Integrated Development Environment for LOOP, WHILE and GOTO

User Manual

The Theo-IDE Developers

Contents

Section 1

Introduction

This document serves as an introduction to the use of Theo-IDE, a program developed for a project named *Integrated Development Enviroment for LOOP, WHILE and GOTO* at the University of Applied Sciences Würzburg-Schweinfurt. LOOP, WHILE and GOTO [1] (hereafter referred to as *the project languages*) are the names of a range of simple concept programming languages used within lectures such as *Foundations of Theoretical Computer Science* and are used for demonstrating a range of concepts and isomorphisms in computability theory. Theo-IDE provides a simple explorative enviroment for developing, running and debugging programs written in these languages.

1.1 Requirements

During the project, great emphasis was placed on accessability and portability. As a result, Theo-IDE is available on a range of different desktop and mobile platforms. We officially support Linux (X11, Wayland), MacOS and Windows desktops and provide an application package for Android (ARMv7, ARMv8). If you're willing to get your hands dirty yourself, the project has also been verified to build for iOS.

1.2 Setup

Binary distributions for most major platforms are available in the project repository at **https://github.com/Theo-IDE/Theo-IDE/**. If there happens to be no binary artifact for your platform, the repository also includes a list of dependencies and build instructions.

Section 2

Basic Functionality

Once the Theo-IDE application is opened, you will be greeted with the user interface as pictured in **Figure 2.1**. Depending on the window width of your platform, the sidebar may be located at the bottom, and some of the buttons in the primary action bar at the top may be hidden in drop-down menus. We will now proceed with an explanation of the functionality of the Theo-IDE.

