COMPUTARIUM
Lycée classique de Diekirch
32 av. de la gare L-9233 Diekirch

HP-65 FIFTIETH ANNIVERSARY PROJECT

COMPUTARIUM.LCD.LU PRESENTS

Python compiler for the HP-65 calculator

PART I: Recursive expression evaluator

Author:
Claude Baumann

Co-author:
Francis Massen

This project has primarily educational objectives
March 22, 2024

Abstract

This year 2024, the seminal HP-65 –the very first handheld programmable calculator– celebrates its incredible 50th anniversary. For this occasion, the COMPUTARIUM initiates a series of contributions starting with the description of a cross-compiler that should convert minimal PYTHON code to valid HP-65 code. For didactic reasons, the compiler is written in LABVIEW. The project is broken into two parts, first the development of a recursive expression evaluator and second the actual cross-compiler.

History:

- Version 1.0: February 5, 2024 : first draft
- Version 1.1: February 8, 2024 : first correction
- Version 1.2: February 26, 2024 : first revision
- Version 1.3: March 22, 2024: final version

Note: The main frame of the cover painting has been generated with CHATGPT in Pieter Claesz' (1597-1661) Still life style. The HP-65 picture has been manually drag and dropped into the frame.

Contents

Foreword

HP-65, the world's first programmable pocket calculator, celebrates its 50th anniversary. For this occasion, the COMPUTARIUM has started the impressively delicate project of bringing back to life its only HP-65 exemplary, a donation by late professor Jean Mootz.[1] After almost half a century passed in hibernation, the HP-65 showed no sign of life, as its battery pack was in a miserable state. Also, the magnetic card reader didn't make a sound. Undoubtedly, the model required a thorough overhaul. (We will present the details of the restoration in a further paper.) Although we could reactivate the card reader and all other functions, the main processor board presented a bug that we were not able to fix. Pressing the **STO 1** key also sets register 3 and vice-versa. The same thing happens to registers 2/4 and 5/7. Sometimes the second false copy of a value doesn't match the original, so that the content of the registers is totally unpredictable... and, by mystery, the bug disappears and reappears for confusing periods. Although our HP-65 model can be used for all calculations, programming, etc., many programs using more variables won't work reliably due to this memory interference. The origin might be our first attempts of starting the calculator with its bad battery pack and the AC adapter being connected. Indeed, HP urged its consumers never to run the HP-65 from the AC line with the battery pack removed (or dead?), because it may result in damage to the calculator.

We found a workaround for the CPU bug by using a replacement board generously donated by Australian specialist Tony Nixon. Currently, the board is en route from Australia to Luxembourg. The replacement of the original board will be documented in the forthcoming repair report.[2] Further, we could run a couple of excellent HP-65 emulators that are available on the web.

Next we came up with the idea of a PYTHON cross-compiler for the HP-65, because the calculator's **Reverse Polish Notation (RPN)** appears as exotic to students as the PYTHON language is popular among them. We believed that such a project could make RPN and also recursive compiler techniques accessible to high school students. We consider this indicated in the time of AI, which extensively applies recursion in propositional logic, for instance.[3]

For didactic reasons, we chose to program the cross-compiler in the graphical programming environment LABVIEW. In fact, it actually doesn't matter in which language a cross-compiler is written, as program sizes normally are very limited. Also, we don't think that this has been done in LABVIEW so far. We must point out that one of the educational advantages of LabVIEW, compared to text-oriented languages, lies in its code diagrams, which can be interpreted much like classic flowcharts.

We must warn the purist that this document might have some inconsistency or imprecision in terminology. And now, dear reader, please enjoy!

Further readings

1. J. A. Farrel, *Compiler basics* https://www.cs.man.ac.uk/~ pjj/farrell/compmain.html [retrieved Feb. 2024]. Although there are countless good books and articles about compilers, this often cited paper stands out from the rest by its crystal-clearness; a MUST!

2. https://learn.ni.com/learn/article/labview-tutorial [retrieved Feb. 2024]. This probably is the most condensed and clear description of the rudiments of LABVIEW.

3. D. B. Blazie, L. S. Levy, *A cross compiler for pocket calculators*, The Computer Journal, Vol. 20, Is. 3, pp. 213–221, (1977). Blazie and Levy's paper is an early attempt of writing an iterative BASIC compiler for the HP-65. The reader gets a good insight in the complexity of iterative compiler solutions.

[1] https://computarium.lcd.lu/obituaries/OBITUARY_JeanMOOTZ/Obituary_Jean_MOOTZ.pdf.
[2] https://www.teenix.org/, [retrieved Feb. 2024].
[3] S. Russel, P. Norvig, *Artificial Intelligence, A Modern Approach*, Pearson, US, p. 217ff., (2020)

4. HP-65 Owner's Handbook: https://literature.hpcalc.org/items/967 [retrieved Feb. 2024]. High quality

5. HP Journal May 1974 Issue: http://hparchive.com/Journals/HPJ-1974-05.pdf. Engineer Chung C. Tung, one of the developers of early HP calculators, qualifies the HP-65 **The „Personal Computer": A Fully Programmable Pocket Calculator**.

6. HP Journal June 1972 Issue: http://hparchive.com/Journals/HPJ-1972-06.pdf. The reader will find background knowledge of the invention of the HP35 model, the world's first pocket calculator. You'll learn about the calculator's use of Reverse Polish Notation, 4 level stack manipulation, sine algorithm (in fact an implementation of J. Volder's famous CORDIC algorithm). [4]

7. Tony Nixon's website: https://www.teenix.org/. This site is a gold mine for HP-65 background information. You'll find an excellent emulator for a long series of HP calculators.

8. HP-65 online emulator: https://archived.hpcalc.org/greendyk/hp65-emulator/ [retrieved Feb. 2024].

9. HP-65 emulator for download: https://www.hpcalc.org/details/9532 [retrieved Feb. 2024].

10. A. Doerr, K, Levasseur, *Applied Discrete Structures*, Univ. of Massachussets Lowell, LibreTexts, (2021), extract: The many faces of recursion, [retrieved Feb 2024].

11. C. Newstead, *An Infinite Descent into Pure Mathematics*, preprint, (2021), https://infinitedescent.xyz/dl/infdesc.pdf, [retrieved Feb. 2024].

12. J. S. Rohl, *Recursion via Pascal*, Cambridge University Press, 2nd Edition, (1984, 2008).

Chapter I
Top-down development of a recursive evaluator

1 Recursion in mathematics

Recursion is a powerful tool applied in mathematics –especially in number theory– to describe functions, or sometimes prove theorems and propositions. A recursive function may be basically understood as a function calling itself.

Example 1.1 (Factorial function)**.**

$$\forall n \in \mathbb{N} \begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \end{cases}$$

Normally the factorial function is presented in its iterative form: $\forall n \in \mathbb{N}, n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$ with the additional oddity $0! = 1$. By contrast, Equation 1.1 defines the function as a recursive process. It demonstrates well that recursion needs at least two expressions. The first fixes the special case, where the second equation runs into a conflict, in this case, if n is zero. As one can see, the special case also stops the recursive invokes of the main expression. Otherwise the process would continue forever with illegal successive n getting negatively larger and larger. For this reason the first equation is called **termination condition**.[5]

Example 1.2 (Prove that $\sqrt{2}$ is irrational)**.**

Proof. The recursive method shown here is denoted **infinite descent**. It was used by Pierre de Fermat in the 17th century. The idea is proving that a wrong hypothesis leads to contradiction, because no valid termination of the thought process can be reached. Therefore the contrary of the hypothesis must be true.

1. First, one supposes that $\sqrt{2}$ is rational. In this case, there are two distinct integer numbers $x > 0$ and $y > 0$ holding:

$$\sqrt{2} = \frac{x}{y} \iff 2 = \frac{x^2}{y^2} \iff x^2 = 2y^2$$

[4] J. E. Volder, *The CORDIC Trigonometric Computing Technique*, in IRE Transactions on Electronic Computers, Vol. EC-8, No. 3, pp. 330-334, (Sept. 1959)

[5] V. R. Mudunuru, M. V. Bhargavi, R. Ethiraj, *Zero Factorial*, Sch. J. Phys. Math. Stat.; 4(4), (2017), pp. 172-177.

2. One knows that:

$$\begin{aligned} & 1<2<4 \\ \Leftrightarrow & 1<\sqrt{2}<2 \\ \Leftrightarrow & 1<x/y<2 \\ \Leftrightarrow & y<x<2y \\ \Leftrightarrow & y-y<x-y<2y-y \implies 0<2y-x \\ \Leftrightarrow & 0<x-y<y \end{aligned}$$

3. The equation $x^2=2y^2$ may now be transformed by subtracting $(x\,y)$ from both sides:

$$x^2-x\,y=2y^2-x\,y \Leftrightarrow x(x-y)=y(2y-x) \Leftrightarrow \frac{x}{y}=\frac{2y-x}{x-y}$$

   This is allowed, as all expressions involved >0.

4. This signifies that $\sqrt{2}$ has a new rational representation:

$$\sqrt{2}=\frac{x_1}{y_1}, \text{ with } x_1=2y-x \text{ and } y_1=x-y$$

5. (Step 2.) showed that $y_1<y$. Similarly, x_1 and y_1 deliver a new fraction x_2/y_2, and this procedure can be repeated forever, so that

$$0<\ldots<y_{i+1}<y_i<\ldots<y_2<y_1<y$$

   This means that there is no termination of the recursive process, and consequently an infinitive amount of numbers would exist between y and 0. That, however, contradicts the elementary truth that only $y-1$ integer numbers are located between y and 0. Hence, $\sqrt{2}$ cannot be a rational number.

□

2 Recursion in computer sciences

Recursive functions can easily be transferred into a computer program, provided, the programming environment allows recursive function calls. This feature represents a complex scaffold behind the scenes, since the environment must produce a code that is capable of dynamically generating a new instance of the function when needed. An efficient way to solve this problem is generating function clones that share the same memory space for the function definition, but build different instances of the implied variables, which perish after use.

For the novice this may sound awfully complicated. The following paragraphs will try to elucidate how recursive processes basically work. Let's consider the PYTHON snippet of Listing 1, which depicts a recursive definition of the factorial function.

Listing 1: PYTHON recursive factorial definition

```
def factorial(n):
    if n == 0:
        return n
    else:
        return n*factorial(n-1)
```

What happens somewhere within the computer, if this function is called by:

```
print(factorial(2))
```

If we assume that the machine code generated by some PYTHON compiler uses the same function code at each new call, the variable n must be created in memory for each instance. This is done in a special memory section called the **stack**. During program execution, data is **pushed** to this pile and **popped** from it, always at the uppermost location. Note however that any location of the stack can be written and read. The compiler must take care that functions and procedures can only access that particular portion of the stack, which is affected to them. The stack therefore exclusively handles local variables. These are created dynamically via **push** and destroyed via **pop** instruction, as soon as they are no longer needed.

The computer then executes something comparable to Fig. 1. As one can easily see, recursive function calls present a fundamental risk of blowing up the stack size, which eventually can run into stack overflow error, if there is insufficient memory space assigned to the stack. Another issue with recursive functions appears, if no termination condition is reached. In such a case, the program hangs up in an everlasting loop.
Despite these inherent dangers, recursive functions are widely used for **divide and conquer** algorithms, such as the FAST FOURIER TRANSFORM, or divers search algorithms and code compilers that excel in highest execution speed and clearness. Interestingly, program code may often be rather simple and short, by contrast to iterative solutions that can turn into utter chaos. A good example for clean code is the recursive solution for the famous TOWER OF HANOI problem[6] shown in Listing 2.

Listing 2: PYTHON Tower of Hanoi

```
def hanoi(n, source, target, spare):
    if n > 0:
        # move n - 1 disks from source to spare
        hanoi(n - 1, source, spare, target)

        # move the nth disk from source to target
        print('Move␣disk␣%i␣from␣%s␣to␣%s' % (n, source, target))

        # move the n - 1 disks that were on spare to target
        hanoi(n - 1, spare, target, source)

# initiate call from source A to target C with spare B
hanoi(3, 'A', 'C', 'B')
```

3 Recursive expression evaluator

In this section, we will develop a simple recursive expression evaluator. Consider the following expression:

$$5 \cdot 3^4 + 3 \cdot 2^2 + 4 \cdot 7 - 45/9$$

Because of the priority rule: EXPONENTIATION BEFORE $\cdot/$ BEFORE $+$-, this expression yields:

$$5 \cdot 81 + 3 \cdot 4 + 4 \cdot 7 - 45/9 = 405 + 12 + 28 - 5 = 440$$

We humans normally proceed iteratively from the left to the right by first searching for exponentiation occurrences, then products and divisions, and only at the end we do operate addition and subtraction. Following the priority rule, we go down to the lowest level and work our way through to the highest. This method is called **bottom-up** approach.

However, the transfer of this simple *marche à suivre* into a computer turns out to be a rather tricky task. Oh, for such a simple straightforward expression like the one depicted here, we could simply scan the expression for the exponentiation symbol. (By the way, there is no such symbol in our expression, just another way of writing the powered parts using superscript style. In fact, computer languages require some unambiguous operator for the exponentiation such as **, or ^, for instance.) If an exponentiation symbol is found, the

[6] https://www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/tower-of-hanoi-algorithm/, [retrieved Feb. 2024].

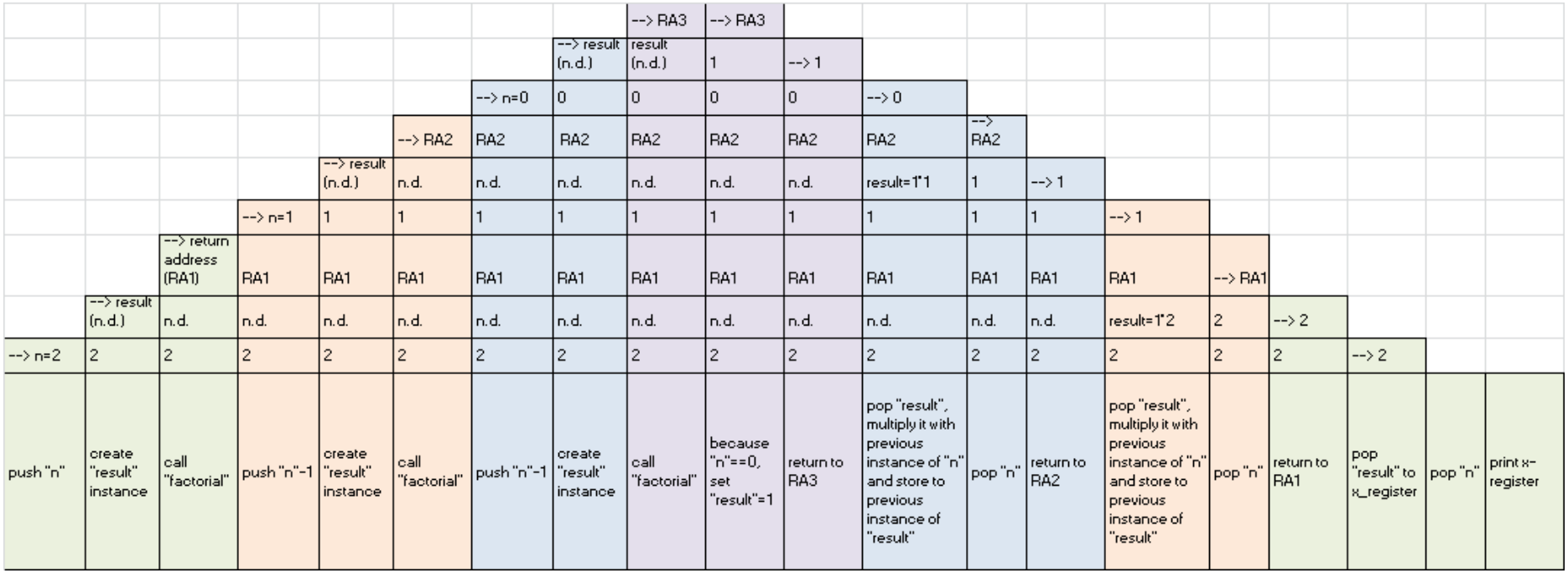


Figure 1: What happens behind the scene, when a recursive function is called; here the example of print(factorial(2)).

algorithm could select the constants to the left and right and do exactly what we did in the development above. The computer would then replace the exponentiation term by the result, and repeat the process until no exponentiation is being found anymore. Then it would go further to the multiplication and division, and end up with the addition and subtraction.

But, what if the expression had the following representation?

$$5 \cdot 3^{2+2} + 3 \cdot 2^2 + 4 \cdot 7 - 45/9$$

There are two alternatives, how to evaluate the first exponentiation occurrence: either do the addition first and then the exponentiation. This somehow violates the priority rule. The second possibility is transforming the critical term to:

$$5 \cdot 3^2 \cdot 3^2 + 3 \cdot 2^2 + 4 \cdot 7 - 45/9$$

And, things even get worse! What about the following form?

$$5 \cdot 3^{23^{5-\sqrt{9 \cdot 3 - 2}}+3} + 3 \cdot 2^2 + 4 \cdot 7 - 45/9$$

So far, we consciously avoided the use of brackets in the notation. However, a computer-friendly transcription of the expression will not be possible without brackets:

$$5*3**(23**(5-\text{sqrt}(9*3-2))+3)+3*2**2+4*7-45/9$$

For simplicity reasons, we will use the following valid PYTHON notation:

$$5*\text{pow}(3,\text{pow}(23,5-\text{sqrt}(9*3-2))+3)+3*\text{pow}(2,2)+4*7-45/9$$

We see here that the developer of an iterative solution runs into serious difficulties, as he or she must spot the lowest level bracket, before being able to apply any other rule. But, what, if we broke up the expression into its addition/subtraction terms, and for each of them continue our way down to its factors, which could be a product or a division, or another expression within brackets? You guessed it, this **top-down** approach might help us out of the trap.

