Compiler Design

Project #3. Semantic analysis

2018008904 이성진

본 프로젝트는 Tiny Compiler를 수정하여 C-minus를 컴파일할 수 있게 하는 과정에서 lexer와 parser를 거쳐 만들어진 AST를 바탕으로 semantic analysis를 구현하는 과제입니다.

1. Compilation environment and method

- 프로젝트를 개발하고 컴파일한 환경은 Intel CPU를 사용하는 Windows 10 Edu 64bit에서 WSL 2 버전으로 Ubuntu 20.04를 설치하여 사용했습니다. C compiler는 gcc를 사용했습니다.
- 프로젝트를 빌드하기 위해서는 make 명령을 활용하면 되며, 아래 Makefile의 내용을 바탕으로 세부 옵션을 설정할 수 있습니다.

- **make**: make cminus_semantic과 동일합니다.
- **make cminus_semantic**: lexcial analysis와 syntax analysis를 거쳐 생성된 AST를 바탕으로 symbol table을 구성하고 이를 바탕으로 type check 등 semantic error가 존재하지 않는지 검사하는 프로그램을 생성합니다.
- 생성된 바이너리 파일은 cminus_semantic [C-minus code file]과 같이 command argument를 통한 입력 파일을 지정해 주면 해당 파일에 semantic error가 존재하는지 확인하여 존재하는 경우 에러 정보를 출력해줍니다.

2. Brief explanations about how to implement and how it operates

- 이 프로젝트는 C-minus로 작성된 프로그램의 semantic analysis를 위해 적절한 reserved keyword, symbol, 그리고 token 등을 정의한 다음 Yacc/Bision을 이용해 정의하는 grammar를 이용해 parsing을 진행한 결과로 AST가 생성되면, 이를 바탕으로 symbol table을 생성하고 type checking 등 semantic analysis를 진행합니다.

2.1. Semantic errors in C-MINUS

• **Un/Redefined Variables and Functions**
- Scope rules are same as C language
- Function overloading is not allowed

• **Array Indexing Check**
- Only *int* value can be used as an index

• **Built-in Functions**
- input() and output(value) must be able to accessed globally

• **Type Check**
- *void* variable
- Operations such as *int*[] + *int*[], *int*[] + *int* and *void* + *void* are not allowed
- assignment type
- *if*/*while* condition: Only *int* value can be used for condition

- function arguments: number of parameters, types
- return type

2.2. Changing options for semantic analysis

main.c: NO_CODE만 TRUE로 설정하고 나머지 모든 옵션(TraceXXX..., NO_PARSE, NO_ANALYZE 등)을 FALSE로 설정하여 AST를 바탕으로 semantic error를 출력하도록 했습니다. 디버그 용도로 symbol table을 출력하려면 TraceAnalyze = TRUE로 설정하면 됩니다.

```
/* set NO_PARSE to TRUE to get a scanner-only compiler */
#define NO_PARSE FALSE
/* set NO_ANALYZE to TRUE to get a parser-only compiler */
#define NO_ANALYZE FALSE
```

```
int EchoSource = FALSE;
int TraceScan = FALSE;
int TraceParse = FALSE;
int TraceAnalyze = FALSE;
int TraceCode = FALSE;
```

globals.h: Project 1, 2에서 구현한 lexer와 parser를 거의 그대로 활용하여 만들어진 AST를 이용하기 때문에, treeNode의 정의 등 이전 구현 내용은 크게 수정하지 않았습니다.

