
RecSPL is strongly and statically typed. Its type checking shall be implemented as a *recursive* Boolean procedure that analyses a given syntax tree (from the parser) by "tree crawling", and which

- returns **true** *if* the syntax tree (with all its branches) was correctly typed
- returns **false** *otherwise*.

It is thereby assumed that a suitable symbol table is already in place, and that the correctness of the name scopes (for variable names and function names) has already been verified *before* the type checker gets launched.

The type checker's **semantic rules** are *attributed* to their corresponding grammar rules as follows:

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

typecheck(PROG) = (*typecheck*(GLOBVARS) ^ *typecheck*(ALGO) ^ *typecheck*(FUNCTIONS))

GLOBVARS ::=

typecheck(GLOBVARS) = **true** *// base-case of the type-checking-recursion*

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

typeckeck($GLOBVARS_1$) = **{** let **T** := *typeof*(VTYP)

let **id** := *symboltable*(VNAME) *// access the existing symbol table!*

link (T,id) in the symbol table *// symbol table now knows the type!*

typeof(VNAME) = *typeof*(VTYP)

return *typecheck*($GLOBVARS_2$) **}**

VTYP ::= **num**

typeof(VTYP) = **'n'** *// the auxiliary function typeof returns a character that represents the type*

VTYP ::= **text**

typeof(VTYP) = **'t'** *// the auxiliary function typeof returns a character that represents the type*

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

// at this point we assume that the compiler's Scope Analyser has
// already entered some ID for this variable name into the above-
// mentioned symbol table, such that symboltable(VNAME) will
// yield use-able information about that node of the syntax tree.

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

typecheck(ALGO) = *typecheck*(INSTRUC)

INSTRUC ::=

typecheck(INSTRUC) = **true** *// base-case of the type-checking-recursion*

$INSTRUC_1$ ::= COMMAND **;** $INSTRUC_2$

typecheck($INSTRUC_1$) = (*typecheck*(COMMAND) ^ *typecheck*($INSTRUC_2$))

COMMAND ::= **skip**

typecheck(COMMAND) = **true** *// base-case of the type-checking-recursion*

COMMAND ::= **halt**

typecheck(COMMAND) = **true** *// base-case of the type-checking-recursion*

COMMAND ::= **print** ATOMIC
if *typeof*(ATOMIC)==**'n'** then *typecheck*(COMMAND) = **<u>true</u>**
if *typeof*(ATOMIC)==**'t'** then *typecheck*(COMMAND) = **<u>true</u>**
else *typecheck*(COMMAND) = **<u>false</u>**

COMMAND ::= **return** ATOMIC *// must stand 'inside' of a Function-Scope!*
typecheck(COMMAND) = { Let a **tree-crawler find** the **FTYP** node which belongs to the **same Function-Scope** inside of which also this COMMAND is standing inside the function's BODY.
*// We assume that **Scope Analysis was already done**!*
Let ***X*** be this 'matching' function-type-node in the tree.
if *typeof*(ATOMIC) == *typeof*(***X***) == '**n**' *// functions can return only **n***
then <u>return **true**</u>
else <u>return **false**</u> }

COMMAND ::= ASSIGN *typecheck*(COMMAND) = *typecheck*(ASSIGN)

COMMAND ::= CALL
if *typeof*(CALL) == **'v'** *// the void-type*
then *typecheck*(COMMAND) = **<u>true</u>**
else *typecheck*(COMMAND) = **<u>false</u>**

COMMAND ::= BRANCH *typecheck*(COMMAND) = *typecheck*(BRANCH)

ATOMIC ::= VNAME
typeof(ATOMIC) = *typeof*(VNAME) *// as per symbol table, which gets consulted at this point*

ATOMIC ::= CONST
typeof(ATOMIC) = *typeof*(CONST) *// as per symbol table, which gets consulted at this point*

CONST ::= <u>a token of **Token-Class N** from the Lexer</u>
typeof(CONST) = **'n'** *// this is obviously a base-case*

CONST ::= <u>a token of **Token-Class T** from the Lexer</u>
typeof(CONST) = **'t'** *// this is obviously a base-case*

ASSIGN ::= VNAME < **input** *// we only allow numeric user-inputs in RecSPL*
if *typeof*(VNAME) == **'n'**
then *typecheck*(ASSIGN) = **<u>true</u>**
else *typecheck*(ASSIGN) = **<u>false</u>**