3.1 Simple +-*/ evaluator

In the following paragraphs, we will develop a most simple recursive expression evaluator for the four basic arithmetic operations. We start by briefly analyzing the final part of our sample expression:

$$4*7-45/9$$

Observed from scratch, this is nothing but a sequence of symbols that may have no meaning at all for the computer, unless there is some kind of rule, how the characters should be read an interpreted. Technically, we are in the presence of a string, which is a chain of characters. How can we tell the computer to draw meaning from a string.

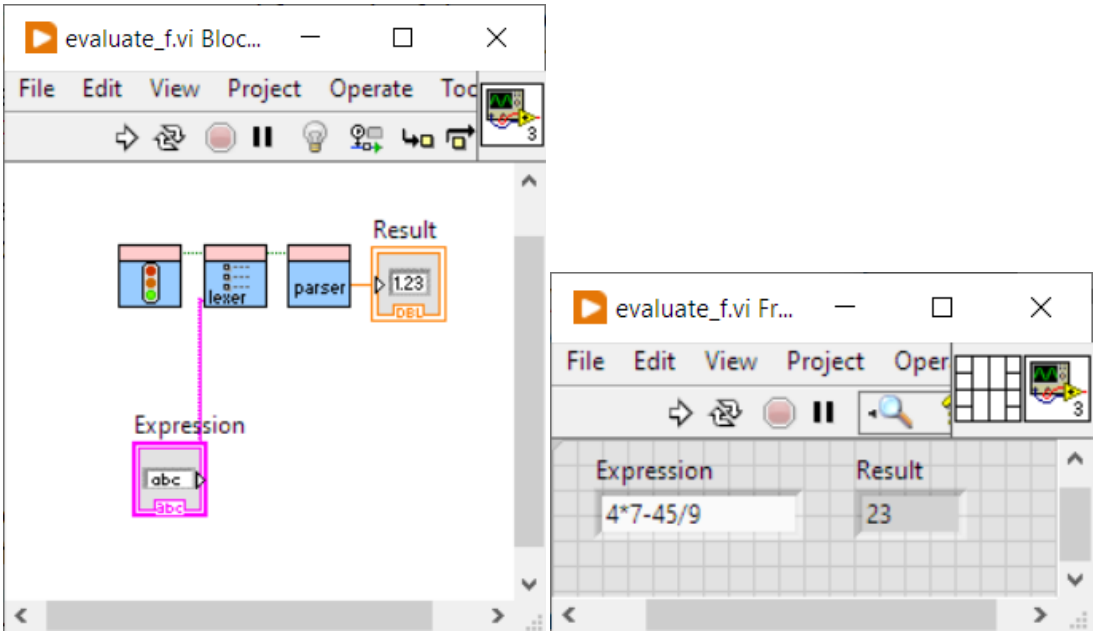


Figure 2: LABVIEW diagram and front panel of a 4-operation arithmetic expression evaluator. A LABVIEW diagram depicts the program flowchart. The dataflow starts at constant values or at input boxes, called **controls**, and passes via **wires** to icons that represent either output boxes –so-called **indicators**– or new subroutines (=**sub.vis**).

The process in question is typically divided into two parts. The first part is executed by the tokenizer, more commonly known as the **lexer**, and the second part is handled by the **parser**. Fig. 2 shows an additional part consisting of variable initialization (the traffic light icon). Visibly on the right picture, the elementary evaluator has yielded the correct result.

Remark: Just two words about LABVIEW for better understanding. (Please don't miss reference 2. of our further reading list at the beginning of this document.)

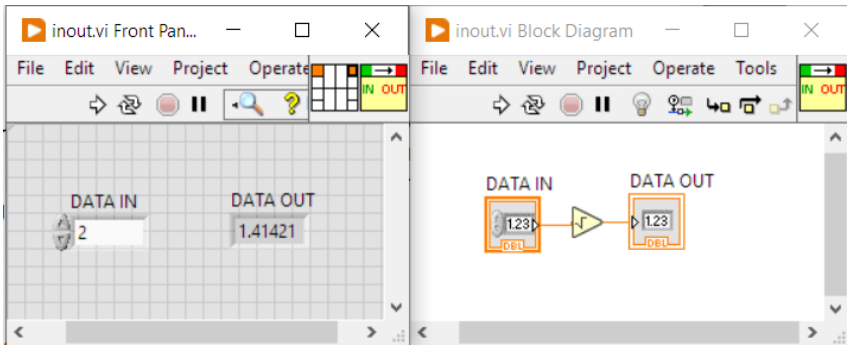


Figure 3: LABVIEW controls and indicators, which in fact represent the environment's variables, are color-coded, for instance: ===== floating point numbers; ===== integer numbers; ===== Boolean; ===== strings. Thin wires indicate single value connections, whereas thick ones show arrays.

Fig. 3 demonstrates well the dataflow in a LABVIEW program –also denoted **virtual instrument (vi)**. In the upper right corner of the front panel, the patterned square shows two smaller orange squares. This are called **terminals**. In fact, the user can freely link the input (control) box and output (indicator) box to one of the 16 squares. (He or she even can choose other patterns from a list.) This defines, how the **vi** as an icon may be wired to other elements of the program diagram.

3.1.1 Lexer

The essential work of the lexer is verifying whether the string elements belong to the set of allowed symbols. While this could easily be done during the semantic analyzing process, it is nonetheless a good idea to do it along with some pre-digestion, in order to detect possible errors in a very early stage, and above all, classify the symbols into a stream of significant tokens.

The environment LABVIEW offers many powerful features making programming clear and comprehensible, as can be seen in Fig. 4. Much of the code should be self-explanatory. The inexperienced reader might first want to have a glance at the tutorial (item 2. of the initial further reading list), in order to understand the **while** structure and the functionality of **shift registers** marked with a down pointing arrow on the left side of the while structure, and an up pointing arrow on the right side. The left instance delivers the value from the former iteration or from the initial value in the case of the first loop passage. Filling a data array of any type is easily done by choosing the **indexing** tunnel mode. If the array should only be filled under some condition, one can also select the **conditional** tunnel mode. Fig. 5 shows the program front panel, which controls program execution and debugging.

The program uses an enumerator with a list of valid tokens. It must be underlined, comparably to other programming languages, that a LABVIEW enumerator in fact represents an integer number, just the index of the chosen item on the list. As can be seen from the **Tokens detected** array, the allowed tokens are:

$$+ \; - \; * \; / \; \%\mathrm{f}$$

where **%f** is a placeholder for floating-point numbers. The LABVIEW built-in token scanner uses this placeholder for the recognition of numbers. This is an important feature that allows rapid extraction of constants within a string.[7]

Important note: Fig. 4 shows that we are using **global variables**. Most programmers consider this a bad practice. Normally they are right! Data should be passed to functions via their arguments and returned to callers via result values. This however blows up the stack, in the case of recursive function calls, since new variable instances must be generated at each new call. There is another side effect of local variables, which concerns execution speed. Because a compiler –and that's what we intend to develop in this project– typically has to manage ever growing code strings and data arrays, the time needed for data transfers from function to function can not be neglected. For this reason, global variables is a valuable choice, although we must take care that there will never any write/read conflict.

[7] An often forgotten issue with floating-point numbers is the form of the decimal separator. Most environments require the decimal point. LABVIEW like MICROSOFT EXCEL uses the system decimal separator, which might be the comma. In that case, **%f** will not read a **0.0** number correctly, and the program will throw an error, because it stumbled over a symbol, which is not on the token list. The straightforward solution is to change the system separator, and we're done. If this is not desired, we must design a workaround.

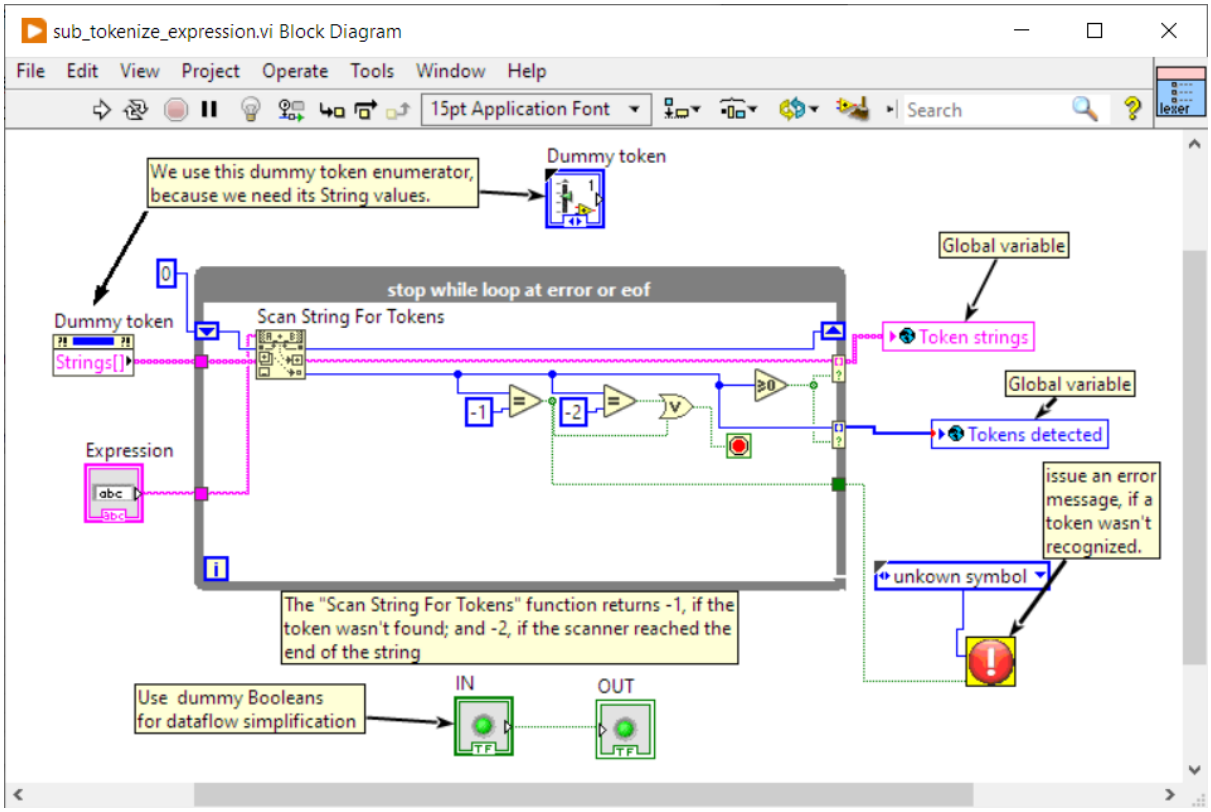


Figure 4: LABVIEW diagram of a lexer for arithmetic expressions.

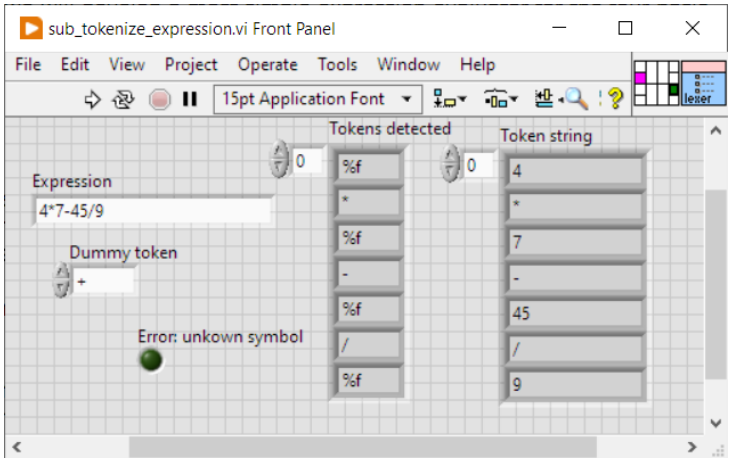


Figure 5: LABVIEW programs are control via **Front Panel**.

3.1.1.1 Error handler

The program has to halt, if an error is encountered. So, we propose a simple error message handler shown in Fig. 6.

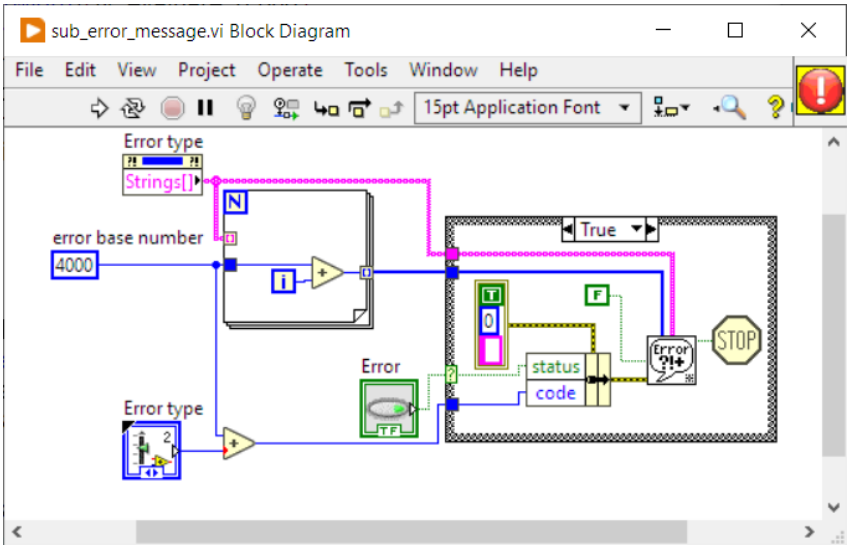


Figure 6: This sub.vi throws an error message and stops the program.

3.1.2 Parser

It is the parser's task of extracting something meaningful from the token list. Unlike the lexer, which only checks for valid strings in the expression, the parser has to verify its grammar, and eventually initiate some action corresponding to the semantics. For instance, the expression:

$$3*4+ \tag{1}$$

will pass the lexer, because every symbol is found on the valid token list. By contrast, it will be rejected by the parser, because obviously there is something missing. Parser actions for a valid expression may be calculation, or code generation. In this section, we will have the parser do the arithmetic operation in the computer.

Our parser consists of only two subroutines. The first **pick token** initiates the parsing process, and the second **expression** evaluates the expression step by step. **pick token** is called at every valid token recognition.

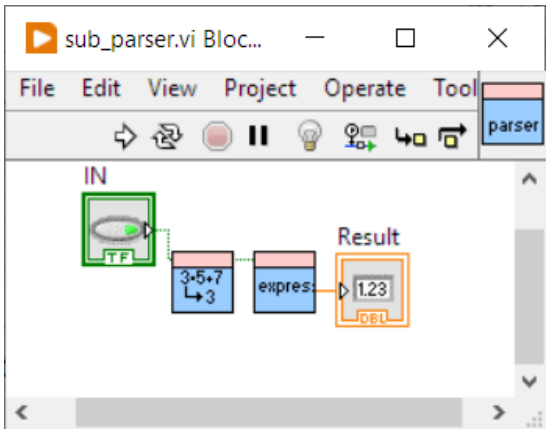


Figure 7: The parser is made of two subroutines only. The first initiates the parsing process by picking the very first token from the list. The second is the actual expression evaluator.

3.1.2.1 Pick token

The program must pick tokens from the detected token array. Fig. 8 reproduces the diagram of this subroutine.