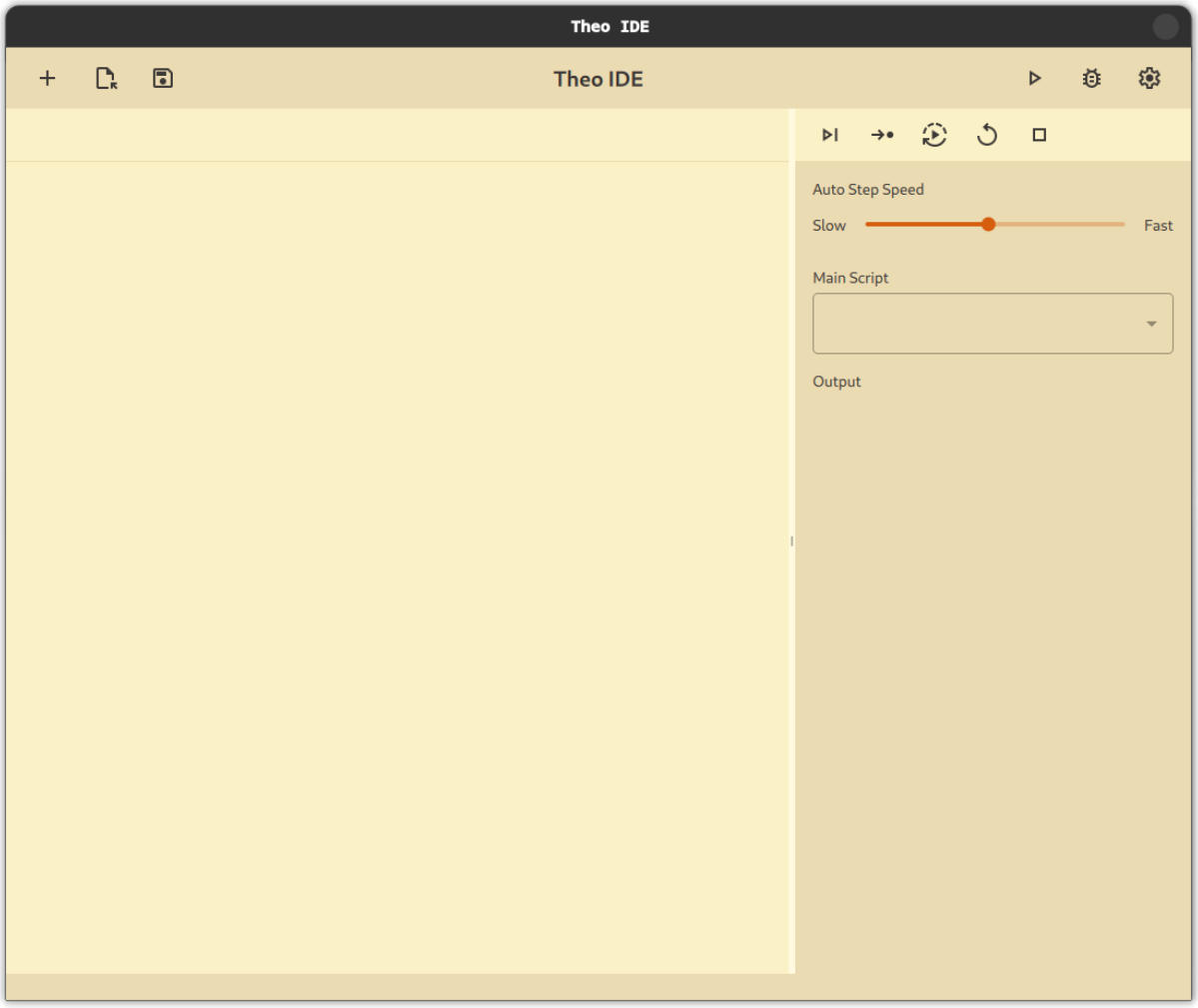

Figure 2.1 User Interface of Theo-IDE

The action bar at the top (which also contains the "Theo-IDE" label) is refereed to as the *primary action bar*. Below the primary action bar the view is split into the *editor* (on the left side in **Figure 2.1**) and the *sidebar*. The sidebar further is segmented into the *secondary action bar* at its top and the *control area* below it.

2.1 Creating, Opening and Saving Files

You may create new program files by clicking on the button with the plus icon in the primary action bar (the leftmost button in **Figure 2.1**). In situations where the window is narrow, this button may be hidden in a drop-down menu, which is accessible from the primary action bar.

Upon clicking this button, a new editor tab named *Temporary-** will open, and you'll be able to enter your source code.

You may load source code from files contained in your file system by clicking on the button with the file icon in the primary action bar (next to the creating button in **Figure 2.1**). In narrow windows, this button is hidden in the same drop-down menu as the opening button.

Upon clicking this button, a file dialog will open, prompting you for the files you wish to load. After confirming your selection of files, one editor tab will open for each of the files you opened, where the name of the tab is the file name in your file system.

Analogously to opening files, you may save modified or created files by clicking on the button with the save icon (to the right of the opening button in **Figure 2.1**).

This action will save whatever file is currently selected in the editor.

2.2 Running Programs

Running programs is a two-step process: First you select the main file of your program in the control area by selecting the name of the editor tab that contains it from the drop-down menu labeled *Main Script*. Secondly, in the primary action bar, you click on the button with the triangular icon.

This action will first compile your program. If any errors occur during compilation you will be shown a message dialog notifying you of the incident. If no errors occur, your program will be executed. When execution finishes successfully, a table showing the final variable contents will appear in the control area under the label *Output*. Should your program not halt for any reason, you may use the stop button in the secondary action bar (the one with the square iconography), to forcefully halt your program.

2.3 Debugging Programs

If you want to step through your program line-by-line or have the program halt on predefined breakpoints, you will want to make use of Theo-IDEs debugging capabilities.

At first, you will follow exactly the same process as when running a program, except that after selecting your main file, you will click on the debugging button (next to the run button in **Figure 2.1**, with the insect icon).

At this point, your program will once again be compiled, and any errors will be reported with a message dialog. But the program *won't* be executed straight away, and you will be able to make use of the buttons in the secondary action bar to influence execution:

- the leftmost button in **Figure 2.1** will execute the program until a breakpoint is encountered or the program ends (whichever happens first)
- the second button from the left in **Figure 2.1** will execute one line of code
- the third button from the left in **Figure 2.1** toggles auto-stepping mode: When enabled, Theo-IDE will automatically execute lines of code with a delay inbetween,

the length of which is altered with the slider titled *Auto Step Speed* in the control area

- the fourth button from the left in **Figure 2.1** resets execution to the beginning of the program
- the fifth button from the left in **Figure 2.1** stops execution

You may activate or deactivate breakpoints by clicking on the line number in the editor at the location where you want the program to pause. During execution, the output table in the control area will always show the current variable states.

