and therefore cannot be redefined either.

Actions and global variables are visible within the script and all nested scopes. They cannot be redefined or hidden.

Events declarations are not visible at all, therefore statements cannot be invoked directly or indirectly. Event handler invocations are performed by the runtime environment. Statements within event scopes can access the parameters that were passed into event handlers as well as global variables or actions.

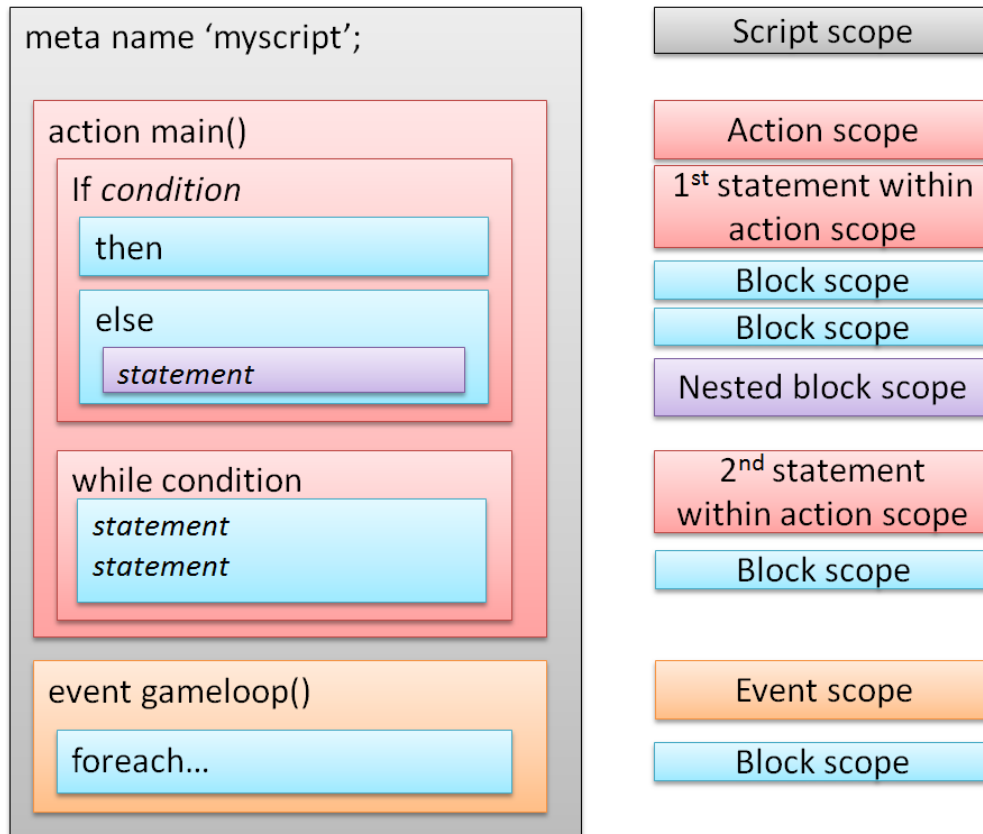


Figure 3.11: Scopes in TouchDevelop.

Local variables exist only within the scope in which they were declared and in all nested scopes. The redefinition of the variable within a nested scope is not possible: the semantics of TouchDevelop does not distinguish variable definition and redefinition, therefore a new definition within the child scope is treated as a re-assignment of the local variable from the upper scope.

In some exceptional cases some variables can be hidden in the nested scopes: for example, the upper bound for a `for` loop is hidden within the loop block scope and cannot be accessed. The loop variable itself is visible, but as a read-only variable.

Actions can have a restricted visibility. Public actions can be directly started by a user, whereas private actions can only be started by other actions. To mark an action as private the `meta private` statement is added as the very last line of the action body.

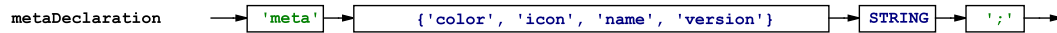
3.7.4 Meta declarations

Meta declarations are mostly for internal use of TouchDevelop applications and contain information about script execution (see Table 3.9).

In our original mining we considered the `meta version` to be the version of the language API, but Horspool et al. [March, 2012] specify it as a TouchDevelop revision that was used to serialize the script. Other meta declarations contain the script name (`meta name 'sample'`), the default icon used to show the script (`meta icon 'clock'`) in the Bazaar, and the foreground color of the application icon with transparency as alpha-channel (`meta color '#FF008080'`).

Meta identifier	Sample value	Semantic meaning
color	#ff00cc99	ARGB representation of the script icon's foreground color.
icon	question	Name of the icon of the predefined icon set.
name	songs and events	Name of the TouchDevelop script in Bazaar.
version	v2.0	Version of the language API or serialization.

Table 3.9: Meta declarations in TouchDevelop.



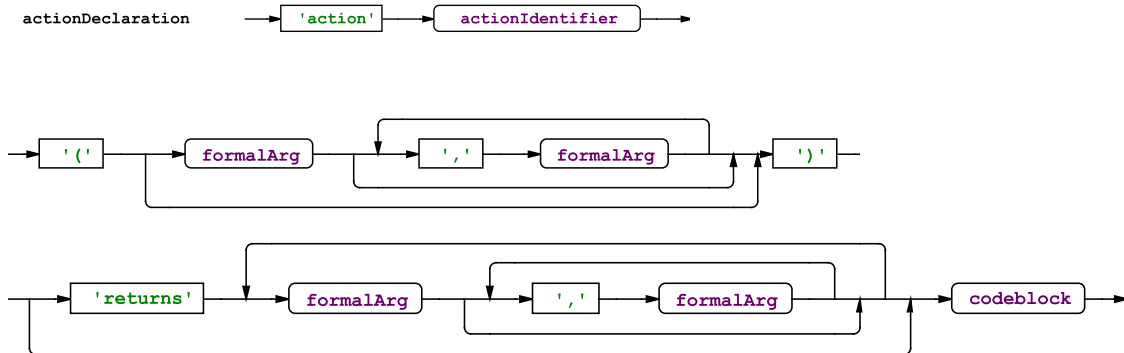
Meta declarations have either global or action scope. In actions they are used to designate certain actions as private (i.e., not directly runnable by the user, **meta private**).

The specification published recently (Horspool et al. [March, 2012]) also covers the usage of meta declarations to bind to other scripts and use these scripts as external libraries. Our specification does not include this feature, because it was not available at the time of writing these lines.

3.7.5 Action declarations

Action is a codeblock that implements computations or an action that can be performed on the type.

Actions are declared using action declarations, which are similar to the function declaration in imperative languages.



Every action must have a unique name and can have multiple (including zero) input and output parameters. A parameter declaration includes a parameter name and type, separated by colon.



Parameter names must be distinct. This applies to both input and output parameters, which are treated as one set of parameters. Return parameters have to be defined (=assigned, due to the TouchDevelop semantics) before the end of the action, otherwise the action is deemed erroneous and will not be executed by the runtime environment.

3.7.6 Event declarations

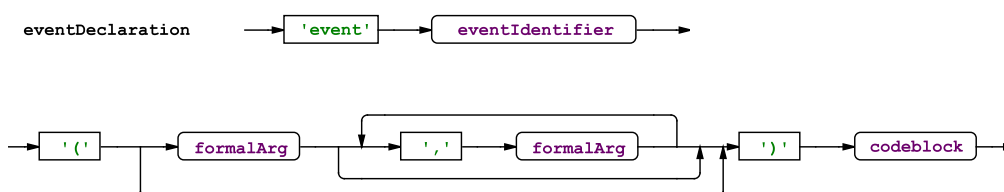
Some TouchDevelop entities support events. Events are external situations that trigger the execution of pre-defined actions (called event handlers). Events are supported only for built-in entities and cannot be defined by users in the script. Only one event handler per event is possible.

Much like types, events in TouchDevelop can be divided into two groups: singleton events and instance events. Single events are unique and not bound to any global variable in the script. Rather, they originate from singleton types, like `phone` or `media`. They represent scenarios that normally happen globally to the phone device: changing its position in space, shaking etc.

Instance events are “attached” to a global variable of the script. Only `GameBoard`, `Sprite`, and `wall` types currently support instance events. The latter one provides the tapping event for every information box that appears upon invocation of the `post to wall` property for any variable or literal.

There is no explicit binding between events and event handlers. Rather, the event handler names are used to map events (similarly to event handler mapping in ASP.NET Webforms): for the name of an event handler (usually) contains the name of the global variable the handler is “attached to” followed by the event name which is usually self-explaining.

Event handler declarations have the same syntax as action declaration, except for the return parameters that they do not have.



Events can be directly called from TouchDevelop applications, similar to actions. There is no difference in execution mode or allowed syntax between actions and events: the only difference is that latter ones are only called by the system and there is no way to call events directly from actions or other events.

Taking another view, events can be categorized depending upon the source of the event triggering signal. One big group of events contains the extrinsic events triggered by phone sensors (currently only accelerometer and camera are supported), these events are shown in Table 3.10.

Another group of events comprises those triggered by the TouchDevelop environment and internal objects, instantiated during script executions. These events are summarized in Table 3.11)

Unlike actions, event handlers cannot have any return parameters: their definitions are strictly predefined and cannot be altered by the user. Event handlers have full access to global variables and can invoke actions (including private ones).

3.7.7 Local variables declarations

Depending upon their visibility scope, the variables can be declared as local or global ones. All global variables that are used in the scripts have to be declared before their use in the same or one of the parent scopes. Local variables are accessed merely through their names and require no prior declaration, therefore the first use of a local variable as the L-value of an expression is considered as both declaration and initialization of the variable. Attempts

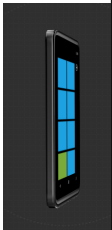
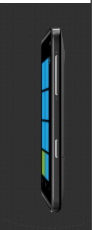
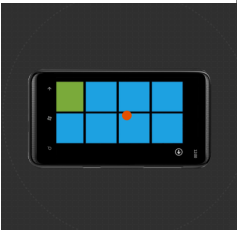



Event name	Event source	Short description	Picture
shake	Accelerometer	Phone is shaken.	
phone face up	Accelerometer	Phone is turned face up.	
phone face down	Accelerometer	Phone is turned face down.	
phone portrait	Accelerometer	Phone is turned to portrait orientation.	
phone landscape left	Accelerometer	Phone is turned to left landscape orientation.	
phone landscape right	Accelerometer	Phone is turned to right landscape orientation.	
camera button pressed	Camera	camera button pressed	To use for the "augmented reality" apps.
camera button half pressed	Camera	camera button half-pressed	
camera button released	Camera	camera button released	

Table 3.10: Extrinsic (sensor) events in TouchDevelop.

Event name and signature	Event source	Short description
active song changed	media library	Changing the currently played song.
tap wall<wallitem>(item<wallitem>)	wall	Tapping any item (of type wallitem) published to the wall.
gameloop	GameBoard	Triggering an event approximately every 50 ms.
player state changed	??	Unknown
tap board <boarditem>(x, y)	board object	Tapping on a specific GameBoard object, with coordinates.
swipe <boarditem>(x,y,dx,dy)	board object	Swiping on a specific GameBoard object, with coordinates and shift.
tap sprite in <spriteitem>(<spriteitem>,index,x,y)	sprite object	Tapping on a specific sprite within SpriteSet collection, with coordinates.
swipe sprite in <spriteitem> (<spriteitem>,index,x,y,dx,dy)	sprite object	Swiping a specific sprite within SpriteSet collection, with coordinates and shift.
drag sprite in <spriteitem>(<spriteitem>,index,x,y,dx,dy)	sprite object	Dragging a specific sprite within SpriteSet collection, with coordinates and shift.

Table 3.11: Intrinsic events in TouchDevelop (“<” and “>” indicate the variable parts of a handler).

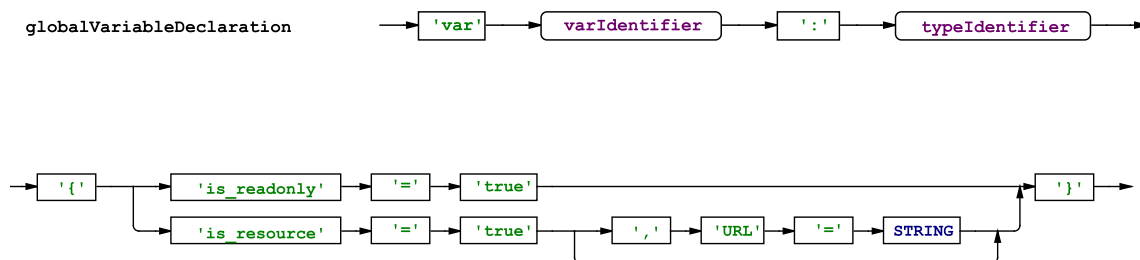
to access the value of a local variable before initialization (possible only for local variables representing return parameters in actions) leads to an error message in TouchDevelop and the whole action or event is deemed non-compilable.

Local variables are labeled with a leading dollar sign (\$, for example \$x) in the web view (see Section 3.4.1), similar to PHP. This designation, however, is only syntactic sugar and not persistent: there is no special designation of local variables in the remaining two views.

3.7.8 Global variables declarations

All global variables are explicitly typed and the type has to be selected from the list of available types.

Global declarations denote the global variables that are available in every action within the same module.



Every global declaration begins with the reserved keyword **var** followed by identifier name and type separated by a colon. Only instantiatable (non-singleton) types can be used for global variables (see Section 3.6). A global declaration also contains a codeblock section with some attributes. This section is normally empty or omitted, but might contain **readonly=true** expression for read-only global variables and **is_resource=true** for assets (termed “arts” in TouchDevelop apps). Assets can optionally contain the URL property to specify a link to an external resource (picture).

Global parameters, unlike the local ones, are not accessible merely via their names. Instead, they are available as properties of a **data** singleton object. That makes them similar to other script entities, like actions, and clearly different from local variables in the source code, so that they can have the same names as local variables without clash or hiding. The name of every global variable must be, however, unique within the global scope.

3.8 Expressions

Expressions are sequences of operators and operands.

An expression can be classified as one of the following:

- A literal. Every literal bears a value with an associated type and has a proper syntax: for example string literals is enclosed in double quotes. For more details on literals supported by TouchDevelop please refer to Section 3.5.4.
- A variable. Every variable has an associated type, either explicitly declared (as a parameter of an action or library type) or implicitly derived taking into account the type of l-value in variable declaration with initialization.
- A library type. Library type can appear only on the left-hand side of the arrow expression (used for property access). In any other context an expression holding a type name would lead to a runtime error.

- A special global namespace. There are predefined namespaces for accessing actions and variables of the current script: **data** (or a special Unicode sign \boxplus , look Table 3.6) is used to access (via an arrow expression) globally declared variables, **art** for assets of the current script and **code** for accessing actions of the current script.
- An operation, consisting of an operator and one or two operands. Unary and binary operators are defined for basic built-in types (see Table 3.12) and for property access (the arrow operator). For property access see for property invocation semantics (see Section 3.8.3).
- Nothing. This occurs if the expression was an invocation of a property or action with a return value of type **Nothing**. This expression can be valid only in the context of statement expression.

The result of expression evaluation is always a value of a certain type. No any other values, like action invocations or partially evaluated expressions (lazy expressions) are possible here.

3.8.1 Operator expressions

TouchDevelop has all three types of operators: prefix, infix and postfix. All operators are predefined. There is no possibility to define own operators or overwrite the properties of existing ones.

