

Compiler Project #3. Semantic Analysis Report

2018008277 노주찬

1. Compilation Environment

아래의 환경에서 프로젝트를 컴파일하고 테스트했습니다.

항목	값
OS	Ubuntu 20.04.4 LTS
Architecture	x86_64
C Compiler	gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Lex/Flex	flex 2.6.4
Yacc/Bison	Bison 3.8.1

POSIX 함수인 `strdup()` 함수를 사용하고 있기 때문에 C POSIX 라이브러리가 제공되지 않는 환경에서는 컴파일이 되지 않을 수 있습니다.

2. Previous Project Modification

이전 프로젝트인 Scanner와 Parser에서 큰 수정 사항은 없었습니다. 상세한 구현 수준에서는 Parser 프로젝트에서 `parameter`, `argument`와 같이 동일한 요소를 `LinkedList`로 저장하는 방법에서 일부 수정이 있었습니다.

3. Symbol Table

3.1. Symbol

기존 Tiny 컴파일러에서 사용하던 `BucketList` 구조체에 심볼의 정보를 저장했습니다.

각 심볼마다 이름, 등장 라인, 메모리 위치 (Semantic Analyzer에서는 같은 이름의 심볼끼리 구분하는 용도로 사용했습니다.), 종류 (함수 또는 변수), Expression 타입 (변수인 경우, 변수의 타입, 함수인 경우, 리턴 타입과 파라미터 타입) 을 저장하고 있습니다.

3.2. Symbol Table

Symbol Table은 Hint로 주어진 `ScopeList` 구조체를 사용하였고, 기존 Tiny 컴파일러에서 구현한 해시 테이블 형태로 심볼을 저장했습니다.

Scope에 대한 정보를 담기 위해서 부모 Symbol Table 포인터를 저장했습니다.

함수 심볼의 타입 바인딩과 리턴 타입 체크를 위해서 global scope가 아닌 Symbol Table에서는 해당 scope를 정의하는 함수 이름과 global scope의 Symbol Table에서 해당 함수 심볼의 메모리 위치를 저장하고 있습니다.

그리고 이 Symbol Table 인스턴스는 새로운 scope를 생성하는 각 AST 노드가 가지고 있도록 했습니다.

3.3. Symbol Table Lookup

Symbol Table을 Lookup 하는 함수는 크게 3가지가 필요했습니다.

1. Symbol을 사용하고 있는 부분에서 Symbol의 정의를 찾기 위해 Recursive하게 Symbol Table을 Lookup하는 함수.
2. Symbol을 정의하고 있는 부분에서 재정의 여부 판단을 위해 현재 Scope에서만 Symbol Table을 Lookup하는 함수.
3. 함수 파라미터 타입 바인딩, 리턴 타입 체크를 위해서 global scope의 Symbol Table에서 심볼 이름과 메모리 위치로 Lookup하는 함수.

또한 심볼의 재정의 여부를 판단할 때, 변수와 함수를 구분해서 체크해야 한다고 생각되어, 1번과 2번의 Lookup 함수에서는 scope, 심볼 이름, 찾고자 하는 심볼 종류 (변수, 함수, 둘 다)를 지정해서 해당하는 종류의 심볼만 찾을 수 있도록 했습니다.

그리고 lookup을 수행할 때, 재정의 오류로 인해 같은 scope에서 같은 이름의 심볼이 존재하더라도 정의한 순서대로 찾도록 하여 빌트인 함수인 input, output이 재정의 되더라도 global scope에서 가장 먼저 추가 되는 빌트인 함수를 먼저 lookup 할 수 있도록 했습니다.

4. Type Checking

타입 체크 과정 중에서 오류가 발생하면 해당 Expression의 타입은 Unknown으로 바인딩하여, 해당 Expression의 타입 정보를 사용하는 타입 체크에서도 연쇄적으로 오류가 발생해 전파되도록 되어 있습니다. 단, 예외적으로 함수 호출에서 파라미터가 맞지 않아서 발생하는 타입 오류는 함수의 return type으로 바인딩되도록 설정했습니다. 그 근거로는 C-Minus에서 function overloading이 없기 때문에, 함수 심볼이 존재하기만 하면 파라미터가 틀렸더라도 해당 심볼의 함수를 호출하겠다는 프로그래머의 의도가 명확하기 때문에 함수의 return type으로 바인딩 하더라도 문제가 없다고 판단했습니다.

