Semantic Analyzer Implementation

컴퓨터소프트웨어학부 2021041385 이예진

Specification

Compiler Design Project 3

: implement a C-Minus semantic analyzer

1. Use source code from C-Minus with Lex, Yacc
2. Find semantic errors by implementing symbol table and type checker
    a. Traverse over the AST to generate a symbol table
    b. Traverse over the AST again and use the symbol table to perform type checking

C-Minus Semantics

1. Built-in Functions
    a. Defined by default, should be always accessible

       Assume that these functions are declared at line 0

    b. `int input(void)` returns a value of the given integer value from the user
    c. `void output(int value)` prints a value of the given argument
2. Scope Analysis
    a. Declare the functions before usage
    b. Undeclared/redefined Variables and Functions

       Assume that the variable(function) is implicitly declared after there is an undeclared variable usage(function call) `undetermined`

       No function overloading (don't check parameters for redefinition)
3. Type Checking: post processing of AST
    a. Cannot declare a void-type variable
    b. Only integer variables are compatible with arithmetic and logical operations (Not allowed: int[] + int[], int[] + int, void + void, ...)

       No need to check if the value is assigned to the variable

    c. Assignment type
        i. No type conversion
        ii. Allowed to assign int [] to int [] (size does not matter)
    d. if / while: only allowed to use int value for condition

    e. Function arguments / parameters: the number and types of the arguments and parameters should match
    f. Return type
    g. Array indexing check
        i. only int value can be used as an index
        ii. No bound checking

4. Modify the Line Number
    a. Function / Variable Declaration: follow the identifier line number
    b. Expressions (including assignments): the line number should be set according to the starting line
    c. Statement: the line number should be set according to the ending line
5. Variables follow scope of each compound statement
6. Error Output Formats follow the given `error_messages.c` file.
    a. should print all errors in the code
    b. check the line number carefully

Compilation Environment

Windows 11

Ubuntu 22.04.3 LTS

GNU/Linux 5.15.153.1-microsoft-standard-WSL2 ×86_64

Implementation

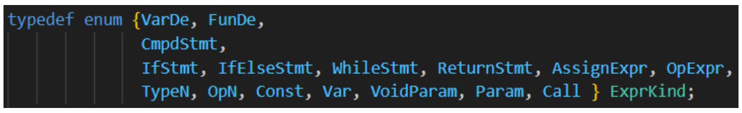
(`FILE` `FUNCTION` `EXPRESSION`)

FILE `main.c`

- Set `NO_ANALYZE`, `TraceParse` to FALSE, `NO_CODE` to TRUE
- Set `TraceAnalyze` to TRUE to print a symbol table, but FALSE for submission

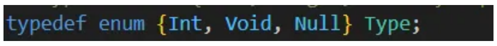
FILE `globals.h`

- Rename some expression kinds, `Expr` to `Stmt`. Corresponding changes are made in `util.c` and `cminus.y`.



```
typedef enum {VarDe, FunDe,
              CmpdStmt,
              IfStmt, IfElseStmt, WhileStmt, ReturnStmt, AssignExpr, OpExpr,
              TypeN, OpN, Const, Var, VoidParam, Param, Call } ExprKind;
```

- Add `Null` as a new member of the `Type` enumeration



```
typedef enum {Int, Void, Null} Type;
```

FILE `cminus.y`

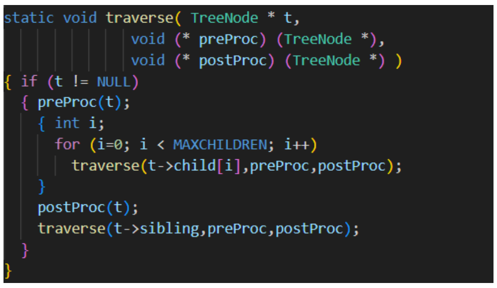
- Initialize `Const` nodes to have the type `Int`
- Modify `lineno` to align with the given specification

FILE `util.c`

- Modify the function `newTreeNode` to initialize tree nodes with the `Null` type, indicating `undetermined`