Prefix operators are unary operators on data types (negation, both arithmetic and Boolean). Operators in infix notation are available for the limited number of built-in data types: **Number**, **String** and **Boolean**. These sets of operators supported by every data type are fully disjoint.

Operators of Boolean logic are only applicable to the **Boolean** type values, there is no any support for bit and bitvector operations.

Besides binary operators in infix notation, there are two unary operators: **not** for Boolean and leading minus sign (-) for arithmetic negation. For Boolean negation the argument is always in parentheses.

3.8.2 Arrow expressions

The Arrow operator is used to access properties of types and their instances.

For library types, properties are the combination of getter or setter functions (properties having no setters are the read-only properties). Depending upon the type, these can be either instance-level (for types supporting instances) or type-level (static properties).

In TouchDevelop scripts the declared actions are mapped onto the properties of a special global object called **code** and the arrow operator is used for action invocations. In the phone app, however, the actions are just designated using a special Unicode sign (see Table 3.6).

Properties can have arguments, and properties called with arguments are similar to function invocations in procedural languages. The only difference is syntactic, that properties without arguments do not require the empty pair of parentheses. Unlike properties in OOP-languages, properties in TouchDevelop are allowed to have no return values (return type **Nothing**), which makes them even more similar to procedures. These properties are used only for side effects, like the most common **post to wall** property, available for every TouchDevelop type. For types that do not implement this property explicitly it is available implicitly due to the implicit conversion to **String**. Under the hood, the **.Net** method **.ToString()** method is called on the underlying data type, sometimes leading to

Operations	Symbol	Unicode	Precedence	Associativity	Position	Operand type	Result type
Function application	function()		10	right	prefix	any	any
Property access	→		10	right	postfix	any	any
Unary negation	-		6	right	prefix	Number	Number
Boolean negation	not()		6	right	prefix	Boolean	Boolean
Multiplication	*		5	left	infix	Number	Number
Division	/		5	left	infix	Number	Number
Addition	+		4	left	infix	Number	Number
Subtraction	-		4	left	infix	Number	Number
Arithmetic comparison	<, ≤, =, ≥, ≠		3	left	infix	Number	Boolean
Boolean operators	and, or		2	left	infix	Boolean	Boolean
String concatenation			2	right	infix	any ^a	Boolean
Assignment	:=		0	right	infix	-	-

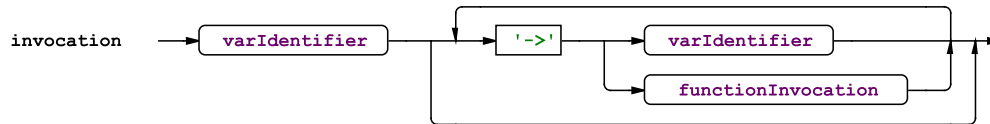
Table 3.12: Operator expressions in TouchDevelop (Precedence: 10 – highest, 0 – lowest).

^aDue to implicit conversion of the arguments to *String*.

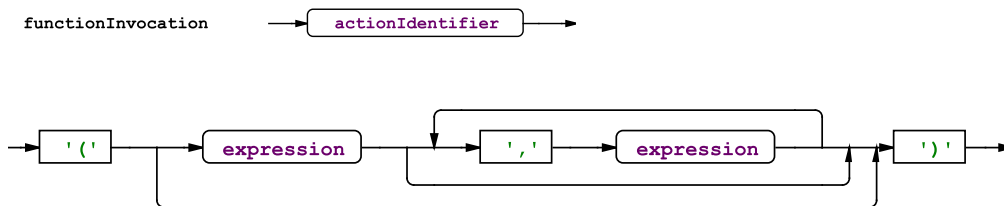
(for a TouchDevelop user) unexpected results.

3.8.3 Action and properties invocation

Properties of library types are called using the name of the library type, an arrow expression to reference the respective property of the type and optionally by providing parameters.



If no parameters have to be specified, the empty brackets can be omitted, and only the name of property is provided.



Property invocation can be either an expression (or its part) or a full-fledged statement. Action from the same script are invoked using the same syntax as properties of library types. The global singleton object `code` is used to refer to the current script with arrow expression referencing a certain action (see Section 3.8.2).

All parameters are passed using call-by-value. This means, that every parameter (including the return ones) corresponds to a local variable that gets its initial value from the corresponding argument supplied in the property invocation. Every argument for input parameters is fully evaluated and a copy of the value is assigned to the local variable with the same name as the parameter. For output parameters local variables with invalid values are created and no assignment happens on invocation, therefore leaving these variables in an invalid, i.e. uninitialized state.

Every action invocations introduces a stack frame: the local variables are re-initialized on every invocation (including recursive ones); there is no option to persist local variables in-between single invocations and one should use global variables for any values that have to be persistent between invocations.

3.8.4 Event calls

Events are not accessible from user script and cannot be accessed using the arrow operator. This is the runtime environment that takes care of detecting the events and invocation of event handlers. Some events have input parameters which are automatically filled by the runtime environment. Even though one can theoretically always check the validity of these parameters by invoking the `is invalid` property, these parameters should never be invalid.

The specification by Horspool et al. [March, 2012] states that the event invocations are queued for execution, so that the execution of an event handler is never interrupted. In our mining experiments we, however, failed to observe any queuing of events: as long as event code is being executed all incoming events are just silently ignored.

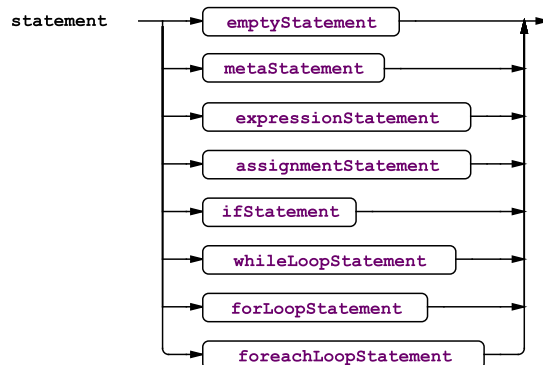
3.8.5 Evaluation order

For assignments the complete right-hand side is first evaluated and upon successful evaluation all values (one or many in case of multiple assignment) are copied to the variables on the left-hand side.

For action invocations only input parameters in actions are evaluated. Parameters are evaluated from left to right and passed to the action as values (see 3.8.3 for details).

3.9 Statements

TouchDevelop supports 8 types of statements.



3.9.1 Empty statement

There is an empty statement, which consists of three dots⁵ and a separator (semicolon).



This statement is probably used by the TouchDevelop environment to mark the last editing site in the source code. In the phone view these statements are shown as empty blocks with small text “do nothing”.

3.9.2 Meta statements

Meta statement are similar to meta declarations (see Section 3.7.4), but used at the action scope to store some information about action.



The only use of this statement in TouchDevelop script we detected was to mark actions as private (not directly runnable by a user).

3.9.3 Expression statements

Expression statements consist of expressions that can be evaluated to return one or several values which is not used in the script.

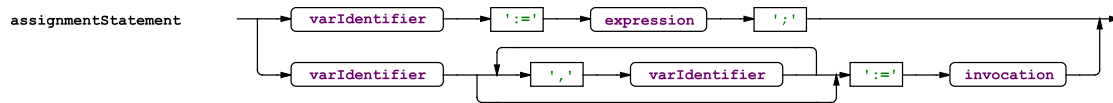
⁵In the latest version of the TouchDevelop cloud one can also see a new keyword **skip** which is used to mark an empty action body together with the three dots.



TouchDevelop always suggests an assignment for the expression statement via a quick fix (see Section 3.4), providing an appropriate local variable name (depending upon the type of the return value).

3.9.4 Assignment statements

The Assignment operator is the only compound operator in TouchDevelop, meaning that it consists of two characters: colon and equal sign (`:=`). Thereby the assignment operator is syntactically different from the equality operator available for **Number** comparisons, fully in the tradition of Pascal and Modula languages.



The assignment operator is used for both declaration and initialization of a local variable; local variables therefore do not require prior declaration and are treated as both declared and initialized after the first assignment: therefore, there are no default values for variables. There is no syntactic difference between a new declaration with initialization and an assignment of the variables. This makes the re-declaration and hiding of variables from outer scopes impossible.

To highlight the new declarations the TouchDevelop application performs a semantic check of the source code on-the-fly and marks the declarations with a preceding **var** keyword in the phone view (see Section 3.4.1). This might be misleading for beginners, because this designation is just a part of syntax highlighting of the TouchDevelop script editor and does not appear in any other views of the source code: trying to edit the respective line of code in the TouchDevelop editor (which makes it switch to a single line editor) one can not see this label.

The right-hand side of an assignment expression is always evaluated before assignment (see the table with operator precedence, Table 3.12) and the resulted value is assigned to the respective variable as a copy. Variables can contain only values or type instances, there is (currently) no way to store an action or event in a variable.

3.9.4.1 Multiple assignment

Multiple assignment is an assignment of several local variables at once. Should the right-hand side of the assignment expression return more than one value, the left-hand side should match in the number of the local variables. The only expressions that return more than one value are invocations of actions that have several return values in their signature. There is no way to group several expressions ad-hoc to produce a tuple-like structure on the statement level.

L-values must exactly match the signature for returning values that is to be of the respective type. Multiple assignment can be combined with variable declaration: in these case the local variables of the type matching the action signature for return values are declared and initialized with return values. Unlike pattern matching, however, all variables are mandatory and there is no a placeholder for an empty value (like underscore in the mentioned languages): every variable has to be provided explicitly. It is also possible to mix already assigned local variables with non-assigned in the multiple assignment.

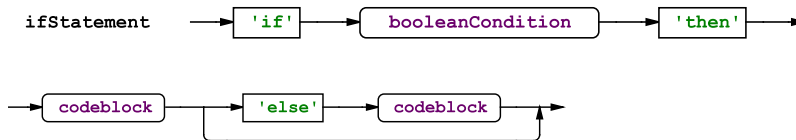
3.9.5 Control flow statements

There are four types of control flow statements in TouchDevelop: one conditional statement (**if**) and three types of loop statements (conditional loop (**while**), loop with counter (**for**) and iterator loop (**foreach**)).

As a condition for conditional statements can serve any expression (Boolean, numeric, function invocation) which can be evaluated to Boolean.

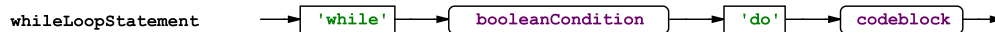
3.9.5.1 if

Control flow can be ramified using the **if...then...else** operator. Both branches (**then** and **else**) are mandatory for the TouchDeveloper app view, but not in the other views (see Table 3.4.1). The condition of the **if** operator must be of type **Boolean**. Both branches introduce subscopes for local variables (see Section 3.7.2), the local variables declared and used here are not available after the end of the branch scope. There is no way to leave the scope prematurely.



3.9.5.2 while

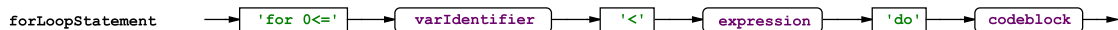
TouchDevelop supports only one type of conditional loops: loops with pre-condition (**while...do**).



It is possible to implement a non-terminating loop using **while (true) do{}** syntax. However, there is no construct that would allow one to break the loop: the only possibility to stop the loop execution is to perform a forbidden operation (for example, division by zero), which would halt the script execution together.

3.9.5.3 for

Classic loops with numeric iterators or counters are also present in TouchDevelop. These loops, however, are rather constrained as regards the iteration domain: the counter always starts at zero and is incremented by one until the provided maximal value reached (**for \leq iterator \leq max do {}**). The counter equals the maximal boundary in the very last iteration of the loop. There is no way to explicitly provide the starting value or to construct a loop with a decrementing counter.



Iterators over collections are only possible for the built-in types of collections (see Section 3.8) using the **at()** property, but the use of a special collection iterator is preferred here (see next section).

The counter variable is read-only within the loop scope: there is no way to skip some iterations by manually setting the counter variables to an appropriate value.

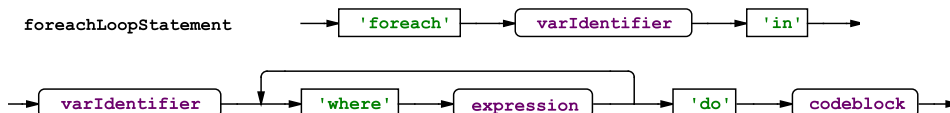
The expression used for the upper bound is calculated once at the beginning of the loop execution and is not updated after every loop iteration. Therefore, the number of loop

3.10 Special types

iterations is determined at the starting point of the loop and cannot be increased or decreased.

3.9.5.4 foreach

For built-in collection types (see Section 3.8) there is support for foreach loops (`foreach iterator in collection where...[,where...]`) with collection iterators.



According to our results from reverse engineering, the `foreach` loop can have only a single `where` guard, which contains an additional criterion that are imposed on every iterated collection item. The recently published alternative language specification (Horspool et al. [March, 2012]) indicates support for multiple `where` keywords in the latest version. These conditions are executed in a lazy way: as long as a certain condition returns false, the iterator automatically moves to the next item in the collection without evaluating the remaining conditions.

For collection iterators it is possible to modify the properties of collection items if the items support it. The collection itself cannot be modified within the loop body: doing so would result in an error message.

3.10 Special types

To support easy script authoring and sharing by hobby developers TouchDevelop provides several types with simple API and hidden implementation details. `GameBoard` allows user to implement 2D board games with smooth animation without manual redrawing of every single frame. `Wall` servers as a substitute for system console as regards basic input and output operations.

3.10.1 Game Board

`GameBoard` is a special feature-rich, instantiatable type in TouchDevelop, which provides APIs useful for 2D game development. The properties of the `GameBoard` type can be classified in the following groups:

1. **Board API** represents the properties necessary for creating `GameBoard` and `SpriteSet` objects and iterators over `SpriteSet` collections associated with the board variable. An important part of the Board API are the board events: tapping, swiping, and dragging. These events can be resolved at the board level (if there is no internal object at the screen position where the event took place) or directly at the sprite level. Therefore, every sprite within the `SpriteSet` can be seen as having its own events with a unified event handler on the board level.
2. **Sprite and SpriteSet API** contains the functionality to output and manipulate a sprite's position, color, and rotation angle.
3. **Physics Engine API** is useful to implement games simulating real-world physics. `GameBoard` provides the methods for adding obstacles and walls with different elasticity along with the methods for setting gravity and friction of the medium. The position of single sprites is automatically calculated on every next step using a

built-in differential equation solver and the sprites are automatically animated to fall along the gravity gradient and to bounce from obstacles.

The **GameBoard** type is described in details in a technical report from Microsoft Research (Fahndrich [2012]).

3.10.2 Wall

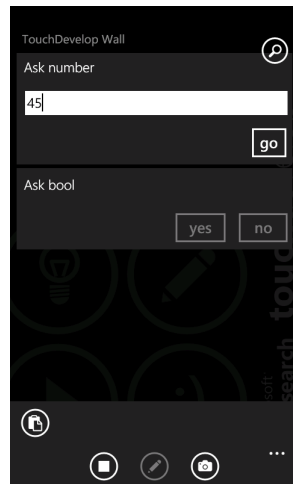


Figure 3.12: Wall with prompts for Boolean and numeric values.

