# COS341, 1st Semester of 2022: Semester-Project Part B: Static Semantic Analysis

In **static semantic analysis**, everything else depends on whether two syntactic entities (AST nodes) are in the same scope or in different scopes. Therefore, scope analysis must precede all subsequent analyses.

To every node *N* (identified by its unique node ID) in the abstract syntax tree there must exist some corresponding scope information in the semantic table. This information determines unambiguously to which scope the node *N* belongs.

The scope information for each node is itself a triple **S=(*id/p/C*)** whereby

- ***id*** is a unique scope ID number
- ***p*** is a reference-pointer to this (***id***) scope's Parent-scope
- ***C*** is a finite **list [*c1, c2, ...* ]** of reference-pointers to this (***id***) scope's Children-scopes.

Thus, an **SPL** program's **scope-information** within the semantic table **is a tree** data-structure, too!

In our project's **SPL**, the scope-**id** of all AST nodes belonging to the **main** program is **0**, and the ***p*** information of the **main**'s scope is the empty "nullpointer" (no parent). The **main**'s Children-scopes are the scopes of the procedures that are (*possibly – if any*) declared at the start of an **SPL** program according to **SPL**'s grammar (provided previously in Project Part A). These procedures can possibly contain also their own "inner" procedure declarations (... and so on...) which have their own scopes, too. At the "deepest level", for those procedures which do not contain any further "inner" procedure declarations, their Children-scope lists are empty **[]**.

*Implementation advice:* use a suitable *auxiliary-method* to generate ever more ever new ***id*** numbers. *Cautioning:* If the scoping of the AST nodes is not correct then all subsequent analyses will fail too!

In the **SPL** semantics, two *syntactic* entities with *different* UserDefinedNames (see Project Part A) are always *semantically different* objects; (i.e.: *no "aliasing"* in **SPL**). In the case that two syntactic entities (nodes in the AST) carry the same UserDefinedName, however, they might *perhaps* be *the same* object semantically, or they might *perhaps* be *two different* objects semantically, which must be found out by further analysis.

A *procedure* with some UserDefinedName=**u** and, a *variable* with *the same* UserDefinedName=**u,** are *always* semantically *different objects:* This is a consequence of **SPL**'s grammar which makes a clear distinction between variables and procedures already at the syntactic level: For example, in the term **call X**, the entity **X** *cannot* be a variable, whereas in the term **add(X,Y)** the entity **X** *cannot* be a procedure – hence there is *no identification problem* at all in such cases.

Consequently, we *only* need to compare variables against variables –respectively procedures against procedures– when we want to find out whether two equally-named syntactic entities are the same (or different) semantic objects.

*Aim of the analysis:*

- If two *different syntactic* objects (AST nodes with *different node IDs*) are *semantically the same*, then they get *the same* new **Semantic ID** in the Semantic Table.
- If two *different syntactic* objects (AST nodes with *different node IDs*) are also s*emantically different* objects, then they also get *different* new **Semantic IDs** in the Semantic Table.

*Implementation advice:* use a suitable *auxiliary-method* to generate ever more ever new ***id*** numbers also for the Semantic Identifiers.

**Semantic Rules for the SPL Procedures (*depending on Scoping*)**

*Motivation.*

Whereas *auxiliary procedures* are typically called "downward" (as sub-procedures of a procedure), the well-known feature of *recursion* typically entails "upwards" calls or also self-calls on the same "level" of scoping. Since both "upwards" and "downwards" procedure calls are thus possible, we must **avoid same-name-confusion** *along Ancestor-Offspring lines of scoping;* otherwise we would not know whether some **call *f*** refers to an upwardly-declared ***f*** or to a downwardly-declared ***f***. For example: a recursive procedure that calls itself *cannot also* call an auxiliary sub-procedure of the same name – otherwise we would not know which is the recursive self-call and which is the call to the auxiliary sub-procedure. The semantic rules given below serve the purpose of avoiding such same-name-confusion.

- The unique "main" *cannot* be a UserDefinedName for any declared procedure in an **SPL** program – otherwise: *semantic error.*
- Let *S* be a scope opened by a procedure declaration with some UserDefinedName=**u**. Let *S1, S2, ..., Sn* be the Child-scopes in ***C*** of *S*. Then no procedure declaration in any of *S1, S2, ... Sn* may have the same name **u** – otherwise: *semantic error*.
- Let *S* be a scope opened by a procedure declaration. Let *S1, S2, ..., Sn* be the Child-scopes in ***C*** of *S*. Then no *two* procedure declaration in any of *S1, S2, ... Sn* may have the same UserDefinedName **u** – otherwise: *semantic error,* in other words: Procedure declarations in Sibling-scopes must have different names – otherwise a Parent would not know which Child procedure is being called by that name.