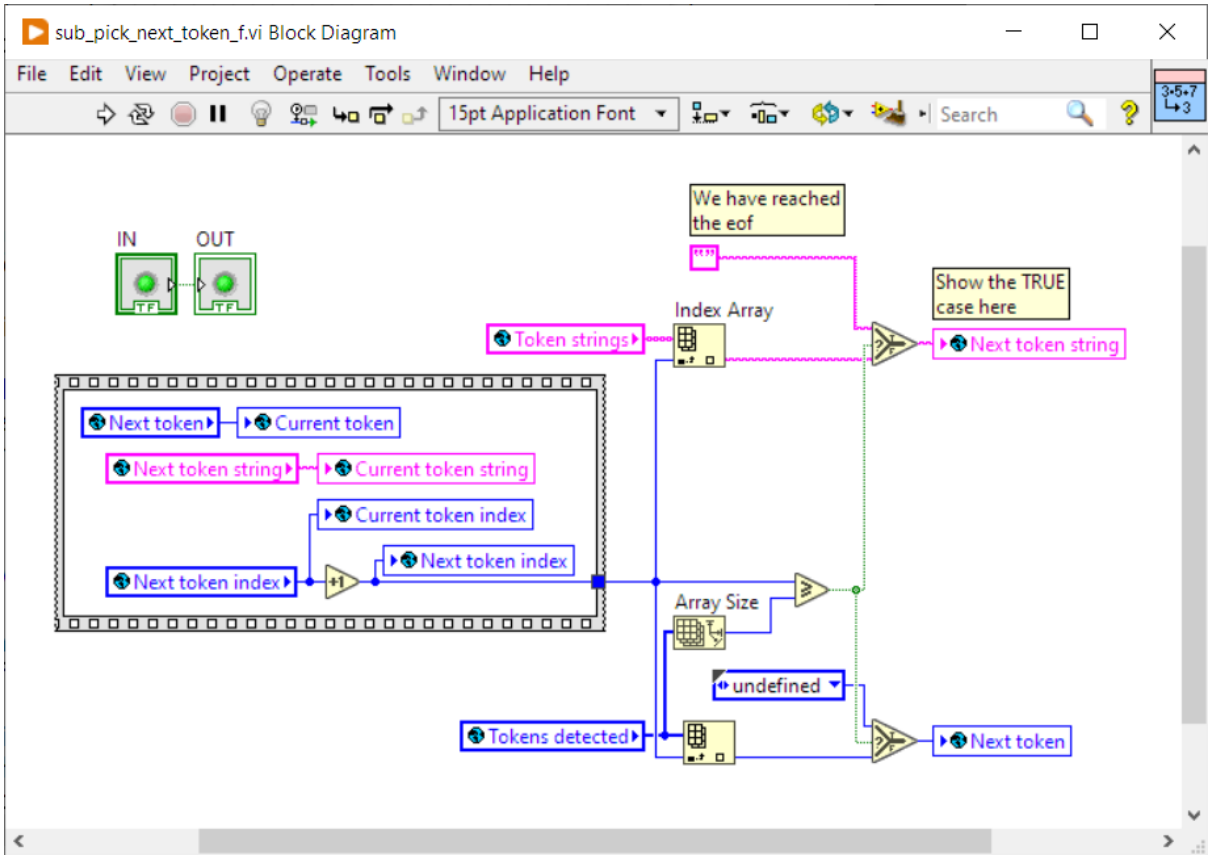


Figure 8: This sub.vi picks the next token and conserves the current one.

3.1.2.2 Expression

An expression is composed of a chain of **terms** that are linked by one of the operators + or −. Because the chain can be as long as desired, the subroutine runs a loop that is stopped at the last term. At this point the expression subroutine calculates the result of successive term operations. Fig. 9 shows the LABVIEW implementation of the expression subroutine. Note that there is a second case for the subtraction not visible here.

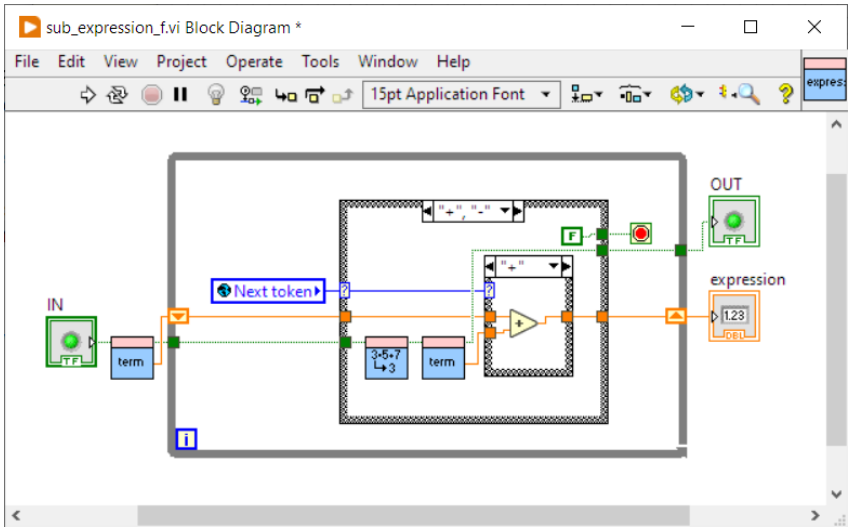


Figure 9: This sub.vi adds or subtracts a chain of terms.

3.1.2.3 Term

Comparably to the expression subroutine, the term routine operates the chain of **factors** that are linked by one of the operators $*$ or /. The structure of this subroutine is identical to the one of the expression routine, as can be seen in Fig. 10.

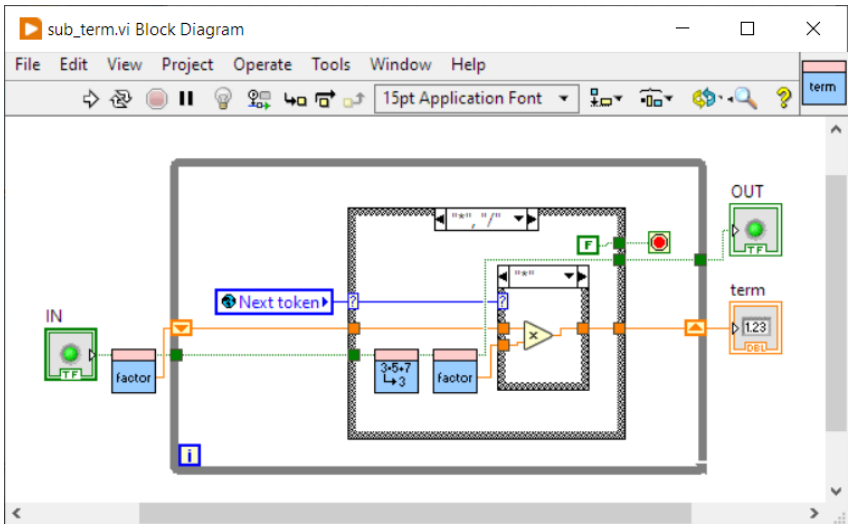


Figure 10: This sub.vi multiplies or divides a chain of factors.

3.1.2.4 Factor

For this simple arithmetic expression evaluator it is sufficient to consider constant factors only. The subroutine asks for the specific **%f** placeholder token. This reduces the program to the minimum of converting strings to numbers. Fig. 11 depicts this procedure. The result is passed to the next level.

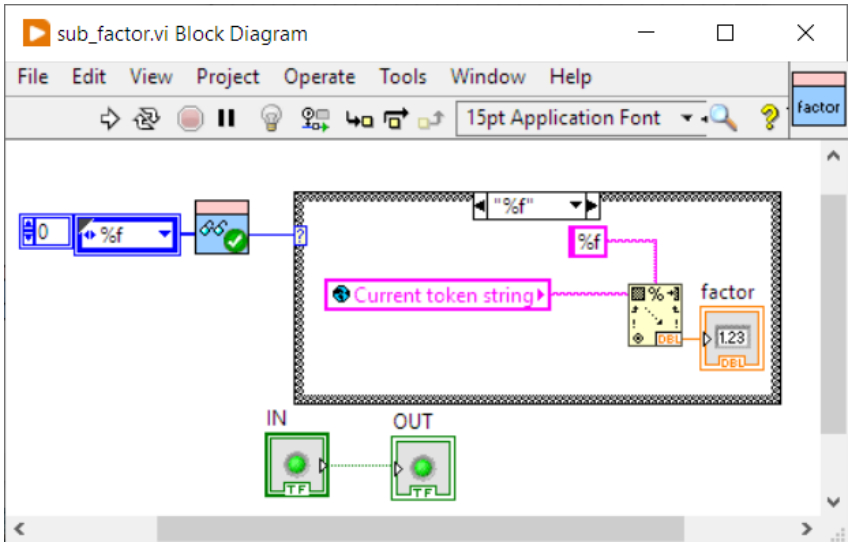


Figure 11: This sub.vi defines the factor, in this case simply the string conversion to the specific number constant.

3.1.2.5 Expect specific token

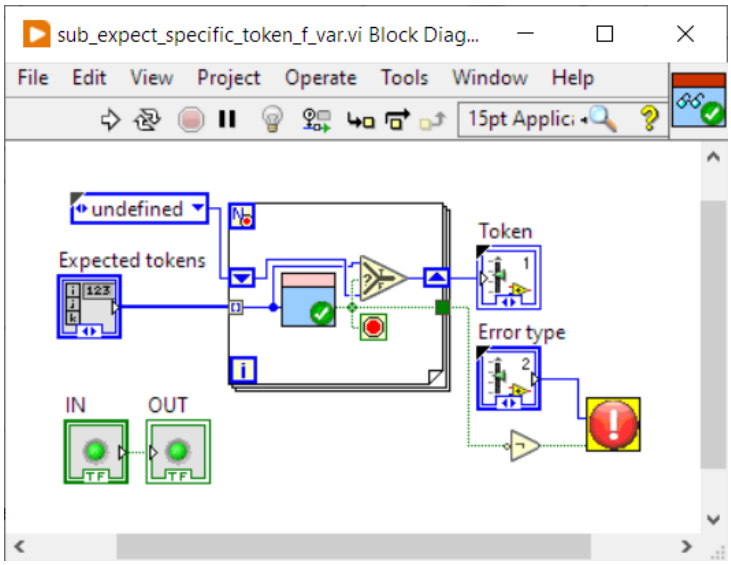


Figure 12: This sub.vi checks if the next token is among the expected ones. If the token is not found in the expected list, the program will issue an error.

3.1.2.6 Accept specific token

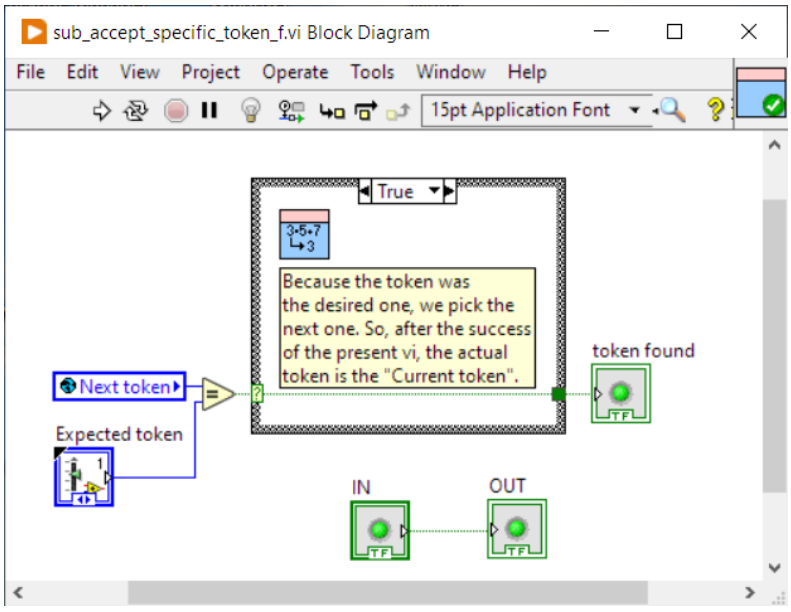


Figure 13: This sub.vi checks if the next token is the expected one.

And that's all for this very basic evaluator!

3.2 Leading -/+ issue

The attentive reader may have noticed that the arithmetic evaluator issues an error for expressions with leading plus or minus signs. The reason for this behavior is that the **factor** subroutine doesn't consider the

minus or plus sign as part of the first number, but rather as an operator. In order to fix this, we need to add a special handler for this particular case.

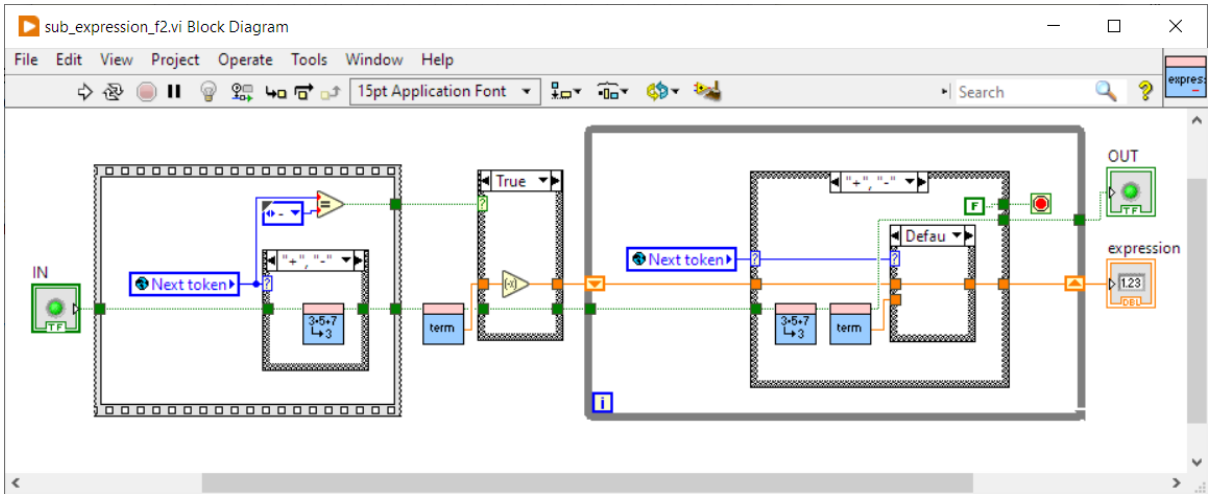


Figure 14: This new version of the expression subroutine solves the leading +/- issue.

3.3 Adding brackets

So far, there has been no recursive descent, because terms in an atomic arithmetic expression all belong to the same operative level. This dramatically changes, if we add brackets. Now, everything between brackets represents a complete expression on its own, which we must evaluate before doing anything else. And this may recursively continue down to any desired level. Here, the top-down concept proves its value, because the programming expense is incomparably smaller than for any bottom-up attempt.

Before passing to the actual implementation, we consider the formal description of our essential structures: **expression, term, factor**. The canonical (Backus-Naur) form of the definition of an arithmetic expression using integer numbers is:[8]

```
S -> EXPRESSION
EXPRESSION -> TERM | EXPRESSION + TERM | EXPRESSION - TERM
TERM -> FACTOR | EXPRESSION * FACTOR | EXPRESSION / FACTOR
FACTOR -> NUMBER | ( EXPRESSION )
NUMBER -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
          1 NUMBER | 2 NUMBER | 3 NUMBER | 4 NUMBER |
          5 NUMBER | 6 NUMBER | 7 NUMBER | 8 NUMBER |
          9 NUMBER | 0 NUMBER
```

where **S** can be seen as the start symbol and the right arrow as an expansion direction to the right side production rule. Only the right side may apply recursion. The left side must be considered the higher level caller. „|“ stands for the logical **OR** operator. There are 3 categories of symbols here:

1. Non-terminal (or intermediate) symbols: EXPRESSION, TERM, FACTOR, NUMBER
2. Terminal symbols: 1, 2, 3, 4, 5, 6, 7, 8, 9, 0
3. Operators: |, +, -, *, /,(,)

Remark: Notice that this definition doesn't solve the leading -/+ issue, because there is no **empty** terminal symbol. Also, as this definition concerns integer numbers only, there obviously is no rule for the decimal point. Anyway, because our LabVIEW **lexer** is capable of identifying whole floating-point numbers, we can change the formal definition to:

[8] cf. Ref. [1] on the introductory reference list.

```
S -> EXPRESSION
EXPRESSION -> TERM | EXPRESSION + TERM | EXPRESSION - TERM
TERM -> FACTOR | EXPRESSION * FACTOR | EXPRESSION / FACTOR
FACTOR -> %f | ( EXPRESSION )
```

Floating-point numbers taken as a whole represent the exclusive termination symbols. Note that the leading -/+ sign exception remains unsolved in this formal definition.

The question arises, how this recursive definition ever comes to halt. In other words, what prevents this production from trapping into an infinite descent? In fact, any valid production grammar must provide symbols that cannot be expanded. These are the termination symbols, in our case floating-point numbers.

In this particular implementation of the production rules, we are using a hybrid form made of recursion and iteration. We continue considering an **expression** as a chain of **terms** that we parse through a while loop, as long, as + or - operators are being found. Similarly, a term is still seen as a chain of **factors** that are parsed in a loop, as long as * or / exist. Although this differs from the formal definition, the mixed method is a valid transcription. As it is visible in Fig. 15, only the subroutine **factor** has a new appearance, as it adds a new case to expected token list, which now has the **%f** and the „(“ tokens. If the latter is detected, there is a recursive call of expression. The factor then must be closed with a „)“ token. Fig. 16 shows that the minimally altered evaluator now correctly computes expressions using brackets.

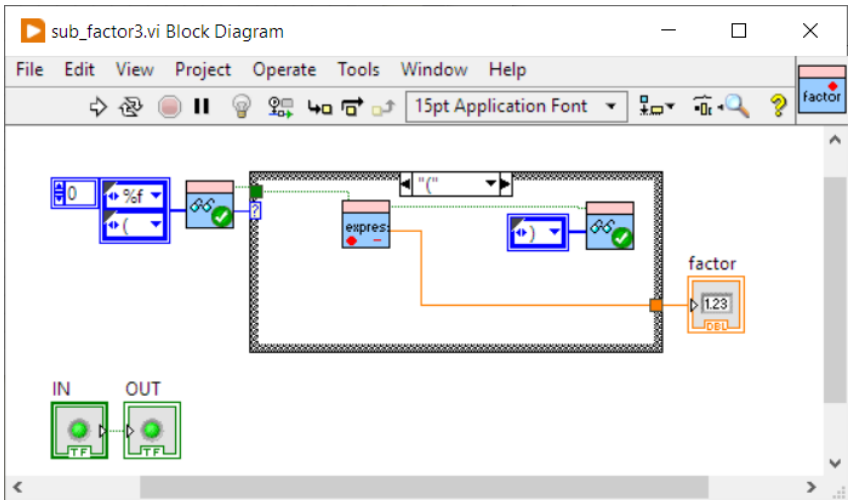


Figure 15: The new **factor** allows the recursive call of **expression**.