2.3. Building symbol tables from AST

- C-minus의 규칙에 맞게 AST가 생성된 이후, buildSymtab 함수를 AST에 대해 실행시켜 tree traverse를 하며 symbol table을 생성합니다.
- buildSymtab 함수는 tree의 각 노드를 traverse 하며 해당 노드가 변수 또는 함수의 declaration일 경우 symbol table에 insert하는 작업을 preOrder로 수행합니다.

```
void buildSymtab(TreeNode * syntaxTree)
{ init_global_scope_with_builtins();
  traverse(syntaxTree,insertNode,postInsert);
  if (TraceAnalyze)
  { fprintf(listing,"\n\n< Symbol Table >\n");
    printSymTab(listing);
    fprintf(listing,"\n\n< Functions >\n");
    printFuncSymTab(listing);
    fprintf(listing,"\n\n< Global Symbols >\n");
    printGlobalSymTab(listing);
    fprintf(listing,"\n\n< Scopes >\n");
    printScopeTab(listing);
  }
}
```

- 기존 Tiny와 다르게 각각의 compound statement마다 scope가 달라야 하기 때문에, scopeList 구조체와 현재 진입한 scope를 알려주는 static variable currentScope를 정의했습니다. 또한 scope마다 symbol table을 가질 수 있도록 scope가 생성될 때마다 각각에 해당하는 bucketListRec의 해쉬 테이블을 생성해서 사용했습니다.

```
typedef struct ScopeListRec
   { char * name;
     int curloc;
     int compidx;
     int isFunc;
     BucketList bucket[BUCKET_SIZE];
     struct ScopeListRec * parent;
   } * ScopeList;
```

```
static ScopeList currentScope = NULL;
```

- Scope를 생성하고 찾기 위해 create_scope와 find_scope 함수를 정의하여 사용했고, 기존 symbol table에 insert하는 것과 line number를 추가하는 기능이 둘 다 포함되었던 st_insert 함수를 분리하여 st_insert와 st_addLineNo 함수 2가지로 나누어서 코드의 직관성을 높였습니다. 또한 symbol table lookup을 위해 st_lookup과 st_lookup_excluding_parent 함수 2가지 버전을 만들어서 현재 scope와 모든 범위에서 찾는

과정을 별도로 진행할 수 있게 했습니다.

```
ScopeList create_scope(char * name, ScopeList currentScope)
{ int i;
  ScopeList s = NULL;
  s = (ScopeList) malloc(sizeof(struct ScopeListRec));
  s->name = name;
  s->isFunc = FALSE;
  s->curloc = 0;
  s->compidx = 0;
  s->parent = currentScope;
  for(i = 0; i < BUCKET_SIZE; i++) s->bucket[i] = NULL;
  for (i = 0; i < SCOPE_SIZE; i++)
    if (scopeList[i] == NULL)
    { scopeList[i] = s;
      break;
    }
  return s;
}
```

```
ScopeList find_scope(char * name){
  int i;
  ScopeList s = NULL;
  for(i = 0; i < SCOPE_SIZE; i++){
    s = scopeList[i];
    if(s != NULL && strcmp(s->name, name) == 0){
      break;
    }
  }
  return s;
}
```

```
void st_insert( ScopeList s, char * name, int lineno, TreeNode * t );

void st_addLineNo( BucketList l, int lineno);
```

```
BucketList st_lookup ( ScopeList s, char * name );

BucketList st_lookup_excluding_parent ( ScopeList s, char * name );
```

- Debug 용도로 symbol table의 각종 정보를 출력하기 위한 출력 함수들도 정의했습니다.

```
void printSymTab(FILE * listing);
void printFuncSymTab(FILE * listing);
void printGlobalSymTab(FILE * listing);
void printScopeTab(FILE * listing);
```

- Redefined symbol definition이 있는지 탐지하는 과정은 symbol table을 build하는 과정에서 이루어집니다.

```
case DeclK:
  switch(t->kind.decl)
  { case VarK:
      l = st_lookup_excluding_parent(currentScope, t->attr.name);
      if (l == NULL){…
      else
      { /* already in table, so the symbol is redefined. */
        redefSymbol(t->attr.name, t->lineno);
      }
      break;
    case FuncK:
      l = st_lookup_excluding_parent(currentScope, t->attr.name);
      if (l == NULL)…
      else
      { /* already in table, so the symbol is redefined. */
        redefSymbol(t->attr.name, t->lineno);
      }
      break;
    default:
      break;
  }
  break;
```

2.4. Building symbol tables from AST

- C-minus의 규칙에 맞게 AST가 생성된 이후, buildSymtab 함수가 정상적으로 종료되면 symbol table과 각각의 scope가 준비됩니다. 이후 typeCheck 함수를 AST에 대해 호출하면 tree를 traverse하며 각각의

expression들이 정상적인 type을 가졌는지 확인합니다.