One of the most common types of applications for regular PCs are so-called console applications: they use the system console to read the user input and output their results again to the console. A system console can be seen as a sort of legacy concept for non-distributed operating systems or for systems where the graphical interface is deeply interwoven with the system core, as it is true for most mobile OS.⁶ Mobile phones do not support any kind of system console; to facilitate user interaction TouchDevelop provides a special structure called „Wall” (see Fig. 3.12), which is similar to the personal wall in the famous web-portal Facebook (<http://www.facebook.com/>).

Every type in TouchDevelop supports the `post to wall` property that is used to output the value of the variable of this type to

the wall. Many types (for example **Song**) provide their own implementation to publish their content and to allow some type of interactivity or context-dependent user actions (in case of *Song*, the “wall posting” allows one to play, pause, and resume the song).

For the types without own implementation of this property the conversion to string can be used to display the value of the variable. Depending upon the type of the variable the value can be numeric (for the type **Number**), “true” or “false” for **Boolean**, or just text for **String**.

Besides the possibility to post the variable content on the wall, the wall supports some interaction with the phone user: the posted song can be played, paused, rewound, etc. Similar to the console, the wall supports the input of Boolean, numerical and string literals⁷ using the phone keypad via the `wall→ask boolean()`, `wall→ask number()` and `wall→ask string()` methods. In the upcoming version 2.4 of the language there is a way to use the built-in phone controls for data input directly from the console: `wall→pick date()` and `wall→pick time()` should invoke a date or a time picker.

⁶The only modern mobile OS that still features the notion of the system console is Android by Google, where the system architecture has been completely inherited from Linux and comes originally from UNIX systems. All other OS, like Windows Phone (Microsoft), iOS (Apple), BlackBerry OS (Research In Motion), Baidu (Samsung), Maemo (Nokia), feature no system console.

⁷Despite the fact that there are two different methods for string and numerical literals, the user input is not constrained and there is the possibility to enter any alphanumerical sequence for both inputs. The only difference between these two methods is that the input string is converted into a number before assignment. This conversion fails if there is any other symbol in the string besides numbers and the decimal separator.

3.11 Program execution

A TouchDevelop phone application provides not only support for authoring scripts on mobile devices, rather it also serves as a runtime environment during script execution. TouchDevelop is known to interpret the scripts, providing the mechanism for catching errors during code execution: a pop-up window titled “something fishy happened” and a short explanation of the error appears if the error caused an exception during code interpretation.

There are several aspects of the runtime environment that we had to reverse engineer.

3.11.1 Entry point

A TouchDevelop script can contain multiple public actions, directly runnable by the user. Nonetheless, there is only one default action, which is started when tapping the script icon. This action is marked with a special icon.

The action with no input parameters and with the name `main` is automatically marked as the default action. If this action is missing in the script, the first public action in the list (sorted alphabetically, case sensitive) is selected as the default action.

Should the default action depend on input parameters, these will be asked from the user using an interactive area generated automatically on the wall depending upon the types of the required parameters. However, this is supported only for the predefined value types: `Boolean`, `String`, and `Number`.⁸ Starting the action with reference parameter types directly leads to unexpected behavior: the parameter is passed uninitialized and any attempt to use it causes the runtime system to report unexpected behavior.

3.11.2 Exceptional situations

TouchDevelop does not provide any mechanisms to deal with exceptions in the code except for the invalid property of the variables described in Section 3.6.3.1.

All exceptional situations during the code execution lead to the immediate halt of the execution. Depending upon the severity of this situation, a pop-up message can appear with the text „something glitchy has happened.” After the script has been halted there is no possibility to continue the execution – the runtime data are completely discarded.

3.11.3 Event loops

TouchDevelop as a language encourages the development of reactive applications that execute certain actions as a reaction to user interaction. Those interactions are mapped onto events with self-explaining names (see Section 3.7.6 for more details on events) representing the handlers that execute when a certain event occurs.

The script execution begins with the default action (see previous section) or a user selected action.

After the execution of script actions is complete the TouchDevelop environment enters the infinite event loop by listening to the incoming events and executing the respective event handlers, which are allowed to call other actions.

This means that no event handlers are executed before the execution of the default action (including all actions invoked from there) completes. Until then the script stops reacting

⁸For other types the input controls contain the grayed-out (read-only) full class names from the TouchDevelop implementation: these classes are located within the *Microsoft.TouchDevelop* namespace; due to the size of controls on the phone device there is no way to read the actual (short) class names.

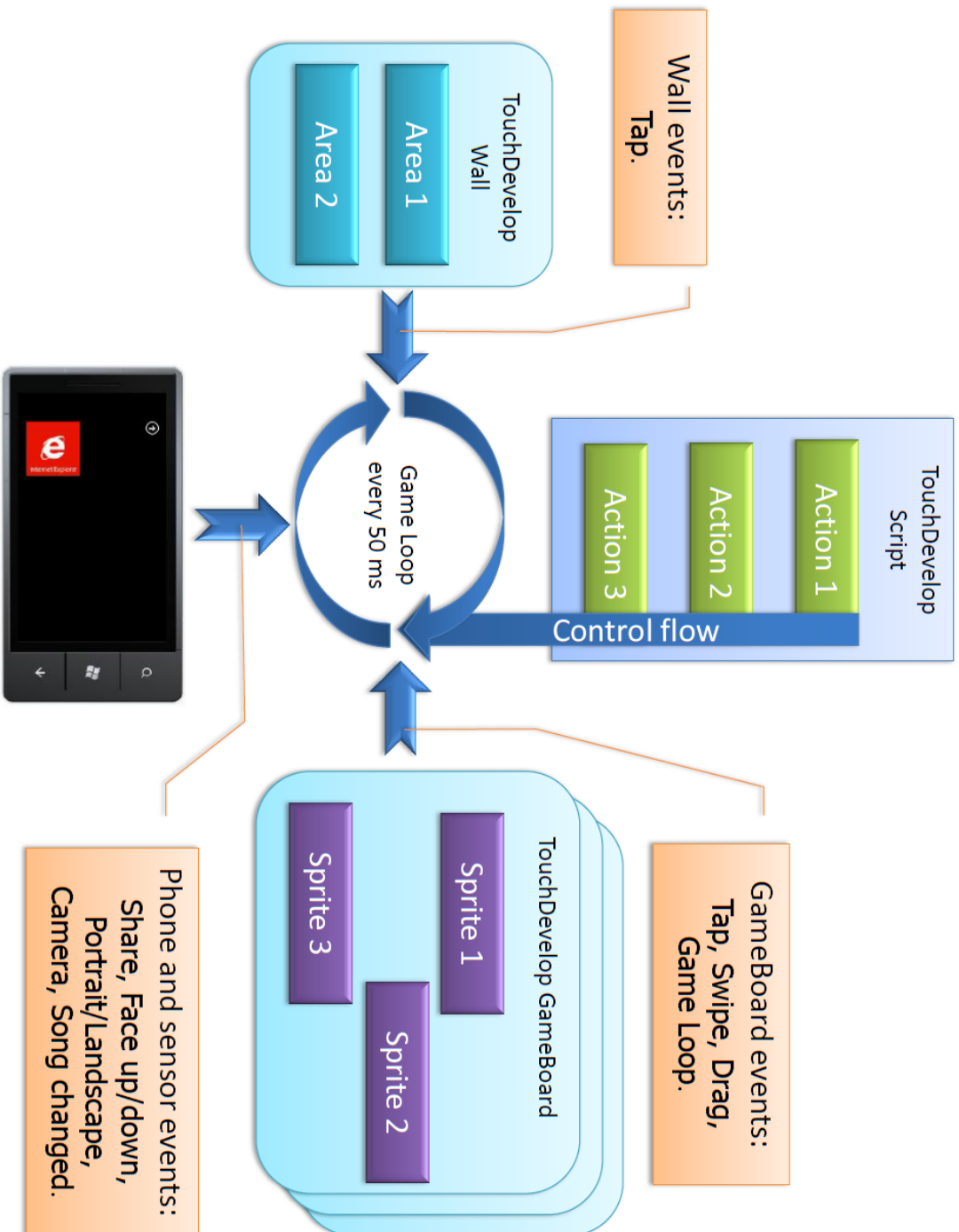


Figure 3.13: Execution model of TouchDevelop.

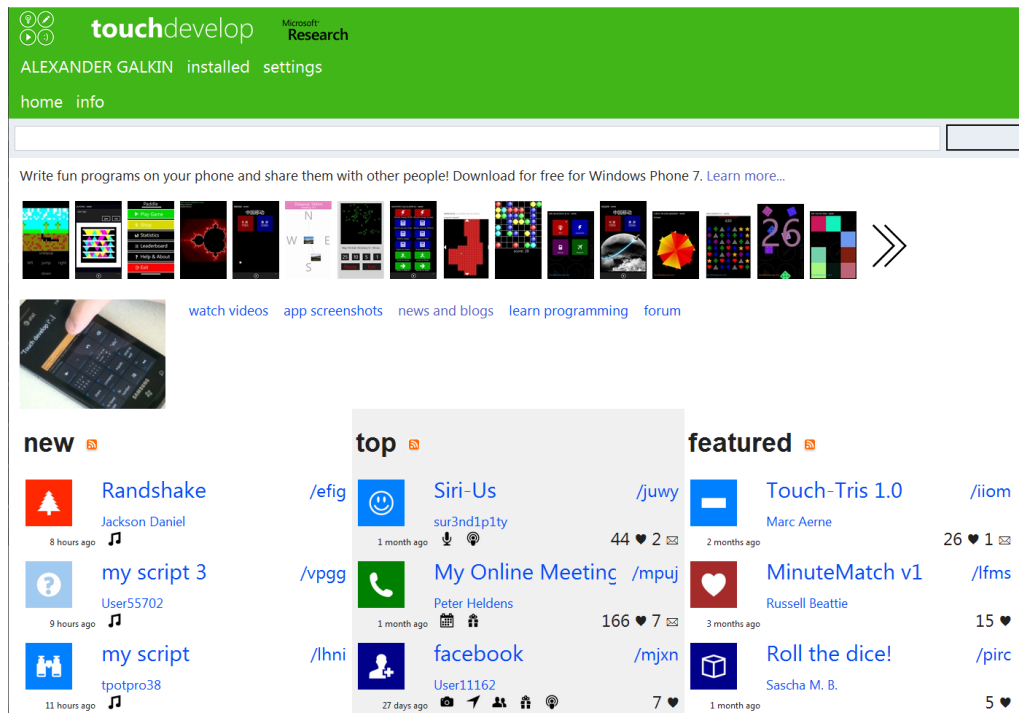


Figure 3.14: Screenshot of the TouchDevelop web portal.

to any user input except for the hardware buttons. Furthermore, all other incoming events are ignored until the execution is over. Since events can call actions holding extensive computations the script might look non-responsive for some time after the event. In case of the `gameLoop` event, which is timer-triggered approximately every 50 ms and is supposed to be used to update the position of graphical game objects and redraw them (similar to the `gameLoop` in XNA) – the long-lasting calculations can easily deteriorate the game performance due to a lowered FPS (frame-per-second) rate.

The event loop is stopped if the executed code calls `time→stop`, if the user presses the “Back” button on the phone, or if any error happens during event handling.

3.11.4 Asynchronous execution

The TouchDevelop phone application was implemented on the basis of Silverlight for a Phone, a managed framework, which, in turn, is based upon a subset of the .Net runtime and class library. To prevent blocking of user interfaces certain actions with unknown execution time are only available as asynchronous calls with callback methods. Examples of the typical asynchronous methods in Silverlight are web request methods, which download content from Internet. TouchDevelop provides the synchronous execution of this calls, awaiting for the data to arrive before a user accesses the value of the web content (similar to the `await` operator in C# 5.0).

3.12 Cloud services

One of the important parts of the TouchDevelop architecture are the cloud services, known under the name Bazaar.⁹ The cloud services keep track of every script, providing the information about script rating, number of reviews, and the total number of script runs. For the derived scripts the complete history is preserved, similar to how it is done by version control systems like SubVersion. The complete statistics on every script, including the number of runs, is then shown on the community web page¹⁰ (see Figure 3.14).

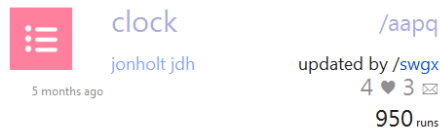


Figure 3.15: Information about a single script on the TouchDevelop web portal. The tile contains information about the script name (top middle), its unique ID (top right), the author (middle), availability of a new version for the base script (middle right), publication date (below the icon), script rating, number of script reviews, and the total count of script runs (right block).

Not only published scripts, but every locally authored script is stored in the cloud for the current user. The cloud contains the complete information about a user's local actions (writing scripts, rating other scripts, or writing reviews for them) and serves as the central repository for the distributed application state. Upon installation of TouchDevelop on a new device (or emulator) and log-in to the TouchDevelop cloud this state is automatically synchronized on all connected devices and all changes will be propagated almost instantly. Newly created scripts are kept private until they are published. The unpublished scripts are designated by a small upper arrow (publishing button) located next to the script

name in the script manager. Publishing the script makes it available for the community for downloading and creating derived works. Every published script is assigned a unique short identifier, which consists of four Latin letters in lower case preceded by a slash (see Figure 3.15).¹¹ A personal page for every published script is available under its ID appended to the URI of the web portal, for example <http://www.touchdevelop.com/lfwq>, providing the options for automatic download of the script to the TouchDevelop application.

In addition to the web-based interface for script sharing and reviewing, the cloud service also provides special services for developers that allow the download of the script information including the complete source code and the pre-processed version of the scripts in form of the abstract syntax tree with flattened leaves in JSON¹² format. This intermediate format can be used to quickly implement third-party tools for interpretation or compilation of the code. However, the flattening of the tree leaves to mere token sequences necessitates statement-level parser. This makes the available webservice much less attractive for tool development.

The web-portal also contains the complete reference for the library functions and data types that are currently available.¹³ Due to the ongoing further development of the language and its continuous changes there is a specification versioning that is independent from the versioning of the phone application.

⁹The first letter is capitalized to distinguish between the singleton class **bazaar** and the actual system behind it.

¹⁰<http://www.touchdevelop.com/>

¹¹The total number of possible scripts in the system is then limited to $26^4 = 456976$.

¹²JSON = JavaScript Object Notation, a lightweight string-based format for storing information about objects, widely used in JavaScript applications for serialization and persistence of objects.

¹³<https://www.touchdevelop.com/help/api>

4

Chapter 4

Implementation aspects

This chapter describes the implementation details of our compiler for the language specification presented above. This is the first compiler for TouchDevelop and its primary goal is to provide a proof-of-concept for the derived language specification. For this reason, the compiler does not have any industrial strength and currently supports only a minimal set of the TouchDevelop standard library.

This chapter follows the roadmap adapted from Aho et al. [1986] and describes the implementation details of the all four steps of this roadmap shown on Figure 4.1, including the description of the CAD-tools used for the derivation of language specification in the first section. The second section deals with the syntax analysis and describes the implementation of a high-performant parser for TouchDevelop. The third section describes the static semantic checker which was implemented as a part of the compiler pipeline. The last section covers the standard library subset, implemented to test the compiler, and the code generation.