ASSIGN ::= VNAME = TERM *// texts or numbers can be assigned to variables*
if *typeof*(VNAME) == *typeof*(TERM)
then *typecheck*(ASSIGN) = **<u>true</u>**
else *typecheck*(ASSIGN) = **<u>false</u>**

TERM ::= ATOMIC *typeof*(TERM) = *typeof*(ATOMIC)

TERM ::= CALL *typeof*(TERM) = *typeof*(CALL)

TERM ::= OP *typeof*(TERM) = *typeof*(OP)

CALL ::= FNAME(ATOMIC$_1$, ATOMIC$_2$, ATOMIC$_3$)
    if (*typeof*(ATOMIC$_1$) == **'n'** ^
        typeof(ATOMIC$_2$) == **'n'** ^
        typeof(ATOMIC$_3$) == **'n'**) *// all three parameters of the function must be numeric*
    then *typeof*(CALL) = *typeof*(FNAME) *// symbol table may be consulted at this point*
    else *typeof*(CALL) = '**u**' *//* ***u****ndefined which will cause the type-checker to return false*

OP ::=UNOP(ARG)
    if *typeof*(UNOP) == *typeof*(ARG) == '**b**' **then** *typeof*(OP) = '**b**' *//* ***b****ool-type*
    if *typeof*(UNOP) == *typeof*(ARG) == '**n**' **then** *typeof*(OP) = '**n**' *// numeric type*
    else *typeof*(OP) = '**u**' *//* ***u****ndefined which will cause the type-checker to return false*

ARG ::= ATOMIC *typeof*(ARG) = *typeof*(ATOMIC)

ARG ::= OP *typeof*(ARG) = *typeof*(OP)

UNOP ::= **not** *typeof*(UNOP) = '**b**' *// bool-type*

UNOP ::= **sqrt** *typeof(UNOP)* = '**n**' *// numeric type*

OP ::= BINOP(ARG$_1$, ARG$_2$)
if *typeof*(BINOP) == *typeof*(ARG$_1$) == *typeof*(ARG$_2$) == '**b**' **then** *typeof*(OP) = '**b**' *// bool-type*
if *typeof*(BINOP) == *typeof*(ARG$_1$) == *typeof*(ARG$_2$) == '**n**' **then** *typeof*(OP) = '**n**' *// numeric type*
if *typeof*(BINOP) == '**c**' *//* ***c****omparison-type, which "takes" numbers and "yields" a boolean result*
^ *typeof*(ARG$_1$) == *typeof*(ARG$_2$) == '**n**' **then** *typeof*(OP) = '**b**'
else *typeof*(OP) = '**u**' *// undefined which will cause the type-checker to return false*

BINOP	::=	**or**	*typeof*(BINOP) = '**b**'
BINOP	::=	**and**	*typeof*(BINOP) = '**b**'
BINOP	::=	**eq**	*typeof*(BINOP) = '**c**' *// comparison-type*
BINOP	::=	**grt**	*typeof*(BINOP) = '**c**' *// comparison-type*
BINOP	::=	**add**	*typeof*(BINOP) = '**n**'
BINOP	::=	**sub**	*typeof*(BINOP) = '**n**'
BINOP	::=	**mul**	*typeof*(BINOP) = '**n**'
BINOP	::=	**div**	*typeof*(BINOP) = '**n**'

BRANCH ::= **if** COND **then** ALGO$_1$ **else** ALGO$_2$
// Attention: do not confuse the syntactic if-then-else in the Syntax Tree
// with the semantic if-then-else of the recursive type analysis procedure!
if *typeof*(COND) == '**b**'
then *typecheck*(BRANCH) = (*typecheck*(ALGO$_1$) ^ *typecheck*(ALGO$_2$))
else *typecheck*(BRANCH) = **false**

COND ::= SIMPLE *typeof*(COND) = *typeof*(SIMPLE)
COND ::= COMPOSIT *typeof*(COND) = *typeof*(COMPOSIT)

SIMPLE ::= BINOP($ATOMIC_1$, $ATOMIC_2$)
if *typeof*(BINOP) == *typeof*($ATOMIC_1$) == *typeof*($ATOMIC_2$) == '**b**' **then** *typeof*(SIMPLE) = '**b**'
if *typeof*(BINOP) == '**c**' *// comparison-type*
^ *typeof*($ATOMIC_1$) == *typeof*($ATOMIC_2$) == '**n**' **then** *typeof*(SIMPLE) = '**b**'
else *typeof*(SIMPLE) = '**u**' *// undefined which will cause the type-checker to return false*