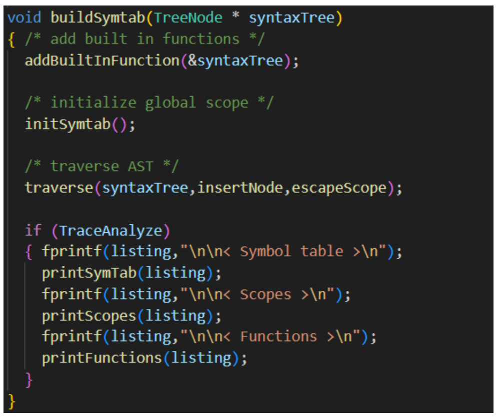
FILE `analyze.c`

- It has two primary functions, `buildSymtab` and `typeCheck`, invoked in `main.c`, to build a symbol table and to perform type checking by traversing the AST(Abstract Syntax Tree). It has `preProc` which is executed before it traverses to its children and `postProc` which is executed after its children and before its siblings.



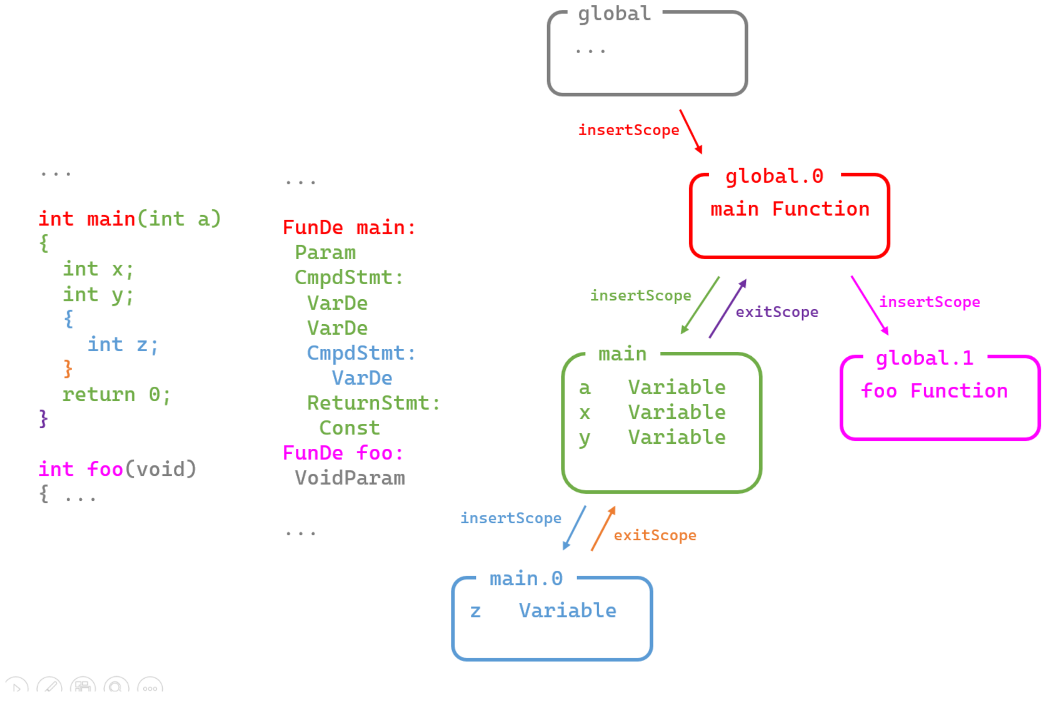
```
static void traverse( TreeNode * t,
               void (* preProc) (TreeNode *),
               void (* postProc) (TreeNode *) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc,postProc);
    }
    postProc(t);
    traverse(t->sibling,preProc,postProc);
  }
}
```

- Function `buildSymtab` generates the symbol table while traversing the AST. It includes `insertNode` as a pre-processing and `escapeScope` as a post-processing procedure. `addBuildInFunction` is first called to add given predefined functions to the `syntaxTree`. However, this will not change the original `syntaxTree`, so it should be called once again in `typeCheck`. Then we initialize the symbol table and the top `globalScope` by calling `initSymtab`. Then we traverse the AST and if `TraceAnalyze` is set to TRUE, it will also print out all information of the symbol table.



```
void buildSymtab(TreeNode * syntaxTree)
{ /* add built in functions */
  addBuiltInFunction(&syntaxTree);

  /* initialize global scope */
  initSymtab();

  /* traverse AST */
  traverse(syntaxTree,insertNode,escapeScope);

  if (TraceAnalyze)
  { fprintf(listing,"\n\n< Symbol table >\n");
    printSymTab(listing);
    fprintf(listing,"\n\n< Scopes >\n");
    printScopes(listing);
    fprintf(listing,"\n\n< Functions >\n");
    printFunctions(listing);
  }
}
```

