The following **translation advice** for *RecSPL* syntax is provided under the assumption that Scope Analsysis, Internal Renaming of user-defined names, and Type Checking were already successful in foregoing phases of the compilation process. It is thus assumed that a Symbol Table already exists, which can be consulted by the translator function (as explained in **Chapter #6** of our textbook).

GLOBVARS ::=
GLOBVARS_1 ::= VTYP VNAME **,** GLOBVARS_2

The variable declarations were needed only for Scope-Analysis and for Type-Checking. They remain un-translated (can be ignored by the translator).

VTYP ::= **num**
VTYP ::= **text**

The type declarations were needed only for Scope-Analysis and for Type-Checking. They remain un-translated (can be ignored by the translator).

VNAME ::= a token of **Token-Class V** from the Lexer

The user-defined names were already re-named in the foregoing Scope Analysis. The translator function can find their new names in the Symbol Table.

PROG ::= **main** GLOBVARS ALGO FUNCTIONS

*The source-word **main** remains un-translated (can be ignored by the translator).*
We translate ALGO, *and append behind the* ALGO-*code the translation of* FUNCTIONS.
*Also important is the generation of a **stop** command behind* ALGO, *such that the running main-code will not continue to run into the program code of the functions in target code within the same target-code-file!*
Thus:
translation(PROG) *must return the target-code-string **aCode**++"* **STOP** *"++**fCode***
*whereby **aCode = translation***(ALGO), *and **fCode = translation***(FUNCTIONS)

ALGO ::= **begin** INSTRUC **end**

*The source-words **begin** and **end** remain un-translated (can be ignored by the translator).*
*Thus: **translate***(ALGO) = ***translate***(INSTRUC)

INSTRUC ::=

For this case, the translator function shall return the target-code-string " **REM END** *"*
// Comment: In our Target-Language, REM represents a non-executable remark

INSTRUC_1 ::= COMMAND **;** INSTRUC_2

*Translate this sequence such as **Stat$_1$; Stat$_2$** in **Figure 6.5** of our Textbook.*

COMMAND ::= **skip**

For this case, the translator function returns the code-string " **REM DO NOTHING** *"*

COMMAND ::= **halt**

For this case, the translator function must return the code-string " **STOP** *"*

COMMAND ::= **print** ATOMIC

codeString = ***translate***(ATOMIC)
return("**PRINT**"++" "++***codeString***)

COMMAND ::= **return** ATOMIC *// **Only** for Project **Phase 5b**, NOT for Project Phase 5a!*

*// The **return ATOMIC** command must stand 'inside' of a Function-Scope!*
// We assume that Scope-Analysis has checked this already! If the return ATOMIC command
// was found inside the MAIN program then a semantic error must have already been thrown
// in the Semantic Analysis phase, such that the translation phase would not even start.
Advice: Translation as per Chapter #9 of our Textbook, or per INLINING (as lectured).

COMMAND ::= ASSIGN *translate*(COMMAND) = *translate*(ASSIGN)
COMMAND ::= CALL *translate*(COMMAND) = *translate*(CALL)
COMMAND ::= BRANCH *translate*(COMMAND) = *translate*(BRANCH)

ATOMIC ::= VNAME
***translate**(ATOMIC) → returns as code-string the **new name** of* VNAME *as found in the Symbol Table*

ATOMIC ::= CONST *translate*(ATOMIC) = *translate*(CONST)

CONST ::= a token of **Token-Class N** from the Lexer
CONST ::= a token of **Token-Class T** from the Lexer

Constants are translated to themselves.
*Example for a number constant: **translate**(**235**) → return " **235** "*
*Example for a **text** constant: **translate**(**"hello"**) → return " **"hello"** "*
*// Note that the returned code-string must also contain these **"**quotation marks**"** !*

ASSIGN ::= VNAME < **input** *// The symbol < remains un-translated*
codeString = ***translate***(VNAME)
return("**INPUT**"++" "++***codeString***)

ASSIGN ::= VNAME = TERM
*Translate this case such as **id := Exp** in **Figure 6.5** of our Textbook.*

TERM ::= ATOMIC *translate*(TERM) = *translate*(ATOMIC)
TERM ::= CALL *translate*(TERM) = *translate*(CALL)
TERM ::= OP *translate*(TERM) = *translate*(OP)

CALL ::= FNAME**(** ATOMIC$_1$ **,** ATOMIC$_2$ **,** ATOMIC$_3$ **)**

*The internally generated **new name** for FNAME can already be found in the **Symbol Table**.*