As a consequence of these naming rules, any two procedure declarations with the same name must be located in different un-related sub-branches of the AST, and are therefore semantically different entities. For example: Procedure ***f*** has to sub-procedures, ***g*** and ***h***, whereby ***g*** has a sub-procedure ***f*** and ***h*** has a sub-procedure ***f***, too: then these three ***f*** are semantically different entities, because also the following call-conventions must be adhered to:

- Any procedure can only call itself (recursion: procedure declaration is in the same scope as the call) or a sub-procedure that is declared in an *immediate* Child-scope – otherwise: *semantic error*.

Thus, in other words: in **SPL** *it is not allowed to make far-ranging procedure calls* to procedures that are defined somewhere in sibling-scopes, nor in cousin-scopes, nor in grand-parent scopes, nor in grand-children scopes, etc. That is also the reason why two semantically different procedures *can* have the same UserDefinedName if they are only "sufficiently far away from each other" in the AST of an **SPL** program.

Under these circumstances, we finally report *also* the following two kinds of *semantic errors:*

- Procedure call without existence of a matching procedure declaration (APPL-DECL error),
- Procedure declaration without existence of a matching procedure call (DECL-APPL error).

**Semantic Rules for the SPL Variables (*depending on Scoping*)**

- Definition *"ancestor":* Parent, or parent of parent, or parent of parent of parent, etc...
- Definition *"offspring":* Child, or child of child, or child of child of child, etc...

Like most "real" programming languages, SPL "knows" the semantic difference between "local" and "global" variables. This is the reason for the *possibility* that two semantically different variables carry the same UserDefinedName, if only the following additional rules are adhered to.

- The declaration of some variable ***v*** can *never* be in any "offspring"-scope of ***v***'s scope.

- The declaration of ***v*** can only be either in ***v***'s own scope, or in some "ancestor"-scope.
- Within any *one* scope, there *cannot be more than one declaration* for the same variable ***v*** – otherwise *semantic error: conflicting declaration.*
- If there is no declaration for ***v*** in its own scope, there must exist a matching declaration in any of ***v***'s "ancestor"-scopes.
- If there is a declaration for ***v*** in its own scope and another declaration for ***v*** in some "ancestor"-scope, then the declaration in ***v***'s own scope is the "matching" declaration: local variable!
- If there is no declaration for ***v*** in its own scope, and there are two declarations for ***v*** at two different hierarchy-levels in two different "ancestor"-scopes, then the "nearer" declaration is the "matching" declaration.

**For example:** Procedure ***f*** has a variable declaration ***x*** and a sub-procedure ***g***. Sub-procedure ***g*** has a variable declaration ***x*** and a sub-procedure ***h***. Sub-procedure ***h*** uses variable ***x*** without declaration in its own scope: Then the ***x*** in ***h***'s scope is the ***x*** declared in ***g***'s scope, whereas the ***x*** declared in ***f***'s scope is a semantically different entity. Let ***f*** also have yet another sub-procedure, ***p***, in which some ***x*** is used without declaration in ***p***'s own scope: Then the ***x*** used in ***p*** is semantically the same as the ***x*** declared in ***f***.

Under these circumstances, we finally report *also* the following two kinds of *semantic errors:*
- Variable usage without existence of any matching variable declaration (APPL-DECL error),
- Variable declaration without existence of any matching variable usage (DECL-APPL error).

All the same applies for **SPL** *Array*-declarations, too! Due to their different syntax (in the **SPL** grammar) it is not possible to confuse UserDefinedNames of *Arrays* with UserDefinedNames of "normal" variables. Thus we *can* have an *array* ***a*****[...]** *co-existing* with a *procedure* ***a*** and *variable* ***a***.

**SPL Type-Checking.**
*The type of a variable is the type of this variable's "matching" declaration,* and "in general" we type-check **SPL** programs in a way that is very similar to the way described in our COS341 text-book.

Some subtle differences (in comparison against our textbook) are explained as follows. In SPL we have the following **types:**
- **N** for numbers,
  - with the additional special *sub-type* **NN**, *// comment: not-negative*
- **B** for Booleans, with the additional two special sub-types of **B**:
  - **T** is the always-true sub-type, *// comment: will be used later for program-optimisation*
  - **F** is the always-false sub-type, *// comment: will be used later for program-optimisation*
- **S** for strings,
- **U** for "unknown",
- **M** for "mixed", which is "**N or S**", *// comment: only for print-screen commands.*

Initially, every syntactic entity had the **U**-type which gets replaced by proper types as type-checking progresses. If any **U**-type remains standing after the termination of the type-checker, then something is wrong and a type-error must be emitted.

Moreover:

For an **SPL** command of the kind **output := Expr**, the **Expr** must be of type **M**. In other words: we can print numbers or strings to the screen – however in **SPL** we *cannot* print Booleans to the screen.

For type-checking **Array**-expressions, the following rules are applicable:

- in userDefinedName**[Var]** expressions, **Var** must be of type **N**
- in userDefinedName**[Const]** expressions, **Const** must be of type **NN**

**SPL**'s **true** *constant* has the Boolean sub-type **T**, and **SPL**'s **false** *constant* has the Boolean sub-type **F**.

