Project Specification Document for the final Phase 5b of the Semester-Project

In our COS341 Semester Project of the year 2024, old-fashioned **BASIC** will be our Target Code. Thus the Target Code will be really executable via some BASIC Emulator that can be found on the Internet.

- For a general overview see → **https://en.wikipedia.org/wiki/BASIC**

From **Chapter #7** of our Textbook you already know that *Pattern Matching* is the typical method of Target Code generation from some already existing Intermediate Code (as per Chapters #6 and #9). In this project specification document, which is relevant for the final phase **5b** of your project, some guidelines are provided along the lines of which a diligent student should be able to complete the project.

Line-Numbering

All BASIC programs are line-numbered. As the Intermediate Code (as per Chapters #6 and #9) is not yet line-numbered, the project ***student's Compiler must generate such line numbers*** in its final compilation phase (5b), such that every previously un-numbered command is thereafter numbered.

String Variables

In old-fashioned BASIC, a string variable looks like ***C$***, where ***C*** is one capital letter in the Roman Alphabet { A, B, C, ... , X, Y, Z }. In other words: an old-fashioned BASIC program has typically not more than 26 different string variables (as 26 is the size of the Roman Alphabet). With help of your **Symbol Table** which still captures **Type Information** you can identify the Text-type variables in the input program and *"match"* them *consistently* onto the above-mentioned form of the old-fashioned BASIC string variables.

- ***Also note***: *Newer/Later versions of BASIC do not follow this convention any longer.* ***IF*** *your BASIC Emulator, which executes given BASIC programs via some interface on the internet, does not support this old-fashioned notation any more, then your finally generated target code does not need to present the string variables in the above-mentioned old* ***$****-notation.*

Simulation of the Run-Time-Stack

Our run-time-stack shall be simulated in BASIC by a **two-dimensional array** of the kind (**7**,***n***) where ***n*** is some *not-too-large* number just good enough for "Toy Examples". The number n limits the number of repetitions "through" which a recursive function can run. Let's say you would have chosen ***n***=**30**, then any recursive function could go through maximally **30** recursions - and thereafter the program would "crash".
Because a **RecSPL** function has exactly three parameters and exactly three internal local variables, for every "**frame**" (***x***,**f**) of the array the usage of the 7 fields is as follows:

- ***x*** = **0**: This field shall hold the Return Value
- ***x*** = **1**: This field shall hold the 1st Parameter
- ***x*** = **2**: This field shall hold the 2nd Parameter
- ***x*** = **3**: This field shall hold the 3rd Parameter
- ***x*** = **4**: This field shall hold the 1st Local Variable
- ***x*** = **5**: This field shall hold the 2nd Local Variable
- ***x*** = **6**: This field shall hold the 3rd Local Variable

Stack-"**Push**" is thus simulated by changing the array-index: ***f = f + 1***
Stack-"**Pop**" is thus simulated by changing the array-index: ***f = f – 1***

The two-dimensional array's name shall be **M** (for "**M**emory"), such as in Chapter #9 of our book.

The **declaration of the Array** must be the **Target Program's <u>first</u> command**, such that that all the remaining commands of the generated BASIC program can utilise the Array when it is needed. The declaration of such an array in BASIC has the syntactic form:

LineNumber **DIM** M(7,*n*)

Further Details of the "Pattern Matching" from Intermediate Code to BASIC Target Code

The following Matching-Table is provided ***approximately*** *"in the spirit of"* **Chapter #7** (see for example Fig.7.1), although *<u>not all</u> details are explicitly stated*. Whereas the most essential <u>hints</u> are given, which are necessary for a successful completion of the Semester-Project, the *diligent project-students must still do some "thinking on their own"* in order to come up with a solution that finally "works".
Throughout the following table, ***LN*** means Line Number, whereby *different* line numbers are shown as *LN*, *LN'*, *LN''*, (and the like).

Pattern from Chapter #6 or Chapter #9	Corresponding BASIC Code Pattern
x := *y // assignment command*	There are <u>two</u> translation options: a short option and a long option, which are <u>both valid</u> BASIC: • ***LN*** *x* = *y* *// short option* • ***LN*** **LET** *x* = *y // long option*
Boolean Equals Comparison Operator	The single equation symbol = is used in BASIC *also* for Boolean Comparisons, *not only* for the above-mentioned assignment commands.
LABEL *LabelName*	***LN*** **REM** *LabelName* *// comment: REM is the Comment of BASIC ;* *// that's basically a non-executable "command"*
GOTO *LabelName* *// comment: that this <u>the "normal" GOTO</u> from* *// **Chapter <u>#6</u>**, which has nothing to do with* *// Functions.*	***LN*** **GOTO** ***LN'*** *// whereby **LN'** is the line number in which we* *// can find the corresponding "Label" command* *// **REM LabelName** (mentioned above)*
IF *cond* **THEN** *SomeLabel* **ELSE** *OtherLabel*	***LN*** **IF** *cond* **THEN** ***LN'*** ***LN''*** **GOTO** ***LN'''*** *// Thereby:* *// In line LN' we can find* **REM** *SomeLabel* *// In line LN''' we can find* **REM** *OtherLabel.* *// Note that the most primitive **old-fashioned*** *// BASIC does <u>not</u> have ELSE.*
M[SP] := *dynamicReturnAddress* **GOTO *FunctionNameLabel*** LABEL *dynamicReturnAddress* *// Function-Call, as per **Chapter #9, Fig.9.4***	***LN*** **GOSUB** ***LN'*** *// In LN' we find **REM FunctionNameLabel*** *// The two command lines containing the red* *// dynamicReturnAddress remain <u>un</u>-translated,* *// because the BASIC Emulator System can* *// automatically remember internally from* *// where a Function had been called by means* *// of the GOSUB command*
GOTO *M[SP]* *// Return to caller, as in **Chapter #9, Fig.9.3***	***LN*** **RETURN** *// <u>No</u> dynamic address is needed, because the* *// BASIC System can internally "remember"...* →

	// from where a Function had been called. The *//* RETURN *command will thus automatically* *// "jump back" to the correct address.*

As mentioned above there are some more technical details that are **not** described in this Document. Thus: **diligent Project Students will have to do some additional "research and tinkering" by themselves** in order to make their whole Compiler Software Package work correctly in the end.

ASSESSMENT

For assessing Phase **5b** of your Compiler, the tutors will *not only* check whether *some* BASIC code gets generated; the Tutors will *also run* the generated BASIC code in an Emulator in order to see whether the BASIC code faithfully represents the original RecSPL program from which the BASIC program was compiled. If the generated BASIC program "is doing nonsense" while running in the Emulator, then the tutors will know that your Compiler's code generation software was not correctly implemented.