COMPOSIT ::= BINOP($SIMPLE_1$, $SIMPLE_2$)
if *typeof*(BINOP) == *typeof*($SIMPLE_1$) == *typeof*($SIMPLE_2$) == '**b**' **then** *typeof*(COMPOSIT) = '**b**'
else *typeof*(COMPOSIT) = '**u**' *// undefined which will cause the type-checker to return false*

COMPOSIT ::= UNOP (SIMPLE)
if *typeof*(UNOP) == *typeof*(SIMPLE) == '**b**' **then** *typeof*(COMPOSIT) = '**b**'
else *typeof*(COMPOSIT) = '**u**' *// undefined which will cause the type-checker to return false*

FNAME ::= a token of **Token-Class F** from the Lexer
// at this point we assume that the compiler's Scope Analyser has
// already entered some ID for this variable name into the above-
// mentioned symbol table, such that symboltable(FNAME) will
// yield use-able information about that node of the syntax tree.

FUNCTIONS ::=
typecheck(FUNCTIONS) = **true** *// base-case of the type-checking-recursion*

$FUNCTIONS_1$ ::= DECL $FUNCTIONS_2$
typecheck($FUNCTIONS_1$) = (*typecheck*(DECL) ^ *typecheck*($FUNCTIONS_2$))

DECL ::= HEADER BODY
typecheck(DECL) = (*typecheck*(HEADER) ^ *typecheck*(BODY))

*// **Attention!** This is exactly the "area" in the tree where the above-mentioned*
*// Tree-crawler must find the above-mentioned **return** ATOMIC command for*
// comparing its type against the function's return-type that is specified in the
// HEADER!

HEADER ::= FTYP FNAME($VNAME_1$, $VNAME_2$, $VNAME_3$)
*// **Attention!** This is exactly the "area" in the tree where the above-mentioned*
*// Tree-crawler must find the above-mentioned **return** ATOMIC command for*
// comparing its type against the function's return-type that is specified in the HEADER!

typeckeck(HEADER) = **{** let **T** := *typeof*(FTYP)
let **id** := *symboltable*(FNAME) *// access the existing symbol table!*
link (T,id) in the symbol table *// symbol table now knows the type!*
typeof(FNAME) = *typeof*(FTYP)

if *typeof*($VNAME_1$) ==
typeof($VNAME_2$) ==
typeof($VNAME_3$) == '**n**' *// RecSPL allows only numeric arguments*

then return **true**
else return **false**
}

FTYP ::= **num** *typeof*(FTYP) = '**n**' *// numeric return type*
FTYP ::= **void** *typeof*(FTYP) = '**v**' *// void, for return-less functions*

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

typecheck(BODY) = (*typecheck*(PROLOG)
^ *typecheck*(LOCVARS)
^ *typecheck*(ALGO)
^ *typecheck*(EPILOG)
^ *typecheck*(SUBFUNCS))

PROLOG ::= **{**

typecheck(PROLOG) = **true** *// base-case of the type-checking procedure*

EPILOG ::= **}**

typecheck(PROLOG) = **true** *// base-case of the type-checking procedure*

LOCVARS ::= VTYP_1 VNAME_1 **,** VTYP_2 VNAME_2 **,** VTYP_3 VNAME_3 **,**

typecheck(LOCVARS) = **{** let **T** := *typeof*(VTYP_1)
let **id** := *symboltable*(VNAME_1) *// access the existing symbol table!*
link (T,id) in the symbol table *// symbol table now knows the type!*
typeof(VNAME_1) = *typeof*(VTYP_1)

let **T** := *typeof*(VTYP_2)
let **id** := *symboltable*(VNAME_2) *// access the existing symbol table!*
link (T,id) in the symbol table *// symbol table now knows the type!*
typeof(VNAME_2) = *typeof*(VTYP_2)

let **T** := *typeof*(VTYP_3)
let **id** := *symboltable*(VNAME_3) *// access the existing symbol table!*
link (T,id) in the symbol table *// symbol table now knows the type!*
typeof(VNAME_3) = *typeof*(VTYP_3)

return **true** **}**

SUBFUNCS ::= FUNCTIONS

typecheck(SUBFUNCS) = *typecheck*(FUNCTIONS)
