INTELLIGENT TUTOR TO LEARN
THE EVALUATION OF MICROCONTROLLER
I/O PROGRAMMING EXPRESSIONS

MASTER THESIS IN SOFTWARE ENGINEERING

STUDENT
STUDENT NUMBER 851695207
PRESENTATION 02/03/2017
INTELLIGENT TUTOR TO LEARN
THE EVALUATION OF MICROCONTROLLER
I/O PROGRAMMING EXPRESSIONS

MASTER THESIS IN SOFTWARE ENGINEERING

OPEN UNIVERSITY OF THE NETHERLANDS,
FACULTY OF MANAGEMENT, SCIENCE & TECHNOLOGY
MASTER SOFTWARE ENGINEERING

STUDENT: ING. HUGO ARENDS
STUDENT NUMBER: 851695207
PRESENTATION: 02/03/2017
CHAIRMAN: PROF. DR. JOHAN JEURING
SUPERVISOR: DR. BASTIAAN HEEREN
2ND SUPERVISOR: IR. HIEKE KEUNING
COURSE: IM9906
SUMMARY

Learning how microcontroller I/O programming expressions evaluate to a normal form is a vital task for students learning how to create embedded systems. Bitwise and logic operators for manipulating variables are a key factor for this domain. Typical expressions for this domain involve looping until a bit is set and updating specific bits in an hardware register. Each expression can have numerous evaluations, and all evaluations lead to the same normal form.

The intelligent tutoring system (ITS) prototype for learning the evaluation of microcontroller I/O programming expressions that was designed, implemented, and tested for this research project, solves the problem that, although no human tutor is available, students are guided step-by-step towards a solution. Such an ITS consists of two major building blocks: a front-end and a back-end. The front-end is realized as a web application. The main part of the back-end is the domain reasoner, which is realized using the IDEAS framework. A domain reasoner is a software program that helps students solve interactive exercises for a specific problem domain.

The domain of microcontroller I/O programming expressions is characterized by a diversity of microcontrollers and programming languages. The main contribution of this research project is the answer to the question how an ITS for this domain is capable of handling this diversity. The answer is a domain reasoner that is configurable by dynamically creating exercises from configuration files. Multiple microcontrollers are supported by parsing definitions from files as keyword and value pairs into a lookup environment. This environment is used whenever a definition or variable must be substituted. Multiple imperative programming languages are supported by allowing tokens from the grammar to be specified in a language definition file. This definition is also parsed into a lookup environment that is used during parsing and pretty printing.

The diagnose service is used to analyse a student’s step and calculate a feedback message. It was unexpected that the default diagnose service of the IDEAS framework is not suitable for this domain. The problem is that the IDEAS framework assumes that if two expressions are semantically equivalent, the student always takes a correct evaluation step. This is not necessarily true. The problem is solved by creating a custom diagnose service, which introduces a new equivalence relation that determines the semantic equivalence of all delta pairs. A delta pair is the maximum subexpression that is different when two expressions are compared.

It helps students to understand the evaluation of microcontroller I/O programming expressions, when the nature of feedback messages is related to explanations on subject matter, solution errors, and task-processing steps. These types of messages guide students step-by-step towards a solution.

The research project is validated by questionnaires filled in by students and lecturers. Quantitative results show that the prototype behaves as expected from a student’s point of view and that the new diagnoses are relevant for this domain. Qualitative results show that the feedback and hint messages help students towards a solution and contributes in understanding how microcontroller I/O programming expressions evaluate.
ACKNOWLEDGEMENTS

Today, an exciting period comes to an end: the finishing touch on this thesis. A lot of people were helpful to me during this period and I would like to take the opportunity to thank them here.

First of all I would like to thank the first year HAN Electrical and Electronic Engineering bachelor students, main phase students from the HAN minor Embedded Vision Design, and colleagues from HAN University of Applied Sciences who participated in the surveys.

My family and friends were always very understanding when I spent time on the research project. Many thanks to all of you for being who you are!

I would also like to thank my employer HAN University of Applied Sciences for giving me the opportunity to follow this education.

Last, but not least, I would like to thank my research committee for their support and guidance during the research project. I would particularly like to single out my supervisor Bastiaan Heeren. Bastiaan, I am grateful for your feedback and hints and I enjoyed our skype meetings that always guided me in the right direction. You encouraged me to get to the bottom of things, which helped me in making this research project and thesis as they are today.

Thank you very much, everyone!

Hugo Arends

Huissen, February 2017
CONTENTS

1 Introduction ................................................................................................................................. 1

2 Microcontroller teaching ............................................................................................................ 3
  2.1 Software development ............................................................................................................. 3
  2.2 Teaching methods .................................................................................................................... 8
  2.3 Expression evaluation .............................................................................................................. 9

3 Research assignment ................................................................................................................ 13
  3.1 Research questions ................................................................................................................. 13
  3.2 Research method ....................................................................................................................... 14
  3.3 Research scope ......................................................................................................................... 15

4 Example session ......................................................................................................................... 17
  4.1 Student interaction ................................................................................................................... 17
  4.2 Instructor interaction ............................................................................................................... 21

5 An ITS for microcontroller I/O programming ............................................................................. 23
  5.1 Web application ....................................................................................................................... 23
  5.2 Domain reasoner ..................................................................................................................... 24
  5.3 Diagnosis ................................................................................................................................. 39

6 Validation .................................................................................................................................... 49
  6.1 Survey setup ............................................................................................................................. 49
  6.2 Quantitative results ................................................................................................................. 50
  6.3 Qualitative results ..................................................................................................................... 54

7 Related work .............................................................................................................................. 59
  7.1 Intelligent tutoring systems ..................................................................................................... 59
  7.2 Evaluation of programming expressions ............................................................................... 60
  7.3 Comparison to other programming tutors ............................................................................. 63

8 Discussion .................................................................................................................................. 67

9 Conclusion ................................................................................................................................. 71

Bibliography ................................................................................................................................. 73

A Exercise configuration file ......................................................................................................... 77

B Language definition files ........................................................................................................... 79

C Script file ..................................................................................................................................... 81

D Diagnosis with a binary decision tree ....................................................................................... 83

E Expressions evaluated in surveys ............................................................................................. 85

F Answers to open questions ......................................................................................................... 89
LIST OF FIGURES

Figure 1. Typical steps from source and library files to executing on target hardware or simulator in an SDE for microcontroller software. .................................................................3

Figure 2. Simplified microcontroller block diagram showing how RAM, machine instructions and peripherals all are parts within the memory map. .................................................................4

Figure 3. Visualisation technique used by Dolman to explain the evaluation of a composite expression (Dolman, 2010). ........................................................................................................8

Figure 4. Initial screen of the web-based front-end of MicK. ..................................................17

Figure 5. After pressing the Start button, the student is asked to enter the next evaluation step. ......18

Figure 6. The student has finished a task successfully...............................................................20

Figure 7. The building blocks of the ITS prototype .................................................................23

Figure 8. Several components of an exercise and their dependencies ......................................25

Figure 9. Parse tree for the expression while(!(UCSR0A & (1<<UDRE0))){}; ..........................26

Figure 10. Simplified AST for the expression while(!(UCSR0A & (1<<UDRE0))){}; ..........27

Figure 11. Configuration files from the Exercises folder are parsed and exercises are dynamically generated. ..................................................................................................................29

Figure 12. Parsers for the definition files and the internal representation for each exercise.........30

Figure 13. The two simplified ASTs for x, y = while1 & 2 a = 3 + 4; while0 a = 7; and visualizing the two delta pairs with corresponding colours .........................................................40

Figure 14. Algorithm for simultaneously traversing two ASTs and checking if nodes are identical or semantically equivalent .................................................................41

Figure 15. Venn diagram for visualizing the characteristics of the subsets ..................................42

Figure 16. A debugging session in Atmel Studio to visualize the data from the ATmega328P on-chip debugger ..............................................................................................................61

Figure 17. Practicing with bitwise operators in Kumar’s online tutor on expression evaluation ......62

Figure 18. The practice part of Olmer’s tutor for practicing with the evaluation of Haskell expressions. .................................................................63
LIST OF TABLES

Table 1. Example elements $x, y \in E_2$ and if the element is an element of each of the six subsets. ........42

Table 2. Truth table for diagnosing an element $x, y \in E_2$ in the prototype. The highlighted diagnosis is new with respect to the IDEAS framework (v1.5). .................................................................46

Table 3. Results from quantitative analysis of the surveys with students and instructors. ...............50

Table 4. Results from quantitative analysis of the log database. ..................................................51

Table 5. Log entries that pointed out a bug in the pretty printer (session id 3472d6142c5f19e0c57a). .........................................................................................................................52

Table 6. Log entries showing how a student creates a worked-out example (session id a09e2dbe5df4154a2385). .........................................................................................................................52

Table 7. Log entries showing how a student only validates expressions in their normal form (session id 9a8a1f16ce1be6e80e30). .........................................................................................................................53

Table 8. First entries from a logging session, showing how a student submits a semantic equivalent expression, but takes a wrong step (session id 3c57c1c6c3205ca43a35). ........................................54

Table 9. Features in example-based environments (Gómez-Albarrán, 2005) and their implementation in the prototype. .........................................................................................................................65
1 INTRODUCTION

The internet of things, robotics, virtual home assistants, smart systems, and many more are vital systems within today’s connected society. What these systems all have in common is that they use one or more microcontroller to get information from sensors, to manipulate this information, and to control actuators based on this information. To build these microcontroller-based systems, students need to learn both hardware and software skills. These skills are taught in embedded systems education both at the bachelor and master level, but there are also many communities where one can learn how to build systems with microcontrollers. The most effective way to learn in general, also for developing microcontroller systems, is the availability of an expert human teacher for one-to-one instruction (Bloom, 1984). With an increasing popularity of embedded systems education, distance education, and online learning, the constant availability of experts is nowhere near this ideal situation.

This is where intelligent tutoring systems (ITS) offer a solution. An ITS is a software application designed to simulate a human tutor’s behaviour and it can play many different roles in the student’s learning process (VanLehn, 2006). The development of this technology started in the late 1960s and although significant progress has been made, the developmental complexity requires a considerable amount of manual effort. For this reason, ITSs have not been widely available. Over the past decade there has been a modest revival (Keuning, 2014) with a growing number of web-based tutors, which probably is related to the increasing popularity of online learning.

The main purpose of the research described in this thesis is to investigate how an ITS is useful within the microcontroller teaching domain. Although it would be interesting to also investigate this for hardware development, the scope of this research project is microcontroller software development. This is introduced in Section 2 by discussing how software is developed for microcontrollers and teaching methods for microcontroller programming. Section 3 discusses the research assignment by presenting the research questions, the research method, and the research scope. An example session from a student’s and an instructor’s point of view is shown in Section 4. Section 5 discusses the realization of an ITS for microcontroller I/O programming, which is validated in Section 6 by analysing the answers from surveys and log entries. Related work is discussed in Section 7. Section 8 reflects on the research project and discusses future work. Finally, the research project is concluded by providing an answer to the research questions in Section 9.
2 MICROCONTROLLER TEACHING

This section provides an overview of subjects that are related to teaching microcontroller programming. Section 2.1 describes the typical steps for developing software for microcontrollers. Teaching methods for microcontroller programming are described in Section 2.2. Finally, Section 2.3 presents a detailed description of the evaluation of microcontroller I/O programming expressions.

2.1 Software development

A microcontroller is the combination of a microprocessor core, memory and programmable I/O peripherals, all in a single integrated circuit (IC). Microcontrollers are used in embedded systems. They are preferred in applications where physical dimension, high performance and long battery life are important design decisions.

There is no microcontroller specific programming language. Software can be written in many different low-level and high-level programming languages as long as there is a compiler, interpreter, assembler, linker or other translation tool available that is capable of creating microcontroller-specific machine instructions. Nowadays, the majority of microcontrollers is programmed in C or C++. This is due to the nature of these languages, because they provide constructs that efficiently map to machine instructions and I/O operations.

Figure 1. Typical steps from source and library files to executing on target hardware or simulator in an SDE for microcontroller software.
The typical steps taken within a microcontroller software development environment (SDE) are depicted in Figure 1. Source files are created by developers, or are provided as a library, for instance by microcontroller vendors or communities. Translation software, such as compilers and assemblers, turn source files into object files. Although these files already contain machine code, a linker is used to combine and map several object files to the microcontroller specific memory layout, which is prescribed in a scatter loading file. Within an SDE, compiling and linking is often merged into one step with the use of makefiles. The resulting executable file is ready for use in a simulator or is programmed in a target microcontroller with a programmer or debugger.

For embedded systems engineers, software development for microcontrollers starts with the creation of source files, just as in application development for personal computers. The software design for microcontroller applications, however, is generally speaking less comprehensive, because software is created for a very specific purpose. Another difference is caused by what is called I/O programming as depicted in Figure 2. The central processing unit (CPU) executes machine instructions, stored in flash memory, to directly control hardware that is connected to the I/O pins of the microcontroller on a printed circuit board (PCB). The hardware inside the microcontroller that is controlled by software is called a peripheral. Peripherals are
controlled by software by means of registers. These registers are mapped to memory addresses and by addressing a specific memory location a register is read and written. This is called memory mapped I/O and is depicted in Figure 2. Another way of performing I/O operations is called port mapped I/O where only a dedicated set of CPU instructions perform input and output operations. When programming in a high-level programming language the compiler takes care of selecting the appropriate machine instructions.

All microcontroller software applications utilize the same basic high level design. After initialization of peripherals and variables, an endless loop reads sensors, processes data and controls actuators. To initialize and use a peripheral, bits in the registers must be set to specific logic values. The meaning and usage of the registers and bits is described by the microcontroller vendor in a document called the datasheet.

An example for initializing a general purpose I/O (GPIO) pin of the Atmel ATmega328P 8-bit microcontroller will be discussed next. The following code snippets are written in the C programming language and assume that code is compiled with the AVR/GNU C compiler. The different code snippets show several strategies that achieve the same result, namely to set the least significant bit (LSB) of the register at memory address 0x24\(^1\) to logic one. For this specific microcontroller, the register at this address is called the ‘data direction register port B’ (DDRB). A bit in this register selects the direction of the corresponding hardware pin. When a bit is written to logic one, the hardware pin is configured as an output pin. Code snippet 1 shows how to do this. The address is a constant in the left-hand side of an assignment. It is type casted to a volatile uint8_t pointer, which means that it now points to an 8-bit memory location that might be changed between different accesses. However, not the value of the pointer should be updated, but the memory location it points to, which requires an additional dereferencing operator. The right-hand side of the assignment operator is a constant. The prefix 0b is an indication for the compiler that the following number is represented in the binary numbering system.

\textit{Code snippet 1. Initialize the LSB of a GPIO port as output pin for an ATMEG 8-bit microcontroller.}
\[
*(\text{volatile uint8_t } *)(0x24)) = 0b00000001;
\]

An alternative implementation uses a definition of the register’s address as shown in Code snippet 2. This improves readability and maintainability. Code snippet 2 also shows that different numbering systems can be used to achieve the same result.

\textit{An alternative implementation uses a definition of the register’s address.}

\textsuperscript{1} The prefix 0x is an indication for the compiler that the following number is represented in the hexadecimal numbering system.
Another possibility is to use bitwise operations. This is actually preferred compared to an assignment as used in the previous examples, because it does not affect the other bits in the register. The downside is that the value of the register must be read first. This sequence is known as read-modify-write. Code snippet 3 shows two ways of programming a bitwise or. It also shows how other operators, such as a bitwise xor or even addition, might be used to achieve the same result.

Finally, macros and functions can be used for making I/O programming even more abstract. Examples are given in Code snippet 4 and Code snippet 5.


```c
#define DDB0 0
#define bit_set(reg, bit) (reg |= (1<<bit))

bit_set(DDRB, DDB0);
```

Code snippet 5. Using a function from a library.

```c
#include <gpio.h>

GPIO_setBit(DDRB, DDB0);
```

Code snippet 1 to Code snippet 5 illustrate that there is great variability in programming strategies that achieve the same result. A more comprehensive example will be discussed in Code snippet 6 to Code snippet 8. For the same microcontroller, the ‘universal synchronous and asynchronous serial receiver and
transmitter’ (USART) is initialized and used to transmit a character. The code in Code snippet 6 uses several aspects of the C programming syntax, such as evaluation order of operands, assignment operators, calculations, definitions, macros, post fixes, type casts and different styles of comments. For this example, the order of statements is important, because receiver and transmitter should only be enabled after other bits have been initialized.