- Pre-processing `insertNode` adds nodes of the syntax tree into the symbol table, such as variable declarations(`VarDe`), function declarations(`FunDe`), parameters(`Param` or `VoidParam`) by function `addNode`. Scopes for function declarations and compound statements are also managed here.
- In the case of `FunDe`, a scope is inserted by `insertScope` before adding an function symbol to the table to implement a fake scope for the c-minus semantic rules that functions should be declared before their usage. Additional scope for parameters and local variables are then created. The return type of the function is assigned directly by `assignType` to the syntax tree to avoid duplicate insertion of scope by `CmpdStmt`. This is because parameters and local variables of a function should be in the same scope.



```
...

int main(int a)
{
  int x;
  int y;
  {
    int z;
  }
  return 0;
}

int foo(void)
{ ...
```

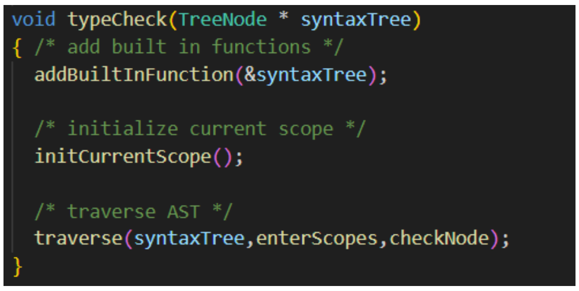
```
...

FunDe main:
 Param
 CmpdStmt:
  VarDe
  VarDe
  CmpdStmt:
    VarDe
  ReturnStmt:
   Const
FunDe foo:
 VoidParam

...
```

insertScope(enterScope) and exitScope(popScope)

  Also, `functionName` and `paramLoc` are modified to let children, which would be `Param` or `VoidParam`, know what their function name was. This can be implemented in a different way, which you can find in function call case (`Call`) in `checkNode`.

- Post-processing `escapeScope` utilizes `exitScope` from `symtab.c` when the function declaration or compound statement is over. Exclusively, fake scopes are never escaped to preserve the order of function declarations.

- Type checking is done in the post-processing because the type inferences are done by traversing the AST upward. `typeCheck` performs type checking during the AST traversal and it also has both pre-processing and post-processing functions, each is `enterScopes` and `checkNode`. Before the traversal, it also calls `addBuildInFunction` to modify the AST as above in `buildSymtab` and `initCurrentScope` to initialize `currentScope` as `globalScope`.



```c
void typeCheck(TreeNode * syntaxTree)
{ /* add built in functions */
  addBuiltInFunction(&syntaxTree);

  /* initialize current scope */
  initCurrentScope();

  /* traverse AST */
  traverse(syntaxTree,enterScopes,checkNode);
}
```

  - `enterScope` from `symtab.c` is called corresponding to the call sequence of `insertScope` in `insertNode` above because the scope is structured as a list. The list is ordered by the sequence of insertion, and it should be entered in the same sequence also in pre-processing of the second traversal.

    The process verifies whether the function is redefined by checking if it is has already been defined in any scopes above. Additionally, for functions with a return type of `Int`, a flag is set to ensure the presence of an appropriate return statement. These checks are performed during pre-processing since the validation of return statements occurs at the conclusion of compound statements, which are nested within the `parent` function declaration. Therefore, variable redefinition checks can be deferred to the post-processing phase on the other hand.

  - `checkNode` represents a critical feature of this project as it performs type checking for various constructs, including `Var`, `VarDe`, `Call`, `OpExpr`, `AssignExpr`, `IfStmt`, `IfElseStmt`, `WhileStmt`, `ReturnStmt`, and `CmpdStmt`. The type inference rules are implemented based on the provided specifications, and error messages for semantic errors are also given via `error_messages.c`.