4.1. Variable Accessing

Recursive Lookup으로 변수 심볼 정의 여부를 확인합니다.

타입 체크에 성공한 경우, Expression의 타입을 변수 심볼의 타입으로 바인딩합니다.

4.2. Array Indexing

먼저 Recursive Lookup으로 indexing이 되는 변수 심볼의 정의 여부를 확인합니다.

두번째로 찾은 심볼이 `int[]` 타입인지 확인합니다.

세번째로 `index` 쪽 `expression`이 `int` 타입인지 확인합니다.

타입 체크에 성공한 경우, `Expression`의 타입을 `int`로 바인딩합니다.

C-Minus에서 `void`, `void[]` 타입 변수는 허용되지 않으므로, indexing 되는 쪽에서 `void[]` 타입의 심볼을 찾고 `index` 쪽 `Expression`의 타입이 `int`였더라도, 타입 체크 성공 / `void` 타입 바인딩으로 처리하지 않고 타입 체크 실패로 처리했습니다.

4.3. Assignment

LHS와 RHS가 모두 올바른 타입이고, 둘의 타입이 동일한지 확인합니다.

Array variable, void type variable 간의 Assignment 허용 여부에 대해 이슈가 있었는데, 명세에서 타입의 동일 여부만 체크하도록 되어있었고 syntax에서도 이를 막지 않아 허용했습니다.

타입 체크에 성공한 경우, `Expression`의 타입을 LHS의 타입으로 바인딩합니다.

4.4. Binary Operation

LHS와 RHS가 모두 `int`타입인지 확인합니다.

타입 체크에 성공한 경우, `Expression`의 타입을 `int`로 바인딩합니다.

4.5. 함수 호출

먼저 Recursive Lookup으로 함수 심볼 정의 여부를 확인합니다.

함수 심볼의 파라미터 타입과 파라미터 AST 노드의 개수, 타입 일치 여부를 확인합니다.

함수 심볼이 존재한 경우, `Expression`의 타입을 함수 심볼의 `return type`으로 바인딩합니다.

4.6. If-(Else)

Condition 영역의 `Expression`이 `int` 타입인지 확인합니다.

4.7. While

Condition 영역의 `Expression`이 `int` 타입인지 확인합니다.

4.8. Return

먼저 Global scope의 symbol table에서 현재 함수 심볼을 탐색합니다.

함수 심볼의 리턴 타입과 리턴하고자 하는 `expression`의 타입 (없는 경우, `void`) 일치 여부를 확인합니다.

5. Traverse

C-Minus에서는 C처럼 global 레벨에서 현재 코드 시점에 정의되지 않았지만 이후에 정의되는 Symbol에는 접근할 수 없기 때문에, 기존 Tiny 컴파일러 구현처럼 Double Pass를 하지 않고 AST Traverse 한번에