4.1 Deriving the language specification

According to the best practices in grammarware re-engineering formulated by Klint et al. [2005] we reconstructed the language syntax of TouchDevelop in a two-step process. A human-readable grammar was first derived manually from existing scripts followed by a manual parser implementation. Instead of the grammar derivation „on paper” as suggested in the publication above, we decided to immediately start with computer-aided approach and used ANTLR for the first step.

4.1.1 Reconstruction of the base-line grammar using ANTLR

ANTLR is a public-domain (licensed under a BSD-compatible license model) parser generator adopted by many educational and commercial facilities, which unites the flexibility of hand-coded parsing with the convenience of a parser generator. Its advantages as a CAD-tool for grammar discovery and reverse-engineering are the following:

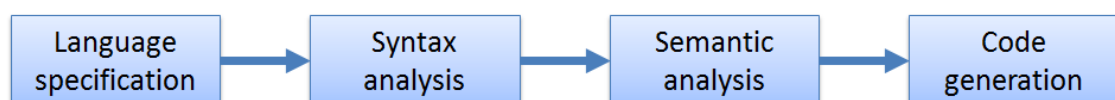


Figure 4.1: Roadmap of a compiler implementation (adapted from Aho et al. [1986])

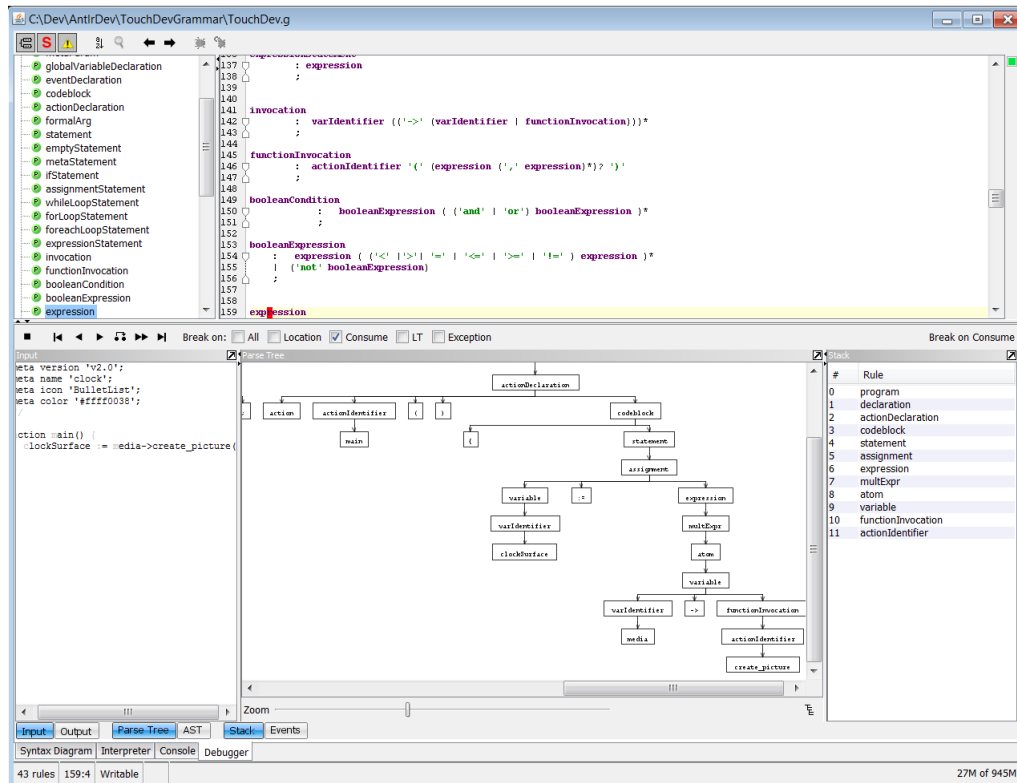


Figure 4.2: Debugging the TouchDevelop grammar in ANTLR.

- ANTLR integrates the specification of lexical and syntactic analysis and generates both lexer and parser. A separate lexical specification is unnecessary because lexemes in form of regular expressions (token descriptions) can be placed in doublequotes and used as normal token references in an ANTLR grammar.
- ANTLR supports an easily readable Extended Backus-Naur Form (EBNF) notation for all grammar constructs.
- ANTLR performs static checks of the grammar to ensure that the rules are not ambiguous and graphically illustrates the offending definitions.
- ANTLR supports grammar debugging by constructing the abstract trees and prompting for parsing shunts.
- ANTLR suggests some grammar transformations to derive context-free grammars even if the original grammars are not context-free.
- ANTLR can automatically test the grammar against the codebase and provide meaningful error messages. The constructed parse tree is very informative and provides valuable feedback for further iterations in grammar engineering (see Figure 4.2).
- ANTLR depicts the grammar rules in form of syntax diagrams and supports the export of those as railroad diagrams. This simplifies the process of creating language documentation and explaining the grammar to non-professionals.
- ANTLR contains a very large set of sample grammars, including those for Java, C/C++, Python, and JavaScript. This grammar corpus significantly facilitates the development of own grammars providing the possibility for copying the existing patterns and applying the best practices from other grammars.

The base-line grammar for TouchDevelop was successfully re-engineered. This grammar, specially designed for human readability, required two tokens of look-ahead and contained two left-recursive rules. We decided not to refactor the recursive rules, because that would result in a less readable grammar definition. To overcome the problem with left recursion we instead used the backtracking option for parser generation. This enabled us to successfully generate the parser and test it against the preselected scripts for correctness.

ANTLR Studio is a tool that is primarily concerned with the concrete syntax of a language. We had difficulties with the definition of how the abstract semantic tree should be constructed for binary operations with different precedence and associativity (expression parser). This problem did not impede the derivation of the language syntax for the base-line grammar and we did not face this problem during the manual implementation of the parser, as explained in the next subsection.

It is a usual practice in ANTLR is to include some piece of code (which is escaped as comments in the grammar file) to control the AST generation and to keep track of certain metadata, like the nesting level of block scopes, etc.

During the process of reverse-engineering of the base-line grammar in ANTLR we faced the following problems:

- ANTLR does not support the testing grammar against a file list: only a single file could be tested at a time.
- It is rather difficult to implement an expression parser for operators with different arity, precedence and associativity.
- ANTLR grammars provide support for Unicode terminals, but generated parsers do not: the automatically generated parser code fails to compile and with numerous compilation errors and warnings.
- The only working target for the parser generator was Java. Several different C# parser generators, including the built-in and customly downloaded ones from the ANTLR webpage, did not bring any results: the generated code could not be compiled, mostly because some referenced classes were obsolete.
- The generator boilerplate code was not easily readable and any, even subtle, modification thereof required considerable efforts.

We used Java as the target language for parser generation in ANTLR. To overcome the problem with Unicode symbols while debugging and testing the grammar specification, we manually substituted all Unicode symbols used in TouchDevelop scripts (see Table 3.6) by their non-Unicode alternatives.

4.2 Parser implementation (syntax analysis)

Despite the parsing power of *LR* algorithms in general and *LARL*(1) in particular, the available tools (YACC and its custom implementations) usually feature a command-line only support for grammar input; no support exists, e.g., for stepwise grammar authoring, debugging, or testing. Besides, automatically generated parsers require some customization to correctly handle Unicode input that is also crucial for TouchDevelop scripts.

Therefore, programmers often choose to implement recursive-descent parsers by hand aiming at flexibility, better error handling, and ease of debugging. Manual parser implementation becomes especially difficult when it goes hand-in-hand with grammar hacking, i.e., if the grammar knowledge is not complete by the time the parser implementation started and the parser is used to derive the syntax (and to some extent also the semantics) of the parsed language.

Following the principles by Klint et al. [2005], we did not try to optimize the reverse engineered grammar to proceed with the parser implementation. Instead, we started a complete new implementation with the goal to implement an effective parsing algorithm. To reach the maximal parsing performance we have chosen to implement a *predictive parser*, that is, a top-down recursive descent parser with one token look-ahead ($LL(1)$) and no backtracking.

One of the limitations for the parser implementation was the compatibility with the .Net platform, because:

- Windows Phone OS supports only Silverlight (as a subset of .Net runtime environment) for mobile application, native code is not allowed to run within the context of a user application. So, for the compiler to run directly on Windows Phone OS it has to be implemented in .Net.
- TouchDevelop itself is entirely based upon .Net, therefore the use of the same platform would allow us to have the same dynamic semantic. For the parsing stage that mostly concerns the support for Unicode: we used the .Net library for all operations with Unicode texts.

Unfortunately, traditional .Net languages, like C# and Visual Basic .Net, are biased towards the object-oriented programming paradigm and are not as flexible as functional languages for compiler implementation: they lack many features that would be useful for a parser, like algebraic data types, pattern matching, and type inference.

Taking this rationale into account we decided to use F#, a relatively new functional programming language for the .Net platform from the ML family, as the core language for both parser and compiler implementation. The language is greatly influenced by OCaml and provides support for all three above-mentioned concepts.¹

To reach a succinct and easy readable implementation we used a parser combinator library for F# called FParsec.

FParsec is an F# implementation of the famous parser combinator library Parsec for Haskell, originally implemented by Daan Leijen from Microsoft Research [Leijen and Meijer, 2001]. While the implementations of Parsec and FParsec are completely different, they share a similar top-level API. Unlike Parsec, FParsec discourages developers to use imperative monadic syntax for parsers and encourages the use of static combinators for parsers. This results in much cleaner parser code with improved readability and an optimized, fast parser. Besides, FParsec also has the following features, useful for the manual parser implementation for TouchDevelop:

- A permissive open source license (simplified BSD license), which allowed us to use it for this diploma project.
- Full Unicode support both for parsers and for text input, which was crucial for TouchDevelop that uses Unicode in keywords (refer to Table 3.1).
- An embeddable, flexible and highly configurable operator-precedence parser component with support for pre-, in- and postfix operators as well as operators of different arity (unary, binary and ternary), which helped us to solve the problems we were facing with the ANTLR auto-generated parser.
- Full support for medium-trust environments for parser execution. This is especially important because one can run the implemented parser in Silverlight runtime environment.

Other important features of FParsec that helped us during the parser development but are

¹Following the established terminology for this language, algebraic data types are called discriminated unions in F#. These constructs have the same properties as the classical algebraic data types in Haskell, including the fact that each constructor is a full-fledged first-order function.

not specific for TouchDeveloper are:

- support for context-sensitive, infinite look-ahead grammars,
- automatically generated highly readable error messages,
- comprehensive documentation with a well-thought tutorials and user manual.

For these reasons the second version of the parser was manually implemented in F# using FParsec. We used LINQPad for code prototyping and Visual Studio 2010 Ultimate for debugging.

The definitions of the algebraic data types used for abstract syntax tree are provided in the Appendix, on page 94.

The final version of the parser comprises about 200 lines of code plus 50 lines for the algebraic types definitions that are used to construct the abstract syntax tree.

This parser successfully parsed 278 out of 282 sample scripts from the code base, requiring less than a second for 35075 lines of code. The four scripts that failed to be parsed were found to contain syntax errors.

4.3 Semantic analysis

The static semantic checker is implemented as a higher-order function in F# that evaluates the abstract semantic tree from the parser and statically checks its validity using a ruleset, implemented as first-order functions. To facilitate the rule checks on the level of statements and expressions we performed an additional transformation of the respective branches of the syntax tree into another data type, `CodeEntry` (see the listing of AST types in the Appendix, page 94).

The semantic checker is implemented to support three different rule-sets with different severity level: hint, warning, and errors. Due to time limitations only the support for the error rules was implemented.

The following rules were checked (for details see the error messages of the static checker in the Appendix, page 95):

- Action, local, and global variables must be declared before first use.
- Global variables, actions, events, and meta declarations must be declared only once.
- Parameter of events and actions must have disjoint names.
- Conditional operators must have the condition of type Boolean.
- The `for each` statement can only be applied to the type of collection type (implementing an `IEnumerable<>` interface in .Net).
- The left-hand side of assignment must not be read-only.
- Only local variables can be on the left-hand side of an assignment statement.
- Loop variables in a `for` loop are not accessible within the loop body.
- The `if` statement must have a non-empty `then` branch.

If one of the rules is violated, the semantic check fails and the compiler pipeline is aborted.

4.4 Code generation

Before to proceed with the compiler implementation we reviewed the existing approaches to code generation for .Net. Because of the complexity of the .Net binary format and the existence of many different libraries for code generation that hide the low-level details of the executable format, we did not consider the manual generation of the executable code. The comparison of these approaches is given in Table 4.1. We decided against the use of

Approach	Examples	Input	Output	Remarks
Use of existing .Net compilers for C# or other languages	MS Compiler for C# (cs.exe) Mono Compiler	C# code as plain text file	Compiled assembly with attached metadata	This approach requires the literal code translation from the source language towards C#. The MS compiler is manually implemented and provides no <i>ad hoc</i> possibilities to cut in or to monitor the compilation process. Mono compiler is open sourced and has a special API
Use of „compiler compiler” libraries for .Net platform	CCI (Common Compiler Infrastructure)	AST	Compiled assembly with metadata	This is a flexible approach that requires the transformation of the source code into a CCI-compatible syntax tree. CCI was developed by Microsoft for internal use and comes with no documentation.
Use of „compiler as a service”	Roslyn	plain text in original language or C#, AST	Compiled assembly, compiler warning or refactoring suggestions.	Roslyn theoretically allows the extension of the existing parser and compiler to directly support all necessary language features of TouchDevelop. The framework is new and is under active development, some major changes might break the code.
Use of the CodeDom provider	C# CodeDom	CodeDom AST	Compiled assembly with metadata	CodeDom is a native part of .Net framework and provides the way of language-agnostic code generation by supplying the code representation in an intermediate form.
Generating code using low-level .Net classes	Reflection.Emit	nothing	Single statements in intermediate language or metadata	This approach is the most flexible one and can be used to generate code for any input language. The code generator needs to be customly implemented.

Table 4.1: Ways to generate executable .Net code.

compilers, because the C# compiler from Microsoft, currently implemented in C++, does not provide any public API to monitor and manage the process of compilation. Besides, the implementation is not modular and there is no way to bypass the compiler front-end. Therefore, a code-to-code transformation from TouchDevelop to C# would be needed.

The Common Compiler Infrastructure (CCI) is an open-source project² from Microsoft Research, a set of libraries and an API that supports some of the functionality that is common to compilers and related programming tools. It is a compiler compiler for .Net with support for static code verification and code re-writing. Despite the attractiveness of this approach, the lack of any developer documentation and the abandoned state of the project were the decisive points for not selecting this option.

The current successor of the CCI project is the ongoing Microsoft initiative „compiler as a service” to provide a managed open-source version of the .Net compiler designed with the ideas of modularity and extensibility. This initiative, known under the working name „Project Roslyn”, has produced a community technical preview (CTP) of this compiler libraries (bearing the same name as the project). Despite the CTP status of the project the project supports customization of the language parser and has a sizable community. Due to the premature status of this project and in attempt to minimize the number of external dependencies we decided to rather consider some of the approaches immanent to .Net.

The .Net foundation classes library (FCL) provides two different APIs for code generation. The low-level API, provided as the all-mighty class `Emit` within the `Reflection` namespace (and therefore often referenced to as `Reflection.Emit` API), is often used by third-party compilers to generate valid .Net binaries. Examples of such compilers are Phalanger³ (.Net compiler for PHP) and the Iron languages, including Iron Python⁴ and Iron Ruby⁵. All these tools implement a manual code generator and use the `Emit` class to generate the assembler code in the Common Intermediate Language (CIL).

