

# Theory of Computer Science

## D2. LOOP- and WHILE-Computability

Malte Helmert

University of Basel

April 19, 2017

# Theory of Computer Science

## April 19, 2017 — D2. LOOP- and WHILE-Computability

D2.1 Introduction

D2.2 LOOP Programs

D2.3 Syntactic Sugar

D2.4 WHILE Programs

D2.5 Digression: the Ackermann Function

D2.6 Summary

## Overview: Computability Theory

### Computability Theory

- ▶ imperative models of computation:
  - D1. Turing-Computability
  - D2. LOOP- and WHILE-Computability
  - D3. GOTO-Computability
- ▶ functional models of computation:
  - D4. Primitive Recursion and  $\mu$ -Recursion
  - D5. Primitive/ $\mu$ -Recursion vs. LOOP-/WHILE-Computability
- ▶ undecidable problems:
  - D6. Decidability and Semi-Decidability
  - D7. Halting Problem and Reductions
  - D8. Rice's Theorem and Other Undecidable Problems
    - ~~Post's Correspondence Problem~~
    - ~~Undecidable Grammar Problems~~
    - ~~Gödel's Theorem and Diophantine Equations~~

## Further Reading (German)

### Literature for this Chapter (German)

Theoretische Informatik – kurz gefasst  
by Uwe Schöning (5th edition)

- ▶ Chapter 2.3
- ▶ Chapter 2.5

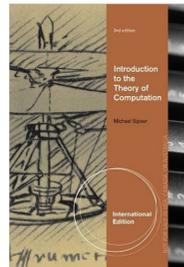


## Further Reading (English)

### Literature for this Chapter (English)

Introduction to the Theory of Computation  
by Michael Sipser (3rd edition)

- ▶ This topic is not discussed by Sipser!



## D2.1 Introduction

## Formal Models of Computation: LOOP/WHILE/GOTO

### Formal Models of Computation

- ▶ Turing machines
- ▶ LOOP, WHILE and GOTO programs
- ▶ primitive recursive and  $\mu$ -recursive functions

In this and the following chapter we get to know three simple models of computation (programming languages) and compare their power to Turing machines:

- ▶ LOOP programs  $\rightsquigarrow$  today
- ▶ WHILE programs  $\rightsquigarrow$  today
- ▶ GOTO programs  $\rightsquigarrow$  next chapter

## LOOP, WHILE and GOTO Programs: Basic Concepts

- ▶ LOOP, WHILE and GOTO programs are structured like programs in (simple) “traditional” programming languages
- ▶ use finitely many variables from the set  $\{x_0, x_1, x_2, \dots\}$  that can take on values in  $\mathbb{N}_0$
- ▶ differ from each other in the allowed “statements”

## D2.2 LOOP Programs

## LOOP Programs: Syntax

### Definition (LOOP Program)

**LOOP programs** are inductively defined as follows:

- ▶  $x_i := x_j + c$  is a LOOP program for every  $i, j, c \in \mathbb{N}_0$  (**addition**)
- ▶  $x_i := x_j - c$  is a LOOP program for every  $i, j, c \in \mathbb{N}_0$  (**modified subtraction**)
- ▶ If  $P_1$  and  $P_2$  are LOOP programs, then so is  $P_1; P_2$  (**composition**)
- ▶ If  $P$  is a LOOP program, then so is **LOOP**  $x_i$  **DO**  $P$  **END** for every  $i \in \mathbb{N}_0$  (**LOOP loop**)

**German:** LOOP-Programm, Addition, modifizierte Subtraktion, Komposition, LOOP-Schleife

## LOOP Programs: Semantics

### Definition (Semantics of LOOP Programs)

A LOOP program **computes** a  $k$ -ary function

$f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$ . The computation of  $f(n_1, \dots, n_k)$  works as follows:

- ① Initially, the variables  $x_1, \dots, x_k$  hold the values  $n_1, \dots, n_k$ . All other variables hold the value 0.
- ② During computation, the program modifies the variables as described on the following slides.
- ③ The result of the computation ( $f(n_1, \dots, n_k)$ ) is the value of  $x_0$  after the execution of the program.