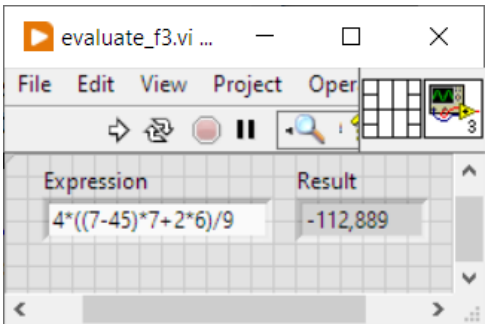


Figure 16: As can be seen on this front pane, the new evaluator correctly computes the expression.

Fig. 17 shows the dependencies of the used subroutines, global variables and user controls. Recursive subroutine calls are only allowed, if their execution method is changed to **reentrant**. This is done in the **File/VI Properties/Execution dialog** as shown in Fig. 18.

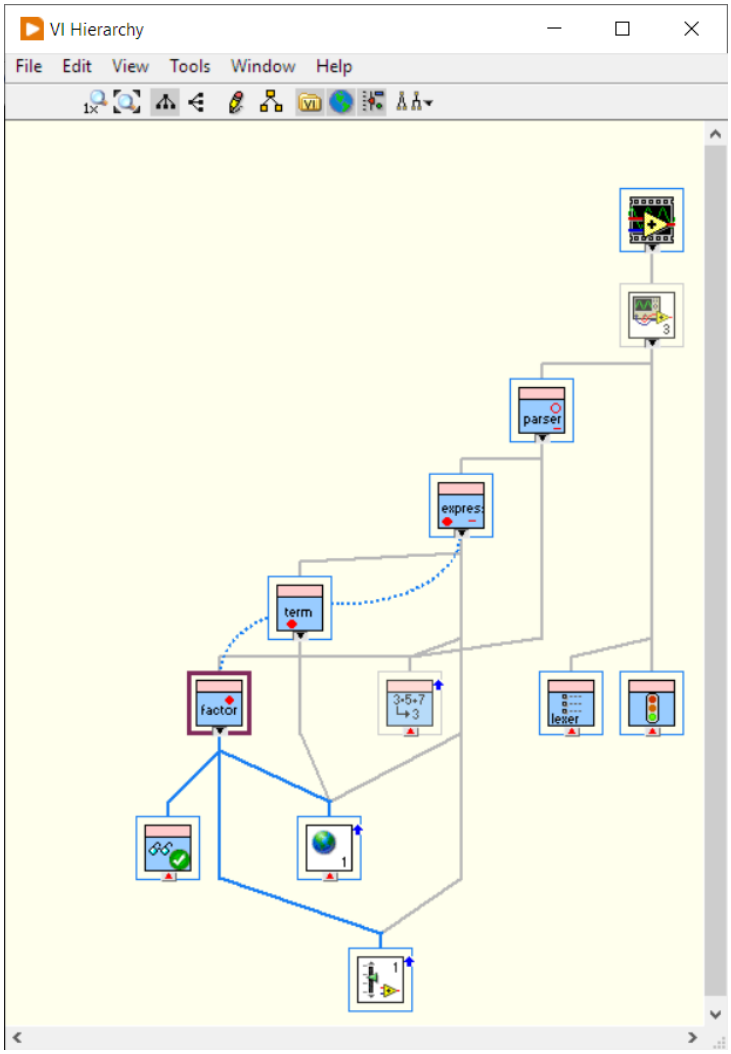


Figure 17: LABVIEW gives an extraordinarily clear overview about the subroutine hierarchy. The dotted lines indicate that the implied subroutines are recursively called. For better visibility, we added a red dot to reentrant subroutines, and a red circle to the **parser** icon, as it is calling a reentrant function.

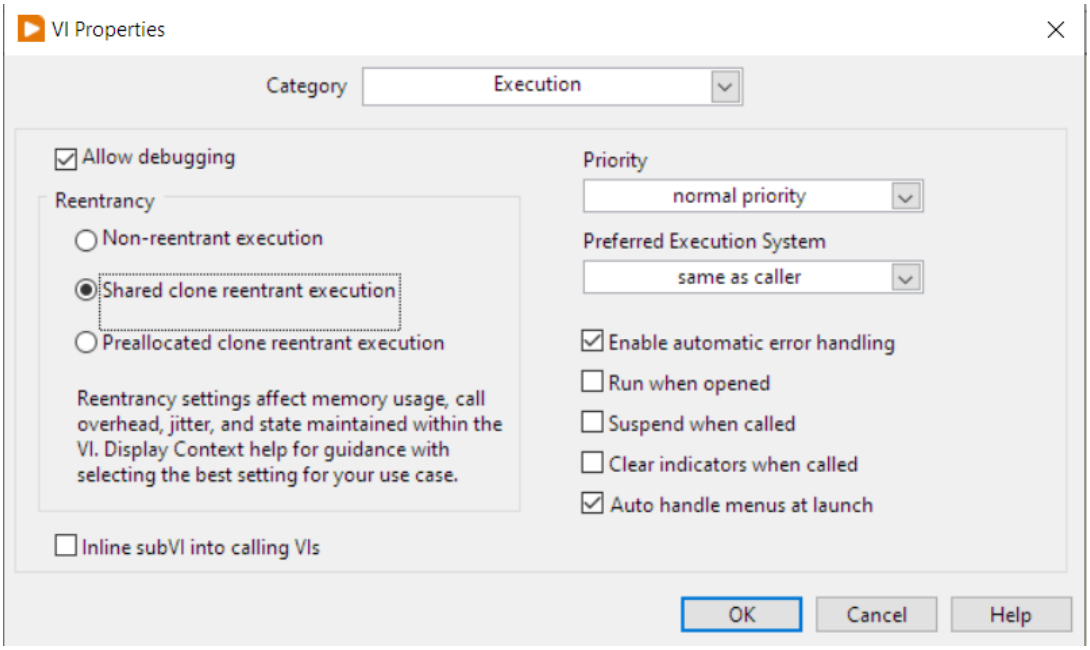


Figure 18: LABVIEW allows recursive subroutine calls, if their execution method is changed to **reentrant**.

3.4 Adding functions

We now are able to add functions to **factor** with minimal effort (cf. Fig. 19-21). The token list must be stocked up to:

+ - * / %f ( ) , sqrt pow sin cos, tan exp exp10 log log10 asin acos atan abs int pi

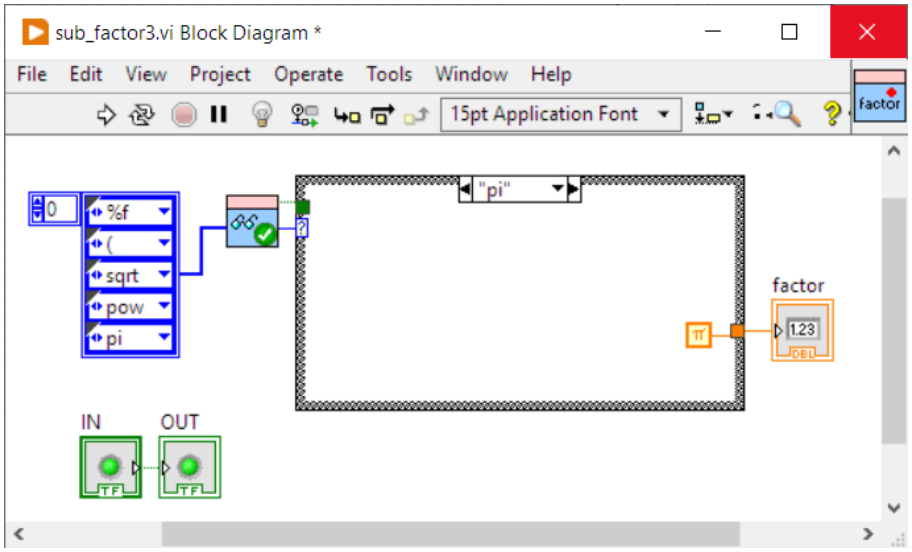


Figure 19: This code snippet adds pi to **factor**. This works similarly for any constant function.

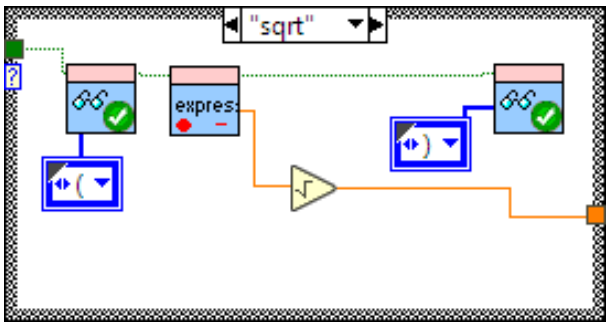


Figure 20: This code snippet adds the square-root to **factor**. This works similarly for any other one-argument function.

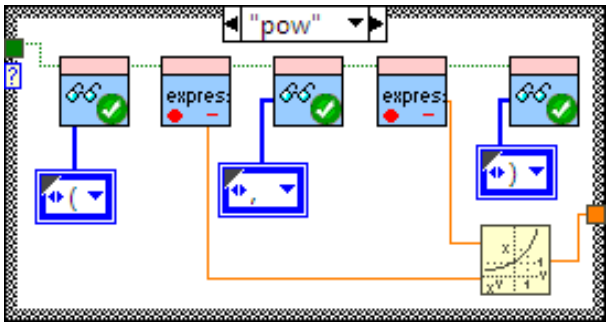


Figure 21: This code snippet adds the 2-argument power function to **factor**.

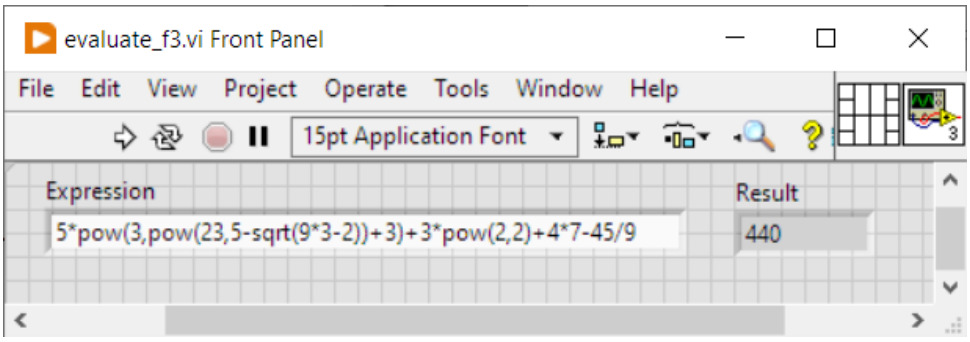


Figure 22: The expression evaluator yields the correct result.

TADA: we have written a working expression evaluator, as can be seen in the calculation of Fig. 22.

Chapter II
Calculating with the HP-65

4 Reverse Polish Notation (RPN)

Figure 23: The HP-65 was the first programmable scientific calculator.

Dear reader, if it is your first encounter with the HP-65, you probably will be stunned that you sought in vain for an „=“ key on this device. You might have no clue how to calculate with it. In fact, the particularity of the early HP calculator models –starting from the legendary HP35– was the use of **Reverse Polish Notation (RPN)**, more commonly known today as **postfix** notation. The name refers to the Polish philosopher and logician Jan Łukasiewicz, who invented Normal Polish Notation (NPN) or prefix notation in order to get rid of brackets. In RPN one writes the operator after the operands, instead of between (infix). So, there is no need for a „=“ or brackets key on a calculator working with RPN. Instead, the calculator uses an **ENTER** button.

Example 4.1. *Reverse Polish Notation*

$$\begin{aligned} &3*(4*7+45/9) \\ \Longleftrightarrow &4 \quad 7 \quad * \quad 45 \quad 9 \quad / \quad + \quad 3 \quad * \end{aligned} \tag{2}$$

Example 4.2.

$$\begin{aligned} &-\sqrt{4*7+45/9} \\ \Longleftrightarrow &4 \quad 7 \quad * \quad 45 \quad 9 \quad / \quad + \quad \sqrt{\ } \quad - \end{aligned} \tag{3}$$

Example 4.3. *HP-65 instructions corresponding to Ex. 4.2.*

```
4
ENTER
7
*
45
ENTER
9
/
+
```

```
f SQRT
CHS
```

Internally, to cope with such a series of symbols, the HP-65 needs registers, where to intermediately store the numbers before doing some operation. Fig. 24- Fig. 26 show the functionality of the stack, which fulfills this task. Fig. 27 shows additional stack manipulation functions.

The Operational Stack

There are four working registers in the HP-65 called **X**, **Y**, **Z**, and **T**. They are arranged in a 'stack' with **X** on the bottom (*see below*).

Contents	Location
t	T
z	Z
y	Y
x	X

To avoid confusion between the name of a register and its contents, the register is designated in this handbook by a capital letter and the contents by a small letter. Thus **x, y, z,** and **t** are the contents of the **X, Y, Z,** and **T** registers.

When you key in a number, it goes into **X**, the displayed register. When you press ENTER↑, this number is also reproduced in **Y**. At the same time **y** is transferred to **Z**, **z** is transferred to **T**, and **t** is lost (*see below*):

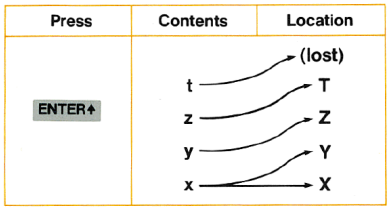
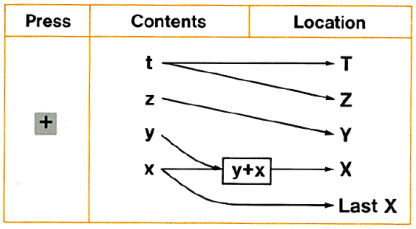


Figure 24: What happens to the HP-65's operation stack, when entering a number (cf. HP-65 Owner's Handbook p.8).

Arithmetic and the Stack. When you press the addition key the contents of **X** and **Y** are added together. The stack then drops, with **t** reproduced in **T** and **Z**, **z** transferred to **Y**, (y+x) transferred to **X**, and **x** transferred to **Last X.** (*Last X is described in* **Section 1.)**



The same dropping action takes place with any arithmetic operator (+, −, ×, or ÷); the result is placed in **X**.

Figure 25: What happens to the HP-65's operation stack, when executing an addition (cf. HP-65 Owner's Handbook p.9).

Sample Case:

$[(4\times5)\ /\ (2+3)]-6=-2$

Press	See Displayed
4 ENTER↑	4.00
5 ×	20.00
2 ENTER↑	2.00
3 +	5.00
÷	4.00
6 −	−2.00

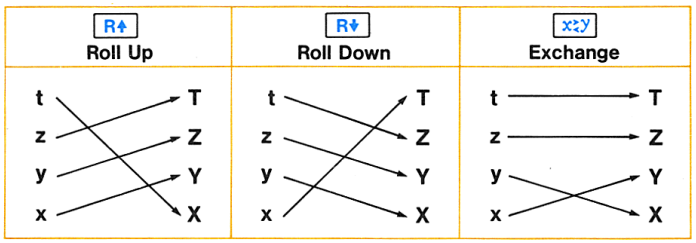
Notice that the numbers are entered in the same order as they appear in the problem. Now consider the stack contents as we do the same example.

Stack Register

T											
Z						20	20				
Y		4	4		20	2	2	20		4	
X	4	4	5	20	2	2	3	5	4	6	-2
Keys	4	↑	5	×	2	↑	3	+	÷	6	−

Note: ENTER↑ is here abbreviated as ↑ .

Figure 26: A sample case (cf. HP-65 Owner's Handbook p.10).



You use these operations to verify (*display*) the contents of stack registers other than **X** and to move information into place for calculation.

Figure 27: Rolling and exchanging stack registers (cf. HP-65 Owner's Handbook p.31).

Chapter III
Generating HP-65 code from an expression

The generation of HP-65 code from an expression can be seen as a process of transforming infix notation to postfix notation. We could apply computer pioneer Edsger W. Dijkstra's brilliant iterative **Shunting yard algorithm** using a stack that he developed in the vivid time of computer language **ALGOL**.[9] However, as we intend to develop a complete PYTHON compiler in this project, we will stick to our recursive approach.

[9] https://www.cs.utexas.edu/ EWD/MCReps/MR35.PDF, [retrieved Feb. 7th 2024]

5 Transforming our expression evaluator

Only minimal changes must be applied to the previously developed evaluator in order to change the parser action from direct calculation to HP-65 code generation! We just have to change the LABVIEW number variable (orange) to an array of strings (pink), and add text code instead of doing mathematical operations. (cf. Fig. 28 to 31).