A high-level API in .Net is represented by the `CodeDom` namespace (see Table 4.3). This namespace contains interfaces and classes for abstract syntax tree representation as well as implementations of several code providers that convert the AST either into the source code or to the .Net binary. The namespace was criticized as biased towards the object-oriented paradigm and provides less support for non-OOP constructs and phenomena.

Having critically reviewed the low-level and the high-level approaches to the code generation, we decided to use `CodeDOM`, because this high-level framework contained all necessary classes to represent the TouchDevelop scripts. It allowed us to confine the compiler implementation to the implementation of the transformation function from the AST to `CodeDOM`, as described in detail below.

4.4.1 TouchDevelop standard library

To test the compiler we implemented a minimal subset of the TouchDevelop standard library that meets the following requirements:

- The subset must contain at least one instantiatable and one singleton TouchDevelop type.
- The subset must provide implementation for every operation (both binary and unary) and implements the types for these operations.
- The subset must contain the types and methods whose implementation can be safely

²<http://cciast.codeplex.com/>

³<http://www.php-compiler.net/>

⁴<http://ironpython.net/>

⁵<http://www.ironruby.net/>

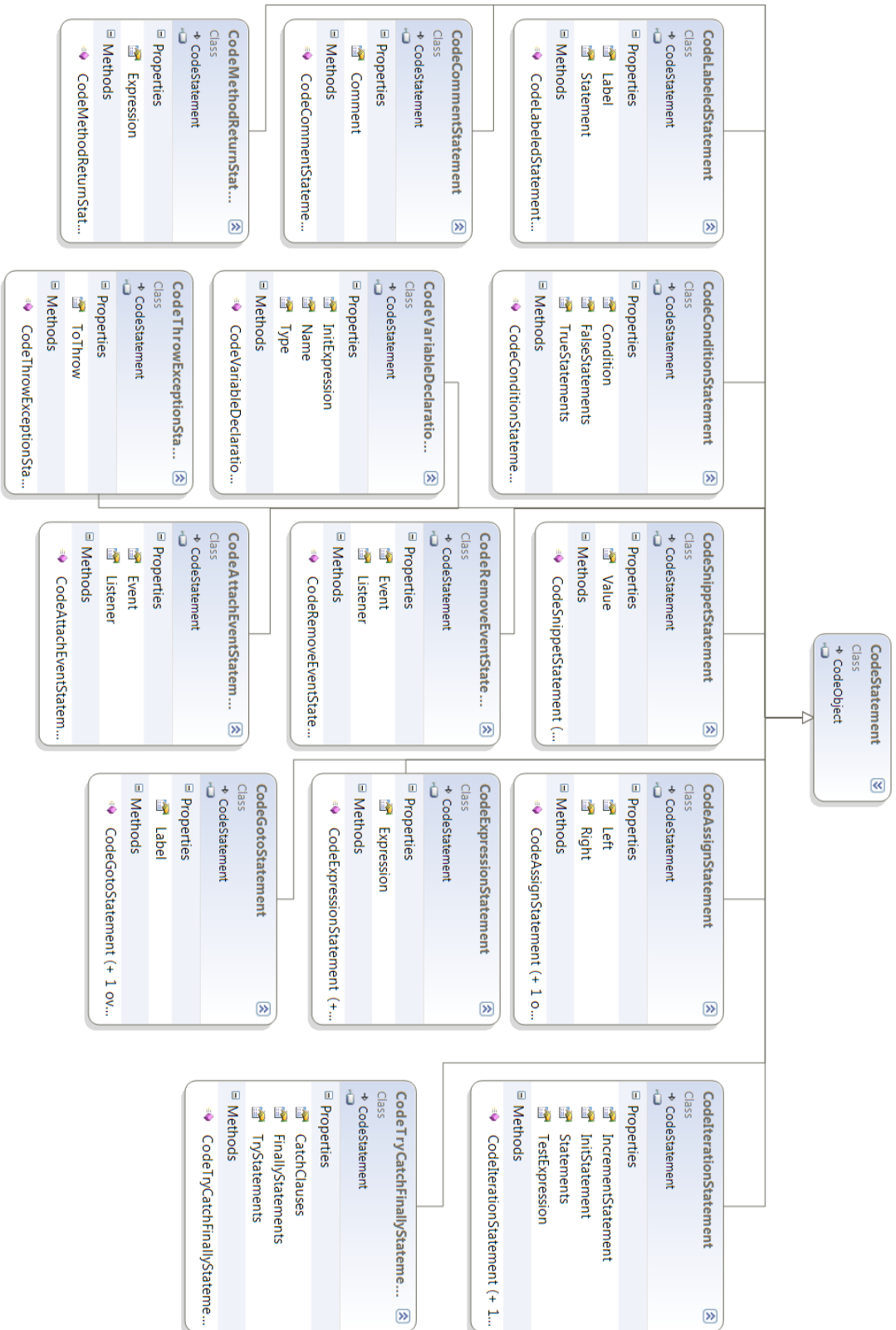


Figure 4.3: Classes of the CodeDom syntax tree.

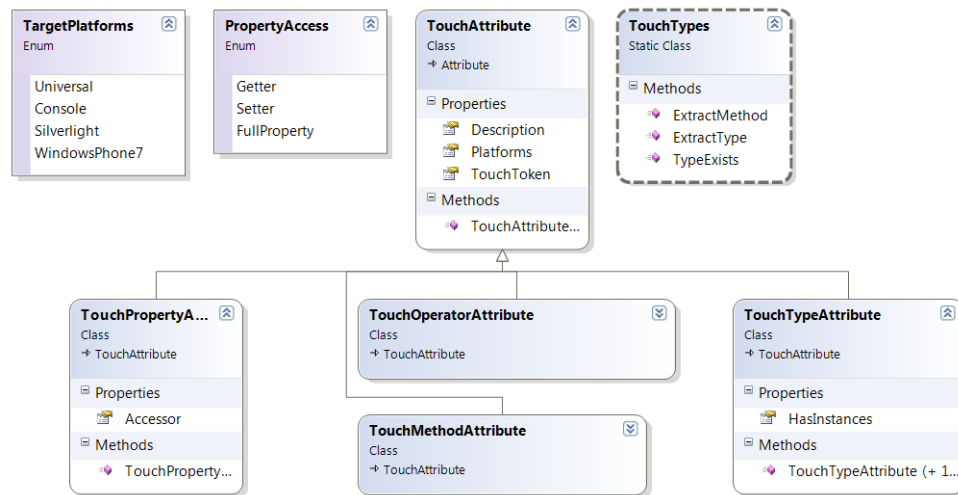


Figure 4.4: The class diagram of compiler library attributes.

shared between different compilation targets, as well as target-specific implementations.

- The subset must contain the implementation of at least one Unicode action identifier that cannot directly represented using ASCII character set. Ideally, several types of Unicode characters (subscripts and superscripts, compound characters) have to be present.
- The subset must support the basic input and output operation for user interaction.

The following five types were implemented as a minimal set of a TouchDevelop library for compiler tests: `Number`, `String`, `Boolean`, `math`, and `wall`.

The library was implemented in C#, the flagship language of .Net platform which is close in its semantics to TouchDevelop.

During the implementation of the standard library we had to overcome the following limitations:

- .Net does not support Unicode in identifiers, however we needed this support because Unicode identifiers are widely used in the TouchDevelop standard library.
- To implement the multi-targeting in the compiler we had to provide alternative implementations for some methods. For example, the console application target uses the system console as a wall object, but Silverlight does not provide any support for a console and we had to instead use a static class to hold the output.

To overcome these challenges we implemented a set of attribute class that contain meta information about the TouchDevelop token names, the instantiability of a type, the compiler target, and optionally the description. The class diagram for these attributes is presented in Figure 4.4.

We used class and methods decorations with our custom attributes to provide information about the mappings between the C# implementation of the library and the TouchDevelop syntax (see Figure 4.5). Thereby we were allowed use Unicode in strings and could map our C# implementation to the TouchDevelop library invocations.

Constructs	AST type ^a	CodeDOM
Top-level of the script, script scope	TopLevel	CompileUnit with the static class TouchScript and standard library reference
Meta declarations and statements	MetaDeclaration, MetaStatement	not implemented
Global variables	Global	Static fields
Actions and events	Action, Event	Static methods
Block statements	Block	CodeStatementCollection
Multiple return values and multiple assignments	Action, Assignment	Out parameters of the static methods
if statement	If	CodeConditionsStatement
while and for loops	While, ForI	CodeIterationStatement
foreach loop	ForEach	not implemented
Single assignment	Assignment	CodeAssignStatement
Expression statement	Expression	CodeExpressionStatement
Unary and binary operations	BinaryOperation	CodeBinaryOperatorExpression

Table 4.2: Transformations performed by the TouchDevelop compiler.

^aSee the Appendix on page 94.

```

[TouchType("Number", TargetPlatforms.Universal, HasInstances = true)]
public class Number : TouchBase
{
    [TouchOperator("-")]
    public static Number operator - (Number left, Number right)
    {
        return left.value - right.value;
    }

    [TouchOperator("*")]
    public static Number operator *(Number left, Number right)
    {
        return left.value * right.value;
    }

    //....
}

```

Figure 4.5: Attribute decorations in the implementation of the TouchDevelop standard library.

4.4.2 CodeDOM transformation

We could confine the compiler implementation to a transformation function that maps the AST of a TouchDevelop script onto the CodeDOM representation. This was possible because of the following properties of TouchDevelop:

- TouchDevelop has a syntax largely based upon the syntax of C#.
- TouchDevelop provides full support for procedural programming and only limited support for object-oriented programming. We did not have any difficulties to map the procedural aspects of TouchDevelop onto object-oriented model of .Net and C# (see Table 4.2 for exact mapping).
- CodeDom provides the implementation of the static semantics for C# and we had only to cover that part of the semantics of TouchDevelop that is different from the basic semantics already covered by the C# static semantics. For example, we did not have to implement the semantics of an assignment (including all necessary checks like check for the existence of the assigned variable, type check etc.), but had to deal with the multiple assignment which is not a part of .Net or C#. We used the output parameters of .Net framework to return several values from a method before assigning the variables.
- The dynamic semantics of the .Net runtime is used for the script execution. For example, we do not perform the checks for division by zero in our implementation, the .Net runtime throws an `DivideByZeroException` during the script execution if this situation happens.

The compiler was implemented as a single higher-order F# function that takes the abstract syntax tree (see Appendix on page 94) and transforms this tree into the respective tree of CodeDOM.

4.4.3 Multi-targeting of the compiler

TouchDevelop provides the possibilities to author and run scripts on Windows Phone device where only Silverlight⁶ applications are allowed to run. For the compiled scripts to run in the same environment one must consider the constraints of the Silverlight runtime, including the following:

- Silverlight is a medium-trust environment where only certain subset of .Net instructions is allowed (for example, no pointer arithmetics is possible here),
- The Silverlight code does not have full access to the drive. Only a small part called isolated storage can be accessed.
- Applications run in isolation in a single process. No inter-process or inter-application communication is possible.
- Silverlight runtime has a preemptive execution model with tombstoning events used to put running applications to sleep. Every application has to serialize its state on this event to its internal storage and re-read this state upon continuation.
- Application code along with the dependencies has to be put in a special XAP package that is digitally signed before deployment on the device. Only signed code is allowed to run.

These aspects of the Silverlight runtime were, however, beyond the scope of this work. Therefore the implemented the compiler was designed to support multi-targeting.

The default compilation target is a .Net 4.0 console application which does not have to deal with the constraints imposed by the Silverlight runtime. As a console application is has access to the system console which was used for the input and output operations and for interaction with user. The interaction with the console were encapsulated in the `wall` type which was specially implemented for this compilation target.

The support for the second goal, Silverlight for Phone, is implemented at the level of both compiler and standard library.

We used special attributes to decorate the types and methods in the standard library either require no special implementation for Silverlight runtime (we call them „universal”) or provide a special Silverlight-tailored implementation.

To avoid the problems with correct packaging of the generated code we implemented the support only for the library output. This means, that the source code is compiled into a Silverlight dynamically linked library (DLL) that can be then used in any Windows Phone application.

4.4.4 Limitations of the compiler

We compiler only implements the transformation of the abstract syntax tree and does not perform the code generation itself. The abstract syntax of CodeDOM can be either compiled to the .Net assembly⁷ or converted to the source code (text representation) using a special language provider for CodeDOM. The .Net FCL contains the CodeDOM providers for C#, VB.NET and JavaScript. F# CodeDOM provider is available as an open-source implementation.

The current version of the compiler has the following limitations:

- The compiler performs the transformation in a single pass. Therefore it cannot

⁶Under Silverlight we everywhere mean the Silverlight for Windows Phone.

⁷Assembly is a binary code for .Net (the bytecode for the Common Language Runtime). Depending upon the meta information (the presence of an entry point) an assembly can be stored as a PE (portable executable) file (.exe) or a dynamically linked library (.dll).

compile the code references (actions, global variables) that are declared in the script after they are referenced. The direct recursion is possible: the action signature is treated as its declaration.

- Only the code translation is performed. There is no support for the functionality provided by the TouchDevelop runtime environment. In particular no support for the event model and for tombstoning is provided.
- Some constructs of the TouchDevelop, like meta statements and **foreach** loop are currently not supported. This means that every action is mapped onto a public static method irrespective of its visibility in TouchDevelop.
- The executable file is only created when compiling against the .Net 4.0 framework. The compilation for Silverlight runtime yields the not directly executable library code.

Despite the limited functionality, this is in fact the first compiler implementation for TouchDevelop⁸. For the completing of the second goal of the current work it was important for us to provide an implementation of the complete compiler pipeline, as it is depicted on Figure 4.1. The existing implementation provides the basic framework for the compilation of the TouchDevelop scripts which can be always extended to accommodate the latest changes in the TouchDevelop syntax and semantics.

⁸It should be noted here that the idea to implement a compiler for TouchDevelop was suggested by Nikolai Tillmann to address the demands of many TouchDevelop users who would like to publish their TouchDevelop scripts as standalone applications on the Windows Phone Application Market. This idea was, however, undermined by the possibility to create standalone executables for Windows Phone that contain the TouchDevelop scripts along with embedded TouchDevelop interpreter and runtime, which was recently made available on the TouchDevelop webportal.

5

Chapter 5

Discussion

In this chapter we identify potential threats to validity and life time of the language specification presented in Chapter 3 and evaluate their impact.

We also assess the current specification in comparison to other scientific publications on TouchDevelop. A particular emphasis is put on the analysis and review of the TouchDevelop manual published by the original authors of TouchDevelop very recently [Horspool et al., March, 2012].

At the end of the chapter we speculate on the design decisions for TouchDevelop as programming language, runtime environment, and integrated developer environment (IDE).

5.1 Threats to validity

In this thesis we made an attempt to re-engineer a complete and accurate description of TouchDevelop syntax and semantics. Despite best efforts we admit that there are reasons for some concerns about the completeness, applicability and timeliness of the derived language specification.

The language specification was reconstructed using the source code corpus containing 284 TouchDevelop scripts downloaded in December 2011 from the TouchDevelop webpage and the TouchDevelop application installed on a Windows Phone device around this time. More recent updates to TouchDevelop that add some new features to the language and runtime environment are not covered in this specification (in Table 5.1 the double line demarcates the changes that are not captured by the current specification).