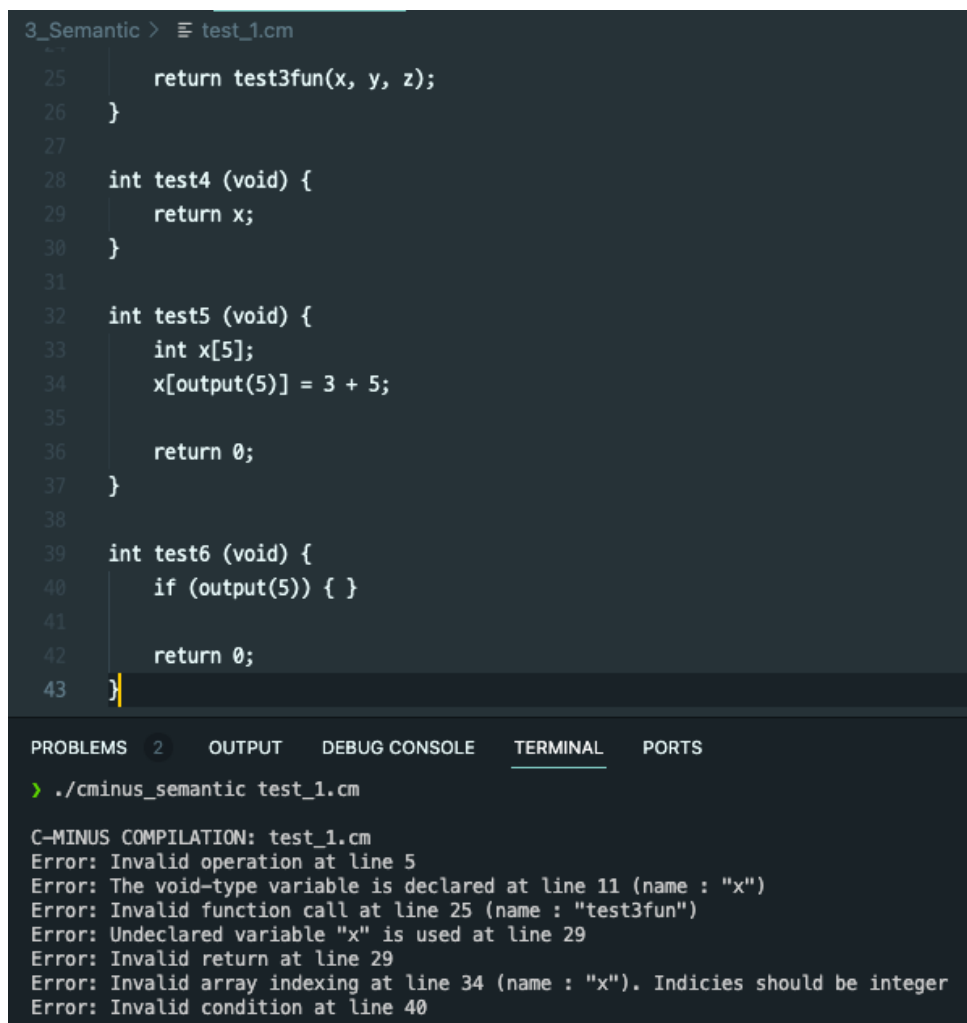
preorder로 Symbol Table 생성을, postorder로 Type Check를 수행하도록 했습니다.

또한 traverse 함수에 scope에 대한 처리를 추가했습니다. traverse 함수의 파라미터로 현재 scope 정보를 받도록 하고, 새롭게 방문한 AST Node가 새로운 scope에 대한 Symbol Table을 가지고 있다면 해당 Symbol Table로 현재 scope 정보를 바꾸고, 해당 AST Node의 방문을 종료할 때 이전 scope를 복구하도록 처리했습니다.

따라서 수정된 traverse의 알고리즘은 아래와 같습니다.

Preorder (Symbol Insert → 필요한 경우, 새로운 scope의 Symbol Table 생성) → 새로운 scope가 생성된 경우, scope 교체 → 자식 순회 → 앞에서 scope를 생성한 경우, 기존 scope 복원 → Postorder (Type Check) → sibling 순회

6. Examples and Results



```
3_Semantic > test_1.cm
25     return test3fun(x, y, z);
26 }
27
28 int test4 (void) {
29     return x;
30 }
31
32 int test5 (void) {
33     int x[5];
34     x[output(5)] = 3 + 5;
35
36     return 0;
37 }
38
39 int test6 (void) {
40     if (output(5)) { }
41
42     return 0;
43 }

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS
> ./cminus_semantic test_1.cm

C-MINUS COMPILATION: test_1.cm
Error: Invalid operation at line 5
Error: The void-type variable is declared at line 11 (name : "x")
Error: Invalid function call at line 25 (name : "test3fun")
Error: Undeclared variable "x" is used at line 29
Error: Invalid return at line 29
Error: Invalid array indexing at line 34 (name : "x"). Indices should be integer
Error: Invalid condition at line 40
```

명세에서 주어진 예제를 모은 파일로 확인해보니, 의도한 대로 Semantic Error를 탐지하고 있음을 확인할 수 있었습니다.