**Code snippet 6. Initialize the USART of an Atmel ATmega328P microcontroller.**

```c
void USART_init(uint16_t baud)
{
    // Enable double speed
    bit_set(UCSR0A, U2X0);

    // Set baud rate
    uint16_t ubbr = (F_CPU / (8ul * baud)) - 1;
    UBRR0H = (uint8_t)(ubbr / 256);
    UBRR0L = (uint8_t)(ubbr);

    /*
    * No other control register needs updating, because the
    * default is async, no parity, 8 data bits, 1 stopbit
    */

    // Enable Receiver and Transmitter
    bit_set(UCSR0B, RXEN0);
    bit_set(UCSR0B, TXEN0);
}
```

The example in Code snippet 7 shows how the USART can be used for transmitting data after initialization. It shows a very common practice in I/O programming. A bit in a status register is polled until the hardware has finished a task. In the example, the ‘USART0 data register empty’ (UDRE0) bit of the register ‘USART0 control and status register A’ (UCSR0A) is polled until it is logic one. As soon as this happens, more data can be transmitted by writing the data to the ‘USART0 data register’ (UDR0). In this example the order of statements is also important.

**Code snippet 7. Transmitting data with the USART by polling a single bit in the status register.**

```c
uint8_t USART_putchar(uint8_t c)
{
    // Wait for UDR to be empty
    while(!(UCSR0A & (1<<UDRE0))) {};

    // Transmit the data
    UDR0 = c;

    return(c);
}
```
There are several possibilities to implement a while-loop such as used in Code snippet 7. These are shown in Code snippet 8. All of these expressions make sure that only the bit of interest is logically compared. The third possibility shows the most abstract way to wait while a bit is clear by means of a macro. This way enhances readability and is used when there is no need for students to exactly know how these expressions evaluate.

**Code snippet 8. Alternative implementations for waiting until exactly one bit is logic one in a register.**

```c
while((UCSR0A & (1<<UDRE0)) == 0)) {} // Mask
while((UCSR0A & (1<<UDRE0)) != (1<<UDRE0) ) {} // Mask
while(bit_is_clear(UCSR0A, UDRE0)) {} // Macro
```

### 2.2 Teaching methods

The code snippets in the previous section show that bitwise and logic operators are a key factor in I/O programming for microcontrollers. The way these operators are used is specific for the domain of microcontroller programming. For instance, a while-loop with an empty body that waits for a logical change of a bit in a hardware register, is a technique that is not relevant to other programming domains. Within microcontroller programming education, these programming techniques are explicitly taught. The book about microcontroller basics by Davies (2008) for example, only discusses the important aspects of C for embedded system. These aspects are declarations, shifts, low-level logic operations, masks to test and modify individual bits, bitfields, and unions. These aspects are explained by simple examples. Dolman (2010) uses a visualisation technique, as depicted in Figure 3, to explain the step-by-step evaluation of composite expressions. The expression explained in Figure 3 is:

\[
\text{PORTB} = (\text{PORTB} \& 0xF9) | ((x << 1) \& 0x06);
\]

![Figure 3. Visualisation technique used by Dolman to explain the evaluation of a composite expression (Dolman, 2010).](image)
Pardue (2005) uses a similar visualisation technique to explain the evaluation of single bitwise and logic operators. He uses a discussion to explain the evaluation of composite expressions, such as the following condition in an if-statement:

\[
\text{if}( \neg (\text{TCC0RA} \& \text{WGM01}) \& \& \neg (\text{TCC0RA} \& \text{WGM00}) ) ...
\]

Such an expression is explained by the following discussion:

“\text{The (TCC0RA} \& \text{WGM01)} test will be 1, true, only if the WGM01 bit is 1, likewise for the \text{(TCC0RA} \& \text{WGM00)} state. The \neg (\text{TCC0RA} \& \text{WGM01)}, adding the ‘!’ \text{or NOT to the statement means that it is true only if the innards of the (}) are false. The ‘if’ statement will only be true if both the first and (logical AND = \&\&) the second are true. So we’ve used two AND in this statement.”

Pardue notes that if this explanation is not clear, the reader should “... get out the pencil and paper computer and work through it till it is.”, emphasising the importance of understanding the step-by-step evaluation of such statements.

There are dozens of teaching methods that instructors can use in their classes for teaching bitwise and logic operators, such as lectures, individual and group reports, textbook assignments, laboratory experiments, blended learning, etcetera. According to Gilibert et al. (2006) the best way to introduce microcontrollers is by practical exercises and to avoid theoretical explanations. To overcome the drawbacks related with the availability of laboratory resources, they implemented a “remote microprocessor work bench”. This system allows students to remotely debug their code and interact with the hardware.

Besides practical exercises, personal feedback in an early stage and progress tracking is important. For this purpose, Weiss et al. (2005) assign one senior student to five students for one hour each week in a microcontroller course. They found that these senior students “have a tremendous influence on the motivation and performance of the students and are in fact the pivotal factor in determining the motivational impact of a course”. To overcome the drawback of planned sessions they suggest to make tutoring available with the use of a webcam. They note that distance learning will also benefit from such an approach.

2.3 Expression evaluation

Learning how to develop software for microcontrollers does not only concern the bottom-up creation of programs. It also involves understanding how expressions evaluate, which is a top-down approach. This section presents two expressions and how these expressions evaluate to a unique representation that cannot be evaluated any further, also known as the normal form. The first expression will be the running example throughout the rest of this document.
Expression 1

```c
while(!(UCSR0A & (1 << UDRE0))) {;}
```

A possible first evaluation step is substituting the definition UDRE0. For the ATmega328P microcontroller, UDRE0 is defined as the decimal value 5.

⇒ Substitute definition UDRE0

```c
while(!(UCSR0A & (1 << 5))) {;}
```

A possible next evaluation step is the bitwise left shift operator. The expression (1<<5) yields 32. However, when the left operand of the bitwise shift left operator is in binary representation, it is clearer how the bitwise shift left operator evaluates. Therefore, before bitwise left shifting, the left operand is first converted to its binary representation.

⇒ Decimal to binary

```c
while(!(UCSR0A & (0b00000001 << 5))) {;}
```

⇒ Bitwise left shift

```c
while(!(UCSR0A & 0b00100000)) {;}
```

From this point, further evaluation depends on the value of the bits in register UCSR0A. This is a volatile memory location, which means that the actual value can change at any moment in time. For instance, if UCSR0A is equal to 2, then the evaluation continues as follows:

⇒ Substitute register contents (e.g. 2 in binary representation)

```c
while(!(0b00000010 & 0b00010000)) {;}
```

For evaluation of the bitwise and operator, binary representation for both operands is most convenient. In this example both operands are already in binary representation, therefore no evaluation step is needed to transform either of the operands.

⇒ Bitwise and

```c
while(!(0b00000000)) {;}
```

Finally, there are two more steps:

⇒ Substitute numeric by Boolean representation

```c
while(!(false)) {;}
```

⇒ Logic negate

```c
while(true) {;}
```

This normal form shows that the condition is true.
The evaluation of this expression could also have started by substituting the contents of UCSR0A. This gives the following possible evaluation:

```
while(!(UCSR0A & (1<<UDRE0))) {};
```

⇒ Substitute register contents (e.g. 2 in binary representation)

```
while(!(0b00000010 & (1<<UDRE0))) {};
```

⇒ Substitute definition UDRE0 (e.g. 5)

```
while(!(0b00000010 & (0b00000001 << 5))) {};
```

⇒ Decimal to binary

```
while(!(0b00000010 & 0b00100000)) {};
```

⇒ Bitwise left shift

```
while(!(false)) {};
```

⇒ Substitute numeric by Boolean representation

```
while(true) {};
```

Both evaluations are correct and there is not a preference for one of them.

**Expression 2**

`PORTB = (PORTB & 0xF9) | ((x<<1) & 0x06);`

The first evaluation step of this expression would be to substitute the volatile value of register PORTB, the volatile value of the variable x, or both. A possible evaluation is the following:

```
⇒ Substitute register PORTB (e.g. 4)
⇒ Substitute variable x (e.g. 2)
PORTB = (4 & 0b11111001) | ((2<<1) & 0b00000110);
```

⇒ Decimal to binary

```
PORTB = (4 & 0b11111001) | ((0b00000010<<1) & 0b00000110);
```

⇒ Bitwise left shift

```
PORTB = (4 & 0b11111001) | (0b00000010 & 0b00000110);
```

⇒ Bitwise and

```
PORTB = (4 & 0b11111001) | 0b00000100;
```
Decimal to binary
PORTB = (0b00000100 & 0b11111001) | 0b00000100;

Bitwise and
PORTB = 0b00000000 | 0b00000100;

Bitwise or
PORTB = 0b00000100;

An alternative evaluation would be:
PORTB = (PORTB & 0b11111001) | ((x<<1) & 0b00000110);  # Substitute register PORTB (e.g. 4)
PORTB = (4 & 0b11111001) | ((2<<1) & 0b00000110);  # Substitute variable x (e.g. 2)

Bitwise and
PortB = 0 | (0b00000100 & 0b00000110);  # Bitwise shift left

Bitwise and
PORTB = 4;

These examples show that evaluations use a bottom-up strategy, taking precedence and associativity of the operators into account. It also shows that there can be many different ways to evaluate a given expression and that multiple evaluation steps can be taken at once. The number of evaluation steps depends on:

- The number of substitutions of registers, variables and definitions.
- The number of operators.
- The transformation to other number representations for operands, which depends on the operator and the previous evaluation steps.
- Definitions that might contain composite expressions.
- The steps a student combines into one evaluation step.

Although the same expression can be rewritten in many different ways, it should always yield the same normal form. A rewriting system for these expressions must therefore be confluent and terminating.
3 RESEARCH ASSIGNMENT

There are many ITSs available, especially for programming exercises. The goal of this research project is to investigate how an ITS can be created for learning the evaluation of microcontroller I/O programming expressions, by using the generic framework from the interactive domain-specific exercise assistants (IDEAS) project. The framework provided by the IDEAS project can be used for developing domain-specific reasoners. A domain reasoner is a software program that helps students solve exercises for a specific problem domain. The IDEAS project was started by a group from the Faculty of Computer Science of the Open University of the Netherlands and the Department of Information and Computing Sciences of Utrecht University.

Section 3.1 formulates the research questions. The research method is described in Section 3.2. Finally, the research scope is discussed in Section 3.3.

3.1 Research questions

Section 2 describes characteristic challenges for teaching microcontroller I/O programming. Students not only need to learn the syntax and semantics of a programming language, they also need to acquire knowledge and skills related to I/O programming for different microcontrollers, such as manipulating bits at specific I/O locations and using loops to wait for a logical change of a specific bit. The evaluation of these composite expressions can be hard to comprehend for novice embedded systems engineers. Most ITSs for programming would address such a task bottom-up, by giving feedback and hints with the goal of guiding students towards a solution. It is also valuable for students to learn the top-down evaluation and interpretation of complex expressions, which is also recognized by Kumar (2005).

Another characteristic for the domain of microcontroller I/O programming is the diversity of libraries and programming languages. An ITS for this domain should be capable of handling this diversity, rather than the need to create separate tutoring systems for each difference.

These observations narrow down the research question to the following:

**HOW TO SUPPORT MULTIPLE MICROCONTROLLERS WITHIN AN INTELLIGENT TUTORING SYSTEM THAT HELPS STUDENTS TO UNDERSTAND THE EVALUATION OF I/O PROGRAMMING EXPRESSIONS?**

An answer can be formulated after answering these sub-questions:

i. How can feedback and hints be generated from a student expression, an exercise and instructor feedback using the IDEAS framework?

ii. How can multiple microcontroller definitions be supported?
iii. How can multiple imperative programming languages be supported?
iv. What types of feedback and hints help students to understand the evaluation of microcontroller I/O programming expressions?
v. What are differences and similarities between this tutor for learning the evaluation of microcontroller I/O programming expressions and existing programming tutors?

The answers to these research questions are formulated in Section 9.

3.2 Research method

This section describes what research methods are used to answer and validate each sub-question listed in Section 3.1.

Literature and tutorials\(^2\) for implementing a domain reasoner using the IDEAS framework are studied to formulate an answer to sub-question i. This information is used to conduct an experiment. A first prototype of a domain reasoner is implemented with static support for expressions. No front-end is created for providing a graphical user interface. The calculated evaluations and feedback generation are validated by a demonstration and a code walkthrough with the supervisor.

For answering sub-questions ii and iii, the outcome of the experiment of sub-question i is used as a starting point for a new experiment. A new domain reasoner is implemented and tested. A simple web-based front-end is created for the purpose of validating the prototype. Students and colleagues are asked to use the prototype, and fill in a questionnaire, which is used to validate the calculated evaluations. It is expected that the IDEAS framework can be used for creating an ITS that supports multiple microcontroller definitions and programming languages. The answer to sub-questions i to iii is described in Section 5. The validation with students and colleagues is described in Section 6.

The categories for feedback generation for learning programming, as described by Keuning et al. (2016), are used to formulate an answer to sub-question iv. It is expected that the types of feedback that guide students step-by-step will help students in their learning process. The answer to this sub-question is formulated in Section 5 and validated in Section 6.

The comprehensive list of tools for learning programming, as identified by Gómez-Albarrán (2005), and a literature study are used to formulate an answer to sub-question v. It is expected that there are not many ITSs for the purpose of expression evaluation. The answer is formulated in Section 7.

\(^2\) http://ideas.cs.uu.nl/tutorial/, August 2016.
3.3 Research scope

The prototype created for this research project is a tutoring system that can be used to learn the evaluation of single expressions at a specific moment in time. It is not a tutor for the bottom-up creation of programming algorithms. The scope of the research project is limited to two typical I/O programming tasks. The first task is looping until a bit is set. For example:

```c
while(!(UCSR0A & (1<<UDRE0))) {};
```

The second task is assigning specific bits of a variable \(x\) to an output register while making sure the other bits in this register do not change. For example:

```c
PORTB = (PORTB & 249) | ((x << 1) & 6);
```

The only supported task is rewriting an imperative expression step-by-step to its normal form. Although software for microcontrollers can be written in many programming languages, two imperative languages are supported. Style issues are out of scope. The prototype provides a means to cope with volatile data, such as registers and variables. Furthermore, the prototype enables instructors to customize the tutoring environment for at least two different microcontrollers: the ATmega328P and the STM32F051R8.

Finally, the prototype will not decide on which task a student should do next. Students can select an expression from a menu or manually enter an expression.
3 RESEARCH ASSIGNMENT
This section demonstrates the tutoring system prototype. The tutoring system is called MicK, which is an abbreviation for Microcontroller Knowledge. It uses a web application front-end for student interaction. Section 4.1 introduces the web application and describes typical student interactions. The customization options are described in Section 4.2.

4.1 Student interaction

A typical session by a hypothetical student starts with opening the web application in a web browser. The interface as depicted in Figure 4 is presented. The student starts by selecting a microcontroller and programming language from the first dropdown box. Upon selection, relevant example expressions are automatically added to the second dropdown box. The student either selects an example expression and optionally changes it, or manually enters an initial expression in the input field below the Start button.

![Initial screen of the web-based front-end of MicK.](image)
Let us assume that the student selects the following expression from the examples and does not change it:

\[
\text{while}( \neg ((\text{UCSR0A} \& (1 << \text{UDRE0}))) ) \{ ; \}
\]

A tutoring task is then started by clicking the Start button. The tutoring system analyses this initial expression and presents a value for the register UCSR0A and the definition UDRE0:

- Value of definitions, registers and volatile variables for this microcontroller and programming language:
  - UCSR0A = 0b00001111
  - UDRE0 = 5

At this point, all information required to do an evaluation task is available for the student. An additional input field is presented for the student to enter the next evaluation step, as depicted in Figure 5.

![Figure 5](image)

**Figure 5. After pressing the Start button, the student is asked to enter the next evaluation step.**

The student copies the expression by clicking the Copy button. The copied expression is then edited by the student by substituting both the register and definition:

\[
\text{while}( \neg ((0b00001111 \& (1 << 5))) ) \{ ; \}
\]

The student validates this evaluation step by clicking the Validate button. The tutor responds with the message:

- That is correct.

Another input field is added to the list of evaluation steps, which allows the student to enter the next evaluation step. The student enters the next step by removing the redundant parentheses and evaluating the shift left operator, but makes a common mistake by reversing the order of operands of the shift left operator:

\[
\text{while}( \neg (0b00001111 \& 10) ) \{ ; \}
\]

The student clicks the Validate button and the tutor responds with the message:

- That is incorrect. The operands of the shift left operator are reversed

This message tells the student exactly what went wrong. By clicking the hyperlink `shift left`, a new window is opened with an external website that explains the shift left operator.
The student does not know how to proceed and clicks the Hint button. The tutor clears the incorrect input field and responds with the message:

\[
1
\Rightarrow \text{rewrite this } \text{decimal number to its binary representation}
\]

This message tells the student which operation must be performed on which part of the subexpression \((1 \ll 5)\). Again, the hyperlink points to an external website for more information about the operation.