5.1.1 Threats to the correctness and completeness of this specification.

Reverse engineering is an approach for which it is very crucial but at the same time also difficult to reason about the correctness of results. In our particular case, this reasoning was even impossible due to lack of an existing specification.

We had to use the existing phone application along with the downloaded script files to perform the reverse engineering of language syntax and semantics. It has to be pointed out that this application is not a compiler; rather, it can be regarded as a thick client for the cloud infrastructure with support for script authoring, syntax and semantic checks, and the execution of these scripts in interpreter mode. This work presents an attempt to generalize the acquired knowledge to an implementation-agnostic specification.

In addition to the challenges to specification mining mentioned in Section 3.1, the following peculiarities of the system under analysis could have contributed to eventual inexactitude

Ver	Most important changes in the language and the libraries	Date
v2.0	The project was renamed to TouchDevelop. TouchDevelop website with cloud support launched.	2.08.2011
v2.1	Support for Leaderboards. Full support for IEEE 754-2008 for float-point arithmetics. String selection from preset list of string („string picking”) on the app wall.	18.08.2011
v2.2	Support for Windows Phone 7 themes querying. Support for data arithmetic and custom data object creation with Invalid . Access to built-in media library objects (images and image names). Cropping of picture object.	13.09.2011
v2.3	Support for Windows Phone calender objects. Access to Bing maps and driving directions. Access to Motion API of Windows Phone, querying the presence of gyroscope and compass and requesting their readings. Access to contact address, saving a new contact locally and search for appointments and contacts using Social Networks API of Windows Phone. Support for internet-based media. Access to songs collection, support for song playing. Support for clipboard (only copying). Working with application tiles. Previous APIs for working with social networks, application tiles and contact collections are now obsolete. Dropped support for panorama tiles.	7.10.2011
v2.4	Board supports camera output as background. Camera settings can be programmatically set (via <i>Camera</i> class). Support for color manipulations (darkening, lightening) and for HSB (hue-saturation-brightness) color model. Full support for contact creating and editing, access to all built-in contact properties.	11.11.2011
v2.5	Small improvements in picture, song, playlist and social networks functions (support for random picking). Improved app wall with date/time user input and background picture. Some Bing searches (phone numbers) and Bazaar options are now obsolete.	20.12.2011
v2.6	Support for external libraries in TouchDevelop scripts. New events for items on wall. New API to access devices in home network.	21.02.2012
v3.0	Support for TouchDevelop query language is expected.	tbd

Table 5.1: Overview of the specification versioning for TouchDevelop as language.

of this document:

1. *Limited support for script authoring.* The only way to implement a new script for TouchDevelop is to author it on a mobile device. Every script has to be manually typed on a virtual keypad and tested, no automatic testing of (randomly) generated script code is possible.
2. *Non-transparent storing and serialization of the scripts.* The tokenized input of the user scripts allows the mobile application to parse the syntax of the script in a much easier way than the plain text with source code. The text version of the script requires several script transformations along the following change chain: mobile device \Rightarrow cloud \Rightarrow text version.
3. *Limited possibilities to debug and test scripts.* There is no built-in debugger that would allow one to peek the values of local and global variables or examine the properties of script objects. The only way to debug the scripts is to output critical values either to the wall or as a pop-up message, which means that the semantics of the script execution is only „palpable” through the semantics of the wall object or pop-up message.
4. *Limited information about errors in scripts.* Despite the static analysis of the script code and some helpful error messages and quick fixes („fix-it” option of the IDE, see Figure 3.4) covering many syntactic and some semantics features of TouchDevelop, many errors are revealed only during execution and are not reported by the code editor. Therefore, to reverse engineer the dynamic semantics of the code we needed to run every script we used.
5. *Limited possibilities to distinguish intentionally loose design from potential bugs.* Having encountered an unexpected or strange behavior of TouchDevelop one can never be sure if this is loose design decision, a bug in the language design or a bug in its implementation. Therefore, this document describes potentially unexpected behavior as it was observed.

5.1.2 Threats to the life time of this specification

As every project under active development, TouchDevelop undergoes rapid changes in the design of the Windows Phone UI, the standard code library, and the language itself. The high pace of project development casts concerns about the life time of this specification and especially about obsoleting of certain parts of our specification. We identify the following threats to the validity of the document:

1. *Rapid development and short iteration time between versions.* Table 5.1 contains information about the most important changes in every public version of TouchDevelop since version 2.0. The average time interval between two subsequent versions comprises one month, meaning that every month some new features are published. This is an extremely high tempo for a developing environment. Taking into account that the final goal declared in the first draft of the language and platform (Tillmann et al. [2011]) has not been reached yet (see the last line in Table 5.1) it is expected that the tempo will stay high until version 3.0 of TouchDevelop is here.
2. *Opaqueness of the development process.* The platform is developed by a group of scientists from Microsoft Research. The leader of the group, Nikolai Tillmann publishes updates with detailed overview of the new features after every public release in his official blog. This blog remains the primary source of information about the TouchDevelop development progress: there is neither any public code repository, nor a feature requests page. The webpage of TouchDevelop provides minimal support for

a forum, but this is bound to users and their scripts and is intended to discuss the existing scripts rather than the evolution of the platform.

3. *No specification or any public information about serialization of TouchDevelop scripts.* If the rapid development of the platform is balanced by the ever growing user communities and the necessity to maintain backward compatibility to the existing script pool, the serialization of the TouchDevelop scripts authored on a mobile device remains uncovered and can be changed at any time. This is the serialization that accounts for three different source code views. Any changes to the serialization, even subtle ones, might have a drastic impact on the syntax of TouchDevelop and require duly adaptation of parser.

Because it is impossible to keep pace with the steadily evolving platform, the complete specification is derived from version v2.4 of the phone application. Newer changes, as shown in Table 5.1 with a double demarcating line, are not reflected here.

Putting aside the improvements in the user interface and new capabilities in the standard library, the major compatibility-breaking change in the latest version of TouchDevelop is the support for external user-defined libraries. As of today, third-party scripts in the TouchDevelop cloud can be referenced from a user program. This allows one to invoke the public actions from these scripts in the user code, similarly to the libraries in procedural languages. With the current support for libraries, only actions of referenced scripts are available, but the developers divulged the plans to add support for custom collections exported as tables (Horspool et al. [March, 2012], p. 111). This example shows that no definitive specification can be provided at the moment and the life time of every specification, even that provided by developers themselves, is limited.

5.2 Comparison to existing works

In this section we compare our specification against the currently existing publications on the TouchDevelop language and environment.

As of today, there are two publicly available descriptions of the TouchDevelop language (besides this document) that can be seen as a sort of language description. Those are the original publication of Microsoft Research Group (Tillmann et al. [2011], available both as conference submission and as technical report) and the user manual „TouchDevelop: Programming on a phone” (Horspool et al. [March, 2012], self-published in electronic form on the project webpage), written in collaboration with the University of Victoria.

The first publication, dated back in spring 2011, contains no formal description of the language. The paper describes the platform as a whole, with detailed coverage of the user interface on mobile devices as well as the cloud service. The language specification is short and mostly deals with the syntax. The syntax rules for almost every language construct include ellipses implying the rules are incomplete and potentially have more alternatives besides the given ones. Out of four pages dedicated to the language description, one page covers the query language, similar to LINQ in C#, that is to be implemented in the next major version of the system (see the Table 5.1). The goal of the article was most likely to show the place of TouchDevelop among developer environments for mobile devices, that is, the exact description of the language was not the main focus of the publication. Therefore, we will not further consider this publication and will focus on the second publication we will reference to as „published publication” thereafter.

The second publication, which appeared very recently (March 2012), looks more like a draft of the specification than a finished work. Besides numerous spelling and formatting

glitches throughout the text, it contains many commentaries that belong to the internal communication of the group like „TODO: add reference” (p. 110), „TODO: maybe we want to revise this?” etc. Despite its draft style, this publication goes much deeper in describing the language syntax and semantics with the last chapter titled „TouchDevelop language specification.” This chapter on the one side describes the language in a very brief form, but on the other side provides no explanation for certain terms and thus requires the reader to refer to previous chapters to find necessary clarifications or details. For example, value and reference types are just briefly touched here, more information is available in Chapter 3 (pages 37-38), whereas the exact list of value types is given only in Chapter 6 (page 64). We identified 16 potential inaccuracies in this language specification that were summarized in Table 5.2.

One of the main problems of the specification by Horspool et al. [March, 2012] is that it describes mostly the unified syntax of TouchDevelop scripts without paying any attention to the serialization algorithm and any different views to the source code. This leads to the problem that some described syntax rules cannot be applied to the source code in all of the available code representations.

Every communication between the user mobile device with an authored script and the cloud requires serialization and deserialization. In its attempt to keep this transformation private (probably with the goal to be able to adapt it any time to the ever-growing needs and wishes of the user community), the specification by Horspool et al. [March, 2012] loses its precision: it describes the syntax neither as it is directly seen on the screen of a mobile device nor as it is rendered by the cloud-connected web portal. We try to tackle this problem by providing the description of different „views” (refer to Section 3.4.1 for the explanation on views) to the script source code and describing the source code as it is serialized to a plain text file.

5.2.1 Syntax issues

Focusing on an unified language syntax the specification by Horspool et al. [March, 2012] does not define how single declarations or statements are separated: a mobile application uses new lines and wraps longer lines with additional indentation to show the statement (or declaration) continuation on the next line. Other views explicitly use semicolons as separators between expressions (including expression statements) and meta declarations, and these semicolons are clearly visible in the browser or in the plain text file.

Probably for the same reason („unified syntax”) the syntax rule for a conditional `if`-statement declares the `else` block as mandatory. This might be correct in the phone view where an empty `else` branch provides a tapping area to add some code, but this branch is almost always omitted if empty in the web view.

The specification uses the term „keyword” (page 109), but does not provide any list of keywords or explanation how to deal with name collisions. The only phrase „any name can be used for an action” (page 110) is not entirely correct: as long as the name of an action collides with a reserved word or an already existing action, a running number is appended to this name automatically. Therefore, there is no possibility to overload existing actions by providing the same name but different signature, as one might suppose from the quoted statement.

Several production rules contain the non-terminal symbol `identifier` on their right-hand part, but this symbol is never defined in the specification, neither syntactically (for allowed symbols) nor semantically (for case sensitivity, maximal allowed length etc.). The results of our mining experiments show that not every ASCII symbol is allowed in local identifiers

Category	Issues
Spelling mistakes that might result in misunderstandings.	The single pipe symbol () is used instead of the „parallel for” Unicode sign or the double pipe () for string concatenation.
Syntax	<p>Syntax rules include curly parentheses to demarcate code blocks (as in the web view), but do not include semicolons as statement separators (also present in the web view).</p> <p>There is no syntax rule to describe a valid variable or method identifier (even though the non-terminal rule with the name identifier is widely used throughout the specification!), whereas some symbols (like asterisks, signs of arithmetic operations etc) are not allowed in them.</p> <p>The else branch of the if-statement is defined as mandatory, whereas scripts in the web view often omit it.</p> <p>The syntax allows return values for event handlers.</p> <p>Syntax for an empty statement (three dots) is missing.</p>
Meta declarations	<p>The meta recent declaration looks like a temporal storage for code completion in IDE and does not appear anywhere.</p> <p>The meta guid declaration is explained as „purely internal” and does not appear in any view.</p> <p>The meaning of the meta seed declaration explained as „a session seed for table storage” is cryptic.</p>
Semantic	<p>No information exists about the case sensitivity of identifiers, their maximal length, and the length of the significant part.</p> <p>No information exists about keywords and potential name clashes with them (despite the fact that IDE automatically appends a running number to offending action names or prepends local variables with the @-sign).</p> <p>Identifiers and keywords are allowed to be „arbitrary (possibly empty) strings”, which is not correct.</p> <p>No information about the precision of the Number type (probably IEEE binary64).</p> <p>Confusing use of the term „mutable types” to denote reference types.</p> <p>No distinction between events and event handlers.</p>
Versioning	There are seemingly different (but similar!) version numbers for the mobile application, language specification (API), and the serialization of the scripts in the cloud.

Table 5.2: Possible inaccuracies in Horspool et al. [March, 2012]

and some symbols are substituted by escape sequences or removed from the identifier. Besides, the phrase from the specification "Identifiers, strings and keywords are arbitrary (possibly empty) strings..." (page 109) is misleading, because "possibly empty" applies only to the string literals and is not possible for the other two groups.

In contrast to the current document, the published specification does not provide a separate syntax definition for event handlers, merging event handlers and action declarations into one production rule. According to this rule, event handlers are also allowed to have return values. The use of those values is not clear because event handlers are called by the execution environment and not by a user and expect no return values. Notwithstanding, the specification itself implies that event handlers can only have input parameters in the phrase „Local variables are introduced ... by action input/output, and by event input parameters“, separating actions from event handlers. This phrase also shows that events (situations triggering code execution) and event handlers (the triggered code) are not distinguished in the specification (probably for the sake of simplicity).

The published specification does not include the term „number literal“, even though the format of numbers is defined by two production rules. It would be logical to provide a special rule or terminal for number literals in this context, especially considering the fact that the specification uses the terms and provides definitions for string and Boolean literals. While mining the specification by examining the source code corpus for the downloaded scripts we encountered an empty statement. Trying to map this statement to the visual representation of the script in mobile applications we discovered that in some cases the empty statement, represented in the source code by three dots, is mapped onto the „do nothing“ commentary in the mobile application. The specification by Horspool et al. [March, 2012] provides no clue if this type of statement exists.

5.2.2 Semantic issues and versioning

Horspool et al. [March, 2012] provide a detailed description of the meta declarations that are visible in none of the available views: the list of recently used identifiers (probably to provide a better context-sensitive code completion) and unique global identifiers (GUID) are internal („the guid is purely internal“, page 113) for TouchDevelop mobile applications and are not propagated to the cloud. The same applies for some other meta declarations, like „seed“: once mentioned in the specification, this declaration is not explained further and cannot be found in any code view.

The published specification uses the terms „mutable type“ and „reference type“ as direct synonyms (see page 111, for example) while explaining the copying semantics for value and reference types as parameters. Parameter passing in TouchDevelop is strictly by value and Horspool et al. [March, 2012] correctly point out that parameter passing semantics differs for reference and value types. Yet, passing a reference type by value does not change the semantics into „call-by-reference“; specifically, it is a copy of the reference that is passed into the action, not the reference itself.

The research paper from 2011 introduced the concept of asynchronous data fetching, which seems to be the way to overcome the limitations of the Windows Phone platform where the calls to external services (including the HTTP requests) are always asynchronous. The published specification from 2012 does not include any mention about asynchrony and when the data are actually fetched.

An important point is the different (but similar!) versioning of the TouchDevelop language and serialization format. The specification describes the latest version of the language and UI of the TouchDevelop. According to the blog by Nikolai Tillmann and the official

web portal, the latest version as of today is „v2.6” (the small „v” belongs to the version identifier), whereas the specification quotes version „v2.7” as one of the past versions („This behavior was introduced in v2.7”). At the same time, the serialization of the scripts, as indicated in the meta declaration on the script level, has currently only version „v2.3” (Horspool et al. [March, 2012], page 110). In the middle of April 2012 a new serialization format for scripts introduced dollar signs as a designation for local variables, but newly authored scripts still bear version „v2.2”, as seen in the web view¹.