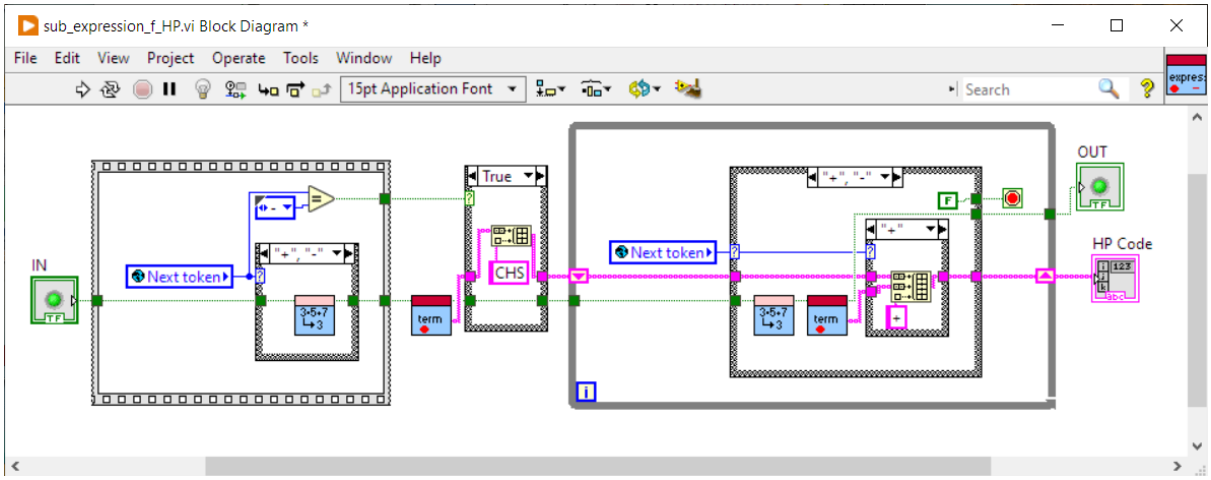


Figure 28: The evaluator action is changed from doing calculations to accumulating text code.

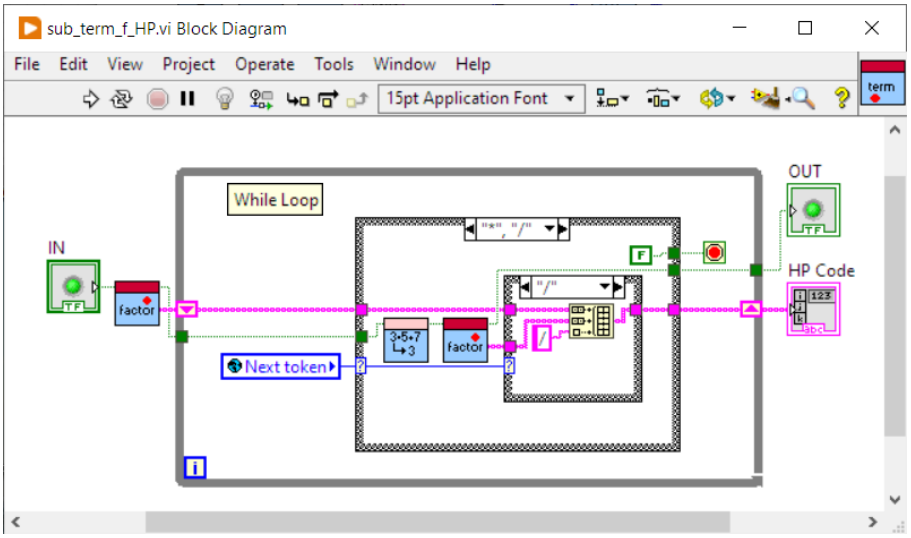


Figure 29: Text code accumulation is continued down the way to **term** subroutine ...

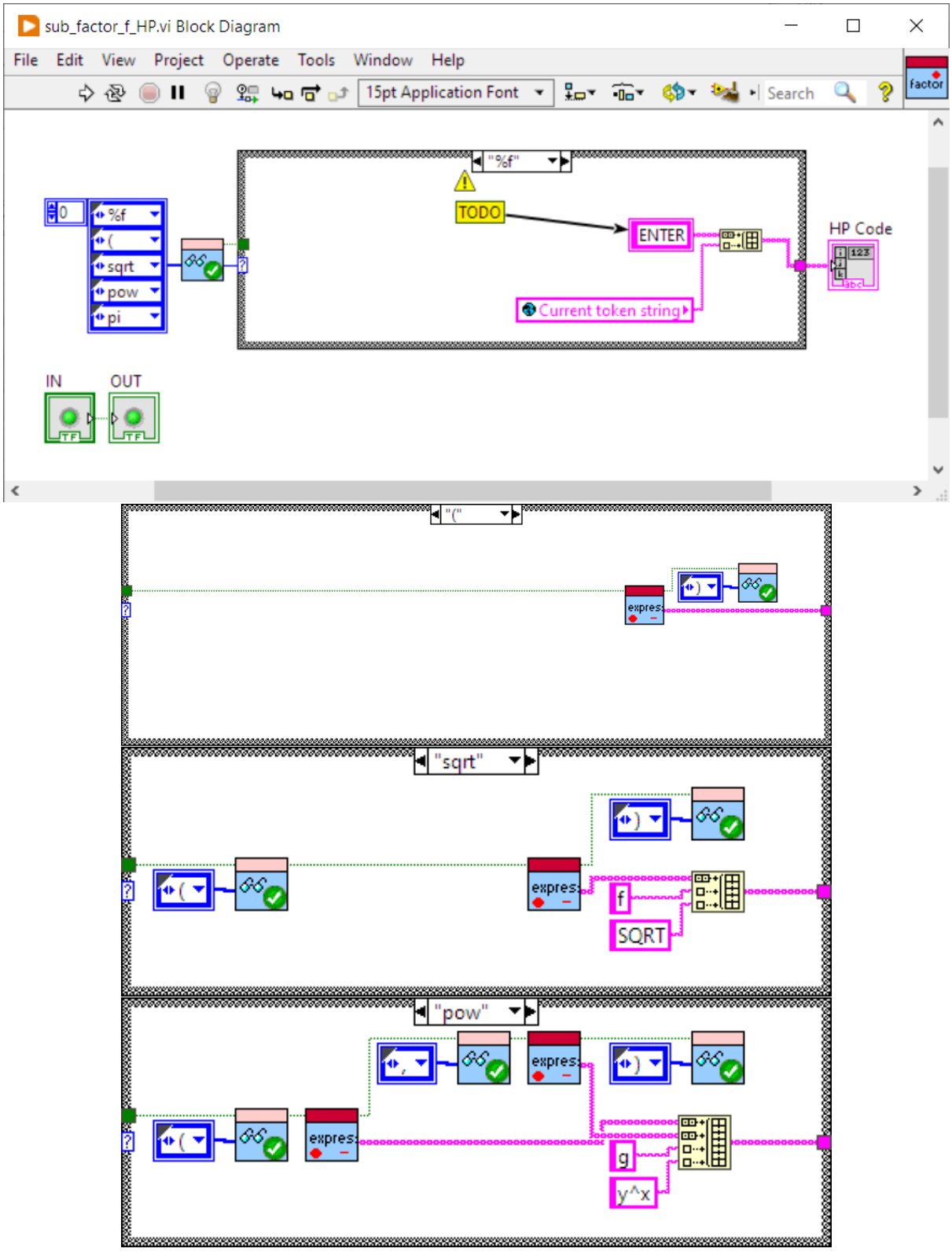


Figure 30: ... reaching **factor** subroutine.

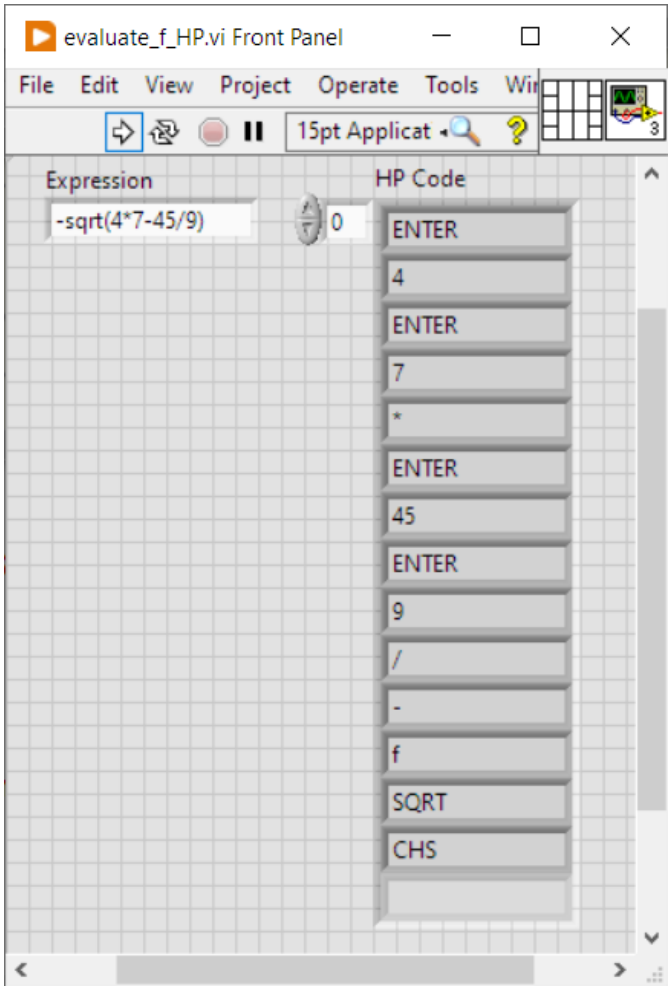


Figure 31: Sample code generation.

5.1 Issues

5.1.1 Not all ENTER instructions are required

The attentive reader may have noticed that there is a superfluous **ENTER** instruction right at the beginning of the code and a second one after the first multiplication sign in Fig. 31 compared to Example 4.3. (That's why we added a **TODO** warning to the **%f** case in **factor**.) The reason is that HP-65's „**X-register**" represents at the same time the data input **and** output register (display) **and** the first stack location **and** the second operation accumulator for the CPU. The varied functionality of this system leads to some side effects. For example, during HP-65 micro-code execution for the SIN function, the display starts flickering due to rapid data exchange with the **X-register**.

As shown in Fig. 24, pressing the **ENTER** key works somehow similarly to a normal **push** operation, where the stack pointer is increased and the new data is placed at the new location of the pointer. Note that most technical stack implementations use **pre-decrementing** of the stack-pointer rather than incrementing, so that they behave more like a cellar than a pile. This corresponds to the way, the stack is shown in Fig. 24 to 27.

So, the initial **ENTER** instruction is superfluous, as the keyed number goes directly to the **X-register**. Also, the internal stack manipulation closing any basic or advanced arithmetic operation, includes an **implicit ENTER** instruction (cf. p. 10 of the Owner's Manual). This explains why the **ENTER** instruction of code line 8 is unnecessary. Being ignored by the HP-65, it doesn't corrupt the calculation though. We have to note that it is impressively difficult to fix this issue in the recursive process. That's why we decided to do it later in an

expression optimizer that will be presented in the second part of this project.

5.1.2 HP-65 stack overflow issue

You guessed it, there is another issue appearing here. In fact, we are talking about something serious! Although the HP-65 can handle many complex expressions delivered by our evaluator, thanks to the 4 level stack depth, it might be possible that nonetheless an expression produces a **stack overflow**. This happens, if an expression needs more than 4 consecutive **ENTER** or **implicit ENTER** that are not separated by any operation (which on its turn would reduce the stack level). Consider the following examples:

Example 5.1. *Correct sample code:* $3*4*sqrt(5+6*7)$

```
(ENTER) -unnecessary
3
ENTER (1st)
4
* (reduces stack level)
(ENTER) (1st, implicit by previous *; ENTER key pressing is ignored by the HP-65)
5
ENTER (2nd)
6
ENTER (3rd)
7
*
+
f
SQRT
*
```

Example 5.2. *Bad sample code:* $3*4*sqrt(5+6*(8-1))$

```
(ENTER) -unnecessary
3
ENTER (1st)
4
* (reduces stack level)
(ENTER) (1st, implicit by previous *; ENTER key pressing is ignored by the HP-65)
5
ENTER (2nd)
6
ENTER (3rd)
8
ENTER (bummer: 4th --> stack overflow)
1
-
*
+
f
SQRT
*
```

The HP-65 code shown in Example 5.2 has one too many **ENTER** instructions in a file. The very first „3“ is kicked out of the **T-stack register** at the moment of pressing **ENTER** „1“.

We can fix this issue by using one or more data registers available on the HP-65 (Registers 1..9) as additional stack registers. We therefore count the number of times the **ENTER** key has been activated in series (**implicit ENTER** included). If this number exceeds 4, we add the following lines at the moment of the supernumerary **ENTER** command:

```
g R^
```

```
STO n
g Rv
```

where „n“ is the address of the first free register in memory. This code snippet rolls the stack, so that the content of the **T-register** changes to the **X-register**. Then the value is stored into memory. Finally, the stack is rolled back.

After the first arithmetic operation following this code insertion, we insert the code snippet recovering the previous **T-register** content:

```
RCL n
g Rv
```

Example 5.3. *A valid HP-65 code version for Ex. 5.2 equation.*

Instruction	X	Y	Z	T	REG 1	
(ENTER)	0.00	0.00	0.00	0.00		unnecessary
3	3	0.00	0.00	0.00		
ENTER	3	3	0.00	0.00		
4	4	3	0.00	0.00		
*	12	0.00	0.00	0.00		
(ENTER)	12	0.00	0.00	0.00		ignored
5	5	12	0.00	0.00		implicit
ENTER	5	5	12	0.00		
6	6	5	12	0.00		
ENTER	6	6	5	12		
8	8	6	5	12		
g R^	12	8	6	5		<–
STO 1	12	8	6	5	12	<–stored
g Rv	8	6	5	12	12	<–
(ENTER)	8	6	5	12	12	(ignored)
1	1	8	6	5	12	implicit
-	7	6	5	5	12	
RCL 1	12	7	6	5	12	<– recovered
g Rv	7	6	5	12	12	<–
*	42	5	12	12	12	
+	47	12	12	12	12	
f	47	12	12	12	12	
SQRT	6.86	12	12	12	12	
*	82.27	12	12	12	12	

Note that this stack extension may have more levels with nested register calling. Because HP-65 variable data registers are involved here, we will present the solution in the second part of this project only.

Summary

In this document we presented the development of a recursive HP-65 expression evaluator and code generator. We therefore briefly introduced recursion in maths and computer sciences, then we went through the evaluator using a top-down approach. Finally, we transformed our calculating evaluator into a code generator for the HP-65. The paper will be completed in the second part coming soon.

COMPUTARIUM
Lycée classique de Diekirch
32 av. de la gare L-9233 Diekirch

HP-65 FIFTIETH ANNIVERSARY PROJECT

COMPUTARIUM.LCD.LU PRESENTS

Python compiler for the HP-65 calculator

PART II: Recursive cross-compiler

Author:
Claude Baumann

Co-author:
Francis Massen

This project has primarily didactic objectives
March 22, 2024

Abstract

This paper introduces the second phase of our PYTHON cross-compiler project, which builds upon the initial development of a recursive expression evaluator. As previously explained in the first part, we have chosen to implement the cross-compiler using the LABVIEW programming environment. Despite the inherent complexity of such a computer program, the fundamental functionality remains consistent.

History:

- Version 1.0: February 23, 2024 : first draft
- Version 1.1: February 28, 2024 : first correction
- Version 1.2: February 23, 2024 : first revision
- Version 1.3: March 22, 2024: final

Note: The main frame of the cover painting has been generated with CHATGPT in Vincent van Gogh Still life style. The HP-65 picture has been manually drag and dropped into the frame. (Compared to Part I document of this project, visibly, ChatGPT didn't generate generate the picture in the desired style. All new attempts resulted in adding van Gogh pictures in the background.)

Contents

Chapter I

The Second Phase of the Python Cross-Compiler Project for the HP-65

While PYTHON was originally conceived as an interpreted programming language, cross-compilation has become increasingly important in various downstream projects. However, cross-compilation poses significant challenges. The absence of standardized infrastructure in the Python packaging ecosystem requires considerable effort. Additionally, the lack of official upstream support further complicates the situation, making cross-compilation delicate and reliant on substantial user engagement.

In this paper, we delve into the second phase of our Python cross-compiler endeavor, which extends the groundwork laid during the development of a recursive expression evaluator. As elucidated in the initial part of our project, we opted to construct the cross-compiler within the LabVIEW programming environment. Our exploration, despite the inherent complexity, maintains consistent fundamental functionality with the first phase of the project. This paper serves as a guide for practitioners navigating the intricate landscape of cross-compilation in Python.

References

1. **PEP 720** – Cross-compiling Python packages: Provides insights into cross-compilation approaches used by distributors https://peps.python.org/pep-0720/
2. **crossenv**: A cross-compiling tool for Python extension modules. https://pypi.org/project/crossenv/ and https://crossenv.readthedocs.io/en/latest/
3. Running Python on ARM Processor: A guide on cross-compilation for ARM processors. https://www.hackster.io/locnnil/running-python-on-arm-processor-891290
4. U. Meyer, *Compilerbau*, Rheinwerk Verlag, Bonn, (2021): excellent guide to compiler internals.
5. S. Bergmann, *Compiler Design: Theory, Tools, and Examples*, Rowan University, Open Educational Resources, (2017); download from https://rdw.rowan.edu/cgi/viewcontent.cgi?article=1001&context=oer: excellent tutorial.

