Theory of Compilation (61304)
Assignment 3 – Semantic Analysis

Submission date: 1.7.2020

The main goal here is to extend the software that already implements lexical and syntax analysis (developed in Assignment 2), and to add a new capability – semantic analysis.

Main issues to be covered here are:

- Type checking
Here it is checked that each object in the compiled program is used in accordance to its definition.

- Scope analysis
The grammar rule that introduces the possibility of multiple nested scopes and definition of local variables is:

COMP_STMT → { VAR_DEC_LIST STMT_LIST }

Also, global variables and functions are considered to be defined in the program (top level) block/scope. The program block is defined by the rule

PROG → GLOBAL_VARS FUNC_PREDEFS FUNC_FULL_DEFS

Semantic actions to be added to the grammar to achieve these goals will need to work with semantic attributes. Implementation should properly address this issue.

Your code should be properly documented (provide comments that explain what the code is supposed to do).

While working on this part of the project, you can fix the problems remained from the previous stages (if any).

Semantic rules for the project language

Whenever an ID is defined in the compiled program, all attributes related to its definition are collected and stored in the symbol table.

When an ID is used in commands and expressions, a check should be performed to ensure that the way it is used fits its definition.

The following semantic rules should be fulfilled:

Definitions and uses of objects

- All used IDs must be declared
- It is allowed that in different scopes variables with same name are declared (local declarations).
- Duplicated declaration of the same name within same scope is forbidden.
  In particular, for functions:
  - When there exist both pre-declaration of a function and its full declaration – this is **not** considered as a case of duplicate declaration
  - Names of all parameters of a function should be different
- Parameter can have same name as a local variable declared in the function's body
- Every declared variable and parameter must be used at least once bonus 2 points

Restriction in assignments

In the following cases, assignment to an ID is forbidden:

- the ID is declared as array (but assignments to array elements are allowed)
- the ID is declared as function

Restrictions related to use of arrays in expressions

- expressions can't refer to entire array, but use of array elements in expressions is allowed
- in variable of the form id[expr_1, … , expr_n]
  - the id must be declared as array
  - n should be equal to the amount of dimensions in the array
  - the type of expr_i must be integer for every $1 \leq i \leq n$
  - if expr_i is a token of kind int_num, then its value should not exceed the size of i-th dimension of the array

Type of expression values

- the type of int_num is integer and of float_num is float
- the type of id that is a variable name is determined as either integer or float once it is declared
- the type of array element is determined in accordance with the array's declaration
- if at least one of the elements in the expression is undefined, then the type of the expression is undefined (error_type)
- if all elements of the expression are defined as integer, then the expression's type is integer

- if all elements of the expression are defined, and at least one is float, then the expression's type is also float
- for type of a function call (returned values), see below.

Type consistency in assignments

- either left and right sides are both of the same type
- or the left side is float and the right side is integer (the opposite is forbidden)

Rules related to functions

- consistency between pre-definition and full definition of a function:
  - if the compiled program contain a function pre-definition, then it should also contain its full definition
  - the amount and types of parameters in pre-definition of a function should be identical to given in its full definition bonus 3 points
- consistency between function definition and calls to this function:
  - the amount of parameters in a function call should match the function's definition
  - the types of parameters in a function call should match the function's definition bonus 3 points
- returned values:
  - a return statement residing in a void function or in the main/program block may not return a value bonus 4 points
  - a function whose returned type is not void, must contain at least one return statement with a returned value bonus 4 points
  - the type of the expression in a return statement must match the type of function's returned value bonus 4 points

TASKS

1. Implementation of symbol table:
   - For each scope there is a separate symbol table; these tables are connected to reflect the hierarchy of scopes
   - For each ID in a table, at least the following information should be stored:
     - For variable:
       name
       role (variable)
       type (integer, float)
       list of sizes for each of the dimensions (in case the variable is defined as array)
     - For function:
       name
       role (pre-definition / full definition)
       type (integer, float) of the returned value
       list of parameter types

2. Selection of a data structure for symbol table:
   - Symbol table is not necessarily implemented as an array (especially because amount of objects declared in the compiled program is not known in advance).
   - The simplest structure for a symbol table is a list of elements stored in it. However, it doesn't allow for efficient search by ID's name.
   - Better structures are:
     - binary search tree (each node holds an element)
     - hash table – this is the best for efficiency; there are ready-to-use packages on Internet (or you can develop it yourself)

3. Interface functions implementation
   Implement all functions needed for the work with symbol table. The list of functions is presented (including some implementation details) in
   מצגת הרצאה #9

4. Define and implement a syntax-directed scheme (combining the grammar rules and semantic actions) that allows to perform type checking and scope checking. This requires:
   - definition and classification of the needed attributes (synthesized, inherited)
   - definition of semantic actions that use calls to symbol table functions for implementation of type and scope checking (see at the end of מצגת הרצאה #9)
   - integration of the semantic actions in functions of the parser; see מצגת הרצאה #8

5. Error handling:
   - Each time a semantic error is discovered, the program should send an appropriate error message that clearly explains
     - what is wrong
     - where the error occurred (line number)
   - No error recovery is done when a semantic error occurs, and compiler should just continue its regular action.