5.2.3 Differences to the actual state of TouchDevelop

Due to the reasons discussed on page 25 our specification compared to the one by Horspool et al. [March, 2012], published 4 months after the beginning of our work, does not cover certain aspects of the actual state of the language. In particular, the following recent changes are important:

- Support for multiple **where** clauses in **foreach** loop statements.
- Support for user-defined libraries, including
 - a new type of scripts (code library) that cannot be directly run, in addition to the standard scripts directly runnable by the user, and
 - the possibility to reference the libraries from user scripts and to use the library actions.

As regards the first change, we added support for optional **where** clauses to the syntax to keep our parser up-to-date, but the current version of the compiler does not provide support for collection iterators and therefore this change was not dramatic.

Concerning the support for user libraries, this change is so drastic that it requires a thorough investigation and major changes on all levels of the compiler infrastructure, from parser to code generator. For this reason we did not incorporate this change into the existing specification and compiler.

5.3 Reflection on design decisions

Unlike many other programming languages where the design decisions are often motivated by the complexity of the parser implementation, the general compiler performance, and the possibilities to perform code optimization in the compiler, we approach TouchDevelop as a language designed for casual developers (see Section 1.3 for discussion on this topic). In this respect, the following rationales should have been taken into account while designing it:

- The language has to be easy enough to learn and to be used by non-professionals.
- The language infrastructure (consisting of the language itself, the runtime, and the IDE) has to hide the unnecessary complexity of the underlying .Net and Silverlight platforms.
- The platform (including the cloud service along with the language infrastructure) has to encourage script authoring, sharing, and collaboration.

Table 5.3 summarizes the design decisions for the language infrastructure. Even though the primary goal of the designers evidently was to make the language easy to learn and use, we categorized these decisions into the two groups: design decisions that aim at the

¹We checked it by opening the following link on May, 9th 2012: <https://www.touchdevelop.com/api/gjto/text>

Rationale(s)	Decisions		
	Language	Runtime environment	IDE
Making TouchDevelop a programming platform for mobile devices targeting students and hobbyists.	<p>Implicit local variable declarations. No compound (like arrays) or custom type definitions. Built-in 2D physics engine (GameBoard).</p> <p>Use of native Unicode symbols in identifiers and as keywords instead of their ASCII equivalents ($\rightarrow, \leq, \geq, \neq$ instead of $\rightarrow, <=, >=$ and $!=$). Use of equal sign ($=$) as comparison operator and colon-equal ($:=$) for assignment (unlike $==$ and $=$ in C-languages)</p>	<p>Event-driven development for phone (position), screen elements (wall), and graphics (sprites). Game-oriented event model (gameLoop event, sprites). Exception leads to the immediate halt of script execution. Built-in rich set of assets and possibility to directly use the assets available online.</p>	<p>Tokenized input of language keywords. „Fix-it” feature for most common glitches. Automatic naming of local variables (using the first letter of the type for the local variable name, avoiding the clash with existing variables). Static analysis of code „on-the-fly”, used for immediate error reporting and advanced highlighting in editor (by using the var keyword to designate the declarations of local variables).</p>
Hiding the complexity of the underlying .Net and Silverlight for Phone runtime environments.	<p>Multiple return values from actions (to hide different calling conventions in .Net: ref, out). Invalid type and is_invalid() property available for every data type (to hide the null value and avoid nullable types).</p>	<p>Transparent managing of the complete user application state (automated serialization on tombstoning^a and deserialization on restoring the application). Mimicking the synchronous execution of inherently asynchronous-only Silverlight operations (for fetching web content, requests to web services).</p>	<p>Automatic detection and declaration of the device capabilities used in the application, no need for a special application manifest etc. Automatic generation of live tiles for single application scripts.</p>

Table 5.3: Design decisions in TouchDevelop.

^aTombstoning – saving the application state when it is interrupted by an incoming call.

simplification of the language and the decisions needed to abstract from the underlying .Net and Silverlight runtime environments and to hide their unnecessary complexity.

Besides, we would like to emphasize the role of the cloud (for synchronization, as a web portal etc.) as a design decision because it contributes greatly to the evolution of the language and its popularity. The following infrastructure provided by the cloud commits to this last goal:

- The cloud-based versioning system for scripts that keeps track of script forking and derived works.
- The local script repository that is synchronized completely with the cloud during occasional Internet connections eliminating the needs for code commits.
- The leaderboards for game scripts that are directly managed by the cloud.

The following points in the design seem to be suboptimal from our point of view:

- The overall closedness of the platform which is reflected in very limited possibilities to author and execute the TouchDevelop scripts.
 - Only mobile device can be used for scripts authoring. There is no possibility to author (or edit) the scripts on the TouchDevelop webpage or to add custom text to the script.
 - No support for script templates. The only possibility to start not from an empty page is to fork somebody's script. This is very helpful, but the existence of templates could facilitate the language learning.
 - Even though many scripts are not directly coupled to the device-specific features and the Silverlight code is generally highly portable, there is no possibility to run scripts as Silverlight application on the web page.
- The approach that hides the details of the Silverlight runtime on Windows Phone makes TouchDevelop not an easy target for compiler implementation. The significant role of the TouchDevelop runtime that deals with many platform-specific issues (like the event model and the `gameloop` event) requires the constant presence of this runtime as a part of the compiled application. Besides, to successfully perform the application state serialization and deserialization on tombstoning events (that is one of the key requirements for the successful certification of the application of on Application Market) this embedded runtime must have access to the complete information about every script object (including local variables, wall, sprites, assets). The need for embedded runtime compromises the idea of compiler implementation and favors a straightforward „canning” approach to the creation of stand-alone applications.

6

Chapter 6

Conclusion

We obtain the following results in this work:

- The language specification for TouchDevelop is reverse-engineered.
 - The language syntax was reconstructed and debugged using ANTLR studio. The grammar implementation in ANTLR format is available in the Appendix.
 - The manually implemented parser successfully parses 100% of the TouchDevelop scripts that were available on the TouchDevelop webpage in December 2011.¹
 - The language syntax and semantics are documented in written form and provided as Chapter 3 of the current thesis.
- The re-engineered knowledge is implemented in form of a TouchDevelop compiler.
 - This is the first compiler for TouchDevelop and the second software (after TouchDevelop itself) which deals with the TouchDevelop scripts.
 - The compiler front-end is a highly performant predictive parser, which was implemented in F# using the FParsec parser combinator library. The 282 files with total 35075 lines of codes are parsed on the developer's machine in less than a second. Out of these 282, four are correctly identified to contain syntax errors.
 - A static semantic checker is implemented as a part of the compiler pipeline to perform semantic check of the parsed TouchDevelop code.
 - The compiler performs the transformation of the abstract syntax tree into the CodeDOM representation of the code.
 - CodeDom foundations from the .Net framework are used as compiler back-end for code generation for the .Net runtime.
- The best practices (as formulated by Klint et al. [2005]) are successfully applied for reverse engineering of the TouchDevelop grammar.
 - The language grammar is reverse-engineered in several iterative steps.
 - On the first step, the human-readable base-line language grammar is derived using ANTLR to catch the essence of the language syntax.
 - On the second step, the base-line grammar is used for the manual implementation of the parser.
 - This two-step approach helps us to decouple the grammar derivation from the implementation, making high optimization of the parser possible.
- We suggest an improvement for the best practices by Klint et al. [2005] to use CAD

¹For the complete list of scripts see page 89.

grammarware tools during the very first stage of the grammar reverse engineering.

- ANTLR was used for the prototyping of the language grammar.
- The use of ANTLR allows us to instantly perform the quality assurance of the grammar on every edit.
- The export feature of ANTLR was used to produce the railroad diagrams that were used in the language specification.

Appendix

List of TouchDevelop scripts used for specification mining

aawt	csix	flnc	hvkq	lchr	oawp	pycw	srdq	velk	xkmz
aewy	ctdb	fnkl	hwtc	lcno	oces	qdph	srvj	vevb	xreh
afnk	cvas	frks	hwyo	leri	ociv	qftc	srxi	vvgu	xrwy
afxx	dace	fuvn	hypt	lfms	odxv	qhuj	suzp	vizt	xynq
ahad	dark	fvhi	iiom	ltheta	ohxf	qhzm	swgx	vlhva	ygnn
aisj	dfzz	fynv	ijck	lney	onux	qitd	swtd	vmcr	yzty
ajlu	dhzd	gbbr	ileo	lqiz	oomv	qkkl	sxoc	vnps	zder
akdt	dicc	gdmr	imqe	lrzu	orto	qkrs	tagy	vrgt	zenx
akto	dphd	gkkw	imsi	lvde	oubt	qmbc	tamo	vrse	zilb
algi	drpc	gnah	iqik	lwtt	ovkh	qogr	tavb	vryl	zimi
amat	dvpr	gpid	iqri	meim	ovkt	qrsf	tbai	vxez	ztlra
aojn	ecvs	gydj	irdy	mfwr	owme	qsvc	tbts	vzkb	ztwi
aojp	efwy	gyti	ired	mmzb	oxvj	qurl	tgff	wawr	zvpj
aoxh	ehmo	hbei	irkb	mrqc	oylo	qvci	tius	wfps	zxwn
aplm	ejjr	hcaj	iufd	mtzs	ozbi	qvxs	tktv	wfvj	zycu
asnma	ejum	hcoh	ixxf	mwzl	ozok	qwxp	tqnd	wglv	zzah
auny	ekdx	hebw	jjuj	mykj	paqt	qyxf	ttky	wiip	yuhe
awic	ekuu	hejn	jmee	mzln	pbkt	rmjf	tulc	wlgr	ywqu
awmq	elpk	hgmt	jmlw	nalq	pddx	rolh	txew	wntg	ywys
aznq	eogy	hhkz	jqer	nclh	pdsf	rpjy	tzvt	wowr	yxbe
azwg	epdc	hkqf	jucv	ndqv	peju	rsey	ucjf	wqvj	
bdrx	eqcf	hllw	julr	nlik	pgcv	rsyo	ucvw	wtbg	
bjft	erzn	hqrx	jvkg	nmru	pprs	rvdy	ugny	wtud	
brui	euwv	hqvc	kdmd	nmwz	ppyd	shfv	ujsp	wyjg	
brzp	ezec	hrmw	kdpo	nnky	pqtq	sjji	uovg	xcbk	
bzqt	fbdv	hrvg	keyj	nnlg	pquj	slfs	uovt	xcfu	
cfor	fgau	hsaj	kocc	nofk	prqb	smrw	uqwy	xdsz	
cnyr	fhnw	htgb	kvmi	nwat	ptkr	somi	uwqr	xdzj	
cpob	flal	hvch	kxte	nykc	pxxf	sqak	uxti	xfxh	

TouchDevelop Grammar for ANTLR

```

grammar TouchDev;

// program blocks

program
    : (declaration)+
    ;

declaration
    : metaDeclaration
    | actionDeclaration
    | globalVariableDeclaration
    | eventDeclaration
    ;

metaDeclaration
    : 'meta' ( 'icon' | 'color' | 'name' | 'version' ) STRING ';'
    ;

metaParam
    : 'version'
    | 'name'
    | 'icon'
    | 'color'
    | 'private'
    ;

globalVariableDeclaration
    : 'var' varIdentifier ':' typeIdentifier
      '{'
      (
          (('is_readonly') '=' ('true')) |
          (('is_resource') '=' ('true'))
          (',' 'URL' '=' STRING)?
      )
      '}'
    ;

eventDeclaration
    : 'event' eventIdentifier '(' ( formalArg (',' formalArg)* )? ')'
      codeblock
    ;

codeblock
    : '{'
      statement*
      '}'
    ;

actionDeclaration
    : 'action' actionIdentifier '(' ( formalArg (',' formalArg)* )? ')'
      'returns' ( formalArg (',' formalArg)* )+ ')'
      codeblock

```

```

    ;

formalArg
  : varIdentifier ':' typeIdentifier
  ;

statement
  : emptyStatement
  | metaStatement
  | expressionStatement
  | assignmentStatement
  | ifStatement
  | whileLoopStatement
  | forLoopStatement
  | foreachLoopStatement
  ;

emptyStatement
  : '...' '*' ';'
  ;

metaStatement
  : 'meta' ('private') ';'
  ;

ifStatement
  : 'if' booleanCondition 'then' codeblock
    ('else' codeblock)?
  ;

assignmentStatement
  : varIdentifier ':=' expression ';'
  | varIdentifier (',' varIdentifier)* ':=' invocation
  ;

whileLoopStatement
  : 'while' booleanCondition 'do' codeblock
  ;

forLoopStatement
  : 'for' 0<=' varIdentifier '<' expression 'do' codeblock
  ;

foreachLoopStatement
  : 'foreach' varIdentifier 'in' varIdentifier ('where' expression)+ 'do'
    codeblock
  ;

expressionStatement
  : expression
  ;

invocation
  : varIdentifier (('->' (varIdentifier | functionInvocation))*
  ;

```

```

functionInvocation
    : actionIdentifier '(' (expression (',' expression)*)? ')'
    ;

booleanCondition
    : booleanExpression ( ('and' | 'or') booleanExpression )*
    ;

booleanExpression
    : expression ( ('<' | '>' | '=' | '<=' | '>=' | '!=') expression )*
    | ('not' booleanExpression)
    ;

expression
    : multExpr (('+' | '-' ) multExpr)*
    | stringExpression
    ;

multExpr
    : atom (('*' | '/' ) atom)*
    ;

atom: numLiteral
    | invocation
    | BOOLEAN
    | STRING
    | varIdentifier
    | '('! expression ')'!
    ;

// identifiers
varIdentifier
    : ('@')?('$')? ID
    ;

typeIdentifier
    : ID
    ;

eventIdentifier
    : ID
    ;

actionIdentifier
    : ID
    ;

literal
    : STRING
    | numLiteral
    ;

numLiteral
    : INT
    | DOUBLE
    ;

```



```

    ;

stringExpression
: STRING
| STRING '||' STRING
;

//basics
//===== literals =====

BOOLEAN : ('true'|'false')
;

ID : ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'_'')*
;

INT : '0'..'9'+
;

DOUBLE
: INT+ '.' INT*
;

COMMENT
: '//' ~( '\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
;

WS : ( ' '
| '\t'
| '\r'
| '\n'
) {$channel=HIDDEN;}
;

STRING
: '\'' ( ESC_SEQ | ~('\''|'\'') )* '\'' //
;

//unicode support
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

ESC_SEQ
: '\\' ( 'b'|'t'|'n'|'f'|'r'|'\n'|'\r'|'\'') //
| UNICODE_ESC
| OCTAL_ESC
;

OCTAL_ESC
: '\\' ('0'..'3') ('0'..'7') ('0'..'7')
| '\\' ('0'..'7') ('0'..'7')
| '\\' ('0'..'7')
;

UNICODE_ESC
: '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
;