    `checkNode` determines the types of previously declared identifiers using `findType`, which traverses the symbol table, from `symtab.c`. It also validates array indexing, operations, assignments, arguments type, return type of functions and conditions in selection statements. Afterward, it assigns type information to the corresponding nodes in the syntax tree. If a node's type is invalid or undeclared, then it is implicitly assigned the type `undetermined`.

    Argument validation for `Call` nodes is done by inspecting the types of child nodes, which differs from parameter-adding methodology above in `buildSymtab`.

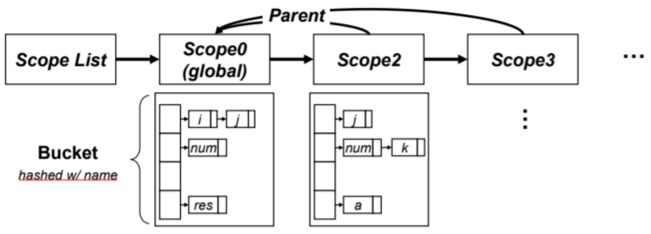
    In cases where a return statement is present, the `intFunctionLineno` is set to -1, to suppress missing return statement errors. Any issues with the return type being invalid for an `Int` function will be handled subsequently in the code.

    Similar to the `exitScopes` in the post-processing phase of `buildSymtab`, `popScope` is invoked to exit scopes properly. To ensure correct traversal using the `next` pointer, exited scopes are removed from the scope list and are no longer accessible.

- The corresponding headre file `analyze.h` are not changed. Every procedure in `main.c` is performed regardless of functions implemented here.

FILE `symtab.c`

- The functions in this file support `analyze.c` by facilitating the construction of the symbol table and efficient type checking. Additionally, it modularizes its role of handling scopes and their nodes' information, ensuring these elements are accessible in `analyze.c` only through the corresponding functions in `symtab.c`.
- A new struct `Scope`, which is missing feature in Tiny language, has been introduced to address the scope management , along with `ParamType` which tracks the types and order of function parameters and arguments. The struct `Bucket` has also been slightly modified by removing unused fields such as `memloc` and adding `params` and `scope`.



structure of scope

  Scope has a hierarchical structure, where new scopes are created as children of the current scope within compound statements or as children of the global scope in function declarations. Despite this hierarchy, `Scope` is implemented as a linked list, with two pointers, `parent` and `next` to make upward traversal and sequential access of scope during the second traversal (type checking). This design integrates seamlessly with the logic in `analyze.c`. Two global variables, `globalScope` (the head of the scope list) and `currentScope`, are maintained to manage scope traversal.

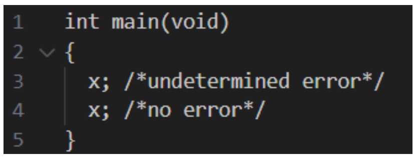
- Additional utility functions include `findInSymbolTable` and `findInScope`, which perform symbol lookups within a scope's hash table. `findInScope` is the same as checkScope() introduced in lecture notes. `findInSymbolTable` function can also traverse upward through parent scopes.
- Although not required in the specifications, printing functions for both functions and scopes have been implemented to enhance debugging and visualization.
- The code is thoroughly documented with inline comments.
- The corresponding header file `symtab.h` is also updated accordingly.

Execution

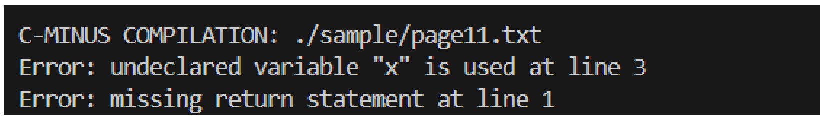
```
apt-get install bison
make cminus_semantic // compile
./cminus_semantic test.cm // execute
```

Result

- `page11.txt`

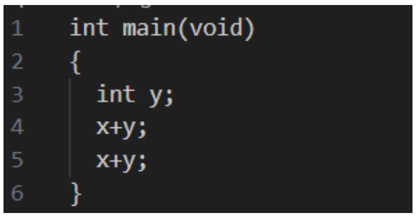


```
int main(void)
{
    x; /*undetermined error*/
    x; /*no error*/
}
```