*For **non**-executable **intermediate** code (Semester-Project Phase 5a),*
*you can translate function calls such as case **id(Exps)** in **Figure 6.3** of our Textbook.*
***In our project**, the translation is indeed **much easier than in the Textbook**, because we know that we have exactly **three** parameters [not an indefinitely long list of parameters], and we also know that our parameters are **atomic** [not the Textbook's parameters which are possibly composite terms].*
In other words:
***translation**(* **FNAME(** ATOMIC$_1$ **,** ATOMIC$_2$ **,** ATOMIC$_3$ **)** *)*
returns simply the non-executable intermediate code-string:
"CALL_"++***newNameforFNAME***++"**(**"++***p1***++"**,**"++***p2***++"**,**"++***p3***++"**)**"
whereby
p1 = ***translation**(* ATOMIC$_1$ *)*
p2 = ***translation**(* ATOMIC$_2$ *)*
p3 = ***translation**(* ATOMIC$_3$ *)*
all of which are either simple variable names or simple constant numbers **:)**

*// For **executable target code for the Function-Call** (in Semester-Project Phase 5**b**),*
// the final translation will continue from there either by way of INLINING (as lectured),
*// or by the code generation method described in **Chapter #9** of our Textbook (with stack).*

OP ::=UNOP(ARG) — *Translate this case such as **unop Exp_1** in **Figure 6.3** of our Textbook, however **with** the brackets**!***
In other words:
return *$code_1$++place++"**:=**"++opName++"**(**"++$place_1$++"**)**"*

ARG ::= ATOMIC — ***translate***(ARG) = ***translate***(ATOMIC)
ARG ::= OP — ***translate***(ARG) = ***translate***(OP)

UNOP ::= **not**
Important! *Our Target-Language does not include in its own syntax any symbolic representation of the Boolean negation operator **not** ! Wherever such a **not** occurs in a* COMPOSIT COND *of any* BRANCH *statement, such a* BRANCH *statement must be translated as described in case **! $Cond_1$** of **Figure 6.8** of our Textbook whereby the **then**-code and the **else**-code of the if-then-else command are getting **swapped**.*

UNOP ::= **sqrt** — *// Numeric operation which yields a number's square root*
translate(**sqrt**) → return "**SQR**" *// That is the operator's syntax in our Target Language*

OP ::= BINOP(ARG_1 , ARG_2)
*Translate this case such as **Exp_1 binop Exp_2** in **Figure 6.3** of our Textbook.*

BINOP ::= **or**
Important! *Our Target-Language does not include in its own syntax any symbolic representation of the Boolean disjunction operator **or** ! Wherever such an **or** occurs in a* COMPOSIT COND *of any* BRANCH *statement, such a* BRANCH *statement must be translated such as described in case **$Cond_1$ || $Cond_2$** of **Figure 6.8** of our Textbook, whereby **cascading jumps to different labels** will be generated by the translator function.*

BINOP ::= **and**
Important! *Our Target-Language does not include in its own syntax any symbolic representation of the Boolean conjunction operator **and** ! Wherever such an **and** occurs in a* COMPOSIT COND *of any* BRANCH *statement, such a* BRANCH *statement must be translated such as described in case **$Cond_1$ && $Cond_2$** of **Figure 6.8** of our Textbook, whereby **cascading jumps to different labels** will be generated by the translator function.*

BINOP	::=	**eq**	***translate*(eq)** → return " = "
BINOP	::=	**grt**	***translate*(grt)** → return " > "
BINOP	::=	**add**	***translate*(add)** → return " + "
BINOP	::=	**sub**	***translate*(sub)** → return " – "
BINOP	::=	**mul**	***translate*(mul)** → return " * "
BINOP	::=	**div**	***translate*(div)** → return " / "

BRANCH ::= **if** COND **then** $ALGO_1$ **else** $ALGO_2$
If the COND *is a* COMPOSIT,
then translate the whole BRANCH *command as in **Figure 6.8** in the Textbook.*
If the COND *is* SIMPLE,
then translate the whole BRANCH *command as in **Figure 6.5** of the Textbook, case:* **if** COND **then** $Stat_1$ **else** $Stat_2$

COND ::= SIMPLE *Translation as explained above*
COND ::= COMPOSIT *Translation as explained above*

SIMPLE ::= BINOP(ATOMIC$_1$, ATOMIC$_2$) *Translation as explained above*

COMPOSIT ::= BINOP(SIMPLE$_1$, SIMPLE$_2$) *Translation as explained above*
COMPOSIT ::= UNOP (SIMPLE) *Translation as explained above*

FNAME ::= a token of **Token-Class F** from the Lexer
The user-defined names were already re-named in the foregoing Scope Analysis.
The translator function can find their new names in the Symbol Table.