Section 3

Language Extensions

We extended the syntax of LOOP, WHILE and GOTO languages in a number of ways either to make their use less tedious without altering their fundamental minimalistic nature or to allow the expression of more complicated syntax in terms of already established functionality. The Theo-IDE accepts programs written in the language *Theo*, which is essentially the project languages, but combined with the afformentioned extensions.

The main extensions are the following:

- multi-file programs using a “copy and paste” include directive
- subroutines that can be used to represent common calculations
- a powerful macro preprocessor that can be used to introduce custom syntax

We will now explain the differences between the project languages and Theo in more detail by showing some examples. We expect that the reader is already quite familiar with the use of the project languages.

Note: In our examples we only make use of `LOOP` syntax. `WHILE` or `GOTO` syntax works just like in the project languages [1] and may be used in the same way as shown in the examples.

3.1 Quality of Life

As already mentioned, we introduced some simple quality-of-life changes that are designed to make the use of Theo less tedious (but not less minimalistic) than the project languages.

As an example take a look at the following program:

```
x0 := 1 ;
// this is a comment !
_coolVariable1 := x0
```

The above program uses the following non-standard extensions:

- constant assignment: `x0 := 1`
- variable copying: `_coolVariable1 := x0`
- variable names that aren't x_n: `_coolVariable1`
- comments

As a result, the above program strictly speaking isn't written in the project languages, but *is* a valid Theo program.

3.2 Subroutines

Consider the following scenario: You have written a program that adds two variables together (a basic excercise when starting to learn the project languages). Now you want to add two variables together at multiple points in your program, but you don't want to copy the same code snippet everywhere. Theo allows the definition of samller *Programs* that you can define in your code and execute on will.

A program definition may look like this for the above scenario:

```
PROGRAM add IN x0, x1 OUT x0 DO
  LOOP x1 DO
    x0 := x0 + 1
  END
END
```

The identifiers after the `IN`-keyword are the input variables. The identifier listed after the `OUT`-keyword is the variable which is used as the result of the subroutine: When execution reaches the `END`-keyword of the program, the contents of this variable are copied to the caller.

You may call subroutines utilizing the `RUN`-keyword:

```
PROGRAM add IN x0, x1 OUT x0 DO
  LOOP x1 DO x0 := x0 + 1 END
END

arg := 4 ;
// x2 = arg + (arg + 5)
x2 := RUN add WITH
  arg,
  RUN add WITH arg, 5 END
END
```

The values passed to a subroutine between `WITH` and `END` are assigned to the identifiers of the `IN`-section. All identifiers used in a subroutine refer to values only accessible by the subroutine:

```
PROGRAM wrong IN x0 OUT x0 DO
  // b = 0, because all variables are initially 0
  b := a ;
  // this modifies the variable 'a' local to this scope
  a := 2
END
a := 1 ;
x0 := RUN wrong WITH 0 END
c := a
// c = 1, as the outer a isn't modified by the program call
```

In other words, you can't access variables of outer scopes, there are always only local variables. However you are able to call other programs as long as they were defined before their use.

Programs are always defined at the top of your code (before the actual program starts), and multiple program definitions are *not* separated by semicolons:

```
PROGRAM add IN x0, x1 OUT x0 DO
  LOOP x1 DO x0 := x0 + 1 END
END

PROGRAM mul IN x0, x1 OUT x2 DO
  x2 := 0 ;
  LOOP x1 DO
    x2 := RUN add WITH x2, x0 END
  END
END

x1 := 4 ;
x2 := 5 ;
x0 := RUN mul WITH x1, x2 END
```

3.3 Programs Consisting of Multiple Files

Consider the following scenario where you have two or more files in your Theo project. One named “math.theo”:

```
PROGRAM add IN x0, x1 OUT x0 DO
  LOOP x1 DO x0 := x0 + 1 END
END
```

And another one, which is the main file of your program:

```
INCLUDE \"math.theo\"
x1 := 2 ; x2 := 3 ;
x0 := RUN add WITH x1, x2 END
```

At the start of processing, the Theo compiler will insert the contents of “math.theo” at the position where it is included. As a consequence of this, you need to respect semicolon rules across file boundaries (possibly requiring a semicolon at the end of a file or after an include statement) and cyclic dependencies are strictly disallowed. We recommend using the include functionality like in the example to outsource program definitions.