Let us assume that the student still does not know how to proceed and clicks the Show button. The tutor responds with the message:

\[
1
\Rightarrow \text{rewrite this } \text{decimal number to its binary representation}
0b00000001
\]

For the most part, this is the same message as when the student requested a hint. The tutor now, however, also fills in the solution:

\[
\text{while( ! } ((0b00001111 \& (0b00000001 \ll 5))) ) \{ ; \}
\]

The student clicks Validate and the tutor responds with the message:

\textit{That is correct.}

Now the student makes a mistake by entering and validating the following expression:

\[
\text{while } ( ! (0b00001111 \& 0) ) \{ ; \}
\]

The tutor responds with the message:

\textit{Although the expressions are equivalent, you took a wrong step.}

This message tells the student that the submitted expression evaluates to the same normal form, but the part that was rewritten is incorrect. This expression is not a valid evaluation step, and therefore no new input field will be added to the evaluation list. The student might edit the expression, or request a hint, or request the solution to this step. Let us assume the student changes the expression to:

\[
\text{while } ( ! (0b00001111 \& 0b00100000) ) \{ ; \}
\]

The student clicks the Validate button and the tutor responds with the message:

\textit{That is correct.}

The student now tries to take multiple steps at once, but forgets a closing parenthesis:

\[
\text{while } ( ! (\text{false} ) ) \{ ; \}
\]

The student clicks the Validate button and the tutor responds with the message:
Syntax error 1:1:
unexpected 'w'
expecting '(', ')', '{', end of input, or operator

The student corrects the mistake:

```c
while( ! (false) ) { ; }
```

The student clicks the Validate button and the tutor responds with the message:

*That is correct.*

The student enters the final step:

```c
while( true ) { ; }
```

The student clicks the Validate button and the tutor responds with the message:

*That is correct.*

*You have finished the task successfully!*

This message tells the student that the expression is in a normal form. The input field with the final solution is marked green and no new input field is added to the evaluation list. The complete worked out solution as seen by the student is depicted in Figure 6. By clicking the Stop button, the task stops and the student can start a new task.

---

*Figure 6. The student has finished a task successfully.*
4.2 Instructor interaction

Instructors can customize the tutoring system by adding or modifying configuration files, which are read from the filesystem. Each configuration file is interpreted as a new exercise. An exercise must be created for the combination of a microcontroller and a programming language. All these exercises are automatically added to the list the student can choose from in the web application. For changes to take effect, the client only needs to reload the web application.

An example exercise configuration file is provided in Appendix A. Each exercise consists of the following configuration options:

- **Exercise id**
  
  This must be a unique combination of a microcontroller and programming language represented as a string, such as ‘ATmega328P’ and ‘ANSI-C’. These two strings are concatenated when presented in the dropdown box in the web application.

- **Word length**
  
  The number of bits of the architecture for the selected microcontroller. Numbers that are displayed in the binary or hexadecimal number representation will use this configuration option to determine the number of digits for zero padding.

- **Programming language file**
  
  Instructors can change the programming language by specifying language specific keywords and tokens in a language definition file. The path to this file must be specified in the configuration file. The language specification can be reused amongst exercises. An example of a customizable token is the assignment operator. It might for instance be specified as the string “=”, or as the string “:=”. Two example language definition files are provided in Appendix B.

- **Definition files**
  
  The back-end allows microcontroller specific definitions to be added in different file formats. This allows for easy integration of existing definition files that are often provided by microcontroller vendors. When, for instance, the ANSI-C programming language is used, the instructor specifies the path to one or more C header files. All definitions from these files are then automatically parsed and available for substitution.

- **Script file**
  
  The feedback and hint messages provided by the tutoring system can be specified in a script file. This enables instructors to easily change feedback and hint messages, for instance to match classroom lectures, or support multiple languages. An example script file is provided in Appendix C.
• Examples

Zero or more example expressions can be specified that will appear in the examples dropdown box in the web application.

• Initial values for volatile variables and registers

In the event that registers or volatile variables are used in an expression, they must have an initial value for evaluation. These initial values are not randomly generated, but must be provided by the instructor. This improves the ability to steer the students’ learning process. The initial values are interpreted as subexpressions. Some examples of initial values are:

- 15
- 0x000F
- 0b0001111
- 1 << 5
5 AN ITS FOR MICROCONTROLLER I/O PROGRAMMING

The ITS prototype created for this research project consists of two major building blocks: a front-end and a back-end. Their dependencies are depicted in Figure 7. The front-end interacts with the user. It filters user input and presents information. For this project, the front-end is realized as a web application. The design decisions and implementation examples for the front-end are described in Section 5.1. The back-end implements the functionality to reason about a problem domain. The IDEAS framework is used for this purpose. The design decisions and implementation examples are described in Section 5.2. Special considerations have been taken for diagnosing expressions for the domain of microcontroller I/O programming. These considerations are described in Section 5.3.

5.1 Web application

The web application is implemented using Bootstrap, JQuery and JavaScript. The look-and-feel is described in Section 4.1.

5.1.1 Model-View-Controller

The web application is implemented using the model view controller (MVC) design pattern. This pattern is depicted in Figure 7. As soon as the controller is triggered by
the student, the controller creates an object in the JavaScript object notation (JSON) and calls the model's `modelRequest()` function with this object as a parameter.

The model holds an abstract representation of the data. It has no intelligence, because it uses the domain reasoner for this purpose. This is transparent for both controller and view. Another JSON formatted object is created by the model and the function `requestToDR()` is used to transmit the request to the domain reasoner.

The model receives the response from the domain reasoner in the callback function `responseFromDR()`. The response is also formatted using JSON. The model extracts the information from the object, interprets it, creates a new JSON formatted object and calls the `viewUpdate()` function, which updates all available views with this new object as a parameter. As soon as a view receives such an object, it knows how to display the information, or flushes it.

5.1.2 Data logging
Logging is enabled in the back-end for investigating interactions. For this purpose, the front-end creates a random user id. This id is stored in a cookie that expires after one day. When requesting specific services from the domain reasoner, such as the create service, the domain reasoner provides a session id that can be used as a token during the remainder of a session to easily filter related interactions from the logging database. Data is logged in an SQL database. Local sqlite3 logging for Windows is enabled by building the HDBC-sqlite3\(^3\) package and rebuilding the IDEAS framework with logging enabled.

5.2 Domain reasoner
The front-end requests a stateless service from the domain reasoner, such as a list of exercises. For providing these services, the domain reasoner consists of two building blocks, as depicted in Figure 7. The first building block is called feedback services. These are already implemented by the framework and use the abstract interfaces of the other building block called exercises. These exercises consist of several components that a developer of a domain reasoner implements in the functional programming language Haskell using the IDEAS package\(^4\). The most important components of an exercise are depicted in Figure 8. This figure shows the hierarchy of the building blocks and their dependencies. A data type must be defined for domain-specific expressions. Rules define how the data type can be transformed. An id is used for identification. Many entities within the IDEAS framework can have such an id, although Figure 8 only shows this for rules and strategies. It is also possible to define buggy rules to recognize commonly made mistakes. A rule is translated to the term data type, which allows the use of a zipper for traversal. Simple exercises can be solved by just applying rules. More complex exercises must be solved


by combining rules into a procedure, which is called a strategy. A label gives the opportunity to provide localized feedback messages.

There are also, among others, the following components (Heeren & Jeuring, 2014):

- Zero or more example expressions.
- A parser to parse a human readable string to the domain specific data type.
- A pretty printer to print the domain specific data type into a human readable string.
- An equivalence function to test whether two expressions are semantically equivalent.
- A similarity function to test whether two expressions are syntactically the same, or nearly so.
- A goal predicate to test whether an expression is in a solved form.

The following sections describe the implementation of these and other components and the rationale for design decisions.

5.2.1 Data type for I/O programming expressions
A data type is used for abstract representation of domain-specific expressions. The data type for this domain is related to the grammar of typical microcontroller I/O programming expressions. The grammar implemented in the prototype is as follows:
expr ::= stmt | stmt ";" | stmt ";" expr | "{" expr "}"
stmt ::= "skip" | identifier "assign" op1 |
"while" "{" op1 "}" "{" expr "}" | op1
op1 ::= op1 "|" op2 | op2
op2 ::= op2 "&" op3 | op3
op3 ::= op3 "<<" op4 | op4
op4 ::= op4 "+" op5 | op5
op5 ::= "!" op6 | op6
op6 ::= "(" op1 ")" | num | bool | identifier
num ::= dec | bin | hex
bool ::= "true" | "false"

The starting non-terminal of an expression is expr. Expr is a statement optionally followed by a semicolon. Expr is optionally placed between braces. Although the grammar allows a sequence of expressions to be parsed separated by a semicolon, the prototype does not support the evaluation of such a sequence of expressions.

The grammar supports four statements:

- Skip: represents an empty statement.
- Assignment: assign the result of an operation to an identifier.
- While: looping structure with a condition between parentheses and a body between braces.
- Op1: an operation with no side-effect. Although such a statement would not make sense in a programming language, the prototype uses such a statement for parsing definitions.

Figure 9. Parse tree for the expression while(!(UCSR0A & (1<<UDRE0))){}, showing only the terminal and non-terminal symbols.
Infix operators are left associative. The grammar is not ambiguous, because of the operator precedence.

The numeric terminal symbols are numerical values in the decimal (dec), binary (bin) or hexadecimal (hex) number representation. An identifier is a string that starts with a letter followed by zero or more alpha numerical characters.

The parse tree, showing only the terminal and non-terminal symbols, for the expression `while(!(UCSR0A & (1<<UDRE0))){};` is presented in Figure 9.

Sentences belonging to this grammar are parsed to an abstract syntax, which is represented by the data type as shown in Code snippet 9. A sequence of expressions is represented as a list. The left-hand side of an assignment is a string, although most programming languages would also allow expressions in the left-hand side of an assignment.

**Code snippet 9. Data type for expressions.**

```haskell
data Expr = Seq [Expr] | Skip | Assign String Expr | While Expr Expr | Infix String Expr Expr | Prefix String Expr | Dec Integer | Bin Integer | Hex Integer | Bool Integer | Var String | Unknown String deriving (Eq, Show)
```

The operator of infix and prefix expressions is represented as a string. This allows to easily add more operators and support for programming language specific tokens.

The numerical values Dec, Bin and Hex are internally represented as integers. A Bool is also represented as an integer. This allows for calculation and comparison between numerical and Boolean values without the need of conversions. The parser converts “false” to the integer value 0 and “true” to the integer value 1.

![Simplified AST for the expression while(!(UCSR0A & (1<<UDRE0))){};](image-url)
All identifiers other than the identifier in the left-hand side of an assignment, are assigned the \texttt{Var} constructor. This distinction is made, because for typical microcontroller I/O programming expressions the identifiers in the left-hand side of an assignment will not be substituted. If an identifier with the \texttt{Var} constructor cannot be substituted, the constructor is changed to \texttt{Unknown}.

The simplified AST in Figure 10 illustrates the result after parsing for the expression \texttt{while(!(UCSR0A & (1<<UDRE0)))\{;\}}.

The parser is implemented using the monadic parser combinator library \texttt{megaparsec}\textsuperscript{5}. One particular advantage of this library is the support for well-typed error messages instead of string-based error messages, which provides flexibility in describing parse errors. This helps students to analyse the syntactic mistakes made when submitting expressions. The type declaration of the parser is shown in Code snippet 10. It takes a language definition and a string as an argument, and either returns a string with a syntax error message or an expression. A language definition is a mapping of programming language specific keywords to the keywords used by the parser. A detailed description of language definition is provided in Section 5.2.4.

A pretty printer turns an expression into a human readable string. The prototype pattern matches on the classifications of the data type and does not remove redundant parenthesis. The pretty printer takes the word length of the selected microcontroller into account, which is provided as a setting in the configuration file as discussed in Section 5.2.2, for zero padding numbers in the binary and hexadecimal representation. The function’s type declaration is shown in Code snippet 10.

\textit{Code snippet 10. Type declarations for the parser and pretty printer.}

\begin{verbatim}
parseStringM :: LanguageDef -> String -> Either String Expr
ppExpr       :: LanguageDef -> Expr -> String
\end{verbatim}

5.2.2 Customizable exercises

Each microcontroller and programming language pair have distinct configurations for an exercise. These configurations are:

- A unique exercise identifier
- The microcontroller’s word length
- One or more microcontroller definition files
- One programming language definition file
- One script file
- Example expressions
- Initial values for volatile data, such as registers and variables

\textsuperscript{5}https://hackage.haskell.org/package/megaparsec, November 2016.
For each exercise, these configurations must be provided in a separate configuration file in the XML format. Exercises are dynamically generated for each configuration file, by parsing all XML files from the exercises folder at start-up of the domain reasoner as depicted in Figure 11. From each exercise configuration file, the exercise parser determines which other files are required. This means that the exercises folder is the only hard coded file location in the domain reasoner. An example exercise configuration file is shown in Appendix A.

Dynamically generating exercises allows instructors to easily add new exercises or update existing exercises. There is no need for programming or recompiling the domain reasoner.

5.2.3 Parsing microcontroller definitions
Definitions supported by a microcontroller must also be supported by the domain reasoner. The prototype assumes that a definition consists of a keyword, which is a string, and a value, which is an expression. All of these keyword and value pairs are internally represented in a list, which is called an environment.

Microcontroller definitions are often provided in header files. To make support for these definitions as easy as possible for instructors, the domain reasoner is capable of parsing several definition file formats. This is depicted in Figure 12. The prototype supports C header files, but does not support macro expansion. There is also support for a custom file format called DEF, which parses definitions with the following syntax:

```
DEF <identifier> <value> <newline>
```
This custom format allows instructors to add support for unsupported file formats. It requires a one-time translation from the original format to the DEF format.

Each exercise has its own environment. In the configuration file for an exercise, the instructor provides the paths to zero or more definition files and for each definition file the type of parser to use.

![Diagram of IDEAS framework](image)

*Figure 12. Parsers for the definition files and the internal representation for each exercise.*

### 5.2.4 Parsing language definitions

Support for multiple imperative programming languages is realized by allowing instructors to customize keywords and tokens from the grammar as discussed in Section 5.2.1. Typical expressions for microcontroller I/O programming do not require differentiation in the abstract syntax. The customizable tokens are:

- “while”
- “assign”
- “true”
- “false”

Instructors can supply a language definition in XML format. Two example definition files are presented in Appendix B. For these two examples, the expressions shown in Code snippet 11 are semantically the same.

*Code snippet 11. Support for custom keywords and operators.*

```
while(!(True)) {
}
ZOLANG(!(WAAR)) {
}
```
The language definition parser creates an environment for each exercise, in a similar way the parser for microcontroller definitions creates an environment. The result is a list of keyword and value pairs, both represented as a string. This language definition environment is used for parsing and pretty printing expressions.

5.2.5 Rules
Rules define how values of the data type can be transformed. Generally speaking, a rule takes a name and a function of type \(a \rightarrow \text{Maybe } a\) as its arguments. If a rule can be applied, the rule returns \text{Just} the transformed expression. If a rule cannot be applied, the rule returns \text{Nothing}.

The prototype implements four groups of rules. The first group are rules for operators that transform expressions. For each operator, a similar rule is defined as given in Code snippet 12. In this example, both operands of the bitwise and operator must be of constructor \text{Bin}, because it is more illustrative to perform this operation on numbers in binary representation. The result is a new expression represented as a binary number with the value being the result of the bitwise and operation.

### Code snippet 12. Rule for the bitwise and operator.

```haskell
-- |Rule to bitwise-and two numbers
bitwiseAndRule :: Rule Expr
bitwiseAndRule = describe "Bitwise and two numbers" $
    makeRule "rule.bitwiseand" f

where
    f :: Expr -> Maybe Expr
    f (Infix "&" (Bin x) (Bin y)) = Just $ Bin (x Bits..&. y)
    f _ = Nothing
```

The second group of rules are transformations to other number representations, such as conversions to decimal, binary, hexadecimal, and Boolean. Code snippet 13 shows a rule for converting a decimal number to a binary number. It changes the constructor, which is used by the pretty printer for representation. This allows the internal representation for all number representations to be the same, namely of type \text{Integer}.

### Code snippet 13. Rule for converting a number from decimal to binary representation.

```haskell
-- |Rule to convert a decimal to binary
decToBinRule :: Rule Expr
decToBinRule = describe "Decimal to binary" $
    makeRule "rule.dectobin" f

where
    f :: Expr -> Maybe Expr
    f (Dec x) = Just $ Bin x
    f _ = Nothing
```
The third group is a substitution rule that is shown in Code snippet 14. This rule transforms a definition, a register, or a variable, all represented by the \texttt{Var} constructor, into an exercise-specific expression. The rule therefore takes an environment, as discussed in Section 5.2.3, as an argument. The \texttt{lookup} function is used to search the environment for a substitution. If a substitution cannot be found, the lookup function returns \texttt{Nothing}. In such a case, the rule returns the same value \((x)\), but changes the constructor to \texttt{Unknown}.