Here ends the work-task for those students who only wish to carry out Phase 5a of the Project, i.e.: the generation of non-executable Intermediate-Code. For Phase 5a (only) you do not need to generate code for FUNCTIONS: these remain un-translated in Project Phase 5a. Generated code must be written out into a legible *.txt file, which our Tutors can read for assessment and marking.

Here begins the work-tasks for those students who also want to accomplish Project Phase 5b, in which executable target code shall ultimately be generated. For this purpose it is of course also necessary to generate target-code for the FUNCTIONS to which the Main-Program can make calls. Project Phase 5b can be fully accomplished as soon as Textbook Chapter #9 and Textbook Chapter #7 have been discussed in the lectures.

FUNCTIONS ::=
For this case, the translator function shall return the target-code-string " **REM END** *"*

FUNCTIONS$_1$ ::= DECL FUNCTIONS$_2$
We translate DECL, *and append behind the* DECL-*code the translation of* FUNCTIONS$_2$.
Also important is the generation of a ***stop*** *command behind* DECL, *such that the running* DECL *code will not continue to run into the program code of the subsequent functions in the same target-code-file!*
Thus we must return the target-code-string ***dCode***++" **STOP** "++***fCode***
where ***dCode = translation***(DECL), *and* ***fCode = translation***(FUNCTIONS$_2$)

DECL ::= HEADER BODY
The HEADER *will be treated either by the method of INLINING (as explained in lecture), or by the method explained in* ***Chapter #9*** *of our Textbook.* ***Ultimately the* HEADER *will vanish****, as it does not contain any do-able algorithm. Only the* BODY *contains a do-able algorithm, and thus only the* BODY *will eventually appear in the generated target code.*

HEADER ::= FTYP FNAME(VNAME$_1$, VNAME$_2$, VNAME$_3$)
The HEADER *will be treated either by the method of INLINING (as explained in lecture), or by the method explained in* ***Chapter #9*** *of our Textbook.* ***Ultimately the* HEADER *will vanish****, as explained above.*

FTYP ::= **num**
FTYP ::= **void**
The type declarations were needed only for Scope-Analysis and for Type-Checking.
They remain un-translated (can be ignored by the translator).

LOCVARS ::= VTYP$_1$ VNAME$_1$ **,** VTYP$_2$ VNAME$_2$ **,** VTYP$_3$ VNAME$_3$ **,**

The variable declarations were needed only for Scope-Analysis and for Type-Checking.
They remain un-translated (can be ignored by the translator).
As usual, their new names are kept in the Symbol Table.

BODY ::= PROLOG LOCVARS ALGO EPILOG SUBFUNCS **end**

translate(BODY) → *return the code-string* ***pCode++aCode++eCode++sCode***
whereby:

*pCode = translate(*PROLOG*)*
*aCode = translate(*ALGO*)*
*eCode = translate(*EPILOG*)*
*sCode = translate(*SUBFUNCS*)*

PROLOG ::= **{**

If the code-generation-method for its corresponding function is INLINING (as lectured),
then ***translate***(PROLOG) → *return* " ***REM BEGIN*** "

If the code-generation-method for its corresponding function is the method from ***Chapter #9***,
*then translate(*PROLOG*) will generate the boiler-plate-code (with runtime-Stack) as explained in Chapter #9.*

EPILOG ::= **}**

If the code-generation-method for its corresponding function is INLINING (as lectured),
then ***translate***(EPILOG) → *return* " ***REM END*** "

If the code-generation-method for its corresponding function is the method from ***Chapter #9***,
*then translate(*EPILOG*) will generate the boiler-plate-code (with runtime-Stack) as* ***explained in Chapter #9.***

SUBFUNCS ::= FUNCTIONS ***translate***(SUBFUNCS) = ***translate***(FUNCTIONS)

For any given RecSPL input program, your translator function's output –i.e.: the generated target code– shall be written into a *.txt file, which the Tutors can inspect for the purpose of assessment and marking. Thereby, please make sure that your output *.txt file also contains some meaningful LINE BREAKS such that your entire target code file does NOT appear as ONE enormously long line of cumbersome text, which our Tutors would not be able to read.