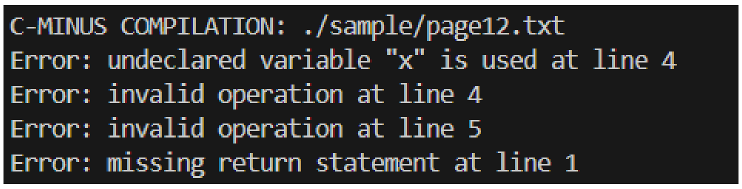


```
C-MINUS COMPILATION: ./sample/page11.txt
Error: undeclared variable "x" is used at line 3
Error: missing return statement at line 1
```

- `page12.txt`

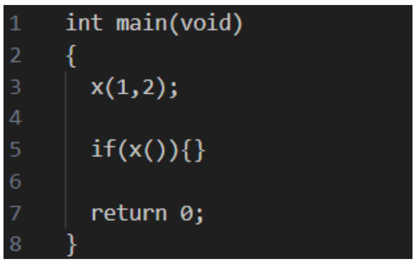


```
int main(void)
{
    int y;
    x+y;
    x+y;
}
```

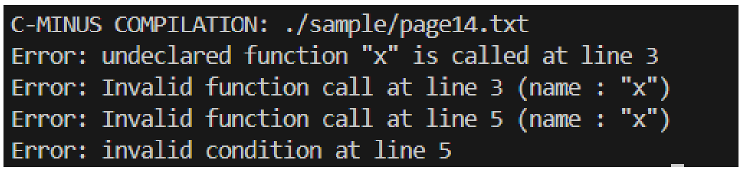


```
C-MINUS COMPILATION: ./sample/page12.txt
Error: undeclared variable "x" is used at line 4
Error: invalid operation at line 4
Error: invalid operation at line 5
Error: missing return statement at line 1
```

- `page14.txt`

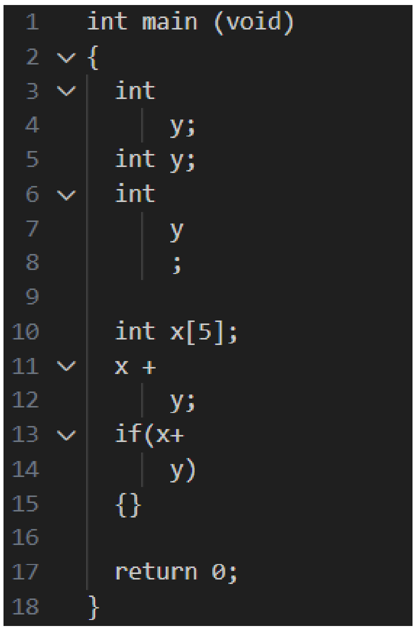


```
int main(void)
{
    x(1,2);

    if(x()){}

    return 0;
}
```

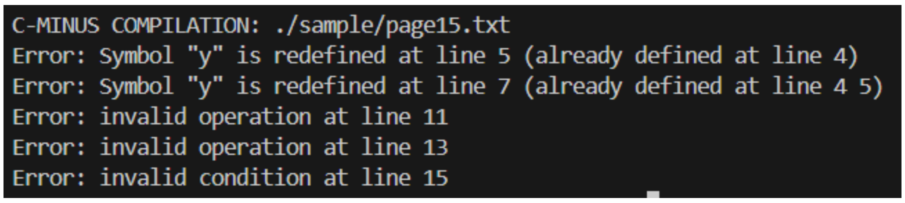


```
C-MINUS COMPILATION: ./sample/page14.txt
Error: undeclared function "x" is called at line 3
Error: Invalid function call at line 3 (name : "x")
Error: Invalid function call at line 5 (name : "x")
Error: invalid condition at line 5
```

- `page15.txt`



```
int main (void)
{
    int
        y;
    int y;
    int
        y
        ;

    int x[5];
    x +
        y;
    if(x+
        y)
    {}

    return 0;
}
```



```
C-MINUS COMPILATION: ./sample/page15.txt
Error: Symbol "y" is redefined at line 5 (already defined at line 4)
Error: Symbol "y" is redefined at line 7 (already defined at line 4 5)
Error: invalid operation at line 11
Error: invalid operation at line 13
Error: invalid condition at line 15
```

Reference

-