\textit{Code snippet 14. Rule for substituting definitions and variables.}

\begin{verbatim}
-- |Rule to substitute definitions and variables
substituteRule :: Env -> Rule Expr
substituteRule env = describe "Substitute a definition or a variable" $
  makeRule "rule.substitute" f

where
  f :: Expr -> Maybe Expr
  f (Var x) = Just $ fromMaybe (Unknown x) (lookup x env)
  f _ = Nothing
\end{verbatim}

The fourth and final group are buggy rules. The prototype implements one buggy rule that is shown in Code snippet 15. This rule describes the transformation of the shift left operator when the operands are reversed. This common mistake is made when both operands are in decimal representation.

\textit{Code snippet 15. Buggy rule for the shift left operator.}

\begin{verbatim}
-- |Reverse the operands of the shift left operator
shiftLeftBuggy :: Rule Expr
shiftLeftBuggy = describe "Shift left operands reversed" $
  buggyRule "rule.buggy.shiftleft" f

where
  f :: Expr -> Maybe Expr
  f (Infix "<<" (Dec x) (Dec y)) =
    Just $ Dec (y `Bits.shiftL` fromIntegral x)
  f _ = Nothing
\end{verbatim}

5.2.6 Strategies

Strategies combine rules to solve more complex exercises. Strategies must be provided in an embedded domain-specific language (EDSL) that is interpreted by the IDEAS framework as a context-free grammar. Strategies are used to calculate feedback messages given an exercise, the strategy for solving it, and student input (Heeren et al., 2010).

A strategy can be composed out of rules, but also out of other strategies. Strategies are combined by using strategy combinators. An example is the choice combinator \((\cdot | \cdot)\), which chooses between two strategies. The prototype implements two
strategies: one for rewriting expressions with an assignment and one for rewriting expressions with a while-statement.

An expression with an assignment is rewritten with the strategy as shown in Code snippet 16. The strategy takes an environment as argument, because it uses the `substituteS` strategy, which also takes an environment. Section 2.3 describes that the evaluation of I/O programming expressions follows a bottom-up procedure. This means that a strategy for rewriting expressions with an assignment should try to either substitute a variable, using the `substituteS` strategy, or calculate the result of an operator, using the `operatorS` strategy. After one of these two strategies has been applied, the strategy stops and starts over again from the bottom. This continues until none of the two strategies can be applied anymore, resulting in the normal form. The strategy described here, where a sub strategy is repeatedly applied until it fails and traversing once bottom-up, is called an innermost strategy (Ren & Erwig, 2006).

**Code snippet 16. Strategy for rewriting expressions with an assignment.**

```haskell
-- |This strategy rewrites an Assign expression to a normal form.
nfAssignS :: Env -> LabeledStrategy (Context Expr)
nfAssignS env = label "strategy.nfAssign" $ innermost $ substituteS env .|. operatorS
```

The sub strategy `operatorS` combines a list of alternative strategies that operate on infix expressions. One of those alternative strategies is the `shiftLeftS` strategy. The implementation is presented in Code snippet 17. The function returns an `ExprStrategy` type, which is a type synonym for `Strategy (Context Expr)`. The strategy uses the sequence combinator to:

1. Check if the node in focus is the shift left operator.
2. Make sure the left operand is of type `Bin` and the right operand is of type `Dec`. The strategy `convOperands` is used for this purpose. This strategy converts the operands to the desired representation by trying to convert none, only the left, only the right or both operands in either order.
3. Apply the `shiftLeft` rule.

**Code snippet 17. Strategy for rewriting the shift left operator.**

```haskell
-- |This strategy verifies if the node in focus is of type shiftLeft, makes sure the left operand is Bin, the right operand is Dec and finally applies the shiftLeftRule
shiftLeftS :: ExprStrategy
shiftLeftS =
  check isShiftLeft
  .*: convOperands decToBinS binToDecS
  .*: liftToContext shiftLeftRule

-- |Returns True if Expr in the current context is of type shiftLeft
isShiftLeft :: Context Expr -> Bool
isShiftLeft = maybe False f . currentInContext
```
where
  \( f \) (Infix "\(<<\)" _ _) = True
  f _ _ = False

-- |This strategy executes two strategies. It is used to convert
-- the operands of operators to the desired representation. It
-- tries to convert none, only the left, only the right or both
-- operands in either order.
convOperands :: IsStrategy \( f \) => \( f \) (Context Expr) ->
  \( f \) (Context Expr) ->
  ExprStrategy
convOperands l r =
  option(  applyToFirstChild l
         .|. applyToSecondChild r
         .|. permute[applyToFirstChild l, applyToSecondChild r]
   )

An expression with a while-statement is rewritten into a normal form with the
strategy as shown in Code snippet 18. The strategy is similar to the strategy for
expressions with an assignment. The strategy uses the left bias choice combinator (|>)
to first try to either substitute a variable, using the substituteS strategy, or
calculate the result of an operator, using the operatorS strategy. If this is not
possible, the strategy tries to rewrite the condition to a Boolean value, using the
whileCondToBoolS strategy. Finally, the strategy tries to calculate the result of the
logic not operator, using the logicNotS strategy. The strategies whileCondToBoolS
and logicNotS are optional, meaning they will not fail if they cannot be applied. This
means that the nfWhileS strategy can also be used to rewrite expressions with an
assignments into a normal form.


-- |This strategy rewrites a While expression to a normal
-- form.
nfWhileS :: Env -> LabeledStrategy (Context Expr)
nfWhileS env = label "strategy.nfWhile" $ innermost $
  (substituteS env .|. operatorS) |> whileCondToBoolS |>
  logicNotS

These innermost strategies for writing expressions into a normal form ensures
confluency, because every possible evaluation path will produce the exact same
result. The strategies nfAssignS and nfWhileS ensure a terminating rewriting
system, because the operators use a predefined number representation for the
operands. As soon as both operands are in the correct representation, the result of the
operator is calculated and rewriting continues.

5.2.7 Equivalence and similarity
Functions for semantic equivalence and similarity are used for calculating feedback
messages by several feedback services, for instance in the diagnose service and the
test report service. It therefore is important that these functions are implemented according to the definitions as required by the IDEAS framework.

An equivalence function is defined as a binary relation on a set and is expected to be reflexive, symmetric and transitive. Semantic equivalence within the prototype is determined by calculating the normal form of two expressions and logically comparing the results. Prior to calculating the normal form, all nodes in the AST with the `Var` constructor are substituted. This approach assumes that neither of the expressions has an `Unknown` constructor in the AST. If at least one of the expressions does have an `Unknown` constructor, it is not possible to calculate the normal form and the equivalence function returns false.

The similarity function is defined by the IDEAS framework as two expressions being the same, or nearly so. For the domain of microcontroller I/O programming two expressions are considered similar when both expressions have an identical AST with zero or more AC-rewritings. For calculating AC-rewritings, the prototype calculates and compares the integer result of nested infix operators. The function for similarity will therefore return false if one of the expressions has an `Unknown` constructor in the AST, because it is not possible to convert an `Unknown` value to an integer value. The downside to this approach is that two identical expressions with an `Unknown` constructor will not be marked similar. This, however, is not a problem in the prototype, because the prototype will not allow a new task to be created that contains unknown registers, definitions or variables. The suitability predicate of an exercise, which acts as the pre-condition of the strategy, is implemented for this purpose.

5.2.8 Feedback generation

The feedback services building block as depicted in Figure 7 calculates feedback messages by using the rules, the strategies and the functions for equivalence and similarity. For deciding what types of feedback and hint messages help students to understand the evaluation of microcontroller I/O programming expressions, the categories for feedback generation for learning programming, as described by Keuning et al. (2016), are used. Keuning et al. describe the initial results of a systematic literature review on feedback generation for programming exercises. They classify the nature of feedback messages generated by 69 tools and have extended Narciss’s five elaborated feedback components (Narciss, 2008), with eleven representative subcategories. These five components from Narciss and the subcategories from Keuning et al. are listed below. For each subcategory, the following is described:

| Situation | An example situation for the domain of microcontroller I/O programming expressions. All examples assume the student uses the expression `while(!(UCSR0A & (1<<UDRE0))) {;}.` |
| Feedback | An example feedback or hint message that is appropriate for the situation. |
Motivation

How feedback and hint messages for this subcategory helps students to understand the evaluation of microcontroller I/O programming expressions and how feedback and hint messages for this subcategory can be realized in the prototype.

Knowledge about constraints (KTC)

Task requirements (TR)

Situation
The student uses ten evaluation steps to rewrite the expression into the normal form.

Feedback
You should finish this task in eight evaluation steps.

Motivation
Rules, constraints, and requirements for tasks help students know what is expected of them. This subcategory is not implemented in the prototype, because it is out of scope for the research project.

Task-processing rules (TPR)

Situation
The student starts the tutor and wants to start with a new task.

Feedback
Welcome, I’m MicK!
I can help you to evaluate microcontroller I/O programming expressions to their normal form.

To start a session, you should:
1. Select a microcontroller and programming language.
2. Enter an initial expression or select an example.
3. Click the Start-button.

Motivation
It must be clear for a student how to approach a task to help a student getting started. This is realized by providing a feedback message when a new task is selected in the front-end.

Knowledge about concepts (KC)

Explanations on subject matter (EXP)

Situation
The student enters an expression, but does not know how to evaluate the bitwise left shift operator. The student asks for a hint.

Feedback
Evaluate the shift left operator

Motivation
If a student makes a mistake, or does not know how to proceed towards a solution, information should be provided on the subject matter. This is realized by providing a hyperlink to an external website with information about the subject.

Examples illustrating concepts (EXA)

Situation
The student makes a mistake by evaluating the subexpression 1<<2 to 2.
Feedback  
This is not the correct evaluation of the bitwise left shift operator. Let us first go through some bitwise left shift examples.

Motivation  
The tutor should be able to select a new task based on a student’s progress. This subcategory is not implemented in the prototype, because it is out of scope for the research project.

Knowledge about mistakes (KM)

Test failures (TF)
Situation  
The student makes a mistake by evaluating the subexpression $1 << 2$ to 2.
Feedback  
This expression produces an incorrect output.
Motivation  
These feedback messages are used to indicate that a program does not produce the expected output. This domain, however, does not check the output produced by an expression. There is no need to provide such feedback messages.

Compiler errors (CE)
Situation  
The student makes a syntactic mistake.
Feedback  
Syntax error at character 10: unexpected ‘{’
Motivation  
An expression should be checked for syntactic errors and non-existing definitions. A detailed error message is provided by the parser to help the student solve the mistake.

Solution errors (SE)
Situation  
The student makes a mistake by evaluating the subexpression $1 << 2$ to 2.
Feedback  
That is incorrect. Try again. You may ask for a hint.
Motivation  
Whenever there is a mistake in an evaluation step, an incorrect expression is submitted by the student. A feedback message is calculated by the feedback services building block indicating that the student made a mistake.

Style issues (SI)
Situation  
The student uses redundant parentheses in the submitted expression.
Feedback  
This step is correct. There are, however, redundant parentheses in your expression.
Motivation  
Good programming style is considered important for programmers. A feedback message could provide information about style issues. This subcategory is not implemented in the prototype, because it is out of scope for the research project.
Performance issues (PI)
Situation The student submits an expression using unnecessary resources.
Feedback This expression is correct. Your expression, however, uses unnecessary resources.
Motivation Although performance issues are generally speaking important in microcontroller programming, performance issues are not relevant for understanding expression evaluation. A tutor for the domain of microcontroller I/O programming expressions does not need to provide feedback messages for performance issues.

Knowledge about how to proceed (KH)

Error correction (EC)
Situation The student makes a mistake by evaluating the subexpression 1<<2 to 2.
Feedback This is not a correct step. The subexpression 1<<2 equals 4.
Motivation It is useful to provide context specific feedback at each step. However, instead of providing feedback on what the student should do to correct the mistake, the student can ask for a hint. Therefore, this subcategory is not implemented in the prototype.

Task-processing steps (TPS)
Situation The student submits the expression \(a=(0b0000001\ll2)\), but does not know how to evaluate the bitwise left shift operator. The student asks for a hint.
Feedback \( (0b0000001 \ll 2) \)
=> evaluate the shift left operator
Motivation In order to proceed towards a solutions, detailed feedback should be provided about a next possible evaluation step. This is calculated by the feedback services building block.

Knowledge about meta-cognition (KMC)
Situation The student correctly evaluates the expression to \(\text{while(true) \{;\}}\).
Feedback You have finished the task correctly. What could you have done differently?
Motivation It is useful for a student’s learning process to check if a student is able to critically analyse the solution. This subcategory is not implemented in the prototype, because it is out of scope for the research project.

Textual feedback messages can be provided by instructors in a script file. An example script file is shown in Appendix C. Script files are parsed by the IDEAS framework and a textual feedback message will be presented instead of an identifier. For each exercise, the path to a script file can be provided in the exercise configuration file.
5.2.9 Custom services
Besides the services provided by the IDEAS framework, the prototype implements several custom services, such as a custom diagnose service and a substitution service. The custom diagnose service is described in more detail in the next section. The substitution service can be requested for presenting the substitution values of definitions, registers and variables. This service takes an expression as an input and returns a list of tuples. Each tuple contains the name of the definition, register or variable and its substitution value.

5.3 Diagnosis
The diagnose service is one of the services provided by the feedback services building block as depicted in Figure 7. It is used to analyse a student’s step and calculate a feedback message. Diagnosing a student’s step for the domain of microcontroller I/O programming expressions, requires an additional diagnosis that is not provided by the default diagnose service in the IDEAS framework (v1.5). The reason is that expressions can still be semantically equivalent, although an incorrect step has been submitted. An example of such an evaluation starts with the following expression:

\[
\text{while}(!(0b00000010 \& (0b00000001 \ll 5)))\ {;}
\]

The following incorrect evaluation step is submitted:

\[
\text{while}(!(0b00000010 \& 0b00000000))\ {;}
\]

Due to the bitwise and operator, both of these expressions evaluate to the same normal form:

\[
\text{while}(\text{true})\ {;}
\]

Despite the same normal form, the student’s step clearly is incorrect. Using the default diagnose service (Heeren & Jeuring, 2014), the step will be diagnosed ‘correct’. This is not a suitable diagnosis for the domain of microcontroller I/O programming expressions. The tutor must be capable of providing feedback on incorrect steps, but still be able to determine semantic equivalence. The same phenomenon has been recognized for the domain of propositional logic. Lodder et al. (2016) note that their learning environment fails to recognize rules in case the student takes an incorrect step, but the expression as a whole still is equivalent. The problem also shows up in other domains, such as mathematics. Suppose a student needs to solve the following equation:

\[
\frac{x}{2^{-1}} \times 0 = 0
\]

The student rewrites this to:

\[
\frac{x}{3} \times 0 = 0
\]

Although the step is wrong, both expressions are semantically equivalent. A solution to this problem is described in the remainder of this section.
5.3.1 Definitions
Let E denote the set of all microcontroller I/O programming expressions. A diagnosis is calculated between a pair of expressions. The Cartesian product $E^2 = E \times E = \{(x,y) \mid x \in E, y \in E\}$ denotes the set of all ordered microcontroller I/O programming expression pairs.

Within the IDEAS framework (v1.5), two equivalence relations are calculated over $E^2$:

- **sem** = \{ (x,y) ∈ $E^2$ | x and y are semantically equivalent \}
- **syn** = \{ (x,y) ∈ $E^2$ | x and y are syntactically the same (or nearly so) \}

Additionally, three binary relations are calculated over $E^2$, which are parameterized over a rule or strategy. These relations are:

- **bug** = \{ (x,y) ∈ $E^2$ | y follows from x by applying a buggy rule \}
- **rul** = \{ (x,y) ∈ $E^2$ | y follows from x by applying a valid rule \}
- **str** = \{ (x,y) ∈ $E^2$ | y follows from x by applying a valid rule according a strategy \}

The following equivalence relation over $E^2$ is new. It relates to the parts of elements (x,y) that are different with respect to each other. Let $x\Delta y \in E$ be the maximum subexpression from x with the root node being the node that is different compared to the node at the same location in the AST of expression y. Then $dp = (x\Delta y, y\Delta x)$ is called a **delta pair** iff $x\Delta y$ and $y\Delta x$ concern the same difference of (x,y). For example:

Let (x,y) be (2 & (1 ≪ 3), 2 & 8). Then ((1 ≪ 3), 8) is a delta pair for (x,y).

Let (x,y) be (2 & (1 ≪ 3), 3 + 8). Then (2 & (1 ≪ 3), 3 + 8) is a delta pair for (x,y) and ((1 ≪ 3), 8) is not a delta pair for (x,y).