Negative-number-Tokens (identifiable by the minus – symbol) have type **N**; all other numbers have **N**'s sub-type **NN**.

*A sub-type stands in the "is-a" relation with its super-type,* such that a sub-type can always be used where its super-type is allowed to be used: keep this in mind when programming your type-checker.

A "speciality" of **SPL** is the *possibility of equality-comparing un-comparable entities,* the result of which will always be false: Thus, in cases of **BinOp → eq(Expr,Expr)**

- **BinOp** is of type **B** *if* the type of the 1st **Expr** is the same as the type of the 2nd **Expr**
- **BinOp** is of sub-type **F** *if* the type of the 1st **Expr** differs from the type of the 2nd **Expr**

**SPL**'s **input(Var)** command requires type **N** for **Var**: the user is not allowed to enter strings from the screen-console.

**SPL**'s **not(...), and(...), or(...)** are all of type **B**, whereas **SPL**'s **add(...), sub(...), mult(...)** are all of type **N**.

For the comparison operation **BinOp → larger(Expr,Expr)**, **BinOp** is of type **B** if and only if both two **Expr** are of type **N**, (which is *basically the same as in our COS341 textbook*).

**SPL Value-Flow Analysis.**
Last but not least we must also find out how the values are flowing through an **SPL** program in order to determine whether this **SPL** program is statically-semantically acceptable. For example, an assignment command-sequence such as

***x* := add(*y*,*z*)**
***r* := mult(*x*,*x*)**

"makes sense" only if both ***y*** and ***z*** already have values – otherwise no value can be assigned to ***x***, consequently no value to ***r*** either, such that a semantic *lack-of-value error* must be reported.

The method of value-flow analysis is very similar to the method of type-checking: also in this case an analysis-algorithm must "crawl" through the SPL program's AST in order to find everything out.

Note, however, that you cannot assume that a variable "keeps" being valued for all times after it had received some value in the first place! For example, in the SPL program-sequence

**...**
***x* := 1**
**...**
***x* := add(*a*,*b*)**
**...**

the variable ***x***, which initially had some value (1), *can later possibly lose* the property of having a value, namely in case that either ***a*** or ***b*** is value-less.

Therefore it makes sense, from a practical point of view, to search for valued-ness from a program's

end-line, and move the search 'backwards' towards the program's beginning-line. Thereby, special care must be taken of *Procedure Calls*, which *can possibly cause the update of Global Variables!* For example, in the **SPL** program

**...**
**call p**
**...**
***y* := mult(*x*,*x*)**
**...**

the variable ***x***, which is needed to "give value" to variable ***y***, might possibly have received a value as a *side-effect* of the execution of procedure **p**: this must not be forgotten in the semantic analysis.

Obviously we can never know in all cases *which* value a variable *specifically* has – otherwise we would be able to solve the unsolvable Halting Problem! All we can possibly know in this analysis is *that* a variable has *some* value, or *that* a variable has no value at all at a specific program-point.

For this purpose, if you look at the grammar of **SPL**, you see immediately that "value" is definitely "provided" wherever the following syntactic constructs occur in a program's AST:

- **Const**
- **input(Var)**

The occurrences of these syntactic constructs in the AST of a given **SPL** program are the "anchors" of value-flow-analysis, and from these "anchors" it can be analysed whether the property of "having a value" gets *propagated* through to other nodes of the AST as well.

Thereby, special attention must be paid to programs that contain *conditional paths that can possibly be skipped* at run-time! For example in the **SPL** program

**...**
**while (Expr) do { *x* := 1 }**
**if( (larger(*x*,0) ) then {...} else { *y* := 2 }**
***z* := add(*y*,*y*)**
**...**

variable ***x*** might perhaps never get a value (if **Expr** evaluates to *false* at run-time), and also variable ***y*** might perhaps never get a value (if **larger(*x*,0)** evaluates to *true* at run-time), as a consequence of which also ***z*** might perhaps never get any value either. Thus, to be sure that ***z*** will receive a value we must analyse *all* flow-paths that "lead to" **z** – not only one of them; otherwise a *possibly-no-value error* would have to be emitted by the semantic analysis algorithm.

Sometimes, however, we might perhaps even get some lucky assistance from the foregoing type-checking phase: For example if some Boolean expression ***E*** is of sub-type **F**, and this ***E*** is used as a branching-condition, then –because of the guaranteed falsity of ***E***– we know for sure which branch is going to be taken such that we can use this certain knowledge also for our value-flow-analysis.

---

**That's the specification of Part B** of your semester-project! If you have any questions about minor details, then please go to the Tutor's Lab-Hour on a Wednesday and *ask the Tutor for clarification about "how he would like to have it"*.

As previously announced, **all** project parts (A, B, C) are getting assessed within the final four weeks of the semester, and the exact submission-dates will be announced towards the end of month April.

**And now: HAPPY CODING :)**