Include statements may appear at any position in the source code.

3.4 Macro Preprocessor

Whilst subroutines endow the user with considerable power, they are still limited. The compiler used in the Theo-IDE comes with a powerful preprocessor with which you can define macro patterns and replacements for levying a higher level of syntactic modification.

Let's look at a simple example of macro usage. In the examples leading up to this point, we have almost always defined a subroutine `add`, which performs addition. We have then utilized this subroutine by writing `RUN add WITH value1, value2`, which is fine, but actually writing something like `value1 + value2` would have been way nicer.

The following example demonstrates how to translate code snippets of the form `value1 + value2` to the more verbose subroutine call:

```
PROGRAM add IN x0, x1 OUT x0 DO
  LOOP x1 DO x0 := x0 + 1 END
END

DEFINE
  // we look for patterns of "value1 + value2"
  <VALUE> + <VALUE>
AS
  // and replace with this
  // ($0 is the contents of the first <VALUE>, $1 of the second)
  RUN add WITH $0, $1 END
END DEFINE

x0 := 5 ; x1 := 4 ;
// gets translated to RUN add WTIH x0, x1 END
x2 := x0 + x1
```

Macro definitions always start with `DEFINE` and end with `END DEFINE`. Between these keywords, seperated by `AS`, you will first find the macro pattern that will be replaced and then the string that any occurence of the pattern will be replaced with.

In the macro pattern definition, you may use special *matcher tokens* which don't match any token in particular, but a pre-defined class of tokens or strings of tokens. The available matcher tokens are:

- `<VALUE>`, which includes any identifier (variable name), constant or subroutine call
- `<ID>`, which matches any identifier
- `<INT>`, which matches any constant
- `<ARGS>`, which matches comma-separated identifiers (such as `x0, x1, x2`)
- `<PROGRAM>`, which matches entire semicolon-separated program blocks (such as `x0 := 1 ; x1 := x0 + 1`), which includes any program according to the syntax rules of the project languages

With the powerful `<PROGRAM>` matcher, you can for example solve the common excercise of defining a "IF-THEN-ELSE" statement whilst only using `LOOP`:

```
DEFINE
  IF <VALUE> THEN <PROGRAM> ELSE <PROGRAM> END
AS
  #0 := $0 ;
  #1 := 0 ;
  #2 := 1 ;
  LOOP #0 DO
    #1 := 1 ;
    #2 := 0
  END ;
  LOOP #1 DO $1 END ;
  LOOP #2 DO $2 END
END DEFINE

x0 := 1 ;
IF x0 THEN
  x2 := 1
ELSE
  x2 := 2
END
```

In the above example we made use of scratch variables (variables that serve as temporary storage for a macro), which are variables of the form `#n`, where n is any natural number including zero.

If you have multiple macro definitions, and want to influence the order of replacement (which can matter in some instances), you can define a *replacement priority*:

```
... definitions of programs add and mul ...
DEFINE PRIO 10
  <VALUE> + <VALUE>
AS
  RUN add WITH $0, $1 END
END DEFINE

DEFINE PRIO 20
  <VALUE> * <VALUE>
AS
  RUN mul WITH $0, $1 END
END DEFINE
```

If at any point multiple macros could be replaced in the input, definitions with higher priority will always be replaced before definitions with lower priority.

In general, macros are replaced according to the following schema: Replace the macro definition with the highest priority, which was detected at the topmost, leftmost position with the longest sequence of tokens matched against it. Repeat this replacement until nothing changes or an upper limit of allowed replacements has been reached (this upper limit is hard-coded in order to avoid infinite replacing loops).

Macro definitions, just like `include` directives, may appear at any point in the source code and are " filtered out" at an early compilation step.

3.5 Formal Summary

This section is intended only for completeness (and not for learning). We will define a set of non-terminal symbols and grammars for syntactically correct include directives, macro definitions and transformed programs, which the informed reader may use to their benefit.