![Figure 13. The two simplified ASTs for (x,y) = (while(1 & 2)( a = 3 + 4;),while(0)( a = 7;)) and visualizing the two delta pairs with corresponding colours.](image)

An element (x,y) can have multiple delta pairs, because nodes might be different in more than one branch. The subset $dps = \{ dp \in E^2 \mid dp$ is a delta pair of (x,y)\} denotes
all delta pairs for \((x, y)\). This is depicted in Figure 13 for the following example, where
the two delta pairs are visualized with corresponding colours:

Let \((x, y)\) be \((\text{while}(1 \& 2)\{ a = 3 + 4; \}, \text{while}(0)\{ a = 7; \})\).
Then \(\text{dps}(x, y) = \{(1 \& 2, 0), (3 + 4, 7)\}\).

The new homogeneous equivalence relation over \(E^2\) is now defined as follows:
\[
\text{dep} = \{(x, y) \in E^2 \mid \text{all delta pairs from } (x, y) \text{ are semantically equivalent}\}
\]

This relation is calculated by a parallel top-down and left-to-right traversal of the two
ASTs of \(x\) and \(y\). The algorithm is depicted in Figure 14. During traversal, the nodes
at the same location in the AST are compared for being identical. If so, traversal
continues top-down and left-to-right. If the nodes are not identical, the
subexpressions, with these nodes being the root nodes, are compared for semantical
equivalence. If semantically equivalent, traversal continues the next node to the right.
If the subexpressions are not semantically equivalent, then \((x, y) \notin \text{dep}\). If all nodes
are traversed, then \((x, y) \in \text{dep}\).

---

**Figure 14. Algorithm for simultaneously traversing two ASTs and checking if nodes are identical or semantically equivalent.**

The calculation of delta pairs does not account for AC-rewritings. In the following
element, strictly speaking, the expression \((1 + 2)\) has been rewritten into 1, and the
expression 3 has been rewritten into \((2 + 3)\):

Let \((x, y)\) be \(((1 + 2) + 3), (1 + (2 + 3))\). Then \(\text{dps}(x, y) = \{(1 + 2, 1), (3, 2 + 3)\}\) and \((x, y) \notin \text{dep}\).
Another example shows the combination of a rewrite step and a commutative rewriting:

Let \((x, y)\) be \((2 \& (1 \ll 3) , 8 \& 2)\). Then \(dps(x, y) = \{(2, 8), ((1 \ll 3), 2)\}\) and \((x, y) \notin \text{dep}\).

Table 1 provides eleven example elements \((x, y) \in E^2\) and shows if the element is (✓) or is not (✗) an element of each of the six subsets as described in this section. Notice from examples three to six that, although the expressions are semantically equivalent \(((x, y) \in \text{sem})\), at least one delta pair is not semantically equivalent \(((x, y) \notin \text{dep})\). Also notice from the fourth example how an element can be expected by a strategy \(((x, y) \in \text{str})\) while having delta pairs that are not semantically equivalent \(((x, y) \notin \text{dep})\). Finally, notice from the eleventh example how an element can have all delta pairs semantically equivalent, but at the same time a buggy rule is applicable.

Table 1. Example elements \((x, y) \in E^2\) and if the element is an element of each of the six subsets.

<table>
<thead>
<tr>
<th>#</th>
<th>((x, y) \in E^2)</th>
<th>\begin{tabular}{l}Is element of subset \end{tabular}</th>
</tr>
</thead>
</table>
|    | \begin{tabular}{l}x \\
y\end{tabular} | \begin{tabular}{l}sem \\
dep \\
bug \\
syn \\
str \\
rul\end{tabular} |
| 1  | 1 & (1 \ll 3)     | \begin{tabular}{l}✗
✗
✗
✗
✗
✗\end{tabular} |
| 2  | 2 & (1 \ll 3)     | \begin{tabular}{l}✗
 ✓
 ✓
 ✓
✗
✗\end{tabular} |
| 3  | 1 & (1 \ll 3)     | \begin{tabular}{l}✓
✗
✗
✗
✗
✗\end{tabular} |
| 4  | ((1 + 2) + 3) + 4 | \begin{tabular}{l}✓
✗
✗
✓
✓\end{tabular} |
| 5  | 1 & (1 \ll 3)     | \begin{tabular}{l}✓
✗
✓
✓
✗
✗\end{tabular} |
| 6  | 1 & (1 \ll 3)     | \begin{tabular}{l}✓
✓
✗
✗
✗
✗\end{tabular} |
| 7  | 1 & (1 \ll 3)     | \begin{tabular}{l}✓
✓
✓
✓
✓
✗\end{tabular} |
| 8  | 1 & (1 \ll 3)     | \begin{tabular}{l}✓
✓
✓
✗
✗
✗\end{tabular} |
| 9  | 1 & (1 \ll 3)     | \begin{tabular}{l}✓
✓
✗
✗
✓
✓\end{tabular} |
| 10 | 1 & (1 \ll 3)     | \begin{tabular}{l}✓
✓
✗
✓
✓
✓\end{tabular} |
| 11 | 1 \ll 2           | \begin{tabular}{l}✓
✓
✓
✓
✓
✗\end{tabular} |

5.3.2 Subset characteristics

The characteristics of the subsets are visualized in the Venn diagram in Figure 15.
The rationale for these characteristics is as follows:

\[ \text{syn} \subseteq \text{sem} \]

The IDEAS framework prescribes that if \( x \) and \( y \) are syntactically the same, \( x \) and \( y \) must also be semantically equivalent.

\[ \text{dep} \subseteq \text{sem} \]

If all delta pairs from \( x \) and \( y \) are semantically equivalent, then \( x \) and \( y \) must be semantically equivalent. The evaluation of expressions takes place without side-effects: a random rewriting in \( x \) does not have an effect on any other part of \( x \).

\[ \text{rul} \subseteq \text{sem} \]

A valid rule is a correct way of evaluating expressions and hence the expressions must be semantically equivalent.

\[ \text{str} \subseteq \text{rul} \]

If \( y \) follows from \( x \) by applying a valid rule according to a strategy, then \( y \) must also follow from \( x \) by applying a valid rule. Strategies combine valid rules.

\[ \text{syn} \cap (\text{rul} \cup \text{bug}) = \emptyset \]

If \( x \) and \( y \) are syntactically the same, then no rule must have been applied. The reverse must also hold: if \( y \) follows from \( x \) by applying any rule, \( x \) and \( y \) cannot be syntactically the same.

\[ \text{dep} \cap \text{bug} \neq \emptyset \]

If all delta pairs from \( x \) and \( y \) are semantically equivalent, then still \( y \) can follow from \( x \) by applying a buggy rule. This is for instance true for operators, where a buggy rule describes the transformation with the operands reversed, but due to the value of the operands, the delta pairs still are semantically equivalent. See example 11 in Table 1.

\[ \text{dep} \cap \text{syn} \neq \emptyset \]

If \( x \) and \( y \) are identical expressions, they are syntactically the same and all nodes are identical. See example 7 in Table 1.

\[ \text{dep} \cap \text{rul} \neq \emptyset \]

If all delta pairs from \( x \) and \( y \) are semantically equivalent, a valid rule might have been applied. See examples 9 and 10 in Table 1.

\[ \text{rul} - \text{dep} \neq \emptyset \]

A valid rule might have been applied, although not all delta pairs from \( x \) and \( y \) are semantically equivalent. See example 4 in Table 1.

\[ \text{dep} - (\text{rul} \cup \text{bug} \cup \text{syn}) \neq \emptyset \]

If all delta pairs from \( x \) and \( y \) are semantically equivalent, and no valid or buggy rule has been applied, and \( x \) and \( y \) are not syntactically the same, then probably multiple evaluation steps have been applied at once. See example 8 in Table 1.
5.3.3 Diagnosing two expressions

Based on the subset characteristics, a pair \((x, y)\) is diagnosed as follows when submitted for validation:

\((x, y) \in E^2 - (\text{sem} \cup \text{bug})\)

**Not equivalent, unknown mistake**

These are the elements in \(E^2\) for which \(x\) and \(y\) are not equivalent and \(y\) does not follow from \(x\) by applying a buggy rule.

\((x, y) \in \text{bug} - \text{sem}\)

**Not equivalent, common mistake with buggy rule**

These are the elements in \(E^2\) for which \(x\) and \(y\) are not equivalent and \(y\) follows from \(x\) by applying a buggy rule.

\((x, y) \in \text{str} \cap \text{dep}\)

**Rewrite step expected by expert strategy**

These are the elements in \(E^2\) for which \(y\) follows from \(x\) by applying a valid rule according a strategy and all delta pairs are equivalent.

\((x, y) \in (\text{rul} - \text{str}) \cap \text{dep}\)

**Correct rewrite step, but detour from strategy**

These are the elements in \(E^2\) for which \(y\) follows from \(x\) by applying a valid rule, but not according a strategy and all delta pairs are equivalent.

\((x, y) \in \text{dep} - (\text{bug} \cup \text{syn} \cup \text{rul})\)

**Equivalent, correct rewrite step, but unknown**

These are the elements in \(E^2\) for which all delta pairs from \(x\) and \(y\) are equivalent, but \(y\) does not follow from \(x\) by applying a valid or buggy rule.

The five diagnoses above are also available in the default diagnose service in the IDEAS framework. A more specific diagnoses can be added.

\((x, y) \in \text{syn} - \text{dep}\)

**Small rewrite step, not recognized, AC-rewritten**

These are the elements in \(E^2\) for which \(x\) and \(y\) are syntactically the same with one or more AC-rewritings.

\((x, y) \in \text{syn} \cap \text{dep}\)

**Small rewrite step, not recognized, identical**

These are the elements in \(E^2\) for which \(x\) and \(y\) are identical.

Additionally, five new diagnoses can be added.

\((x, y) \in \text{sem} - (\text{dep} \cup \text{bug} \cup \text{rul} \cup \text{syn})\)

**Equivalent, wrong rewrite step, unknown mistake**
These are the elements in $E^2$ for which $x$ and $y$ are semantically equivalent, but one or more delta pairs are not semantically equivalent, and $y$ does not follow from $x$ by applying any rule, and $x$ and $y$ are not syntactically the same.

$(x,y) \in \text{rul} - (\text{dep} \cup \text{str})$

**Equivalent, wrong rewrite step, detour from strategy**

These are the elements in $E^2$ for which $x$ and $y$ are semantically equivalent, but one or more delta pairs are not semantically equivalent, and $y$ follows from $x$ by applying a valid rule, but not according a strategy.

$(x,y) \in \text{str} - \text{dep}$

**Equivalent, wrong rewrite step, expected by strategy**

These are the elements in $E^2$ for which $x$ and $y$ are semantically equivalent, but one or more delta pairs are not semantically equivalent, and $y$ follows from $x$ by applying a valid rule according a strategy.

$(x,y) \in (\text{sem} \cap \text{bug}) - \text{dep}$

**Equivalent, wrong rewrite step, common mistake with buggy rule**

These are the elements in $E^2$ for which $x$ and $y$ are semantically equivalent and $y$ follows from $x$ by applying a buggy rule.

$(x,y) \in \text{bug} \cap \text{dep}$

**Equivalent, correct rewrite step, common mistake with buggy rule**

These are the elements in $E^2$ for which $x$ and $y$ are semantically equivalent and all delta pairs are semantically equivalent and $y$ follows from $x$ by applying a buggy rule.

All of these twelve diagnoses are disjoint subsets of $E^2$. In other words, the union of the subsets is equal to $E^2$. Therefore, all elements $(x,y) \in E^2$ can only be an element of exactly one of these diagnoses.

5.3.4 Implementation

The implementation for such a diagnosis can be realized in several ways. One way is to calculate for each of the subsets if $(x,y) \in E^2$ is an element or not. This yields a truth table that can be used for diagnosis, as shown in Table 2. In this table, all diagnoses that are impossible, such as $(x,y)$ being an element of all subsets, are omitted. Only seven diagnoses are supported in the prototype, because adding more requires significant changes to the IDEAS framework, such as updating the encoders. This is beyond the scope of this research project. Therefore, several of the above diagnoses are grouped with a don’t care, indicated by a hyphen ‘-‘ in Table 2.

If a structure is to be implemented as discussed by Heeren & Jeuring (2014), a binary decision tree helps to determine which subsets must be checked in which order. Such a tree is presented in Appendix D. The tree in this appendix shows, for instance, that if $(x,y) \notin \text{sem}$, then the only other subset that needs to be checked is ‘bug’ to determine if a buggy rule can be recognized or not.
Table 2. Truth table for diagnosing an element \((x, y) \in E^2\) in the prototype. The highlighted diagnosis is new.

<table>
<thead>
<tr>
<th>sem</th>
<th>dep</th>
<th>bug</th>
<th>syn</th>
<th>str</th>
<th>rul</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>Not equivalent, unknown mistake</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>Not equivalent, common mistake with buggy rule</td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>-</td>
<td>Equivalent, wrong rewrite step, unknown mistake</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>Small rewrite step, not recognized</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Equivalent, correct rewrite step, but unknown</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>Correct rewrite step, but detour from strategy</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>Rewrite step expected by expert strategy</td>
</tr>
</tbody>
</table>

The prototype for this research project implements the truth table approach for diagnosing. The implementation of only the truth table part of the custom diagnose service is presented in Code snippet 19.

Code snippet 19. Diagnosing two expressions with a truth table.

```haskell
diagnose :: State a -> Context a -> Diagnosis a
diagnose state new =
  case (sem, dep, bug, syn, str, rul) of
    (False, False, False, False, False, False) ->
      NotEquivalent ""
    (_ , False, True , False, False, False) ->
      let (r, as) = fromJust (discovered True Nothing)
          in Buggy as r
    (True , False, False, False, _, _) ->
      Unknown (finished state) state
    (True , _, False, True , False, False) ->
      Similar (finished state) state
    (True , True , False, False, False, False) ->
      Correct (finished restarted) restarted
    (True , True , False, False, False, True ) ->
      let (r, as) = fromJust (discovered False Nothing)
          in Detour (finished restarted) restarted as r
    (True , True , False, True , False, True ) ->
      let ((r, _, _), ns) = fromJust expected
          in Expected (finished ns) ns r
      ->
      NotEquivalent ""
    where
      sem = equivalence ex (stateContext state) new
      dep = equivalentDeltaPairs ex (stateContext state) new
      bug = isJust (discovered True Nothing)
      syn = similar
      str = isJust expected
      rul = isJust (discovered False Nothing)
```

The function `equivalentDeltaPairs` takes an exercise and two generic data types in their context as a parameter and returns a Boolean to indicate if \((x, y) \in dep\). The generic data types are casted from their context to the exercise-specific data type, of
which the delta pairs are calculated and compared using the function for semantic equivalence as discussed in Section 5.2.7. As this semantic equivalence function takes a substitution environment as a parameter, this environment must also be known to the diagnose service. This is realized by saving the substitution environment as an exercise-specific ‘extra property’ upon exercise creation and reading it from the exercise in the equivalentDeltaPairs function.
Three surveys have been conducted for data acquisition: two surveys amongst students and one survey amongst instructors. The setup of these surveys is discussed in Section 6.1. The quantitative results are discussed in Section 6.2, by analysing the results from the surveys and analysing entries from the logging database. Section 6.3 discusses the qualitative results.

6.1 Survey setup

Two groups of students with different domain knowledge have been asked to participate in the same survey. The first group are main phase Electrical and Electronic Engineering bachelor students from HAN University of Applied Sciences. They participated December 20th, 2016. The second group are first year Electrical and Electronic Engineering bachelor students from HAN University of Applied Sciences. They participated January 9th and 12th, 2017. Prior to each survey, a fifteen minute classroom instruction was given that explains tutoring systems in general, expression evaluation, and how to use the prototype. After that, the students were pointed to the questions in an online survey which they have answered independently for 30 minutes. The survey consisted of the following four sections:

1. Introductory questions, e.g. what year a student is in.
2. Solving a task: expression evaluation 1 of 2.
4. Concluding questions, e.g. how useful a tutoring system such as the prototype is to the student.

Students have been asked in sections 2 and 3 of the survey to evaluate an expression by using the prototype. They have been instructed to use two or three operators in each expression, and to use at least one variable, register or definition. This ensures that several evaluation steps have to be taken, but at the same time it will not take too long to completely evaluate an expression.

Colleagues at HAN University of Applied Sciences have been asked to participate in the survey for instructors. They participated January 16th, 2017. Prior to the survey, a 20 minute classroom instruction was given to introduce tutoring systems in general, how to use the prototype from a student’s point of view, and how to use the prototype from an instructor’s point of view. After that, the instructors were pointed to the questions in an online survey which they have answered independently for 30 minutes. The survey consisted of the following four sections:

1. Introductory questions, e.g. how important it is to an instructor that students learn how expressions evaluate.
2. Solving a task: expression evaluation.
3. Customizing the tutor.
4. Concluding questions, e.g. how useful a tutoring system such as the prototype is for students.