- Symbol table을 구성하는 과정과 반대로, type check의 경우 하위 child node들에 대한 type check가 완료되어야 현재 노드에 대한 type check 역시 가능하므로 각 노드를 방문했을 때 자식 노드에 대해 모든 작업이 완료된 후 작업이 수행되는 post order 순서로 실행됩니다.

```
/* Procedure typeCheck performs type checking
 * by a postorder syntax tree traversal
 */
void typeCheck(TreeNode * syntaxTree)
{ traverse(syntaxTree,preCheck,checkNode);
}
```

- Type checking 과정에서 나머지 대부분의 semantic error들이 탐지됩니다.

3. Examples and corresponding result screenshots

make 명령을 이용해 생성된 cminus_semantic을 test용 프로그램을 컴파일하기 위해 실행한 결과 코드를 분석하여 AST가 생성되고 이를 바탕으로 semantic analysis를 진행하여 semantic error가 발생한 경우 출력하는 것을 볼 수 있습니다. Debug 용도로 traceAnalysis를 TRUE로 설정하면 symbol table의 각 scope와 symbol들의 정보를 볼 수 있습니다.

Sample input: test1.txt

```
1   void foo(int u, int v)
2   {
3       int y[3];
4       return x + y;
5   }
6
7   int main(void)
8   {
9       int x;
10      int y[3];
11
12      foo(x, y[1]);
13
14      return x + y[x];
15  }
```

```
~/repos/2022_ele4029_2018008904/3_Semantic$ ./cminus_semantic ./test1.txt > output.txt
```

Semantic error check result:

```
C-MINUS COMPILATION: ./test1.txt
Error: Undeclared variable "x" is used at line 4
Error: Invalid operation at line 4
Error: Invalid return at line 4
```

Symbol table construction result:

```
C-MINUS COMPILATION: ./test1.txt

Building Symbol Table...


< Symbol Table >
Symbol Name    Symbol Kind   Symbol Type    Scope Name   Location  Line Numbers
------------  -----------  ------------  ------------  --------  ------------
main           Function     int           {global}      3           7
input          Function     int           {global}      0           0
output         Function     void          {global}      1           0
foo            Function     void          {global}      2           1   12
value          Variable     int           output        0           0
u              Variable     int           foo           0           1
v              Variable     int           foo           1           1
y              Variable     int[]         foo           2           3    4
x              Variable     int           main          0           9   12   14   14
y              Variable     int[]         main          1          10   12   14


< Functions >
Function Name   Return Type   Parameter Name  Parameter Type
------------  ------------  --------------  --------------
main           int                           void
input          int                           void
output         void
-              -             value           int
foo            void
-              -             u               int
-              -             v               int


< Global Symbols >
Symbol Name    Symbol Kind   Symbol Type
------------  -----------  ------------
main           Function     int
input          Function     int
output         Function     void
foo            Function     void
```

```
< Scopes >
Scope Name    Nested Level   Symbol Name    Symbol Type    Scope Kind
------------  ------------  -------------  -------------  ------------
output        1             value          int            Function

foo           1             u              int            Function
foo           1             v              int            Function
foo           1             y              int[]          Function

main          1             x              int            Function
main          1             y              int[]          Function


Checking Types...
Error: Undeclared variable "x" is used at line 4
Error: Invalid operation at line 4
Error: Invalid return at line 4

Type Checking Finished
```