Legal Notice

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

And for older versions:

Copyright ©1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

This project makes exclusively use of the PYTHON denomination and elements of the language syntax and structure. No parts of the PYTHON software are used.

1 What is a cross-compiler?

In simple terms, a cross-compiler can generate code that runs on a different system or architecture than the one where the compiler is executed. For example, it could run on a PC but produce code for an ARDUINO module or a RASPBERRY PI PICO. These compilers are particularly useful when developing software for diverse platforms or when the target environment differs significantly from the development environment.

2 Python in the land of Lilliput

Since its early days in the 90s PYTHON's popularity has ever grown, so that meanwhile it counts among the most frequently used programming languages. The main reason for this success might be its simplicity and clearness. Its ease of learning and usage makes it the first choice in educational contexts.

Normally, PYTHON code is interpreted rather than compiled. Although this generally leads to slower code execution, the utility of interpretation outsmarts compilation advantages in many cases, as it reduces program design and debugging time. The code doesn't need to be recompiled at each new program change.

For obvious reasons, the HP-65 cannot run a PYTHON interpreter! We don't need to point out that from today's point of view the HP-65 is a nanoscopic computer with a tiny keyboard, 15 seven-segment LEDs, 9 user registers, 4 stack registers, and 100(!) program steps, of which a 10 digit constant consumes already 10 single steps. With such limited horsepower, one must even wonder whether it's feasible to execute compiled PYTHON code on this diminutive computer. We must necessarily pare down the Python instruction set to a strict minimum. Let's call this set **Lilli-Python**.

2.1 Variables

LILLI-PYTHON can handle floating-point variables only. It will throw an error, if there is an attempt of using more than 8 variables, stack extension included. Note that the HP-65 utilizes Register 9 during the computation of trigonometric functions. That's why LILLI-PYTHON doesn't allow access to this register. LILLI-PYTHON keywords may not be used as variable names.

2.2 Input/output

1. **input()**: This function, when invoked with an empty argument, can exclusively be called within an expression. It causes the HP-65 to stop program execution and wait for a number entry that is completed by pressing the HP-65 **R/S** key.
   - x=input()
   - myvariable=7*sin(input())
2. **print(EXPRESSION)**: This procedure has no return value. So, it may only be used as a regular command.
   - print(x)
   - print(sin(pi/4+x))

2.3 Expressions

2.3.1 Assignments and comparisons

By contrast to part I of our project, expressions are no longer used as simple stand-alone data evaluations, but as part of assignment or comparison statements. Whereas assignments change the content of variables by adopting the values resulting from expressions, comparisons analyze their relationship in terms of equality, difference and order.

Because LILLI-PYTHON uses floating-point variables only, there are no logical operators. Arithmetic operators and mathematical functions in use are the following:

- **+ - * /** (Note: division by zero causes the HP-65 to go into error state with blinking display.)
- **pow(x,y)**=x^y, where x and y may be expressions. If $x \leq 0$ the HP-65 will go into error state.
- **sin(x), cos(x), tan(x)** (Note that tan(90) will issue 9.9999999E99 without producing an error)
- **asin(x), acos(x)**, where $-1 \leq x \leq 1$
- **atan(x)**
- **factorial(x)**, where x must be a natural number (zero included); values>69 return 9.9999999E99
- **abs(x)**
- **int(x)**
- **sqrt(x)**, where x must be positive
- **exp(x)**=e^x
- **exp10(x)**=10^x
- **log(x)**=$\ln(x)$, with $x > 0$
- **log10(x)**=$\log_{10}(x)$, with $x > 0$
- **pi**=π=3.141592653...

2.3.2 +=, -=, *=, /= Assignments

LILLI-PYTHON accepts short-cut assignments, for example: **x+=3**, which signifies: x=x+3

2.4 FOR loop

LILLI-PYTHON can handle three **for** loop variants:

1. **for** i **in range** (EXPRESSION)**:** #do anything
   (Note: Floating-point variable **i** is initialized with zero. The loop is executed as long as **i** < EXPRESSION. Each iteration increments **i** by 1. EXPRESSION must be an integer value.)
2. **for** i **in range** (EXPRESSION1, EXPRESSION2)**:** #do anything
   (Note: Variable **i** is initialized with EXPRESSION1. The loop is executed as long as **i** < EXPRESSION2. Each iteration increments **i** by 1. EXPRESSION1 &2 must be integer values.)
3. **for** i **in range** (EXPRESSION1, EXPRESSION2, EXPRESSION3)**:** #do anything
   (Note: Variable **i** is incremented by EXPRESSION3, where EXPRESSION3 may be positive or negative. EXPRESSION 1..3 must be integer values.)

2.5 WHILE loop

LILLI-PYTHON accepts:

1. **while** EXPRESSION1 **OP** EXPRESSION2**:** #do anything, where **OP**∈{==; !=; <=; >=; <; <}
2. **while** TRUE**:**
   (Note: Although Boolean variables don't exist, LILLI-PYTHON accepts the TRUE constant, in order to provide a structure for infinite loops.

2.6 IF-ELSE structure

LILLI-PYTHON handles:

1. **if** EXPRESSION1 **OP** EXPRESSION2**:** #do anything, where **OP**∈{==; !=; <=; >=; <; <}
2. **if** EXPRESSION1 **OP** EXPRESSION2**:** #do anything **else:** #do anything
3. **if** EXPRESSION1 **OP** EXPRESSION2**:** #do anything {**elif** EXPRESSION1 **OP** EXPRESSION2**:** #do anything} **else:** #do anything

Note that if statements may have more than one occurrences of the **elif** sub-statement. If **elif** is being used, the **else** case must finish the structure.

2.7 BREAK

The **break** instruction forces the program flow to exit the current while or for-loop, if ever there is. Because of the limited number of labels available on the HP-65, LILLI-PYTHON has no **continue** instruction.

2.8 PASS

The **pass** statement is a place-holder that results in empty HP-65 code.

2.9 Subroutines

Because the HP-65 cannot manage nested subroutine calls, LILLI-PYTHON only accepts the following procedures or subroutines, which may not be used as functions inside an expression, as there is no return value. The subroutines must have the following names:

def A():, **def B():**, **def C():**, **def D():**, **def E():** #do anything

Subroutines may be called by pressing one of the A,B,C,D,E, keys on the HP-65, or by using

A(), **B**(), **C**(), **D**(), **E**()

Calling a subroutine within a subroutine is allowed only for one single level (Refer to the HP-65 Handbook pp. 51-52). Note that LILLI-PYTHON will not issue an error message, if there is a nested call attempt.

2.9.1 Special procedure

setmode(MODE), with MODE=rad, deg, grad: this command sets the angular mode.

Chapter II
Top-down development of a recursive cross-compiler for the HP-65

Fig. 1 reproduces a typical code exercise in the **compile1.vi** main front panel. LILLI-PYTHON code is entered in the upper left text box. The program is run by pressing the arrow in the main function toolbar. If no error is notified during compilation, the compiled HP-65 program appears to the right with additional information about variables, procedures and labels. The compiled program can be saved as a text file (with file extension **.hp65**). Note that the PYTHON code text box doesn't work as a regular code-editor. However, code can be copy-pasted from elsewhere. The program example in Fig. 1 searches real roots for the second degree equation $a\,x^2 + b\,x + c = 0$.

The compilation process converts the LILLI-PYTHON code into an intermediate parsable code, which has added code lines. This is helpful, in the case of an error message that refer to the line-numbers of the parsable code. As can be seen in Fig. 1, this parsable code has no indentation. Instead, the compiler has added terminating] brackets that embrace code blocks together with the starting **colons**. Also, multiple statements that are separated with **semicolons** now are changed into single lines.

Indentation errors are recognized **before** the generation of the parsable code, so that this intermediate code doesn't appear in the lower left text box. If multiple statement lines are being used, the user must take care that line numbers take into account their separation into single lines. In other words, each statement is counted as an independent line.

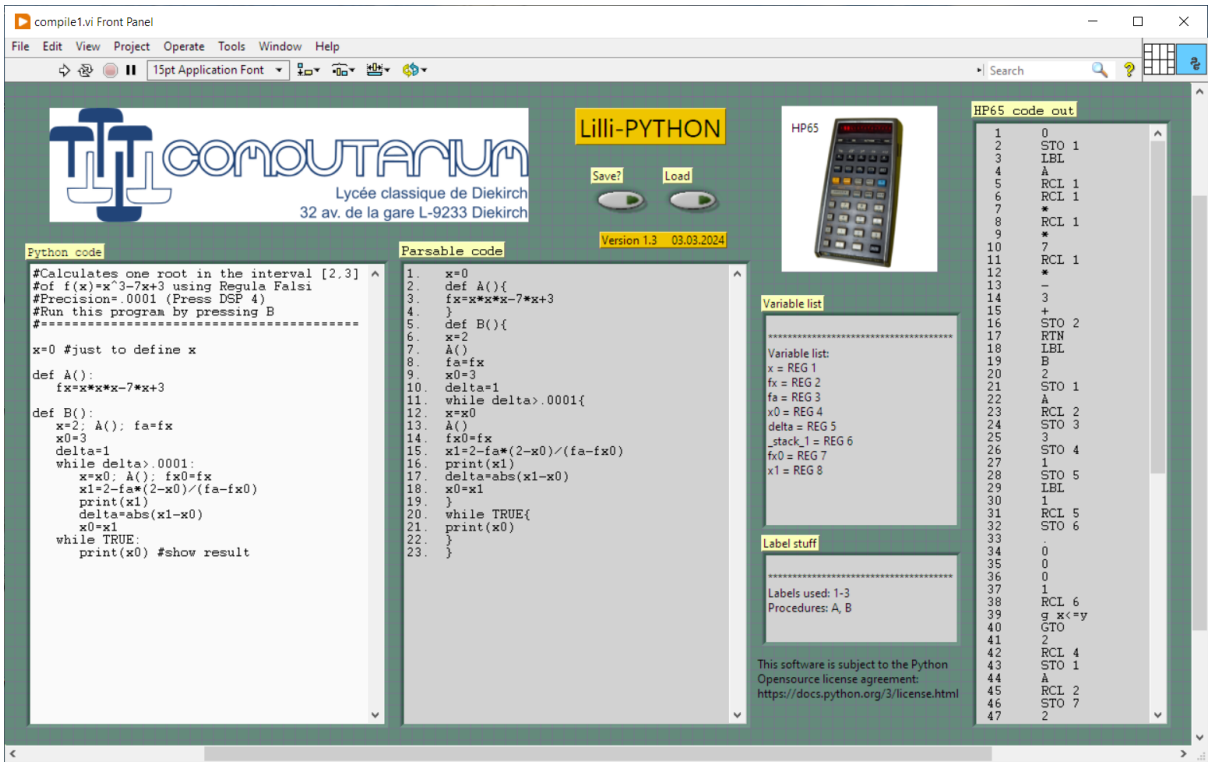


Figure 1: The compiler has converted PYTHON code to HP-65 code passing via the **parsable code**, which is an intermediate structured code.

Fig. 2 shows that the compiler for LILLI-PYTHON code is composed of three major parts: the **compiler-initialization** (cf. traffic lights), the **compiler-lexer** and the **compiler-parser**, which are necessarily more complex than the previous ones met in the expression evaluator part.

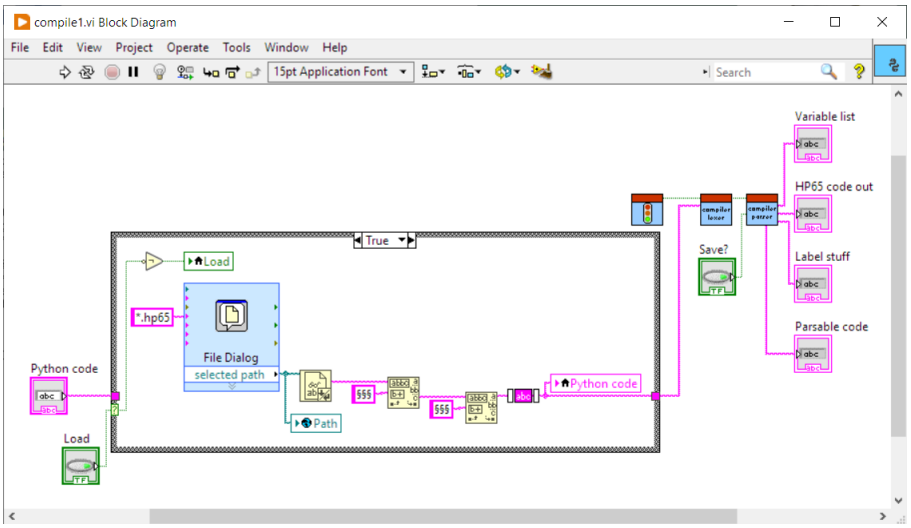


Figure 2: The compiler is made of three major parts: initialization, lexer and parser.

3 From indentation to code blocks

The key distinction of PYTHON lies in the fact that it eschews brackets (as seen in languages like C) to delineate code structures. Instead, PYTHON relies on line indentation to demarcate code blocks. This approach significantly enhances the readability of structured code. However, there is a trade-off. Indeed, a PYTHON lexer must identify code blocks from this representation and convert it into parsable code.

Another particularity of PYTHON is that code lines are not terminated by semicolons, although semicolons can be used as separations between statements.

Comments are preceded by a # symbol. They are ignored by the lexer.

Because of the complexity of the compiler-lexer sub-diagrams, we only present here the high-end program diagram (cf. Fig. 3). The program executes several subroutines for the conversion of regular PYTHON code into parsable code. First, the program replaces all colons „:“ with the non-PYTHON **„{“**; it then eliminates comments, counts indentation spaces and eliminates unnecessary white space. Then, it resolves multiple statement lines that have colons and/or statement separations by semicolons. At this point, the program converts the indentation structure into code blocks that are terminated with the non-PYTHON **„}“** symbol.

Before invoking the tokenizer, the lexer faces a non-trivial challenge. Since a typical compiler has to recognize variable names as identifiers, it must avoid any ambiguity with valid LILLI-PYTHON keywords. These keywords closely align with the token list. Internally, the tokenizer scans this list while processing the current string. When it encounters a valid token, it returns the corresponding index from the token list. Consequently, it's crucial that longer token names appear before shorter ones. Otherwise, valid tokens might be overlooked. The following example illustrates this principle:

Example 3.1. *Bad parsing:*
*Imagine that the keyword **in** precedes the keyword **sin** on the token list.*

*In that case, $x = sin(pi)$ is understood as the token chain: **IDENT = IDENT in (pi)**, because token **in** is found earlier as token **sin***

This also concerns user variables such as: **min, MAX, x1, force, etc.**, which contain the tokens **in, A, %f, for**. For this reason the lexer must spot all identifiers in the LILLI-PYTHON code. Fortunately, because LILLI-

PYTHON exclusively uses floating point variables, these can be located easily, because they either precede the = token or immediately follow the **for** token, while being defined. With this trick, the lexer can draw a sorted list of identifiers, from which it has removed any duplicates. This list is now inserted into the token list right at the beginning.

Only now the **tokenizer** is called with an augmented list of valid tokens compared to the one used in the previous expression evaluator (cg. Fig. 4). Note that there is a significant difference in the **tokenizer**'s functionality, as no error is generated, if a symbol isn't understood. Instead, the **tokenizer** uses the token **IDENT** to tell the parser that it should handle a user defined token.

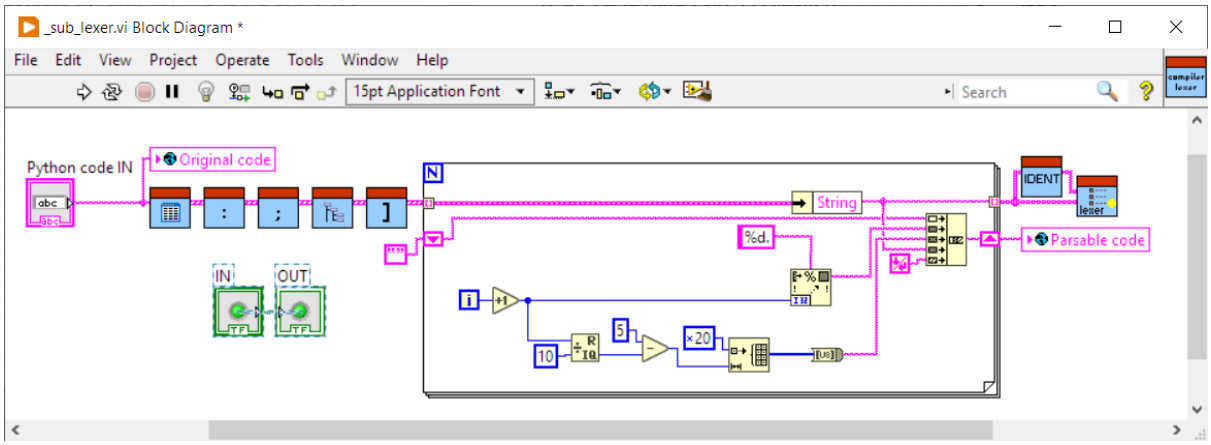


Figure 3: The lexer has multiple subroutines handling the different stages of the lexer process.