3.5.1 Non-Terminals

Table 3.1 lists all non-terminal symbols that the formal grammars will use.

Name	Regular Expression (FLEX)	Note
id	`[a-zA-Z_][a-zA-Z0-9_]*`	identifiers, variable names, program names
int	`(0|([1-9][0-9]*))`	constants
file	`\"[^\"]*\"`	file names
insert	`${int}`	matcher insertion point
local	`#{int}`	macro scratch varialbe
ws	`[ \t\n]+`	whitespace, removed before grammar application
comment	`\/\/.*`	comments, removed before grammar application
,	`\,`	program argument separator
;	`\;`	program statement separator
:	`\:`	label declarator
:=	`\:\=`	assignment operator
!= 0	`!= 0`	unequal zero operator
=	`=`	equality operator

Table 3.1a Non-Terminal Symbols of Theo

Name	Regular Expression (FLEX)	Note
run	`RUN|run|Run`	
with	`WITH|with|With`	
do	`DO|do|Do`	
loop	`LOOP|loop|Loop`	
while	`WHILE|while|While`	
goto	`GOTO|goto|Goto`	
if	`IF|if|If`	
then	`THEN|then|Then`	
stop	`STOP|stop|Stop`	
end	`END|end|End`	
program	`PROGRAM|program|Program|` `PROG|prog|Prog`	
in	`IN|in|In`	
out	`OUT|out|Out`	
include	`INCLUDE|include|Include`	
define	`DEFINE|define|Define|` `DEF|def|Def`	
as	`AS|as|As`	
priority	`PRIORITY|priority|Priority|` `PRIO|prio|Prio`	
enddefine	`END DEFINE|end define|` `End Define|ENDDEF|` `Enddef|enddef`	
<program>	`<PROGRAM>|<program>|` `<Program>|<PROG>|` `<prog>|<Prog>|<P>|<p>`	macro program matcher
<args>	`<ARGS>|<args>|<Args>|<A>|<a>`	macro argument chain matcher
<value>	`<VALUE>|<value>|<Value>|` `<VAL>|<val>|<Val>|<V>|<v>`	macro value matcher
<int>	`<INT>|<int>|<Int>`	macro constant separator

Table 3.1b Non-Terminal Symbols of Theo

Name	Regular Expression (FLEX)	Note
<id>	`<ID>|<id>`	macro identifier matcher
separator	.	any other sequence of characters (may be used for delimiters in macros, when you do not want `<VALUE>` to treat it as an identifier

Table 3.1c Non-Terminal Symbols of Theo

3.5.2 Include Directives

`include` statements are filtered out of the import at the first step of compilation. Include directives start with **include** and must be followed by **file**.

3.5.3 Macro Definitions

At the next step, macro definitions are extracted from the remaining input. The grammar in **Figure 3.1**, with starting symbol S, produces valid macro definitions.

S → ϵ | A S | M S

A → *any terminal symbol which is not* **define**, **as**, **enddefine**, **insert**, **local**, **<program>**, **<args>**, **<id>**, **<int>** *or* **<value>**

M → **define** P D **as** R **enddefine**

P → ϵ | **priority int**

D → B | B D

B → A | **<program>** | **<args>** | **<value>** | **<int>** | **<id>**

R → ϵ | C R

C → A | **local** | **insert**

Figure 3.1 Grammar for macro definitions

3.5.4 Theo

After the preprocessor is applied to the remaining input, the grammar in **Figure 3.2** is used to verify syntactic correctness.

S	→	def S | P
def	→	**program id** input output **do** P **end**
input	→	**in** L
L	→	**id** | L **,** L
output	→	ϵ | **out id**
P	→	P **;** P
P	→	A
P	→	**id :** A
A	→	**id := V**
A	→	**loop id do** P **end**
A	→	**while id != 0 do** P **end**
A	→	**goto id** | **stop**
A	→	**if id = int then goto id**
V	→	**id** | **int** | C
C	→	**run id with** args **end**
args	→	V | args **,** args

Figure 3.2 Grammar of Theo

References

[1] U. Schöning, *Theoretische Informatik - kurz gefasst* (Spektrum Akademischer Verlag, 2008).