Compilers tutorial V: Semantic analysis

Semantic analysis is a crucial phase in the compilation process where the meaning and validity of a program are examined. It involves a series of checks and transformations to ensure that the input program adheres to the rules and constraints of the programming language. Two of the most important goals are: *scope analysis* and *type checking*.

A bit of theory: The usual semantic analysis algorithm performs a depth-first traversal of the AST (abstract syntax tree). It performs tasks such as annotating the tree with semantic information, type checking, symbol table construction, scope resolution, and consistency verification. It ensures that variables and functions are declared before use, enforces type compatibility, detects and reports errors related to mismatched types or invalid operations. Semantic analysis ensures the correctness of the program before proceeding to code generation.

Scope analysis

A *symbol table* is constructed by the compiler to store information about about identifiers (e.g., variables, functions) and their attributes. It tracks the scope, data types, and other properties associated with each identifier.

```
struct symbol_list {
    char *identifier;
    enum type type;
    struct node *node;
    struct symbol_list *next;
};
```

The `symbol_list` structure represents a symbol table as a linked list where each entry contains: the name of the *identifier*, its *data type*, and a pointer to the *AST node* in which it is declared. There are two data types and a special case:

```
enum type {integer_type, double_type, no_type};
```

To handle a symbol table we only need two operations: *inserting* a new symbol and *looking up* a symbol by its name. Two functions provide that functionality:

```
insert_symbol(symbol_table, identifier, type, node);
search_symbol(symbol_table, identifier);
```

The first function inserts a symbol in the table, unless it is already there. The second function looks up a symbol by its identifier name, returning `NULL` if not found. The functions are provided in the `semantics.h` and `semantics.c` files.

Type checking

The compiler performs *type checking* by traversing the AST, using the symbol table to check if variables are declared before use, to enforce type compatibility

rules (e.g., addition of numbers but not strings), to check function calls for argument types, and to validate language-specific rules regarding data types.

AST nodes must be annotated with their data type. Nodes representing *expressions* will be annotated with `integer_type` or `double_type`, while other nodes will be annotated with `no_type`. Therefore, the AST node data structure must now include the `type` field:

```
struct node {
    enum category category;
    char *token;
    enum type type;
    struct node_list *children;
};
```

The semantic analysis algorithm

Semantic analysis is expressed through a recursive traversal of the AST. Symbol table construction typically occurs during the *descending* phase, while type calculations and checking usually take place during the *ascending* phase. However, the order and tasks may vary depending on the specific design of a compiler.

Function `check_program` begins semantic analysis by checking the AST's root:

```
struct symbol_list *symbol_table;

int check_program(struct node *program) {
    symbol_table =
        (struct symbol_list *) malloc(sizeof(struct symbol_list));
    symbol_table->next = NULL;
    struct node_list *child = program->children;
    while((child = child->next) != NULL)
        check_function(child->node);
    return semantic_errors;
}
```

We declare a global symbol table and initialize it in the `check_program` function. To check a `Program` node the compiler recursively checks its children, which are the `Function` nodes. It does so for each child by calling `check_function`:

```
void check_function(struct node *function) {
    struct node *id = getchild(function, 0);
    if(search_symbol(symbol_table, id->token) == NULL) {
        insert_symbol(symbol_table, id->token, no_type, function);
    } else {
        printf("Identifier %s already declared\n", id->token);
        semantic_errors++;
    }
}
```

To check a `Function` node, `check_function` must ensure that its identifier hasn't been declared previously. Notice that we have a single global symbol table (for now). If the function's identifier (`id->token`) can't be found in the symbol table, then it is inserted there. Otherwise, a semantic error is printed.

Semantic analysis should next proceed to check the *parameters* of the function and the *expression* which is forms the body of the function. To maintain our naming convention, those two checks should be called `check_parameters` and `check_expression`. The exercises below challenge you to continue in this direction by writing those two checks.

Exercises

We begin with a simplifying assumption: any identifier may only be used once, globally. This restriction allows us to use *a single symbol table for all identifiers*. As a consequence, functions and parameters cannot share the same name (not even parameters from different functions). The last exercise contributes to removing this restriction.

1. Modify the code to check function *parameters*. Ensure that a semantic error is displayed when a parameter's identifier has already been declared, using the same error message as for functions.

2. Enhance the code to provide the line and column numbers where semantic errors are detected. To achieve this, you need to store the line and column of lexemes (tokens) in the AST. Two effective approaches for this are:

   - Modify the lexical analyzer to pass a new `struct` that includes the `char *token` along with its corresponding line and column.
   - Enable `%locations` in *yacc*, update `yylloc.first_line` and `yylloc.first_column` for each token processed by the lexical analyzer, and utilize, for instance, `@2.first_line` and `@2.first_column` to reference the location of `$2` in a semantic action.

3. Revise the code to check function *calls*. Ensure that the identifier used is the name of an existing function and that the number of arguments in the call matches the number of parameters of the function. Otherwise, show appropriate error messages.

4. Annotate *expression* nodes with their data type by computing the `type` field during the ascending phase of semantic analysis. Begin the process by annotating the leaves falling into the categories `Natural` (`integer_type`), `Decimal` (`double_type`), and `Identifier` (retrieved from the symbol table). Calculate the type of operations based on a simple rule: if the result of the operation involves a double, the type is set to `double_type`; otherwise, if it involves only integers, the type becomes `integer_type`. When displaying the content of the AST, incorporate the type information for each *expression* node (e.g., `Identifier(d) - double`).

5. Revise the code to check for undeclared identifiers. In expressions, variables are restricted to the parameters of the respective function. To implement this, we shall declare a local scope for each function, meaning a dedicated symbol table for each function. One approach is to create a new local `struct symbol_list *scope` within `check_function` and pass this scope down to both `check_parameters` and `check_expression`. Subsequently, `check_parameters` adds the parameters to the local scope, while `check_expression` retrieves identifiers from the local scope.

Author

Raul Barbosa (University of Coimbra)

References

Aho, A. V. (2006). Compilers: Principles, techniques and tools, 2nd edition. Pearson Education.

Barbosa, R. (2023). Petit programming language and compiler. https://github.com/rbbarbosa/Petit

Free Software Foundation (2021). GNU Bison Manual, Tracking Locations. https://www.gnu.org/software/bison/manual/html_node/Locations.html