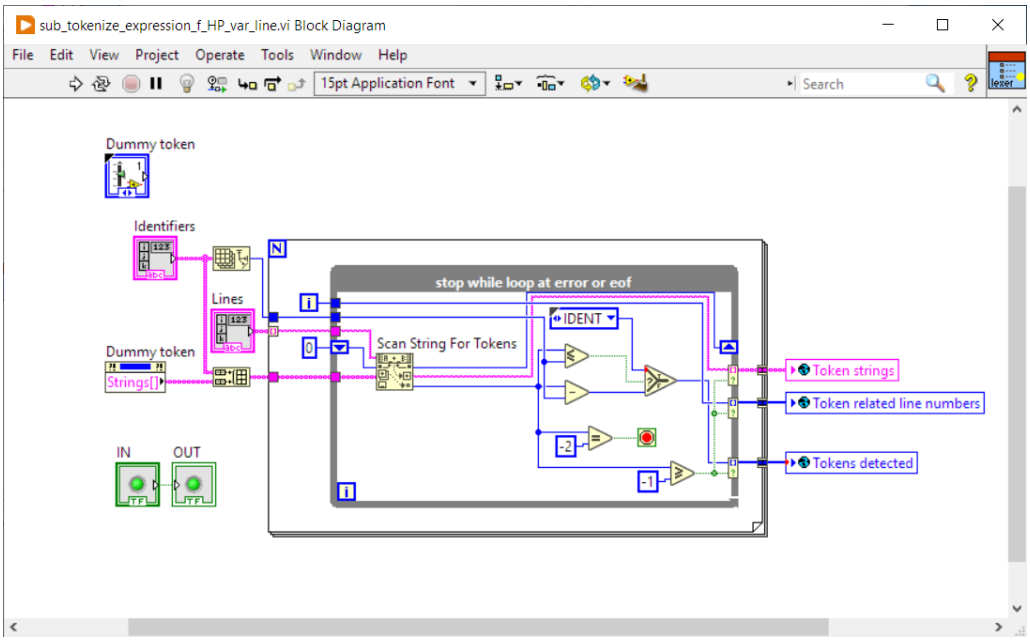


Figure 4: A minimally altered **tokenizer** compared to the one developed in project Part I.

4 Swiss knife

Formally, a computer program consists of a sequence of meaningful statements that the computer should execute in the order they appear. Consequently, the program must be capable of deriving the accurate semantic meaning from the statement structure, which, in turn, must adhere to the specified syntax. The

LILLI-PYTHON parser must handle the following production rules:

(Remind that the lexer has replaced colons with the non- PYTHON { symbol in the **def**, **if**, **elif**, **else**, **for**, **while** statements or sub-statements.)

```
S--> STATEMENT-LIST
STATEMENT-LIST -> STATEMENT-LIST | STATEMENT
STATEMENT --> VAR = EXPRESSION | VAR += EXPRESSION | VAR -= EXPRESSION |
              VAR *= EXPRESSION | VAR /= EXPRESSION |
              for VAR in range ( EXPRESSION ) { STATEMENT-LIST } |
              for VAR in range ( EXPRESSION, EXPRESSION ) { STATEMENT-LIST } |
              for VAR in range ( EXPRESSION, EXPRESSION, EXPRESSION ) { STATEMENT-LIST }
              while EXPRESSION == EXPRESSION { STATEMENT-LIST } |
              while EXPRESSION != EXPRESSION { STATEMENT-LIST } |
              while EXPRESSION <= EXPRESSION { STATEMENT-LIST } |
              while EXPRESSION >= EXPRESSION { STATEMENT-LIST } |
              while EXPRESSION < EXPRESSION { STATEMENT-LIST } |
              while EXPRESSION > EXPRESSION { STATEMENT-LIST } |
              while TRUE: STATEMENT-LIST |
              if EXPRESSION == EXPRESSION { STATEMENT-LIST } [ ELIF-LIST ] [ else { STATEMENT-LIST } ] |
              if EXPRESSION  != EXPRESSION { STATEMENT-LIST } [ ELIF-LIST ] [ else { STATEMENT-LIST} ] |
              if EXPRESSION  <= EXPRESSION { STATEMENT-LIST } [ ELIF-LIST ] [ else { STATEMENT-LIST} ] |
              if EXPRESSION  >= EXPRESSION { STATEMENT-LIST } [ ELIF-LIST ] [ else { STATEMENT-LIST} ] |
              if EXPRESSION  < EXPRESSION { STATEMENT-LIST } [ ELIF-LIST ] [ else { STATEMENT-LIST} ] |
              if EXPRESSION  > EXPRESSION { STATEMENT-LIST } [ ELIF-LIST ] [ else { STATEMENT-LIST} ] |
              break | pass |
              def A ( ) { STATEMENT-LIST } |
              def B ( ) { STATEMENT-LIST } |
              def C ( ) { STATEMENT-LIST } |
              def D ( ) { STATEMENT-LIST } |
              def E ( ) { STATEMENT-LIST } |
              A ( ) | B ( ) | C ( ) | D ( ) | E ( ) |
              input() | print ( EXPRESSION ) |
              setmode ( rad ) | setmode ( deg ) | setmode ( grad )
ELIF-LIST--> ELIF-LIST | ELIF
ELIF--> elif EXPRESSION == EXPRESSION { STATEMENT-LIST }
```

Notes:

1. VAR represents a new terminal symbol in expressions that must be identified by an unambiguous variable name.

2. **end of string (eof)** or] terminate a statement-list

Fig. 5 shows that the compiler-parser has three components, the **launch** of the parsing process, the **Swiss knife** automaton and the **printout**. The third subroutine has no other function than displaying and saving the generated code.

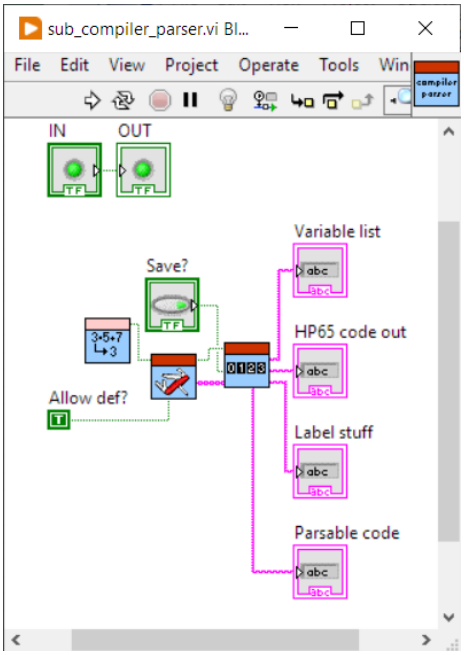


Figure 5: The compiler-parser.

The heart of the compiler-parser is the indirectly recursive **Swiss knife** subroutine (cf. Fig. 6). The program runs a loop that is stopped, if the last token has been parsed, or if a block termination] token has been encountered. It checks whether the current statement begins with a legal token from the expected list. For instance, if the token is **IDENT**, then the parser understands that a new variable should be created and assigned with the result value from an arithmetic or mathematical expression. As presented briefly in the previous project part, the expression evaluator that is called here uses the code optimizer by adding the stack extension functionality and solving the superfluous **ENTER** problem. Fig. 7 shows the **while** case, which runs the recursive **while** sub.vi, which is reproduced in Fig. 8. This subroutine indirectly calls the **Swiss knife**, because the block code belonging to the while structure could be another recursive sub-structure of statements.

The loop statements **for, while, if, else** require **labels** indicating, where the program execution has to jump, in order to run specific code parts or skip undesired ones. This cannot be avoided as computer program steps are executed sequentially. Note that the **break** statement needs special care, because the parser must keep track of the correct label, where to jump.

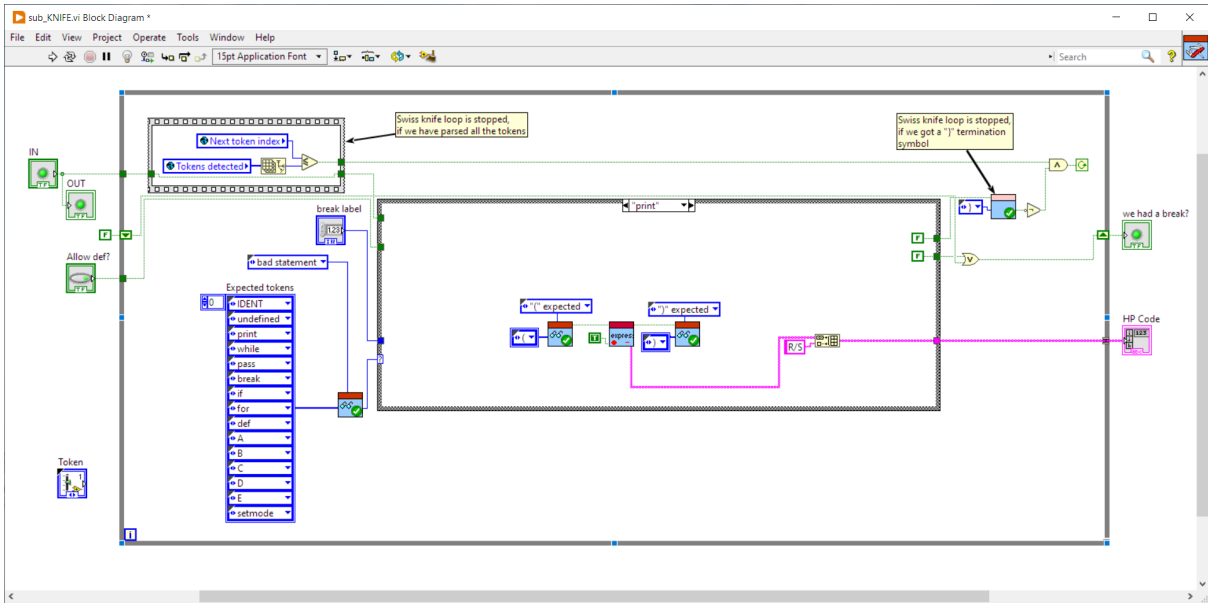


Figure 6: Extract of the **Swiss knife** diagram.

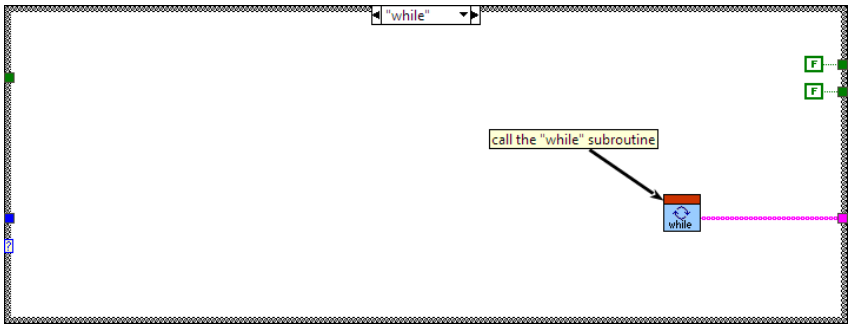


Figure 7: **Swiss knife** case for calling the **while** subroutine.

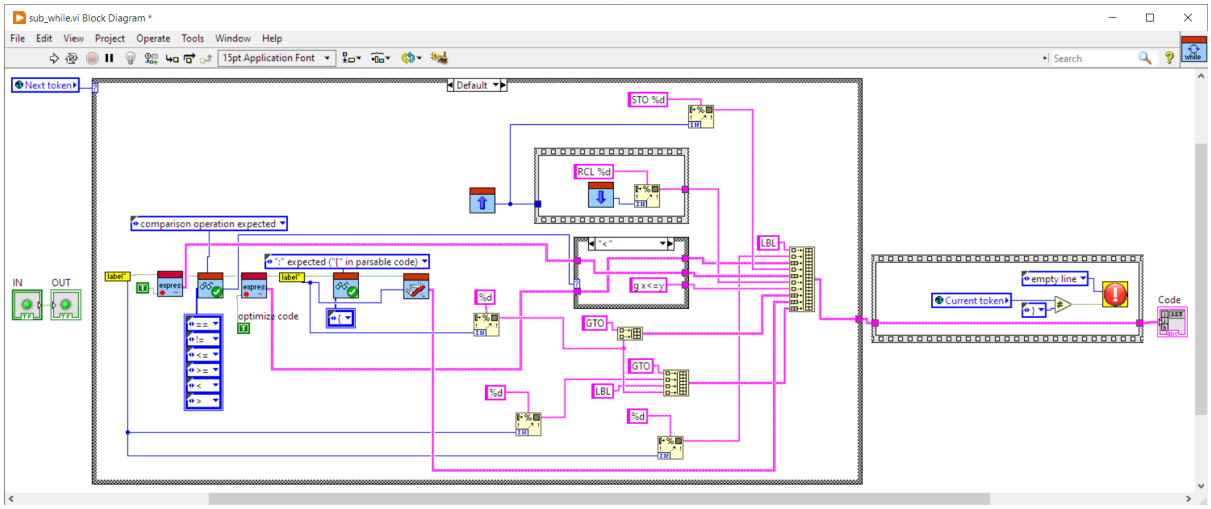


Figure 8: Aspect of the **while** sub.vi.

The HP-65's test functions are limited to **x!=y, x<=y, x==y, x>y** as visible in Fig. 9

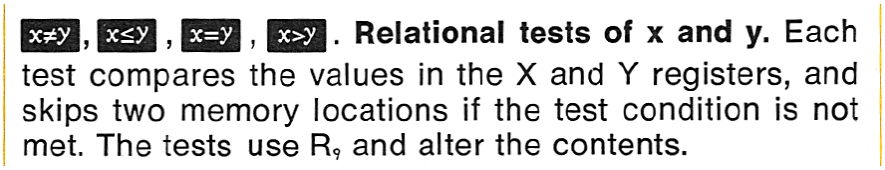


Figure 9: Extract of HP-65 Owner's Handbook p.63.

Example 4.1. *while x<=y*

PYTHON *code snippet:*

```
x=0;  y=1
while  x<=y :  pass
```

is compiled to HP-65 code:

```
0
STO 1
1
STO 2
LBL
1
RCL 1
STO 3
RCL 2
RCL 3
g x>y
GTO
2
GTO
1
LBL
2
```

As one can see, the compiler internally applies the logical ***(x>y) = NOT(x<=y)*** *equation, in order to skip the exit jump* ***GTO 2****, because this skipping is executed only, if* ***(x>y)=FALSE****. Note that the superfluous use of intermediate Register 3 results from the fact that the compiler doesn't optimize the code in the case of EXPRESSION1=VAR yet. This will be added in a further* LILLI-PYTHON *version.*

Example 4.2. *while x<y*

PYTHON *code snippet:*

```
x=0;  y=1
while  x<y :  pass
```

is compiled to HP-65 code:

```
0
STO 1
1
STO 2
LBL
1
RCL 2
STO 3
RCL 1
RCL 3
g x<=y
GTO
```

```
2
GTO
1
LBL
2
```

In this example, the order of the registers is exchanged, so that Register 2 is intermediately stored in temporary Register 3. This time the logical equation ***(x<=y) = NOT(x>y)*** *equation is applied, while –as said–* PYTHON *variables* ***x*** *and* ***y*** *have been exchanged.*

Notes:

1. During the development of LILLI-PYTHON we discovered a further issue concerning the leading minus sign in **Expression** in the case of floating point constants with negative exponents.

   $x = -1E-2$ was **badly** compiled to:

   ```
   1
   EEX
   CHS
   2
   CHS
   STO 1
   ```

   The HP-65 would set Register 1 to 100 instead of 0.01, because both **CHS** affect the exponent only and cancel each other. We fixed this bug, so that now the **correct** code is:

   ```
   1
   CHS
   EEX
   CHS
   2
   STO 1
   ```

2. The **stack extension** sub.vi is shown in Fig. 10

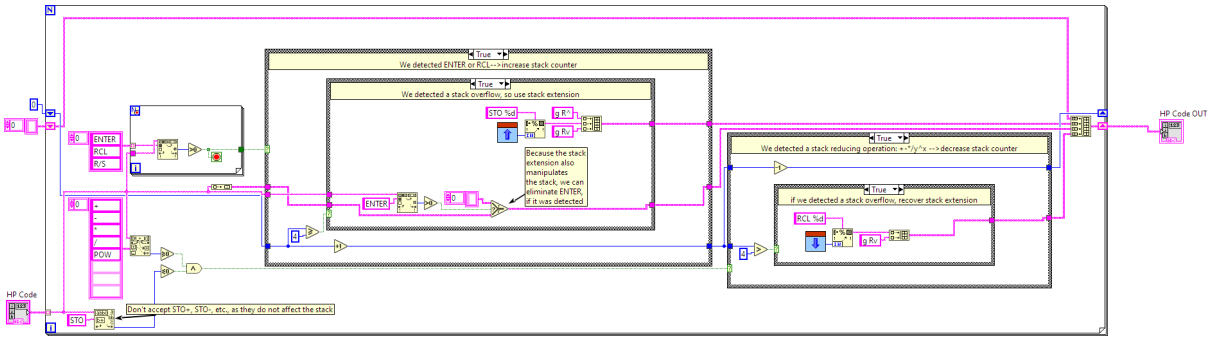


Figure 10: Stack extension method.