```

TouchDevelop AST types (F#)

```

module touchdev_ast
open FParsec
type Name = string
type Operator =
    | Plus | Minus | Slash | Asterisk
    | LessOrEqual | Less | Equal | Unequal | GreaterOrEqual | Greater
    | Assignment // :=
    | And | Or | Not
    | Arrow // ->
    | Concat // ||

type Parameter = Name (*param name*) * Name (*type qualifier*)

type Expression =
    public
    | Nothing
    | Float of float
    | Boolean of bool
    | Variable of string
    | String of string
    | BinaryOperation of Operator * Expression * Expression
    | FunctionCall of Name (*fun name*) * Expression list (*arguments*)

and Statement =
    public
    | Block of Statement list
    | Expression of Expression
    | Assignment of Expression list (*multiple assignment*) * Expression
    | If of Expression * Statement (*then*) * Statement option (*else*)
    | ForI of Name (*loop variant*) * Expression (*upper bound*) * Statement
    | ForEach of Name (*iterator*) * Expression (*collection*) * Expression option (*condition*)
        * Statement
    | While of Expression * Statement
    | MetaStatement of Name

and public FunctionType =
    Name (*fun name*) * Parameter list option (*input values*) * Parameter list option (*return
        values*) * Statement

and TopLevel =
    public
    | MetaDeclaration of Name (*type*) * string (*value*)
    | Action of FunctionType
    | Event of FunctionType
    | Global of Parameter * Statement (*options*)

type CodeEntry =
    public
    | Namespace of string (* name *) * bool (* singleton *) * CodeEntry list option (* properties
        *) * string option (* help *)
    | Property of string (* name *) * bool (* public *) * Parameter list option (* input *) *
        Parameter list option (* return values *) * string option (* help *)

```

```

| Variable of string (* name *) * bool (* assignable *) * bool (* initialized *) * string (* type
  *) * string option (* help *)
| BinaryOperator of string (* name *) * string (* type left *) * string (* type right *) * string
  (* type return *) * string option (* help *)
| Meta of string (* name *) * string (* value *)

```

Issue types and error messages of semantic checker

```

type Issues =
  | Hints
  | Warnings
  | Errors

```

```

type Issue =
  | Hint of string
  | Warning of string * Issue option
  | Error of string * Issue option

```

```

let messages =
  dict [
    ("td001", (Errors, "Identifier not found."));
    ("td001.local", (Errors, "Local variable '{0}' not found."));
    ("td001.global", (Errors, "Global variable '{0}' not found."));
    ("td001.art", (Errors, "Art '{0}' not found."));
    ("td001.action", (Errors, "Action '{0}' not found."));
    ("td001.event", (Errors, "Event '{0}' not found."));
    ("td002", (Errors, "Ambiguous declaration: identifier already exists."));
    ("td002.global", (Errors, "Global variable '{0}' already declared."));
    ("td002.art", (Errors, "Art '{0}' already declared."));
    ("td002.action", (Errors, "Action '{0}' already declared."));
    ("td002.event", (Errors, "Event '{0}' already declared."));
    ("td002.meta", (Errors, "Meta '{0}' already declared."));
    ("td002.params", (Errors, "In action or event '{0}' the parameter
      '{1}' declared more than once."));
    ("td003", (Errors, "Type mismatch."));
    ("td003.g", (Errors, "Expected type '{0}', but found '{1}'."));
    ("td003.binary", (Errors, "Binary operator '{0}' requires '{1}' and
      '{2}', but was provided with '{3}' and '{4}'."));
    ("td003.foreach", (Errors, "ForEach loop requires a collection type,
      the provided type '{0}' is not a collection type."));
    ("td004", (Errors, "Assignable and read-only."));
    ("td004.left", (Errors, "Left side of the assignment expression must
      contain only assignable identifiers."));
    ("td004.loop", (Errors, "Loop variable '{0}' is read-only and cannot
      be re-assigned."));
    ("td005", (Errors, "Other errors"));
    ("td004.nothen", (Errors, "If statement requires a non-empty then-
      branch."));
  ]

```

TouchDevelop Script used for specification mining (in web view)

```

meta version "v2.2";
meta name "Syntax_Deriving";
meta icon "ABC";
meta color "#ffff0038";
//

action Identifiers() {
    // Test for allowed identifier names
    $a\_2 := 53;
    $answer_to_life\_the_universe_and_everything := 42;
    $comma\_bang\_at\_sharp\_percent\_potence\_ampersand\_asterisk\_
    \_rbrackets\_cbrackets\_ := 11;
    $equal\_sbrackets\_apostroph\_quote\_colon\_tilde\_question\_ := 22;
    $\u043A\u0430\u0436\u0434\u044B\u0439\u043E\u0445\u043E\u0442\u043D\u0438\u043A
    \_u0436\u0435\u043B\u0430\u0430\u0435\u0442\u0437\u043D\u0430\u0442\u044C\u0433
    \u0434\u0435\u0441\u0441\u0438\u0434\u0438\u0442\u0444\u0430\u0437\u0430\u043D
    \u043D := 56;
    $flei\u00DFige\_h\u00FCbungen\_h\u00E4rten := 56;
    // Test for identifier length
    $x234567890123456789012345678901234567890123456789012345678901234567890
    := 52;
    $x2345678901234567890123456789012345678901234567890123456789012345678901
    := 33;
    // Test for case sensitivity
    $X := 23;
    $x := 45;
    $res := $x || $X;
    $res->post_to_wall;
    // Testing if reserved words can be used as identifiers
    $@meta := 523.63;
    $@action := "RAIN IS PLAIN IN SPAIN";
    $meta_version\_v2\_0\_ := 52 = 3;
    $e\_ := false;
}

action test_wall() {
    $b := wall->ask_boolean("Ask bool ", "");
    $x := wall->ask_number("Ask number");
    $tb := wall->create_text_box("Textbox", 18);
    $x->post_to_wall;
    $b->post_to_wall;
}

var v : String_Collection {
}

action test_asynchron() {
    $s := web->download("http://www.google.com");
    $links := web->search_images("Esperanto");
    $links->post_to_wall;
    $s->post_to_wall;
}

action test_precision() {
    $pi := - 3.1415;

```

```

}

action test_loops() {
  for 0 <= i < 258 do {
    $i := $i + 3;
  }
}

action test_unequations() {
  $a := true;
  $b := true;
  $c := false;
  while($a = $b) or ($b -> $c) do {
    ... ;
  }
  if $c = false then {
    skip;
  }
  else {
    if ... then {
      skip;
    }
  }
  $x := 1;
  $y := 2;
  $z := 3;
  if not ($x = $y) or ($z = $y) and $x -> $z then {
    $a\_ := $a < $z;
  }
  else {
    if $x then {
      $x;
    }
    $w := 56;
    $h1 := 52;
  }
  if $b then {
    skip;
  }
}

action is_true() returns res: Boolean {
  "true" -> post_to_wall;
  $res := true;
  ... ;
  ... ;
  meta private;
}

action is_false() returns res: Boolean {
  "false" -> post_to_wall;
  $res := false;
  meta private;
}

action test_boolean_shortcuts() {

```

```

"--- OR ---" -> post_to_wall;
if code->is_true or code->is_true then {
    "-- OR checked --" -> post_to_wall;
}
else {
    // Nop
    ... ;
}
if not (code->is_false and code->is_false) then {
    "-- AND checked --" -> post_to_wall;
}
}

action test_division_by_zero() {
    $x := 56 / 0;
    $x-> post_to_wall;
}

action test_lazy_if_semantics() {
    if true then {
        6 >= 5;
    }
    else {
        5 = 63;
    }
}

action test_impossible_operations() {
    "sqrt(-3)" -> post_to_wall;
    math->sqrt( - 3)-> post_to_wall;
    "acos(12)" -> post_to_wall;
    math->acos(012)-> post_to_wall;
    "log(-18)" -> post_to_wall;
    math->log(2, - 18)-> post_to_wall;
    math->log( - 4, 056)-> post_to_wall;
}

action test_overload() {
    skip;
}

action test_overload1(x1: Number) {
    skip;
}

action test_return() returns x1: Number {
    $x1 := 6;
}

action test_return1() returns contact1: Boolean {
    $contact1 := false;
}

action test_implicit_conversions() {
    $x := 45;
    $s := "this is string";
}

```

```

($x -> $x)-> post_to_wall;
($s -> ($x -> "this is literal"))-> post_to_wall;
}

action test_implicit_conversion_by_arguments(s1: String, s2: String) returns res: Boolean {
  $res := $s1->count = $s2->count;
}

action test_implicit_conversion_runner() {
  $s := "Initial value";
  $s := code->test_implicit_conversion_by_arguments((52 -> ""), 18 -> "") -> "";
  $s-> post_to_wall;
}

action test_immutable_LValue() {
  math->\u03C0 := 2;
  math->sqrt(55) := 5;
  "This is literal" := "This is new literal";
}

action test_assignment_value() {
  $x := 52;
  $x1 := ($x := 36);
  $x1 := 56;
  $x1 := "String literal";
}

action test_static_types() {
  $board := media->create_board(640);
  $board := 56;
  $b := true;
  $b := "false";
  $b := 42;
  $s := "string";
}

action test_string_assignments() {
  $s := "String";
  $s-> post_to_wall;
  $s := true -> "";
  $s-> post_to_wall;
  $s := 417 -> "";
  $s-> post_to_wall;
  $s := wall->create_text_box("Test", 18) -> "";
  $s-> post_to_wall;
}

action incrementing_number(x1: Number) {
  $x1 := $x1 + 1;
  meta private;
}

action flipping_boolean(b1: Boolean) {
  $b1 := not $b1;
  meta private;
}

```

```

}

action concatenating_strings(s1: String) {
    $s1 := $s1 || " appended to the string";
    meta private;
}

action intensifying_a_color(c1: Color) {
    $c1 := colors->accent;
    meta private;
}

action playing_with_time(dt1: DateTime) {
    $dt1 := $dt1->add_days(10);
    meta private;
}

action changing_contact(contact1: Contact) {
    $contact1 := social->contacts("facebook")->at(1);
    meta private;
}

action test_parameter_passing() {
    $x := 41;
    $x-> post_to_wall;
    code->incrementing_number($x);
    $x-> post_to_wall;
    $b := true;
    $b-> post_to_wall;
    code->flipping_boolean($b);
    $b-> post_to_wall;
    $s := "text";
    $s-> post_to_wall;
    code->concatenating_strings($s);
    $s-> post_to_wall;
    $c := colors->rand;
    $c-> post_to_wall;
    code->intensifying_a_color(colors->accent);
    $c-> post_to_wall;
    $dt := time->now;
    $dt-> post_to_wall;
    code->playing_with_time($dt);
    $dt-> post_to_wall;
    $contact := social->contacts("facebook")->at(0);
    $contact-> post_to_wall;
    code->changing_contact($contact);
    $contact-> post_to_wall;
}

var a : Picture {
    is\_resource = true;
}

action go(aps1: Appointment_Collection, x1: Number, b1: Boolean, s1: String, aps2: DateTime,
    aps3: Color) {
    $aps3-> post_to_wall;
}

```



```
action Abc() {  
    skip;  
}
```

```
action main() {  
    skip;  
}
```


Bibliography

- IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, 29:1–58, 2008. doi: 10.1109/IEEESTD.2008.4610935.
- Hal Abelson, R.Kent. Dybvig, Christopher. Haynes, Guillermo. Rozas, Norman. Adams, Daniel P. Friedman, Eugene Kohlbecker, Guy Steele, David Bartley, Robert Halstead, Don Oxley, Gerald Jay Sussman, Gary Brooks, Chris Hanson, Kent. Pitman, and Mitchell Wand. Revised report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11:7–105, 1998. URL <http://dx.doi.org/10.1023/A:1010051815785>. 10.1023/A:1010051815785.
- Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1986.
- Noam Chomsky. *Reflections on language*. Fontana, London, 1976.
- Sebastian Deterding. Gamification: Toward a definition. *Design*, pages 12–15, 2011.
- Sophia Drossopoulou and Susan Eisenbach. Towards an operational semantics and proof of type soundness for java. In *Formal Syntax and Semantics of Java*. Springer-Verlag, 1998.
- Manuel Fahndrich. Game Board: Enabling Simple Games in TouchDevelop. Technical report, Microsoft Research, 2012.
- Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine and the lambda-calculus. *Proc. of the IFIP TC 2/WG2. 2 Working Conf. on Formal Description of Programming Concepts Part III, Ebberup, Denmark*, pages 193–217, 1986.
- Dick Grune and Criel J. H. Jacobs. *Parsing techniques: A practical guide*. Springer, New York and London, 2 edition, 2011.
- Elliott Rusty Harold. *Processing XML with Java: A guide to SAX, DOM, JDOM, JAXP, and TrAX*. Addison-Wesley, Boston [u.a.], 2003.
- R Nigel Horspool, Judith Bishop, Arhmand Samuel, Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Faehndrich. *TouchDevelop: programming on a phone*. Microsoft Research, unpublished, March, 2012.
- Jesper Juul. *A casual revolution reinventing video games and their players*. MIT Press, Cambridge and Mass, 2010.
- Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, July 2005. doi: 10.1145/1072997.1073000. URL <http://doi.acm.org/10.1145/1072997.1073000>.
- Ralf Lämmel and Chris Verhoef. Semi-automatic grammar recovery. *Software Practice and Experience*, 31:1395–1438, 2001.
- Ralf Lämmel and Vadim Zaytsev. Recovering grammar relationships for the Java Language Specification. *Software Quality Journal*, 19:333–378, 2011. URL <http://dx.doi.org/10.1007/s11219-010-9116-5>. 10.1007/s11219-010-9116-5.

- Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. *Department of Information and Computing Sciences Utrecht University Tech Rep UU-CS-2001-27*:19, 2001.
- Matt MacLaurin. Kodu: End-user programming and design for games. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 2:xviii–2:xix, New York, NY, USA, 2009. ACM. doi: 10.1145/1536513.1536516. URL <http://doi.acm.org/10.1145/1536513.1536516>.
- Donis Marshall. *Programming Microsoft Visual C-Sharp 2008: The Language*. Microsoft Press, 2009.
- Peter D. Mosses. Formal Semantics of Programming Languages: An Overview. *Electronic Notes in Theoretical Computer Science*, 148(1):41–73, 2006. doi: 10.1016/j.entcs.2005.12.012. URL <http://www.sciencedirect.com/science/article/pii/S1571066106000429>.
- Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software Practice and Experience*, 25:789–810, 1994.
- Gordon Plotkin. A structural approach to operational semantics, 1981.
- Brian Richardson. Terrarium Creature Development. 2003.
- Walter Saumweber. *AntMe: Programmieren und Spielen mit den Ameisen und Visual C#*. Microsoft Press, Unterschleissheim, 2007, c2008.
- David A. Schmidt. Programming language semantics. In *Encyclopedia of Computer Science*, pages 1463–1466. John Wiley and Sons Ltd., Chichester, UK, 2003.
- Dana S. Scott. Logic and programming languages. *Commun. ACM*, 20(9):634–641, September 1977. doi: 10.1145/359810.359826. URL <http://doi.acm.org/10.1145/359810.359826>.
- Kathryn T. Stolee and Teale Fristoe. Expressing computer science concepts through Kodu game lab. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 99–104, New York, NY, USA, 2011. ACM. doi: 10.1145/1953163.1953197. URL <http://doi.acm.org/10.1145/1953163.1953197>.
- Nikolai Tillmann, Michal Moskal, Jonathan Peli de Halleux, and Manuel Fahndrich. TouchDevelop: Programming cloud-connected mobile devices via touchscreen. Technical report, Microsoft Research, 2011.
- Stephen H. Unger. A global parser for context-free phrase structure grammars. *Commun. ACM*, 11(4):240–247, April 1968. doi: 10.1145/362991.363001. URL <http://doi.acm.org/10.1145/362991.363001>.