**German:**  $P$  berechnet  $f$

## LOOP Programs: Semantics

### Definition (Semantics of LOOP Programs)

effect of  $x_i := x_j + c$ :

- ▶ The variable  $x_i$  is assigned the current value of  $x_j$  plus  $c$ .
- ▶ All other variables retain their value.

## LOOP Programs: Semantics

### Definition (Semantics of LOOP Programs)

effect of  $x_i := x_j - c$ :

- ▶ The variable  $x_i$  is assigned the current value of  $x_j$  minus  $c$  if this value is non-negative.
- ▶ Otherwise  $x_i$  is assigned the value 0.
- ▶ All other variables retain their value.

## LOOP Programs: Semantics

### Definition (Semantics of LOOP Programs)

effect of  $P_1; P_2$ :

- ▶ First, execute  $P_1$ .  
Then, execute  $P_2$  (on the modified variable values).

## LOOP Programs: Semantics

### Definition (Semantics of LOOP Programs)

effect of **LOOP**  $x_i$  **DO**  $P$  **END**:

- ▶ Let  $m$  be the value of variable  $x_i$  at the start of execution.
- ▶ The program  $P$  is executed  $m$  times in sequence.

## LOOP-Computable Functions

### Definition (LOOP-Computable)

A function  $f : \mathbb{N}_0^k \rightarrow_p \mathbb{N}_0$  is called **LOOP-computable** if a LOOP program that computes  $f$  exists.

**German:**  $f$  ist LOOP-berechenbar

**Note:** non-total functions are never LOOP-computable.  
(Why not?)

## LOOP Programs: Example

Example (LOOP program for  $f(x_1, x_2)$ )

```

LOOP  $x_1$  DO
  LOOP  $x_2$  DO
     $x_0 := x_0 + 1$ 
  END
END

```

Which (binary) function does this program compute?

## D2.3 Syntactic Sugar

## Syntactic Sugar or Essential Feature?

- ▶ We investigate the power of programming languages and other computation formalisms.
- ▶ **Rich** language features help when writing complex programs.
- ▶ **Minimalistic** formalisms are useful for proving statements over **all** programs.

↔ conflict of interest!

Idea:

- ▶ Use **minimalistic core** for proofs.
- ▶ Use **syntactic sugar** when writing programs.

German: syntaktischer Zucker

## Example: Syntactic Sugar

Example (syntactic sugar)

We propose five new syntax constructs (with the obvious semantics):

- ▶  $x_i := x_j$  for  $i, j \in \mathbb{N}_0$
- ▶  $x_i := c$  for  $i, c \in \mathbb{N}_0$
- ▶  $x_i := x_j + x_k$  for  $i, j, k \in \mathbb{N}_0$
- ▶ **IF**  $x_i \neq 0$  **THEN**  $P$  **END** for  $i \in \mathbb{N}_0$
- ▶ **IF**  $x_i = c$  **THEN**  $P$  **END** for  $i, c \in \mathbb{N}_0$

Can we simulate these with the existing constructs?

## Example: Syntactic Sugar

### Example (syntactic sugar)

$x_i := x_j$  for  $i, j \in \mathbb{N}_0$

Simple abbreviation for  $x_i := x_j + 0$ .

## Example: Syntactic Sugar

### Example (syntactic sugar)

$x_i := c$  for  $i, c \in \mathbb{N}_0$

Simple abbreviation for  $x_i := x_j + c$ ,  
where  $x_j$  is a fresh variable, i.e., an otherwise unused variable  
that is not an input variable.  
(Thus  $x_j$  must always have the value 0 in all executions.)

## Example: Syntactic Sugar

### Example (syntactic sugar)

$x_i := x_j + x_k$  for  $i, j, k \in \mathbb{N}_0$

Abbreviation for:

```
x_i := x_j;
LOOP x_k DO
  x_i := x_i + 1
END
```

Analogously we will also use the following:

- ▶  $x_i := x_j - x_k$
- ▶  $x_i := x_j + x_k - c - x_m + d$
- ▶ etc.