3. The **ENTER elimination** sub.vi is shown in Fig. 11

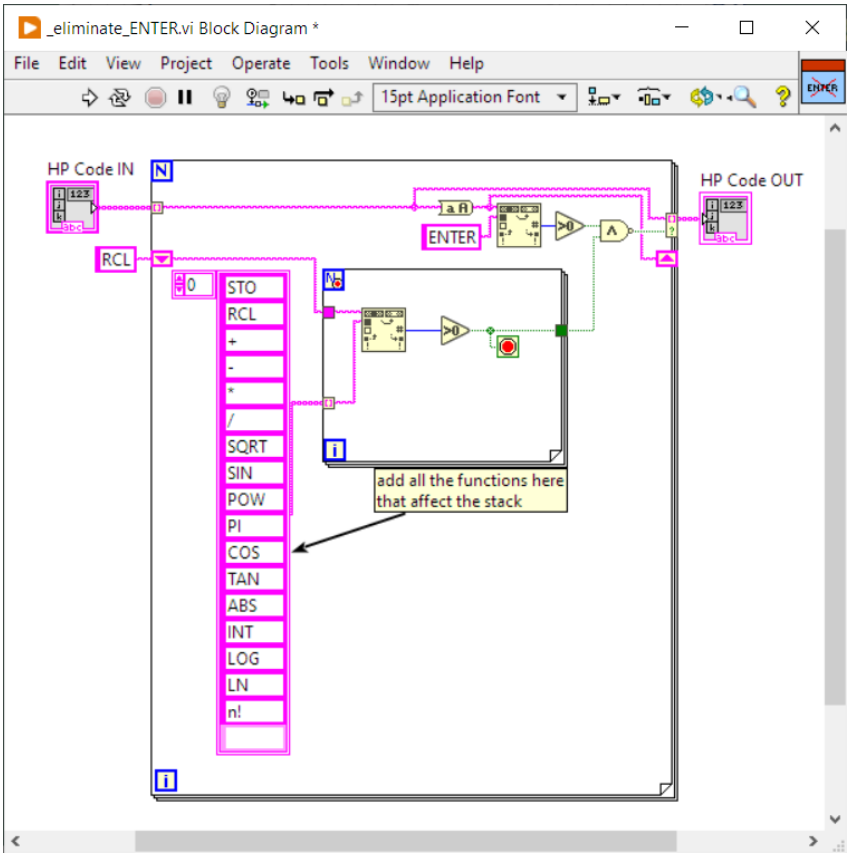


Figure 11: Superfluous **ENTER** elimination.

Because of the overall complexity of the complete compiler program, we will not go into further details here. The compiler source code can be downloaded from the https://computarium.lcd.lu/ web-site. For those interested readers, who do not own a LabVIEW environment, a ready-to-use application is available. It requires the install of the freely available LabVIEW ≥ 2021 SP1 Patch Run-Time Engine (32bit) at:

https://www.ni.com/en/support/downloads/software-products/download.labview.html,

where you need to create an account and be patient during the install.

Important Note: For compatibility reasons with older systems, we chose to build the Lilli-Python compiler as a 32bit program. It will work only with the correct LabVIEW 2021 Run-Time Engine (32bit) version.

5 Sample code

5.1 Naive approximation of a decimal number with a rational number

As an illustrative example, let's begin by introducing a straightforward method for converting a decimal number into a rational number with a specified precision. The algorithm iteratively adjusts test numerators, calculates the corresponding denominator, and checks whether the resulting fraction approximates the input decimal number within the predefined error bounds. However, when executing this code on an HP-65 calculator, users may be astounded by the significant amount of time required to produce a result. For instance, obtaining the approximation for π as 355/113 takes approximately five minutes! In stark contrast, the same algorithm executes in less than a microsecond on a modern personal computer.

```
COMPUTARIUM (C. Baumann 2024)
HP65 Python Compiler
************************************
```

```
#Computes a rational approximation
#    x~=n/d
#with error<1/precision
#First input x, then input precision
#=====================================

x=input()
precision=input()
maxerror=1/precision
for n in range (int(x)+1,precision):
    d=int(n/x)
    if abs(n/d-x)<maxerror:
        print(n)
        print(d)
        break
print(55555) #show that program has ended
```

```
************************************
Parsable code
1.      x=input()
2.      precision=input()
3.      maxerror=1/precision
4.      for n in range (int(x)+1,precision){
5.      d=int(n/x)
6.      if abs(n/d-x)<maxerror{
7.      print(n)
8.      print(d)
9.      break
10.     }
11.     }
12.     print(55555)


************************************
Variable list:
x = REG 1
precision = REG 2
maxerror = REG 3
n = REG 4
d = REG 5
_stack_1 = REG 6

************************************
Labels used: 1-3
Procedures: -

************************************
HP65 code
  1         R/S
  2         STO 1
  3         R/S
  4         STO 2
  5         1
  6         RCL 2
  7         /
  8         STO 3
  9         RCL 1
 10         f
 11         INT
 12         1
 13         +
 14         STO 4
 15         LBL
 16         1
```

```
17          RCL 2
18          RCL 4
19          g x<->y
20          g x<=y
21          GTO
22          2
23          RCL 4
24          RCL 1
25          /
26          f
27          INT
28          STO 5
29          RCL 3
30          STO 6
31          RCL 4
32          RCL 5
33          /
34          RCL 1
35          -
36          g
37          ABS
38          RCL 6
39          g x<=y
40          GTO
41          3
42          RCL 4
43          R/S
44          RCL 5
45          R/S
46          GTO
47          2
48          LBL
49          3
50          1
51          STO
52          +
53          4
54          GTO
55          1
56          LBL
57          2
58          5
59          5
60          5
61          5
62          5
63          R/S
64          RTN
```

5.2 Approximating a decimal number by continued fractions

We may choose a much faster method, which is based on the Euklidian division algorithm.[1] This example shows the evident limits of the HP-65 in terms of data and program memory. The incredibly small memory size requires that the problem is broken into two independent procedures. The first will compute the continued fraction coefficients that the user must physically note on paper.[2] The second program will then convert the coefficients step by step to the rational number.

[1] https://en.wikipedia.org/wiki/Euclidean_division
[2] https://en.wikipedia.org/wiki/Continued_fraction

5.2.1 Yielding the continued fraction coefficients of a decimal number

```
COMPUTARIUM (C. Baumann 2024)
HP65 Python Compiler
************************************


#Input x, which is the number to be converted
#into a continued fraction [a0;a1,...ak]
#The second number to input is the initial
#(den)onimator. The algorithm starts with
#(num)erator=int(x*den)
#=====================================

x=input()
den=input()
num=int(x*den)
while den>0:
    tmp=int(num/den) #get single coefficient
    print(tmp) #copy value of tmp & key R/S to continue
    num=num-tmp*den #get the fractional part
#now exchange numerator and denominator (=inverse)
    tmp=den
    den=num
    num=tmp
print(555) #indicates the end of the program


************************************
Parsable code
1.      x=input()
2.      den=input()
3.      num=int(x*den)
4.      while den>0{
5.      tmp=int(num/den)
6.      print(tmp)
7.      num=num-tmp*den
8.      tmp=den
9.      den=num
10.     num=tmp
11.     }
12.     print(555)


************************************
Variable list:
x = REG 1
den = REG 2
num = REG 3
_stack_1 = REG 4
tmp = REG 5

************************************
Labels used: 1-2
Procedures: -

************************************
HP65 code
  1         R/S
  2         STO 1
  3         R/S
  4         STO 2
  5         RCL 1
  6         RCL 2
```

```
 7          *
 8          f
 9          INT
10          STO 3
11          LBL
12          1
13          RCL 2
14          STO 4
15          0
16          RCL 4
17          g x<=y
18          GTO
19          2
20          RCL 3
21          RCL 2
22          /
23          f
24          INT
25          STO 5
26          RCL 5
27          R/S
28          RCL 3
29          RCL 5
30          RCL 2
31          *
32          -
33          STO 3
34          RCL 2
35          STO 5
36          RCL 3
37          STO 2
38          RCL 5
39          STO 3
40          GTO
41          1
42          LBL
43          2
44          5
45          5
46          5
47          R/S
48          RTN
```

5.2.2 Yielding numerators and denominators from continued fraction coefficients

We use here the recursion method explained in:
https://www.fim.uni-passau.de/fileadmin/dokumente/fakultaeten/fim/lehrstuhl/sauer/geyer/Kettenbrueche.pdf.

```
COMPUTARIUM (C. Baumann 2024)
HP65 Python Compiler
************************************

#This program converts a continued fraction
#[a0,...a_k] into a rational fraction
#The program issues numerator A_k
#and denominator B_k

A_m2=0;A_m1=1;B_m2=1;B_m1=0
while TRUE:
    dummy=55555 #invites to input next a_k
    a_k=input()
    A_0=a_k*A_m1+A_m2
    print(A_0)
```

```
    B_0=a_k*B_m1+B_m2
    print(B_0)
    A_m2=A_m1;A_m1=A_0
    B_m2=B_m1;B_m1=B_0
```

```
************************************
Parsable code
1.      A_m2=0
2.      A_m1=1
3.      B_m2=1
4.      B_m1=0
5.      while TRUE{
6.      dummy=55555
7.      a_k=input()
8.      A_0=a_k*A_m1+A_m2
9.      print(A_0)
10.     B_0=a_k*B_m1+B_m2
11.     print(B_0)
12.     A_m2=A_m1
13.     A_m1=A_0
14.     B_m2=B_m1
15.     B_m1=B_0
16.     }


************************************
Variable list:
A_m2 = REG 1
A_m1 = REG 2
B_m2 = REG 3
B_m1 = REG 4
dummy = REG 5
a_k = REG 6
A_0 = REG 7
B_0 = REG 8

************************************
Labels used: 1
Procedures: -

************************************
HP65 code
  1         0
  2         STO 1
  3         1
  4         STO 2
  5         1
  6         STO 3
  7         0
  8         STO 4
  9         LBL
 10         1
 11         5
 12         5
 13         5
 14         5
 15         5
 16         STO 5
 17         R/S
 18         STO 6
 19         RCL 6
 20         RCL 2
 21         *
 22         RCL 1
```

```
23          +
24          STO 7
25          RCL 7
26          R/S
27          RCL 6
28          RCL 4
29          *
30          RCL 3
31          +
32          STO 8
33          RCL 8
34          R/S
35          RCL 2
36          STO 1
37          RCL 7
38          STO 2
39          RCL 4
40          STO 3
41          RCL 8
42          STO 4
43          GTO
44          1
45          RTN
```

5.3 Calculate a root in the interval [2,3] for $f(x) = x^3 - 7x + 3$ using Regula falsi

Preliminary note: The REGULA FALSI is well explained at https://byjus.com/maths/false-position-method/.

```
COMPUTARIUM (C. Baumann 2024)
HP65 Python Compiler
************************************


#Calculates one root in the interval [2,3]
#of f(x)=x^3-7x+3 using Regula Falsi
#Precision=.0001 (Press DSP 4)
#Run this program by pressing B
#=========================================

x=0 #just to define x

def A():
    fx=x*x*x-7*x+3

def B():
    x=2; A(); fa=fx
    x0=3
    delta=1
    while delta>.0001:
        x=x0; A(); fx0=fx
        x1=2-fa*(2-x0)/(fa-fx0)
        print(x1)
        delta=abs(x1-x0)
        x0=x1
    while TRUE:
        print(x0) #show result

************************************
Parsable code
1.      x=0
2.      def A(){
3.      fx=x*x*x-7*x+3
4.      }
5.      def B(){
```

```
6.      x=2
7.      A()
8.      fa=fx
9.      x0=3
10.     delta=1
11.     while delta>.0001{
12.     x=x0
13.     A()
14.     fx0=fx
15.     x1=2-fa*(2-x0)/(fa-fx0)
16.     print(x1)
17.     delta=abs(x1-x0)
18.     x0=x1
19.     }
20.     while TRUE{
21.     print(x0)
22.     }
23.     }


************************************
Variable list:
x = REG 1
fx = REG 2
fa = REG 3
x0 = REG 4
delta = REG 5
_stack_1 = REG 6
fx0 = REG 7
x1 = REG 8

************************************
Labels used: 1-3
Procedures: A, B

************************************
HP65 code
  1         0
  2         STO 1
  3         LBL
  4         A
  5         RCL 1
  6         RCL 1
  7         *
  8         RCL 1
  9         *
 10         7
 11         RCL 1
 12         *
 13         -
 14         3
 15         +
 16         STO 2
 17         RTN
 18         LBL
 19         B
 20         2
 21         STO 1
 22         A
 23         RCL 2
 24         STO 3
 25         3
 26         STO 4
```

```
27          1
28          STO 5
29          LBL
30          1
31          RCL 5
32          STO 6
33          .
34          0
35          0
36          0
37          1
38          RCL 6
39          g x<=y
40          GTO
41          2
42          RCL 4
43          STO 1
44          A
45          RCL 2
46          STO 7
47          2
48          RCL 3
49          2
50          RCL 4
51          -
52          *
53          RCL 3
54          RCL 7
55          -
56          /
57          -
58          STO 8
59          RCL 8
60          R/S
61          RCL 8
62          RCL 4
63          -
64          g
65          ABS
66          STO 5
67          RCL 8
68          STO 4
69          GTO
70          1
71          LBL
72          2
73          LBL
74          3
75          RCL 4
76          R/S
77          GTO
78          3
79          RTN
```

5.4 Calculate the n'th digit of PI using the BAILEY–BORWEIN–PLOUFFE (BBP) formula

The following code exercise is a good example for demonstrating that short Python code can fill the entire program memory of the HP-65 calculator. Note that BAILEY–BORWEIN–PLOUFFE (BBP) formula has been discovered in 1995 only. [3].

```
COMPUTARIUM (C. Baumann 2024)
HP65 Python Compiler
```

[3] https://en.wikipedia.org/wiki/Bailey%E2%80%93Borwein%E2%80%93Plouffe_formula

```
************************************


# Calculate PI using Simon Plouffe's
# formula
# Enter the desired number of precise
# decimal digits; press R/S
# When the program stops, Press
# RCL 2 to get the yielded value.
#=====================================


precision=input()
my_pi=0
for k in range (precision):
    my_pi+=pow(1/16,k)*(4/(8*k+1)-2/(8*k+4)-1/(8*k+5)-1/(8*k+6))

************************************
Parsable code
1.   precision=input()
2.   my_pi=0
3.   for k in range (precision){
4.   my_pi+=pow(1/16,k)*(4/(8*k+1)-2/(8*k+4)-1/(8*k+5)-1/(8*k+6))
5.   }


************************************
Variable list:
precision = REG 1
my_pi = REG 2
k = REG 3
_stack_1 = REG 4

************************************
Labels used: 1-2
Procedures: -

************************************
HP65 code
  1     R/S
  2     STO 1
  3     0
  4     STO 2
  5     0
  6     STO 3
  7     LBL
  8     1
  9     RCL 3
 10     RCL 1
 11     g x<=y
 12     GTO
 13     2
 14     1
 15     ENTER
 16     1
 17     6
 18     /
 19     RCL 3
 20     g
 21     POW
 22     4
 23     ENTER
 24     8
 25     RCL 3
```

```
26      *
27      1
28      +
29      /
30      2
31      ENTER
32      8
33      g R^
34      STO 4
35      g Rv
36      RCL 3
37      *
38      RCL 4
39      g Rv
40      g R^
41      STO 4
42      g Rv
43      4
44      +
45      RCL 4
46      g Rv
47      /
48      -
49      1
50      ENTER
51      8
52      g R^
53      STO 4
54      g Rv
55      RCL 3
56      *
57      RCL 4
58      g Rv
59      g R^
60      STO 4
61      g Rv
62      5
63      +
64      RCL 4
65      g Rv
66      /
67      -
68      1
69      ENTER
70      8
71      g R^
72      STO 4
73      g Rv
74      RCL 3
75      *
76      RCL 4
77      g Rv
78      g R^
79      STO 4
80      g Rv
81      6
82      +
83      RCL 4
84      g Rv
85      /
86      -
87      *
88      STO
```

```
89      +
90      2
91      1
92      STO
93      +
94      3
95      GTO
96      1
97      LBL
98      2
99      RTN
```

Summary

In this document we presented the development of a recursive PYTHON cross-compiler for the HP-65. The compiler source code can be downloaded from the https://computarium.lcd.lu/ web-site. For those interested readers, who do not own a LABVIEW environment, a ready-to-use application is available. It requires the install of the freely available LABVIEW ≥ 2021 SP1 PATCH RUN-TIME ENGINE (32BIT) at:

https://www.ni.com/en/support/downloads/software-products/download.labview.html,

where you need to create an account and be patient during the install.

Important Notes:

1. For compatibility reasons with older systems, we chose to build the LILLI-PYTHON compiler as a 32bit program. It will work only with the correct LABVIEW 2021 RUN-TIME ENGINE (32BIT) version.
2. Please send remarks, questions and bug reports to claude.baumann@education.lu

---