The main purpose of the quantitative results from all surveys is the validation of the implementation of the prototype. Quantitative data is acquired by counting responses from the surveys and log database. The main purpose of the qualitative results is validating if the prototype generates the right feedback messages and therefore contributes in understanding how microcontroller I/O programming expressions evaluate. Qualitative data is acquired by asking closed questions with a five point Likert scale and open questions.

6.2 Quantitative results

Section 6.2.1 discusses the quantitative results from the surveys. Section 6.2.2 presents the quantitative results from analysing the log entries.

6.2.1 Survey analysis

An overview of the quantitative results from the surveys is presented in Table 3. The table shows the number of participants per group and characteristics of the expressions that were evaluated in the surveys. The 46 participants have evaluated 88 expressions, of which 37 unique expressions. The number of ‘unique expressions in normal form’ is different, because a minor syntactic difference, such as additional braces, is also considered a unique expression. All the initial expressions and the evaluated normal forms were written down in the surveys. These are presented in Appendix E. All calculated normal forms are correct according to the participant, which means that the improved implementation of the diagnose service with a truth table works as designed.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total participants</td>
<td>46</td>
</tr>
<tr>
<td>First year students</td>
<td>18 39 %</td>
</tr>
<tr>
<td>Main phase students</td>
<td>25 54 %</td>
</tr>
<tr>
<td>Instructors</td>
<td>3 7 %</td>
</tr>
<tr>
<td>Expressions evaluated in surveys</td>
<td>88</td>
</tr>
<tr>
<td>Unique initial expressions</td>
<td>37 42%</td>
</tr>
<tr>
<td>Unique expressions in normal form</td>
<td>41 47%</td>
</tr>
<tr>
<td>Normal form is correct according to participant</td>
<td>88 100%</td>
</tr>
<tr>
<td>Participant has asked for one or more hints during evaluation</td>
<td>85 97%</td>
</tr>
</tbody>
</table>

6.2.2 Log analysis

This section presents quantitative results from the logging database and highlights several logging sessions. Table 4 presents an overview of the quantitative results. The
table shows how many times a service was requested, how many syntax errors were generated for each service, and the diagnoses calculated by the ‘Feedback text’ service. Besides showing the totals, the table also show these numbers for each of the three groups.

All participants generated 4029 log entries. The web application was started 91 times, which can be concluded from the number of times the ‘List of exercises’ service is used. Examples are requested more often, because when a participant selects another exercise the ‘List of examples’ service is used. There were 499 attempts to create a new task. This succeeded 247 times, because 31 expression were already in a normal form, and 221 expressions contained a syntax error.

Table 4. Results from quantitative analysis of the log database.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Totals</th>
<th>1st year students</th>
<th>Main phase students</th>
<th>Instructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log entries recorded in database</td>
<td>4029</td>
<td>1580</td>
<td>2333</td>
<td>85</td>
</tr>
<tr>
<td>List of exercises service</td>
<td>91</td>
<td>2 %</td>
<td>28</td>
<td>57</td>
</tr>
<tr>
<td>List of examples service</td>
<td>423</td>
<td>11 %</td>
<td>89</td>
<td>318</td>
</tr>
<tr>
<td>Create new task service</td>
<td>499</td>
<td>12 %</td>
<td>292</td>
<td>198</td>
</tr>
<tr>
<td>Expression is in normal form when created</td>
<td>31</td>
<td>1 %</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>Substitute service</td>
<td>248</td>
<td>6 %</td>
<td>89</td>
<td>153</td>
</tr>
<tr>
<td>One first text service</td>
<td>1113</td>
<td>28 %</td>
<td>471</td>
<td>632</td>
</tr>
<tr>
<td>Feedback text service</td>
<td>1532</td>
<td>38 %</td>
<td>566</td>
<td>928</td>
</tr>
<tr>
<td>Stop clicked</td>
<td>92</td>
<td>2 %</td>
<td>45</td>
<td>47</td>
</tr>
<tr>
<td>Syntax errors</td>
<td>344</td>
<td></td>
<td>217</td>
<td>120</td>
</tr>
<tr>
<td>Create new task service</td>
<td>221</td>
<td>64 %</td>
<td>172</td>
<td>46</td>
</tr>
<tr>
<td>Feedback text service</td>
<td>116</td>
<td>34 %</td>
<td>45</td>
<td>67</td>
</tr>
<tr>
<td>Substitute service</td>
<td>1</td>
<td>0 %</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>One first text service</td>
<td>6</td>
<td>2 %</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Diagnosis</td>
<td>1416</td>
<td></td>
<td>521</td>
<td>861</td>
</tr>
<tr>
<td>Not equivalent, unknown mistake</td>
<td>190</td>
<td>13 %</td>
<td>64</td>
<td>124</td>
</tr>
<tr>
<td>Common mistake with buggy rule</td>
<td>2</td>
<td>0 %</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Small rewrite step, not recognized</td>
<td>52</td>
<td>4 %</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>Rewrite step expected by expert strategy</td>
<td>840</td>
<td>59 %</td>
<td>347</td>
<td>479</td>
</tr>
<tr>
<td>Correct rewrite step, but detour from strategy</td>
<td>18</td>
<td>1 %</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>Equivalent, wrong rewrite step, unknown mistake</td>
<td>145</td>
<td>10 %</td>
<td>33</td>
<td>106</td>
</tr>
<tr>
<td>Equivalent, correct rewrite step, but unknown</td>
<td>169</td>
<td>12 %</td>
<td>39</td>
<td>120</td>
</tr>
</tbody>
</table>

The ‘Substitute’ service is automatically called after successfully creating a task. However, one of the expressions in the ‘Substitute’ service still contained a syntax error. This session is shown in Table 5. The table shows in ascending order the
services requested by the front-end and the response from the domain reasoner. Log entry 1 shows that the created expression is accepted by the grammar as described in Section 5.2.1. The pretty printer, however, returned the expression `while(! ! ! false){;}`, which is not accepted by the grammar when used in the ‘Substitute’ service in log entry 2.

The ‘One first text’ service is used for both generating hints and to show the solution to a next step. The six syntax errors logged in the ‘One first text’ service were caused by the same pretty printer problem.

<table>
<thead>
<tr>
<th>#</th>
<th>Request Service</th>
<th>Parameter 1</th>
<th>Error message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create</td>
<td><code>while(!(!false)){;}</code></td>
<td>Syntax error 1:1: unexpected ‘w’ expecting &quot;0B&quot;, &quot;0X&quot;, &quot;0b&quot;, &quot;0x&quot;, &quot;false&quot;, &quot;true&quot;, '{', '}', end of input, integer, or letter</td>
</tr>
<tr>
<td>2</td>
<td>Substitute</td>
<td><code>while(! ! false)){;}</code></td>
<td></td>
</tr>
</tbody>
</table>

When omitting the syntax errors introduced by the pretty printer, 337 log entries contain a syntax error, as a result of a participant’s input. The ‘Create new task’ service and ‘Feedback text’ service are the only services that contain user submitted expressions. This means that 17% of the submitted expressions contains a syntax error. Of the expressions with syntax errors, 64% is submitted by first year students.

The ‘Feedback text’ service is used for diagnosing expressions. This service uses the new diagnose service as discussed in Section 5.3 and returns a textual feedback message. A total of 1416 pairs of expressions that were submitted for diagnosis did not contain a syntax error. Table 6 shows the log entries for a student session from creating an initial expression until it is rewritten into the normal form. The log entries show the requested service, and for the ‘One first text’ service the reason for using that service between braces. The parameters show the submitted expression or, in case of a diagnosis, the pair of expressions. The response shows the diagnosis and the recognized rule calculated by the domain reasoner. After successfully creating a new task with the ‘Create new task’ service in log entry 1, the ‘Substitute’ service is used to request a value for variable x in log entry 2. The student uses the ‘One first text’ service to make the tutor show a possible next step and uses the ‘Feedback text’ service to diagnose this. This sequence is repeated until a normal form is presented, basically creating a worked-out example.

<table>
<thead>
<tr>
<th>#</th>
<th>Request Service</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>Diagnosis</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create</td>
<td><code>y = ((x + 3) &lt;&lt; 4);</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Substitute</td>
<td><code>y = ((x + 3) &lt;&lt; 4);</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Onefirsttext (show)</td>
<td><code>y = ((x + 3) &lt;&lt; 4);</code></td>
<td><code>y = ((15 + 3) &lt;&lt; 4);</code></td>
<td>Expected</td>
<td>Replace</td>
</tr>
<tr>
<td>4</td>
<td>Feedbacktext</td>
<td><code>y = ((x + 3) &lt;&lt; 4);</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The log entries in Table 7 show how a student creates a task and tries to validate several expressions directly in normal form, without taking any intermediate step. Log entries 3 and 4 show that the parser correctly parses expressions with or without a semicolon. In log entry 7, the variable \( g \) is not known to this exercise, and as a result the expression \( a = g \) is correctly diagnosed as not being equivalent. In log entry 8, however, the variable \( x \) is equal to 15 in the selected exercise. This is not presented to the student, because it is not in the created expression in log entry 1. In this situation the diagnosis still is correct, because \( 2 \& (1 << 3) \) is not semantically equivalent to 15. But when the initial expression would have been \( 14 + 1 \), the ‘Feedback text’ service would have returned the diagnosis ‘Equivalent’ without directly being apparent for the student what the substitution value of \( x \) is. This might be solved by requesting the ‘Substitute’ service after each submitted expression. It could also motivation instructors to only create context specific exercises, rather than one exercise with multiple contexts combined.

<table>
<thead>
<tr>
<th>#</th>
<th>Request Service</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>Response Diagnosis Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Onefirsttext (show)</td>
<td>( y = ((15 + 3) &lt;&lt; 4); )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Feedbacktext</td>
<td>( y = ((15 + 3) &lt;&lt; 4); )</td>
<td>( y = (18 &lt;&lt; 4); )</td>
<td>Expected</td>
</tr>
<tr>
<td>7</td>
<td>Onefirsttext (show)</td>
<td>( y = (18 &lt;&lt; 4); )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Feedbacktext</td>
<td>( y = (18 &lt;&lt; 4); )</td>
<td>( y = (0b00010010 &lt;&lt; 4); )</td>
<td>Expected</td>
</tr>
<tr>
<td>9</td>
<td>Onefirsttext (show)</td>
<td>( y = (0b00010010 &lt;&lt; 4); )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Feedbacktext</td>
<td>( y = (0b00010010 &lt;&lt; 4); )</td>
<td>( y = 0b10000000; )</td>
<td>Expected</td>
</tr>
<tr>
<td>11</td>
<td>Onefirsttext (show)</td>
<td>( y = (0b00010010 &lt;&lt; 4); )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Feedbacktext</td>
<td>( y = (0b00010010 &lt;&lt; 4); )</td>
<td>( y = 0b10000000; )</td>
<td>Expected</td>
</tr>
</tbody>
</table>

The results in Table 4 show that 10% of the submissions for diagnosis, the pairs of expressions were diagnosed with the new diagnosis ‘Equivalent, wrong rewrite step, unknown mistake’. This shows that this kind of diagnosis is relevant for this domain. The log entries in Table 8 show an example of this diagnosis. The student creates an expression and the substitution values are calculated by the domain reasoner. Log entry 3 shows the pair of expressions submitted for diagnosis. The value for UDRE0 is equal to five, but the student makes mistake by substituting UDRE0 for \( 0b0000110 \). Both expressions, however, yield the same normal form. This is
correctly calculated by the domain reasoner with the diagnosis ‘Unknown’. After the student has used the ‘One first text’ service for both a hint and showing the next step in log entries 4 and 5, a pair of expressions is submitted for diagnosis in log entry 6.

Table 8. First entries from a logging session, showing how a student submits a semantic equivalent expression, but takes a wrong step (session id 3c57c1c6c3205ca43a35).

<table>
<thead>
<tr>
<th>#</th>
<th>Request Service</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>Response</th>
<th>Diagnosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create</td>
<td>while(! (UDRE0</td>
<td>PORTB) ) { ; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Substitute</td>
<td>while(! (UDRE0</td>
<td>PORTB) ) { ; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Feedbacktext</td>
<td>while(! (UDRE0</td>
<td>PORTB) ) { ; }</td>
<td>while(0b0000110</td>
<td>0b00001111)</td>
</tr>
<tr>
<td>4</td>
<td>Onefirsttext</td>
<td>while(! (UDRE0</td>
<td>PORTB) ) { ; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Onefirsttext</td>
<td>while(! (UDRE0</td>
<td>PORTB) ) { ; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Feedbacktext</td>
<td>while(! (UDRE0</td>
<td>PORTB) ) { ; }</td>
<td>while( ! (5</td>
<td>PORTB) ) { ; }</td>
</tr>
</tbody>
</table>

6.3 Qualitative results

This section analyses the open and closed questions from the surveys. The answers to the closed questions are presented by means of stacked bar charts. In these charts, the left (green) bars indicate first year students, the middle (blue) bars indicate major phase students, and the right (orange) bars indicate instructors. The number in a bar represents the number of respondents. The questions are answered on a five point Likert scale. The levels of each Likert item are shown on the left-hand side of the chart. The average score is shown on the right-hand side.

All participants agree that it is important to understand how microcontroller I/O programming expressions evaluate, which gives some legitimacy for the development of this tutoring system:

For each of the implemented feedback subcategories, the qualitative results are discussed in the remainder of this section.
Task-processing rules

Textual information on how to approach a task is presented when the web application is started. Most participants think this is sufficient, but for some participants it is not clear how to start with a task.

![Bar chart](chart1.png)

These result show that TPR feedback is appreciated and helps to start with a task, which is the first step towards a solution.

Compiler errors

The answers to the open questions are presented in Appendix F. To the open question ‘What improvement(s) do you recommend?’ three answers directly refer to feedback messages related to syntax errors. Four answers refer to improving error messages, which might also be related to syntax errors. These answers point out that the feedback messages related to compiler errors are hard to understand. The messages are probably not descriptive enough for helping students to solve syntactic problems. This seems especially true for first year students, because the results in Table 4 point out that the majority of syntax errors is made by this group.

Solution errors

A feedback message after each step is useful to students:

![Bar chart](chart2.png)
After making a mistake, students appreciate the ability to ask the tutoring system for a solution:

These results show that feedback messages after each step, or the ability to ask for the solution to a step, help students towards a solution and therefore helps them to understand the evaluation of microcontroller I/O programming expressions.

**Task processing steps**

Generally speaking, participants find the ability to ask for a hint very useful:

Whenever the participant asked for a hint, most participants think that the hint helped them to solve the task. Students were asked to do two evaluation tasks, and instructors were asked to do one evaluation task. Therefore, the students answered the same question twice:
These results show that generally speaking, feedback messages with information about the next step, such as hints, help students towards a solution. The results also show that to some students the hints are not helpful. The survey had no question to clarify this.

The overall conclusion from the qualitative results is that the prototype helps students to understand the evaluation of microcontroller I/O programming expressions. The average scores show that detailed feedback and hint messages for each step are useful to students.

Two answers to the open questions suggest that if a step is diagnosed as not obvious or incorrect, the tutor should explain why. This suggests to additionally implement the feedback subcategory EC, although Section 5.2.8 argued not to.
This section presents work that is related to this research project. Section 7.1 describes related work concerning intelligent tutoring systems. Other tools and methods for the evaluation of expressions are discussed in Section 7.2. Finally, Section 7.3 discusses similarities and differences with respect to other programming tutors.

7.1 Intelligent tutoring systems

The goal of an ITS is to give students individual feedback and guide them to a solution in much the same way a human tutor would do. An ITS can interpret complex strategies and an ITS can learn as it operates (Koedinger & Tanner, 2013). ITSs, however, will not take over the role of a human teacher. They should be used as a complement in the teaching process (VanLehn, 2006).

Many ITSs have been developed for different problem domains. The classical LISP tutor (Farrell et al., 1984) provides students with a series of LISP programming exercises. It interacts with the students and helps them to solve problems. An evaluation study shows that the tutor is about twice as effective as classroom instruction. Ask-Elle is a tutor for the functional programming language Haskell (Gerdes et al., 2016). It helps students step-by-step to develop Haskell programs by providing feedback and hints for each step, although the program might be incomplete. Keuning developed a prototype for an imperative programming tutor (Keuning et al., 2014). This prototype also guides students step-by-step towards a solution that can be solved by multiple strategies and allows instructors to adapt the feedback messages by annotating model solutions. The goal of J-LATTE is teaching a subset of the Java programming language. Learning the language syntax and program design are separated (Holland et al., 2009). Advanced Geometry Tutor assists students in geometry theorem proving. It has been developed as a research tool to compare learning outcomes (Matsuda & VanLehn, 2005). Improving a student’s writing proficiency can be done with Writing Pal. Individualized formative feedback is automatically generated based on a variety of algorithms that are used for assessing essays (Roscoe et al., 2014).

