Report

11910501 鲍志远(A) 11910934 吴泽敏(A) 11810925 周翊澄(A)

1. Overview

In this project, we implement semantic analysis based on the syntax tree generated by the first project. During the traverse of syntax tree, we insert symbol nodes into symbol table and carry on semantic checking. We support both scope checking and structural type equivalence. A symbol table stack is used to support scope checking, and a type system is built to support structural equivalence.

2. Symbol table

2.1 Data structure

Our symbol table is based on doubly linked list. Every node in linked list is a two-tuple, containing `key` for the variable's identifier and `value` for the variable's type.

2.2 When to insert node into symbol table

During the traverse of syntax tree, whenever we meet the node represents nonterminal `ExtDef` or `Def`, we grab the information contained in its children to construct an node and insert the node into symbol table.

There is something tricky about `struct`. When we meet a structure specifier, we will also insert the structure definition into symbol table for the following declaration to use. However, we cannot use the row `id` as the key, since the variable's name may be the same as the `id`. Therefore, we add a prefix `struct` to `id`. Then the `id` becomes `struct id`. The space is necessary, since variable's name cannot contain space, and this insures that the variable's name won't conflict with the `id`.

2.3 Scope check

A syntax table stack is maintained during the traverse of syntax tree to support scope checking. Every scope has its own syntax table. When we enter an scope, we will create a new symbol table and put it on the stack top. When we leave an scope, the symbol table at the top will be popped. In this case, the check of redeclaration only checks the current symbol table, and the check of undefined variable checks the whole symbol table stack.

3. Type System

3.1 Supported Type

- int
- float
- char
- array
- struct
- function

3.2 Data Structures

```
struct Type{
  enum Category category;
  union {
    enum Primitive primitive;
    struct ArrayType *array;
    struct FieldType *structure;
    struct FunctionType *function;
  };
};

struct ArrayType {
  int size;
  struct Type *type;
};

struct FieldType {
  char *name;
  struct Type *type;
  struct FieldType *next;
};

struct FunctionType {
  struct Type *returnType;
  struct ParameterType *parameters;
};

struct ParameterType {
  char *name;
  struct Type *type;
  struct ParameterType *next;
};
```

3.3 Type checking

1. Primitive types are equivalent if and only if they are all `int` or all `float` or all `char`.
2. Array types are equivalent if and only if the array size and the element type are the same.
3. Structure types are equivalent if and only if all the field types are the same and in the same order.
4. Function types are equivalent if and only if their parameters types are the same and in the same order.

4. Semantic check

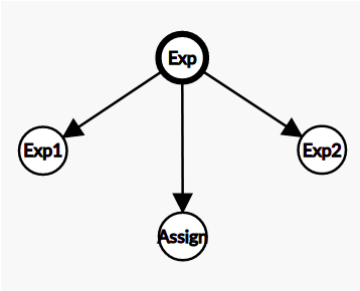
The most error types are only about searching through symbol table and type operating. However, there is one exception: the error about left value. In this section we will explain how we support every error type.

In the syntax tree, variables are only used in the subtree of `Exp`. We make light modifications on the node of syntax tree to support all error types. `left_or_right` is added to specify whether the `Exp` is left value or right value. 0 means left and 1 means right, while the default value is 1. `type` is added to denote the `Exp`'s type.

Error type 1 and 2 are about undefinition. When a variable is used or the function is invoked, we will search through the symbol table stack. If we find nothing, an error will be reported.

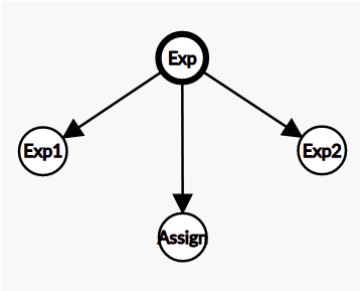
Error type 3 and 4 are about redefinition. When a variable is defined, we will first search through the current symbol table, if we find a duplicate one, an error will be reported. For function definition, only the global symbol table, which is the bottom symbol table, will be searched.

Error type 5 is about mismatching types. When we meet a tree like this:



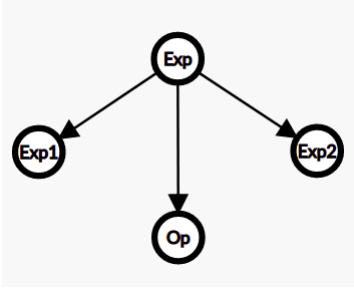
we will check whether the `type` of `Exp1` is equal to the `type` of `Exp2`. If they are not equal, an error will be reported.

Error type 6 is about left value. When we meet a tree like this:



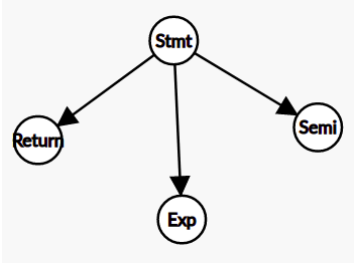
we will check whether the `left_or_right` of `Exp1` is 0. If it is not 0, an error will be reported.