## Example: Syntactic Sugar

### Example (syntactic sugar)

IF  $x_i \neq 0$  THEN  $P$  END for  $i \in \mathbb{N}_0$

Abbreviation for:

```
x_j := 0;
LOOP x_j DO
  x_j := 1
END;
LOOP x_j DO
  P
END
```

where  $x_j$  is a fresh variable.

## Example: Syntactic Sugar

### Example (syntactic sugar)

IF  $x_i = c$  THEN  $P$  END for  $i, c \in \mathbb{N}_0$

Abbreviation for:

```

 $x_j := 1;$ 
 $x_k := x_i - c;$ 
IF  $x_k \neq 0$  THEN  $x_j := 0$  END;
 $x_k := c - x_j;$ 
IF  $x_k \neq 0$  THEN  $x_j := 0$  END;
IF  $x_j \neq 0$  THEN
   $P$ 
END

```

where  $x_j$  and  $x_k$  are fresh variables.

## Can We Be More Minimalistic?

- ▶ We see that some common structural elements such as IF statements are unnecessary because they are syntactic sugar.
- ▶ Can we make LOOP programs even more minimalistic than in our definition?

### Simplification 1

Instead of  $x_i := x_j + c$  and  $x_i := x_j - c$  it suffices to only allow the constructs

- ▶  $x_i := x_j,$
- ▶  $x_i := x_j + 1$  and
- ▶  $x_i := x_j - 1.$

Why?

## Can We Be More Minimalistic?

- ▶ We see that some common structural elements such as IF statements are unnecessary because they are syntactic sugar.
- ▶ Can we make LOOP programs even more minimalistic than in our definition?

### Simplification 2

The construct  $x_i := x_j$  can be omitted because it can be simulated with other constructs:

```

LOOP  $x_i$  DO
   $x_j := x_i - 1$ 
END;
LOOP  $x_j$  DO
   $x_i := x_j + 1$ 
END

```

## D2.4 WHILE Programs

## WHILE Programs: Syntax

### Definition (WHILE Program)

**WHILE programs** are inductively defined as follows:

- ▶  $x_i := x_j + c$  is a WHILE program for every  $i, j, c \in \mathbb{N}_0$  (**addition**)
- ▶  $x_i := x_j - c$  is a WHILE program for every  $i, j, c \in \mathbb{N}_0$  (**modified subtraction**)
- ▶ If  $P_1$  and  $P_2$  are WHILE programs, then so is  $P_1; P_2$  (**composition**)
- ▶ If  $P$  is a WHILE program, then so is **WHILE**  $x_i \neq 0$  **DO**  $P$  **END** for every  $i \in \mathbb{N}_0$  (**WHILE loop**)

**German:** WHILE-Programm, WHILE-Schleife

## WHILE Programs: Semantics

### Definition (Semantics of WHILE Programs)

The semantics of WHILE programs is defined exactly as for LOOP programs.

effect of **WHILE**  $x_i \neq 0$  **DO**  $P$  **END**:

- ▶ If  $x_i$  holds the value 0, program execution finishes.
- ▶ Otherwise execute  $P$ .
- ▶ Repeat these steps until execution finishes (potentially infinitely often).

## WHILE-Computable Functions

### Definition (WHILE-Computable)

A function  $f : \mathbb{N}_0^k \rightarrow_p \mathbb{N}_0$  is called **WHILE-computable** if a WHILE program that computes  $f$  exists.

**German:**  $f$  ist WHILE-berechenbar

## WHILE-Computability vs. LOOP-Computability

### Theorem

*Every LOOP-computable function is WHILE-computable.  
The converse is not true.*

WHILE programs are therefore **strictly more powerful** than LOOP programs.

**German:** echt mächtiger

## WHILE-Computability vs. LOOP-Computability

Proof.

Part 1: Every LOOP-computable function is WHILE-computable.

Given any LOOP program, we construct an equivalent WHILE program, i. e., one computing the same function.

To do so, replace each occurrence of `LOOP  $x_j$  DO  $P$  END` with

```

 $x_j := x_j$ ;
WHILE  $x_j \neq 0$  DO
   $x_j := x_j - 1$ ;
   $P$ 
END

```

where  $x_j$  is a fresh variable. ...

## WHILE-Computability vs. LOOP-Computability

Proof (continued).