Generally speaking, tutoring systems are described as having two loops. The outer loop executes once for each task, where a task is solving a problem that can be broken down into a sequence of steps. The inner loop executes once for each step and provides feedback and hints to guide a student towards a solution. The inner loop also assesses the student’s level of competence, which is used by the outer loop to select the task the student should do next (VanLehn, 2006). There are many types of tutoring systems (Shute & Psotka, 1996). A well-established approach is called model-tracing, which is developed by Anderson and his colleagues at Carnegie-Mellon
University. The approach works by “delineating many hundreds of production rules that model curricular ‘chunks’ of cognitive skill.” A student’s progress of the acquisition of these so-called knowledge chunks, is tracked by the system and feedback is generated.

Although developing software for microcontrollers differs from developing software for personal computers, in both situations students must acquire knowledge and skills about a programming language. Intelligent tutoring systems for the programming domain are therefore closely related to tutors for microcontroller software development. Although there are a lot of tools available that offer programming exercises and provide automated feedback (Keuning et al., 2016), we are not aware of such a tutor for the specific goal of teaching microcontroller I/O programming expression evaluation.

7.2 Evaluation of programming expressions

This section describes three related tools and methods for the evaluation of expressions within a programming context.

7.2.1 Debugger

A common way of inspecting the evaluation of imperative programs is by using the stepper functionality of a debugger. This is also applicable for microcontroller programs. Although microcontroller programs can often be debugged with a simulator, on-chip debugging is nowadays possible at very low cost. An on-chip debugger implements additional hardware within the microcontroller that halts all the internals of a microcontroller and transmits internal data to a host. This internal data represents the system’s state and can be inspected by a student. The host can also transmit commands to the on-chip debugger, e.g. to change the value of a variable or to execute the next statement.

Figure 16 shows how the on-chip debugger of an ATmega328P microcontroller is connected to the Atmel Studio IDE during a debugging session. The screenshot shows the student’s program in the upper-left window. The yellow highlighted statement is where the program halted and which will be executed next. The upper-right window shows which peripherals are available for this microcontroller and the contents of the registers for the selected peripheral. The bottom-left window can be used to inspect the value of variables.

By stepping through the program, a student sees which statements are executed over time based on the microcontroller’s internal state. When stepping over a composite expression, only the result can be inspected. A debugger does not show the step-by-step evaluation of composite expressions. In general, debuggers are not designed to give feedback or hints on the evaluation of a program.
7.2.2 Online tutor on expression evaluation

There are not many programming tutors available in the ITS community to learn the top-down evaluation of expressions. One other tutoring system for this purpose is developed by Kumar (2005) and teaches the step-by-step evaluation of programming expressions in C++/Java. The online tutor helps a student to learn the evaluation of expressions by generating a problem, letting the student solve it and grading the solution. Expressions can be provided by the instructor, or can be generated randomly. The tutor ensures that a student never sees the same problem twice.

Figure 17 shows a screenshot of the tutor for practicing with bitwise operators. When the student starts a new problem, the left pane presents the expression to be evaluated. At this moment, the right pane shows help information on how expressions must be entered. The student is expected to solve each expression one operator at a time. When the student drags the mouse across a sub-expression, the tutor presents a dialog box to enter the intermediate result. When the intermediate result is submitted, the tutor provides two types of feedback. The first is the colour of the underbrace. It is green if the student selected the correct subexpression according to precedence rules, otherwise it will be coloured red. The second type of feedback is the colour of the intermediate result. It is coloured green if the answer is correct and otherwise it is red. In case of an incorrect step and a student does not know how to proceed, the student can undo the incorrect step and try another step until it is marked green. Finally, a student submits the solution and the detailed feedback in

---

the right pane as depicted in Figure 17 is presented. If there is time remaining, which is shown below the right pane, the student is presented a new problem. In case the student does not know how to proceed, and clicks the ‘Don’t Know’ button, the entire evaluation with detailed feedback is provided in the right pane.

Kumar’s tutoring system is made out of problets. A problet is a Java applet built to automatically generate problems calibrated to the needs of a student (Kumar & Singhal, 2000). A problet consists of two components. The first component is a problem template with background information, the stem of the problem, the response options and the format of the feedback. The second component is visualization tools, which support the problem-solving environment. Problets can be used by students for practicing and testing, and by instructors for designing tests.

![Figure 17. Practicing with bitwise operators in Kumar’s online tutor on expression evaluation.](image)

7.2.3 Evaluating Haskell expressions in a tutoring environment

Olmer et al. (2014) developed a prototype of a tutor for evaluating Haskell expressions. The prototype supports students in the understanding of Haskell
programming concepts and evaluation strategies by showing the step-by-step evaluation of Haskell expressions. The tutor supports two evaluation strategies, innermost and outermost, and besides inspecting the evaluation steps it is also possible to practice evaluation steps. A screenshot of the practice part in the front-end of the tutor is depicted in Figure 18. A student selects an expression and presses the start button. The student submits the next evaluation step and clicks the diagnose button to have the input diagnosed by the tutor. In case of a valid step, the applied rule and step are appended to the derivation pane. In case of an invalid step, the tutor provides feedback in the output pane. If a student does not know how to proceed, several hints can be provided by the tutor, e.g. the number of steps left or all rules that can be applied according to the strategy. The prototype uses the IDEAS framework to rewrite expressions and uses its services to diagnose an evaluation step and to give hints about the next evaluation step.

7.3 Comparison to other programming tutors

There are many intelligent tutors available for learning programming. This section describes how this tutor for microcontroller I/O programming expressions fits within this broad landscape. For this purpose, the comprehensive list of tools for learning programming, as identified by Gómez-Albarrán (2005), is used. She classifies the tools in four distinct categories and describes the main features for each of these categories. In the remainder of this section, first a rationale is given for the category the prototype belongs to. Then the features of this particular category are compared to the prototype to identify differences and similarities with other tools for learning programming.

The four categories identified by Gómez-Albarrán are:
7 RELATED WORK

- **Tools with a reduced development environment**
  Tools in this category address the problem that development environments can be hard for novice students to comprehend. This problem is solved by presenting simplified and reduced development environments that students can use to program, compile, and debug a program. The prototype for this research project does qualify for this category, because it uses a web application with a simplified user interface. On the other hand, tools in this category can be used to learn the bottom-up creation of programs and the prototype of this research project is for learning the top-down evaluation of expressions.

- **Example-based environments**
  In example-based environments, examples serve as a basis to solve similar problems. The way these examples are presented varies within these systems. The prototype for this research project does qualify for this category, because examples are provided in feedback messages generated for the EXP and TPS subcategories.

- **Tools based on visualization**
  Tools in this category use mental images of a program’s behaviour to explain algorithms. By animating these images, the explanation becomes dynamic and interactive. The prototype for this research project does not qualify for this category, because textual code is presented instead of a mental image.

- **Simulation environments**
  In simulation environments, the execution of a program is reflected by the behaviour of the inhabitants of an imaginary world. This world is observed by students or students take active part in it. The prototype for this research project does not qualify for this category, because expressions are evaluated and no programs are executed. Furthermore, the prototype does not reflect an imaginary world.

From these observations, the ITS for learning the evaluation of microcontroller I/O programming expressions is a tutor similar to tutors in the example-based environments category. Table 9 shows the features of example-based environments as discussed by Gómez-Albarrán, and how these features are implemented in the prototype. The descriptions for implementation is similar to the descriptions presented in (Gómez-Albarrán, 2005).
Table 9. Features in example-based environments (Gómez-Albarrán, 2005) and their implementation in the prototype.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Implementation in the prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example granularity</td>
<td>Evaluation of each operator for imperative programming expressions in a customizable programming language.</td>
</tr>
<tr>
<td>Example selection approach(es)</td>
<td>Collaborative: the system selects the examples appropriate for an exercise and the student selects an example from a list.</td>
</tr>
<tr>
<td>Exploration of examples sensitive to student knowledge</td>
<td>Not available, because this is a responsibility of the outer loop which is beyond the scope of this research project.</td>
</tr>
</tbody>
</table>
| Support for exercise solving                        | A form-based web application to submit expressions for evaluation. Support on demand:  
  • Contextualized help (a hint for the next step to solve the exercise, the solution for the next step to solve the exercise, explanations) and feedback.  
  • Syntax error detection.  
  • Intelligent analysis of task solutions. |
| Access to student-solved exercises                  | Not available. |
| Other capabilities                                   | Student action logging.  
  Customizable environment. |

Based on the feature descriptions presented in Table 9, the prototype for microcontroller I/O programming expressions has features that are most similar to the tutoring tools ELM-PE (Weber & Möllenberg, 1994) and Javy (Gómez-Martín et al., 2003). The main difference with these tools is that the prototype is specific for the domain of microcontroller I/O programming, the prototype is a tutoring system for learning expression evaluation, and the prototype has a customizable environment, which allows for easy support of multiple microcontrollers and imperative programming languages.
7 RELATED WORK
The goal of this research project is to investigate how an ITS can be created by using the IDEAS framework, for helping students to understand the evaluation of microcontroller I/O programming expressions. This section interprets the result by discussing threats to validity, limitations and future work.

There are not many ITSs available for learning the evaluation of imperative programming expressions. One other tutoring system for this purpose is developed by Kumar (2005) and teaches the step-by-step evaluation of programming expressions in C++/Java. Kumar’s tutor helps students towards a solution by letting the student know if the submitted subexpression is correct or incorrect. In contrast, this research project also enables students to ask for a hint and to submit your own expressions. The IDEAS framework has already been used in a tutoring system for evaluating Haskell expressions by Olmer et al. (2014). Olmer’s tutor, however, is a tutor for the functional programming paradigm, whereas this research project is generally speaking intended for the imperative programming paradigm and specifically speaking for microcontroller I/O programming.

The reader should bear in mind that the timeframe for this research project has been approximately 25 weeks. As a consequence, the timeframe for validating the results was limited and therefore the students and instructors asked for participating in the surveys were selected based on convenience. This causes a threat to generalizing the results. Furthermore, all participating students were taking a course from the researcher at the moment the surveys were held and participation was on a voluntary basis. Another threat to generalizing the results is that only three instructors participated and they are all colleagues of the researcher. However, by selecting participants with different domain knowledge the results are not limited to one group.

When this research project started, it was expected that the IDEAS framework can be used to generate feedback and hint messages from a student expressions, an exercise and instructor feedback, because this framework has been used in past projects for creating domain reasoners, such as tutoring systems described by Olmer et al. (2014) and Lodder et al. (2016). It was unexpected that the default diagnose service could not be used for diagnosis, although the same problem has been described by Lodder et al. (2016). By designing, implementing, and validating a new equivalence relation based on delta pairs, this problem is solved. The results show that this solution works as designed. In fact, the solution is of interest to all domains that allow incorrect transformations on the data type that still lead to semantically equivalent expressions and therefore is an important contribution to the IDEAS project. Further research
should address the question if and how this solution can be adopted by the IDEAS framework. In that light it is worth noting that a similar problem is seen in the tutoring systems for imperative programming (Keuning, 2014) and functional programming (Gerdes et al., 2016). If a submitted program in these tutoring systems does not correspond to a model solution, these tutoring systems are still capable of diagnosing if the produced output is correct. Therefore, instead of allowing incorrect transformations, these domains allow unknown transformations that produce correct output.

By adding this new relation to the framework an interesting question arises. Are there more relations that can be calculated between expressions and if so, what new diagnoses will be possible? A literature study could be a starting point for answering these questions. At least one new relation to consider is the same semantic equivalent delta pairs relation with AC-rewritings taken into account. Further research should focus on the new set of diagnoses this relation makes possible, but more importantly, should address the research question if such diagnoses will help students to learn better for the domain of microcontroller I/O programming expression evaluation, but also for other domains.

At the start of the research project, it was expected that the IDEAS framework can be used to support multiple microcontrollers and programming languages. The results show that this expectation comes true, because the implementation behaves as expected. Dynamically creating exercises from files is also seen in the tutor for functional programming (Gerdes et al., 2016). Using a lookup environment is an obvious approach for substituting parts of expressions, such as definitions and variables. It was a challenge to make this environment available throughout an exercise and also make it available in the services.

An easy way of adding new programming languages to the IDEAS framework has been proposed by Keuning et al. (2014) and also by Jeuring et al. (2012). This research project demonstrates how this is done for a limited set of imperative programming expressions that share the same abstract syntax and therefore contributes to an answer for that question. A research question that still is unanswered is how programming languages that do not share an abstract syntax can be supported by the IDEAS framework without the need of recompilation.

The results show that the nature of the feedback messages that guide students step-by-step towards a solution are important for learning. It was not expected that students also prefer feedback messages for the error correction subcategory, because students already have the possibility to ask for a hint. A threat to the validity of these conclusions is that the conclusions are not based on statistically significant results, but based on the interpretation of qualitative results. Moreover, the survey’s questions were written from a student’s point of view, which made it easier for students to fill in the survey. It required, however, additional interpretation of the results.
Although the results look very promising, the research project focuses on a very small set of programming expressions for the specific domain of microcontroller I/O programming. Further research could be undertaken to explore how these results can be used for the domain of imperative programming in general. Also adding a student model for tracking a student’s progress and selecting tasks is an important area for further research and of interest to the IDEAS project.

Finally, Gilibert et al. (2006) have noted to introduce microcontrollers by practical exercises. This prototype uses a web application and no development environment or microcontroller. It would be very interesting to explore the idea of combining an expression evaluator with the stepper functionality of a debugger. This allows students to learn the evaluation of expressions in the context of an executing program.
9 CONCLUSION

This thesis describes the design, implementation, and validation of an ITS prototype for the domain of microcontroller I/O programming. The overall conclusion is that the IDEAS framework can be used to support multiple microcontrollers and programming languages and that the step-by-step guidance helps students to understand how microcontroller I/O programming expressions evaluate. The remainder of this section concludes the research project by answering the research questions.

i. How can feedback and hints be generated from a student expression, an exercise and instructor feedback using the IDEAS framework?

A grammar is defined for parsing typical microcontroller I/O programming expressions. Sentences belonging to the grammar are parsed to a domain-specific data type. A pretty printer turns the data type into a human readable string. Four groups of rules are defined for data type transformations. Strategies are defined for rewriting typical microcontroller I/O programming expression into a normal form. A custom diagnose service is realized for diagnosing expressions for this domain. The IDEAS framework (v1.5) assumes that if two expressions are semantically equivalent, the student always takes a correct evaluation step. This is not necessarily true for this domain, but also not for other domains. The problem is solved with the introduction of a new homogeneous binary relation called ‘delta pairs’. This new relation calculates semantic equivalence for all maximum subtrees that are different for the two submitted expressions. If at least one of the differences is not semantically equivalent, the student took an incorrect evaluation step. This new relation allows for two more finer grained diagnoses and five new diagnoses compared to the default diagnose service. The conclusion that can be drawn from the results is that the new diagnoses are relevant for the domain and that the implementation works as designed.

ii. How can multiple microcontroller definitions be supported?

Exercises are dynamically generated by reading configuration files. Microcontroller definitions are added to each exercise by supplying the path to one or more definition files. These files are parsed and the definitions are stored in a lookup environment. This environment is used whenever a definition or variable must be substituted. To make support for multiple microcontrollers as easy as possible, several parsers for parsing definition files are available in the prototype.

iii. How can multiple imperative programming languages be supported?
Support for multiple programming languages is realized by allowing instructors to customize tokens from the grammar. For each exercise, a language definition is read from a configuration file. Similar to the microcontroller definitions, the tokens and their values are stored in a lookup environment. This environment is used for parsing and pretty printing.

iv. What types of feedback and hints help students to understand the evaluation of microcontroller I/O programming expressions?

The nature of feedback messages have been categorised by Keuning et al. (2016) by extending Narciss’s (2008) content-related feedback components with eleven representative subcategories. The conclusion drawn from the results of the surveys is that it helps students to understand the evaluation of microcontroller I/O programming expressions, when the nature of feedback messages is related to explanations on subject matter (EXP), solution errors (SE), and task-processing steps (TPS). These types of messages guide students step-by-step towards a solution.

v. What are differences and similarities between this tutor for learning the evaluation of microcontroller I/O programming expressions and existing programming tutors?

Programming tutors can be categorised in four categories (Gómez-Albarrán, 2005). The prototype for learning the evaluation of microcontroller I/O programming expressions is similar to tutors in the example-based environments category, because the prototype also uses examples as a basis to solve similar problems. These examples are provided in feedback messages generated in the EXP and TPS subcategories. The main difference with these tools is that the prototype created for this research project is specific for the domain of microcontroller I/O programming, the prototype is a tutoring system for learning expression evaluation, and the prototype has a customizable environment.