Error type 7 is about mismatching operands. When we meet a tree like this:



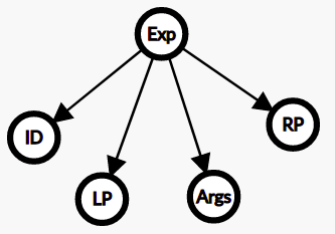
we will check whether the `type` of `Exp1` and the `type` of `Exp2` is compatible with the `Op`. If they are not compatible, an error will be reported.

Error type 8: when me meet a tree like this:



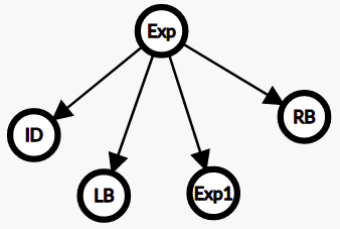
we will first get the current function's return type and compare it with the `Exp`'s `type`. If they are not equal, an error will be reported.

Error type 9 and 11 are about function: when we meet a tree like this:



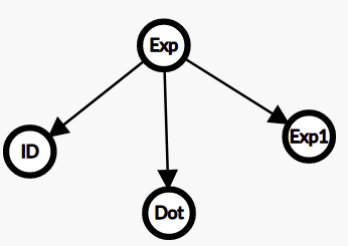
we will first get the type of `id` from symbol table. If the type is not function, an error type 11 will be reported. Then we will grab information from the subtree of `Args`, and get the function's argument list from symbol table by `ID`. If they are not equal, an error will be reported.

Error type 10 and 12 are about arrays: when we meet a tree like this:



we will first get the type of `id` from symbol table. If the type is not array, an error type 10 will be reported. Else if the type of `Exp1` is not `int`, an error type 12 will be reported.

Error type 13 and 14 are about structure: when we meet a tree like this:



we will first get the type of `id` from symbol table. If the type is not struct, an error type 13 will be reported. Then we will get the type of `Exp1`, and search the type in the struct's filed list. If the field list does not contain the type, an error type 14 will be reported.

Error type 15 is about redefining. The only thing different from error type 3 and 4 is that we will search `struct id` instead of `id` through symbol table stack.

5. Extra explanation

There are some test cases that our output is not the same as the provided output. This section will explain the difference.

1. test_2_r01.spl

```
int test_2_r01()
{
  int x1 = 5;
  int x2 = 3;
  int x4 = x1 + x3;
  return 0;
}
```

```
01
Error type 1 at Line 5: undefined variable: x3
Error type 7 at Line 5: unmatching operands
```

```
Error type 1 at Line 5: undefined variable: x3
<
```

In our output, we report an extra error. The reason is that the type of x3 is unknown and it is illegal to add an int with unknown type.

2. test_2_r04.spl

```
int compare(int x, int y)
{
  if (x > y) {return 1;}
  else if (y > x) {return -1;}
  else {return 0;}
}
float compare(int a, int b)
{
  if (b > a) {return -1.0;}
  if (b == a) {return 0.0;}
  return 1.0;
}
int test_2_r04()
{
  return compare(7, 8);
}
```

```
Error type 4 at Line 7: redefine function: compare
Error type 8 at Line 15: incompatiable return type
```

```
Error type 4 at Line 7: redefine function: compare
<
```

In our output, we report an extra error. The reason is that we resolve the function compare as the latest definition, which is defined at line 7.

3. test_2_r10.spl

```
struct Person
{
  int name;
  int friends[10];
};
int test_2_r10()
{
  struct Person tom;
  struct Person people[10];
  int i = 0;
  while (i < 10)
  {
    people[i].name = i;
    tom.friends[i] = i;
    i = i + 1;
  }
  return tom.name[i-1];
}
```

```
Error type 10 at Line 17: indexing on non-array variable
Error type 8 at Line 17: incompatiable return type
```

```
Error type 10 at Line 17: indexing on non-array variable
<
```

In our output, we report an extra error. The reason is that `tom.name[i-1]` is unknown, and unknown is incompatible with `int`.

4. test_2_r11.spl

```
int compare1(int x, int y)
{
  if (x > y) {return 1;}
```

```
  else if (y > x) {return -1;}
  else {return 0;}
}
int test_2_r11(int i, int m, int n)
{
  int compare2 = 10;
  if (i == 0)
  {
    return compare1(m, n);
  }
  else
  {
    return compare2(m, n);
  }
}
```

```
Error type 11 at Line 16: invoking non-function variable: com   Error type 11 at Line 16: invoking non-function variable: com
Error type 8 at Line 16: incompatiable return type           <
```

In our output, we report an extra error. The reason is that `compare2(m, n)` is unknown, and unknown is incompatible with `int`.

5. test_2_r12.spl

```
int test_2_r12()
{
  float arr1[10];
  float arr2[10];
  float a = 1.1;
  int i = 0;
  while (i < 10)
  {
    arr1[i] = a;
    a = a * a;
  }
  i = 0;
  while (i < 10)
  {
    arr2[arr1[i]] = a;
  }
  return 0;
}
```

```
Error type 12 at Line 15: indexing by non-integer              Error type 12 at Line 15: indexing by non-integer
                                                             > Error type 5 at Line 15: unmatching type on both sides of ass
```

In our output, we do not report the mismatching error. The reason is that we guess that the developer wants to access the array `arr2`. Therefore, although his access is illegal, we still make the type of `arr2[arr1[i]]` be `float`.