Semantic Analysis

In this year's Semantics of the RecSPL, **everything is static** - i.e.: known at compilation time - i.e.: **static scoping**, as well as also **static typing**.

For this reason, a **simple Hash Table** - or a **simple Relational Database** - will suffice as Symbol Table; it can be "thrown away" after target code generation (and will thus not itself become part of the target code).

Since **your already-existing parser** has equiped each Tree Node with a unique Node ID, you will use these unique **IDs as "`Foreign Keys`"** in your Database or Hash Table to **"link" the Syntax Tree with the Symbol Table**.

A **Tree-Crawling-Algorithm** must be implemented that **"populates" the Symbol Table** with Semantic Information while "visiting" all the nodes of the Syntax Tree.

The **SEMANTIC RULES** for **Function Names** and for **Variable Names**, which your **Compiler's Semantic Analyser Component** must take into account, are given in the PDF `document attached below`.

More Tips

In our **semester project**, we want to give **system-internal unique new names** for user-defined variable-names and user-defined function-names **already in the scope-analysis-phase**, (NOT "mingled into" the later translation phase): That was the approach which I had recommended in the previous lecture, (in contrast to the approach presented by our textbook).

For this purpose, already **your scope-analyser module needs a sub-function that can generate ever-new identifiers** that were never used before.

Examples:

PROF

- Let there be two variables **X** in one scope (being the same entities to each other), and another two variables **X** in another scope (also being the same entities to each other but NOT to the afore-mentioned other two **X**), then your scope analyser would perhaps give the new names '**v136**' to the former two X, and perhaps '**v419**' to the latter two **X**.
- Let there be a **call** to some function **g**, which has a **g-definition** within the appropriate scope, then both the call to **g** as well as the definition of **g** itself could get an internal new name '**f561**' in its internal representation.

After the consistent re-naming of all user-defined names by unique internal new names, we need not worry about scope-borders any longer, as we can now simply treat each uniquely re-named variable AS IF IT WOULD BE a global variable 😀

And by giving all function-names unique new internal names, we will later be able to simply use a unique function-name as a unique GOTO-Label, to which we can "jump" without any ambiguity when we are "calling" a function 😀👍

More Tips For Semantic Analysis

- Since our already-existing parser has equiped each Tree Node with a unique Node ID, we could use these unique IDs as "Foreign Keys" in our Hash Table to "link" the Syntax Tree with the Symbol Table.
- We want to give system-internal unique new names for user-defined variable-names and user-defined function-names already in the scope-analysis-phase
    - Example 1: Let there be two variables X in one scope (being the same entities to each other), and another two variables X in another scope (also being the same entities to each other but NOT to the afore-mentioned other two X), then your scope analyser would perhaps give the new names 'v136' to the former two X, and perhaps 'v419' to the latter two X.
    - Example 2: Let there be a call to some function g, which has a g-definition within the appropriate scope, then both the call to g as well as the definition of g itself could get an internal new name 'f561' in its internal representation.

Rules Concerning Function Names

Example Code

```
main {
// new scope
call g() // in same scope as declaration
decl f() {
// new scope
call f() // recursion
call g() // in same scope as declaration
decl g()
decl h()
}
decl g() {
// new scope
call h() // in same scope as declaration
decl f()
decl h()
}
}
```

In this example codeL:

- The function g() declared in f() is not the same as the function g() in main although they have the same name.
- The same rule applies to the function f() in main as well as the function f() declared in g(), as well as the function h() in f() as well as the function h() declared in g() are not the same since they are in different scopes.

Rules

- The main program forms the highest level scope, with no "parent".
- Every function declaration opens it's own scope.
- A child scope may not have the same function name as it's immediate parent scope.
- A child scope may not have the same name as any of it's sibling scopes under the same parent.
- A call command may refer to an immediate child-scope.
- A call command may refer to it's own scope: That is RECURSION.
- There may be no recursive call to main function!.
- The compiler's SEMANTIC ANALYSIS MODULE must throw an Error Report, if any of the semantic rules of above are violated!.

Rules Concerning Variable Names

Example Code

```
{
// new scope
decl x
decl y
use y
use z // Error, Z is Undefined
{
// new scope
decl y
use x // from parent scope
use y // from the same scope
{
// new scope
decl x
decl z
use y // from parent scope
use x // from the same scope
use z // from the same scope
}
}
}
```

Rules

- No variable name may be double-declared (twice) in the same scope.
    - Eg. No string X and also number X in the same scope.
- The declaration of a used variable name must be found either within that name's own scope or in any higher ancestor scope.
- If a used variable name has two declarations in twi different scopes, then the "nearest" declaration is the relevant declaration for that variable.
- Every used variable name must have a declaration.

- No variable anywhere in the program may have a name that is also used as a Function name anywhere in the program.
- No variable name anywhere in the program may be identical with any reserved keyword.
- Two variables with the same name are different computational entities if they are rooted in different scopes.
- The compiler's SEMANTIC ANALYTICS MODULE must throw an Error Report, if any of the semantic rules of above are violated!.