**A EXERCISE CONFIGURATION FILE**

```xml
<example val="while (!((UCSR0A & (1<<UDRE0))) {;}" />
<example val="PORTB = (PORTB & 249) | ((x << 1) & 6);" />
<example val="a = 1 & 1<<3;" />

<microcontroller val="ATmega328P" />
<language val="ANSI-C" />
<wordlength val="8" />

<deffile val="/../mick/definitions/iom328p.h" type="CHeader" />
<languagefile val="/../mick/languages/ansic.xml" />
<scriptfile val="/../mick/scripts/atmega328p.txt" />

<variable name="UCSR0A" val="0b00001111" />
<variable name="UCSR0B" val="0b11110000" />
<variable name="PORTB" val="0b00001111" />
<variable name="x" val="15" />
```
Although ANSI-C does not support the concept of Boolean variables, true and false are used by the tutor to visualize the evaluation of conditional statements.
Example script file.
Three dots (...) indicate that more text follows on that line.

String oke = That is correct.
string incorrect = That is incorrect.

string youRewroteInto = you rewrote @diffbefore into @diffafter.
string appliedRule = The step is correct, but is not obvious.

string tryAgain = Try again.

string suggested
  | @hasexpected = Use step @expected.
  | true = Use another step.

string askForHint
  | not @oldready = You may ask for a hint.
  | true = {} # empty

# Feedback
feedback same = This expressions has no rewritings, it is ...
feedback noteq = @incorrect @tryAgain @askForHint
feedback unknown = Although the expressions are equivalent, you ...
feedback correct = @oke
feedback ok = @oke
feedback buggy = @incorrect @recognized
feedback detour = @appliedRule

feedback hint
  | @hasexpected = @expected.
  | true = Sorry, no hint available.

# Rules
text rule.add = evaluate the addition operator
text rule.bitwiseand = evaluate the @bitwiseand operator
text rule.bitwiseor = evaluate the @bitwiseor operator
text rule.shiftdown = evaluate the @shiftdown operator
text rule.logicalnot = evaluate the @logicalnot operator
text rule.tobool = rewrite this number to its boolean ...
text rule.dectobin = rewrite this @dectobin representation
text rule.dectohex = rewrite this @dectohex representation
text rule.bintodec = rewrite this @bintodec representation
text rule.bitohex = rewrite this @bitohex representation
text rule.hextodec = rewrite this @hextodec representation
text rule.hextobin = rewrite this @hextobin representation
text rule.substitute = substitute this definition, register or ...

# Buggy rules
text rule.buggy.shiftdown = The operands of the @shiftdown ...

# Links
string bitwiseand = <a href="https://en.wikipedia.org/wiki/ ...
string bitwiseor = <a href="https://en.wikipedia.org/wiki/ ...
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string shiftleft</td>
<td></td>
</tr>
<tr>
<td>string logicnot</td>
<td></td>
</tr>
<tr>
<td>string dectobin</td>
<td></td>
</tr>
<tr>
<td>string dectohex</td>
<td></td>
</tr>
<tr>
<td>string bintodec</td>
<td></td>
</tr>
<tr>
<td>string bintohex</td>
<td></td>
</tr>
<tr>
<td>string hextodec</td>
<td></td>
</tr>
<tr>
<td>string hextobin</td>
<td></td>
</tr>
</tbody>
</table>
D DIAGNOSIS WITH A BINARY DECISION TREE
Expressions evaluated in surveys

First year students

<table>
<thead>
<tr>
<th>Initial expression</th>
<th>Normal form</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTB = ((PORTB</td>
<td>((x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>while(! (UCSR0A + (3 &lt;&lt; UDRE0)) ) { ; }                                        while (false) { ; }</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b00000000;</td>
<td></td>
</tr>
<tr>
<td>while((x &lt;&lt; 122) ) { ; }                                                             while(True) ;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                0b00000000</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = 0b00000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b00000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 123)</td>
<td>(x</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b00000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b00000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>x = (x &lt;&lt; ! y);                                                                      x = (x &lt;&lt; ! y);</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 255)</td>
<td>(x &lt;&lt; 1) &amp; 15));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b000000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b000000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b000000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b000000000;</td>
<td></td>
</tr>
<tr>
<td>- PORTB = ((PORTB &amp; 249)</td>
<td>(x &lt;&lt; 1) &amp; 6));                                               PORTB = (0b00001111);</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));                                                                a = 0b000000000;</td>
<td></td>
</tr>
</tbody>
</table>
### Initial expression vs Normal form

<table>
<thead>
<tr>
<th>Expression</th>
<th>Normal form</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIOB-&gt;BSRR = (GPIO_BSRR_BS_2</td>
<td>GPIOB-&gt;BSRR = 0x1100;</td>
</tr>
<tr>
<td>GPIO_BSRR_BS_3);</td>
<td></td>
</tr>
<tr>
<td>while( (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>ONWAAR</td>
</tr>
<tr>
<td>a = (4 &amp; (1 &lt;&lt; 5));</td>
<td>a = 0b00000000;</td>
</tr>
<tr>
<td>while( (!UCSR0A &amp; (1 &lt;&lt; UDRE0)) ) { }</td>
<td>while( true );</td>
</tr>
<tr>
<td>while( (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>while( true )</td>
</tr>
<tr>
<td>while( (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>while( true )</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));</td>
<td>a = 0;</td>
</tr>
<tr>
<td>GPIOB-&gt;BSRR = (GPIO_BSRR_BS_2</td>
<td>GPIOB-&gt;BSRR = 0b0001000100000000;</td>
</tr>
<tr>
<td>GPIO_BSRR_BS_3);</td>
<td></td>
</tr>
<tr>
<td>GPIOB-&gt;BSRR = (GPIO_BSRR_BS_2</td>
<td>GPIOB-&gt;BSRR = 0b0001000100000000</td>
</tr>
<tr>
<td>GPIO_BSRR_BS_3);</td>
<td></td>
</tr>
<tr>
<td>while( (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>while( true )</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));</td>
<td>a = 0</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));</td>
<td>a = 0b00000000;</td>
</tr>
<tr>
<td>while( (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>while(1);</td>
</tr>
<tr>
<td>while( ! (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>while(1);</td>
</tr>
<tr>
<td>a = (1&lt;&lt;2</td>
<td>(1 &lt;&lt; 3));</td>
</tr>
<tr>
<td>while( (x</td>
<td>y) &amp; (3 &lt;&lt; 1)) ) { }</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));</td>
<td>a = 0</td>
</tr>
<tr>
<td>a = (0b00000010 &amp; (1 &lt;&lt; 3));</td>
<td>a = 0b00000000;</td>
</tr>
<tr>
<td>while( (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>while(1);</td>
</tr>
<tr>
<td>PORTB = ((PORTB &amp; 249) &amp; ((x &lt;&lt; 1) &amp; 6));</td>
<td>PORTB = (0b00001111);</td>
</tr>
<tr>
<td>while( !(UCSR0A &amp; (1 &lt;&lt; UDRE0)) ) { }</td>
<td>while(1);</td>
</tr>
<tr>
<td>GPIOB-&gt;BSRR = (GPIO_BSRR_BS_2</td>
<td>GPIOB-&gt;BSRR = 0b0001000100000000</td>
</tr>
<tr>
<td>GPIO_BSRR_BS_3);</td>
<td></td>
</tr>
<tr>
<td>while( !(UCSR0A &amp; (1 &lt;&lt; UDRE0)) ) { }</td>
<td>while( true );</td>
</tr>
<tr>
<td>PORTB = ((PORTB &amp; 249)</td>
<td>((x &lt;&lt; 1) &amp; 6));</td>
</tr>
<tr>
<td>PORTB = ((PORTB &amp; 249)</td>
<td>((x &lt;&lt; 1) &amp; 6));</td>
</tr>
<tr>
<td>PORTB = ((PORTB &amp; 249)</td>
<td>((x &lt;&lt; 1) &amp; 6));</td>
</tr>
<tr>
<td>while( (I2C1-&gt;ISR &amp; I2C_ISR_STOPF) ) { }</td>
<td>while( true );</td>
</tr>
<tr>
<td>GPIOB-&gt;BSRR = (GPIO_BSRR_BS_2</td>
<td>GPIOB-&gt;BSRR = 0b0001000100000000</td>
</tr>
<tr>
<td>GPIO_BSRR_BS_3);</td>
<td></td>
</tr>
<tr>
<td>while( !(UCSR0A &amp; (1 &lt;&lt; UDRE0)) ) { }</td>
<td>while( true );</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));</td>
<td>a = 0</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));</td>
<td>a = 0b00000000;</td>
</tr>
<tr>
<td>a = (2 &amp; (1 &lt;&lt; 3));</td>
<td>a = false;</td>
</tr>
</tbody>
</table>
E. Expressions evaluated in surveys

- \( a = (2 \& (1 << 3)) \);
- \( \text{PORTB} = ((\text{PORTB} \& 249) | ((x << 1) \& 6)); \)
- while( !(\text{UCSR0A} \&(2 \ll UDRE0)) ) { ; }  
- \( y = x + 3 << 4; \)
- while( !(\text{I2C1->ISR} \& \text{I2C_ISR_STOPF} ) ) { ; } 
- \text{GPIOB->BSRR} = (0x0100 | 0x1000); 
- while( !(\text{UCSR0A} \&(1 \ll UDRE0)) ) { ; } 
- \( a = (2 \& (1 << 3)); \)
- \( a = (2 \& (1 << 3)); \)
- while( !(0 + 0) ) { ; }  
- \text{GPIOB->BSRR} = (\text{GPIO_BSRR_BS_2} | \text{GPIO_BSRR_BS_3}); 

<table>
<thead>
<tr>
<th>Instructors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial expression</strong></td>
</tr>
<tr>
<td>while( !(\text{UCSR0A} &amp;(1 \ll UDRE0)) ) { ; }</td>
</tr>
<tr>
<td>( \text{PORTB} = ((\text{PORTB} &amp; 249)</td>
</tr>
<tr>
<td>while( !(\text{UCSR0A} &amp;(1 \ll UDRE0)) ) { ; }</td>
</tr>
<tr>
<td>( y = 0b100100000 )</td>
</tr>
<tr>
<td>( a = (0b00000000000000000000000000000000); )</td>
</tr>
<tr>
<td>( a = (0b00000000000000000000000000000000); )</td>
</tr>
<tr>
<td>while(80) { ; } // but does not work</td>
</tr>
</tbody>
</table>
E EXPRESSIONS EVALUATED IN SURVEYS
ANSWERS TO OPEN QUESTIONS

The answers are taken verbatim from both English and in Dutch.

**Question: What do you particularly like about this tutoring system?**

**First year students**
- Copying the former statement
- this is a nice system. I would like to use it.
- stepwise explanation
- You can practice with more types of microcontrollers and programming languages
- Zeer overzichtelijk, en onzettend leerzaam
- Dat het hele kleine stapjes per keer doet.
- Je kunt overzichtelijk de stappen zien het proces.
- het is enwoudig te gebruiken
- Het stap voor stap zien wat er gebeurd, geeft een hoop duidelijkheid
- Het is handig te gebruiken. Eventueel ook te gebruiken voor CPROG. In het eerste blok moest je leren werken met opperators, op deze manier kun je antwoorden gemakkelijk controleren.
- makkelijk te gebruiken, en snel leren
- Het werkt heel gemakkelijk en heeft meestal goede tips. Erg handig als je er nog niet echt kennis van hebt ook
- Yes to a certian degree.
- it could be usefull if people dont understand the subject, if they do it wouldn’t be as helpfull
- I like that you can see where you have problems with and what you understand. This makes it more accurate if you learn for a exam.
- That if you have any problems with in your program then you can check here if it works
- I am intrigued by the idea of a program that can validate and break down your expressions.

**Main phase students**
- accessibility
- Simplicity of use
- that it shows the different steps that you have to take to the result.
- you get hints on how to make the step to the next step.
- the layout seems ok
- It could be nice to evaluate complicated conditions/statements.
- its easy to use.
- It is easy to calculate values if you have a lot of binary operations
- That it is able to show every single step cleary
- the hint option
- The step by step approach
- the received feedback when there is a mistake and doing a lot of these calculations will improve your understanding of these constructions.
- The simple user interface, that you can ask for hints, and that you can make it show the answer per step.
- Intuitive and copying previous statements
- It's also usefull to create your own statements
- direct feedback
- Hoe eenvoudig je elke stap uitlegt.
- you don’t need to have access to a microcontroller to learn the basic logic operations.
- Step by step breakdown.
- Very easy to understand
- It is a good way to learn how to read microcontroller code.

Instructors
- Teaching students without teacher help
- Handige tool, voor gebruik bij flipped-classroom
- Het geven van de stappen om tot een oplossing te komen

Question: What improvement(s) do you recommend?

First year students
- Copying earlier statements and if you give the final answer at the start, the steps that you have skipped have to be shown.
- more specific hints, and syntax errors
- implementation of functions and what they do.
- Hints that explain better what you did wrong
- Meer mogelijkheden (registers en dergelijke) De start balk onder het evaluation form plaatsen lijkt nu namelijk net alsof het veld wat jij in moet vullen automatisch gevuld gaat worden met een expressie
- Meer operators toevoegen.
- Duidelijk maken wanneer er geen hint kan worden gegeven.
- als je op hint drukt dat dan niet je tekst word gewist
- Dat je terug kan kijken wat er bij stappen gebeurd, het commentaar/hints terug kan zien.
- Error tekst iets uitgebreider, kijken of je aan kunt geven waarom het fout is
- Het zou makkelijk zijn als je terug zou kunnen kijken naar de stappen.
- Niet echt kunnen vinden. misschien als je iets gebruikt wat niet mag sneller errors. Wil wel nog een keer even kijken
- I would like to see more information about the mistakes I made.
- when i tried to put 6 in binary ( to have everything in binary ) it said it was wrong because i had to do some operators first, even though i prefer having everything in binary before using the operators.
- I would recommend testing this with students that know how to write complex expressions. It did not seem clear right away what I needed to do. A small detailed description would be appreciated
If you did an operations but didn’t remove the logical operator that it gives you the hint to remove it and not just say you need to do the operator.

**Main phase students**
- If more things have changed say which of the changes is incorrect
- Have standard hints (not as excessive as the current hint, just for the general gist of what is to be done) show up after each step since this tool is meant to teach people with little knowledge about the evaluations. People with prior knowledge will not mind as well.
- It is not really clear what steps have to be taken, and in what sequence they have to be entered.
- Some steps are so simple, like filling in the variables that this can be done in a single step
- Better indication of the expected next step (hint always visible maybe?)
- Copy function does not work for me. Putting explanation text on top.
- Maybe it would be easier to put spaces between every four characters. For example, if you have a line with 30 binary characters, it would be easier to recognize which bit is which.
- It can be hard to translate a number to the binary form, especially if you have a 32-bit microcontroller because its a lot of zeroes. Maybe it’s best to only have assignments with 8-bit numbers and use the 32-bit as a solver rather than an assignment.
- Hint doesn’t clear the text bar
- Het antwoord met show pas laten zien naar een x aantal pogingen
- When the feedback says that the step is not obvious, I like to know why it is not obvious. (Same for you took a wrong step
- More examples and more debug messages
- Show in the information field that the exercise is completed.
- More supported statements
- Teacher should be able to disable hints
- More questions
- Geen
- Fix some things I perceive as bugs. Examples: picking an microcontroller with an expression that has register(s) that belong to another microcontroller is allowed, this is not what I expected, after starting I clicking show+validate and it showed: this is incorrect, that was not what I expected. The implications of UNKNOWN were vague to me in this situation.
- When I skipped a lot of steps and filled in while( 245 ) ; ; and clicked validate there was no error message. This was an obviously wrong answer, but not getting an error message was weird.
- If the user translates a register to a value, include which value was wrong
- When hint is pressed, do not empty the field.
- On the stm32 when working with bits (0b000 for example), the hints and answers are given as 16 bits numbers. This can be a bit confusing
- More error messages.

**Instructors**
- Some error messages related to syntax errors, difficult to read by novices
- Nog iets duidelijker hints
Question: Remarks (Optional)

First year students
- add more statements
- Leuk en handig te gebruiken!
- I like the idea and I would have used it if it was available in class especially if there were more available choices.

Main phase students
- perhaps some syntax highlighting?
- The Show-function makes it a little too easy to just click through. Why would anyone even try themselves? Students are lazy ;)
- Looks nice! !
- sommige opmerkingen staan in het nederlands
- Deze manier werkt wel degelijk. Alleen lijkt me het veel werk om dit allemaal te implementeren voor zoveel voorbeelden...
- Certainly going to use it!

Instructors
- Zeer bruikbare/zinvolle tool