Part 2: Not all WHILE-computable functions are LOOP-computable.

The WHILE program

```

 $x_1 := 1$ ;
WHILE  $x_1 \neq 0$  DO
   $x_1 := 1$ 
END

```

computes the function  $\Omega : \mathbb{N}_0 \rightarrow_p \mathbb{N}_0$  that is **undefined everywhere**.

$\Omega$  is hence WHILE-computable, but not LOOP-computable (because LOOP-computable functions are always total).  $\square$

## D2.5 Digression: the Ackermann Function

## LOOP vs. WHILE: Is There a Practical Difference?

- ▶ We have shown that WHILE programs are **strictly more powerful** than LOOP programs.
- ▶ The **example** we used is not very relevant in practice because our argument only relied on the fact that LOOP-computable functions are always **total**.
- ▶ To terminate for every input is not much of a problem in practice. (Quite the opposite.)
- ▶ Are there any **total** functions that are WHILE-computable, but not LOOP-computable?

## Ackermann Function: History

- ▶ **David Hilbert** conjectured that **all computable** total functions are primitive recursive (1926).  
 ~→ We will see what this means in Chapter D4.
- ▶ **Wilhelm Ackermann** refuted the conjecture by supplying a counterexample (1928).
- ▶ The counterexample was simplified by **Rózsa Péter** (1935).  
 ~→ [here](#): simplified version

## Ackermann Function

### Definition (Ackermann function)

The **Ackermann function**  $a : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0$  is defined as follows:

$$\begin{aligned} a(0, y) &= y + 1 && \text{for all } y \geq 0 \\ a(x, 0) &= a(x - 1, 1) && \text{for all } x > 0 \\ a(x, y) &= a(x - 1, a(x, y - 1)) && \text{for all } x, y > 0 \end{aligned}$$

**German:** Ackermannfunktion

**Note:** the recursion in the definition is bounded, so this defines a total function. (*Why?*)

## Table of Values

	$y = 0$	$y = 1$	$y = 2$	$y = 3$	$y = k$
$a(0, y)$	1	2	3	4	$k + 1$
$a(1, y)$	2	3	4	5	$k + 2$
$a(2, y)$	3	5	7	9	$2k + 3$
$a(3, y)$	5	13	29	61	$2^{k+3} - 3$
$a(4, y)$	13	65533	$2^{65536} - 3$	$2^{2^{65536}} - 3$	$\underbrace{2^{2^{\dots^2}}}_{k+3} - 3$

## Computability of the Ackermann Function

### Theorem

*The Ackermann function is WHILE-computable, but not LOOP-computable.*

(Without proof.)

## Computability of the Ackermann Function: Proof Idea

proof idea:

- ▶ **WHILE-computability:**
  - ▶ show how WHILE programs can simulate a stack (essentially: push/pop with *encode/decode* from Chapter D4)
  - ▶ dual recursion by using a stack
  - ↪ WHILE program is easy to specify
- ▶ **no LOOP-computability:**
  - ▶ show that there is a number  $k$  for every LOOP program such that the computed function value is smaller than  $a(k, n)$ , if  $n$  is the largest input value
  - ▶ proof by structural induction; use  $k =$  "program length"
  - ↪ Ackermann function grows faster than every LOOP-computable function

## D2.6 Summary

## Summary: LOOP and WHILE Programs

two new models of computation for numerical functions:

- ▶ **LOOP** programs and **WHILE** programs
- ▶ **closer to typical programming languages** than Turing machines

## Summary: Comparing Models of Computation

general approach to compare **power** of formalisms:

- ▶ How can features be used to **simulate** other features (cf. **syntactic sugar**, **minimalistic** formalisms)?
- ▶ How can one formalism simulate the other formalism?

## Power of LOOP vs. WHILE

We now know:

- ▶ **WHILE** programs are **strictly more powerful** than **LOOP** programs.
- ▶ **WHILE**-, but not **LOOP**-computable functions:
  - ▶ simple example: function that is undefined everywhere
  - ▶ more interesting example (total function):  
**Ackermann** function, which grows too fast to be **LOOP**-computable

How do **LOOP** and **WHILE** programs relate to Turing machines?

↔ next chapter