Project 3 결과보고서

2015-18525 김세훈

1. 구현 방법

전체적인 semantic check의 동작은 subc.y 파일에서 작동하며, token을 읽어오는 subc.l (NULL token 이 추가된 점 이외에는 이전과 동일), 각종 check 함수들을 포함하는 check.c, 여러 동작들을 정의하는 functions.c 파일과 함께 동작한다. 그리고 primitive type의 decl* pointer들(inttype, chartype, voidtype), scope stack top을 가리키는 pointer 등의 중요한 전역 변수들은 모두 subc.h 파일에 선언되어 있다. 구체적인 구현 내용은 아래에서 다루고자 한다.

2. 구현 내용

2.1 Scope Stack과 Definition Stack 및 Redeclaration/Undeclaration 에러 처리

Scope stack와 definition stack은 각각 scope_stack 구조체와 ste 구조체로 구성되어 있는 단방향 linked list이다. Scope_stack의 멤버변수 top은 해당 scope에서 가장 최근에 선언된 definition stack entry를 가리키도록 하였다. 그리고, scope stack의 top 은 global 변수 sstop에 저장하여 어디서든지 접근할 수 있도록 하였다. 따라서, 현재 scope의 entry들은 sstop->top부터 sstop->prev->top까지, 전체 scope은 sstop->top부터 dummy node까지 scan하면서 확인할 수 있다. 각각은 함수 find()와 find_current_scope()가 담당한다. 변수가 선언 될 때는 find_current_scope() 함수를 통해 redeclaration 에러를 잡아줬으며, 사용될 때는 find() 함수를 통해 undeclaration 에러를 잡아주었다. 또한 수업 자료와 동일한 방법으로 scope_stack에 새로운 local scope를 위한 entry를 잡아주는 push_scope() 함수와 이를 제거하는 pop_scope() 함수를 생성하였다.

Main() 함수가 호출 되자마자 definition stack에는 dummy node와 primitive type들(inttype, chartype, voidtype)에 대한 entry가 들어온다. 추가적으로 globaldef라는 전역 변수를 생성하여, 이때의 sstop->top 값을 저장해 둔다. 이는 이후에 struct가 어느 scope(함수 내부나 다른 struct 내부)에서 생성되든지 간에 definition stack top의 local scope가 아닌 globaldef이 가리키는 global scope 영역에 선언이 될 수 있도록 하기 위함이다. 즉, struct가 생성된 local scope가 pop되어도 해당 struct의 정의는 definition stack의 바닥에 위치하기 때문에 그 정의가 사라지지 않는다.

2.2 Value Declaration

모든 값들은 value decl(편의상 명명)에 encapsulate되며 이들은 VAR(변수), EXP(식), CONST(상수), FUNC(함수), NULL로 구성된다. CONST는 1과 'a'와 같은 값들로, 이들에 대한 binary 및 unary operation들은 CONST를 생성한다. CONST는 유일하게 array를 선언할 때 대괄호 내부에 들어갈 수 있는 값이다(e.g. int a[1+1]은 가능하지만 int a[1+x]는 불가능). VAR은 순수한 변수를 나타낸다. VAR에 대한 incop, decop, *operator는 VAR를 생성하지만, 나머지 연산자들을 적용하면 식(EXP)이 생성된다 (e.g. int *x에 대한 *x는 VAR, int x에 대한 x++는 VAR, x+x는 EXP). VAR은 유일하게 assign 문에서 좌변에 위치할 수 있는 값이다. EXP는 변수의 일종이지만 assign문의 좌변에 위치할 수 없기

때문에 다른 class로 분류를 하였으며 CONST와 VAR들의 operation들로 생성이 된다. 마지막으로 FUNC은 함수로, ( )를 통한 호출이 가능한 유일한 값이다. 추가로 NULL은 assign문의 좌변이 포인터 변수일 때만 우변에 위치할 수 있으며, 어떠한 unary/binary 연산도 불가능한 값이다.

2.3 Type declaration 및 Type 비교 (Assign 관련 에러 처리)

타입은 type decl로 나타내며, 모든 value decl들은 멤버변수 type이 자신의 type decl을 가리키도록 한다. 타입은 VOID, INT, CHAR, STRING, ARRAY, STRUCT, POINTER 이렇게 7가지가 존재하며, 각각은 다음과 같은 type decl로써 정의된다.

- VOID, INT, CHAR (primitive types): 각각 definition stack bottom에 있는 voidtype, inttype, chartype의 decl entry들이다.
- POINTER: 각 포인터 변수마다 type decl entry가 할당되며, 멤버변수 ptrto가 해당 포인터 변수의 type decl을 가리키는 구조이다. (e.g. int 포인터는 ptrto가 inttype을 가리킨다)
- ARRAY: 각 array 변수마다 type decl entry가 할당되며, 멤버변수 elementvar이 해당 array의 원소에 대한 value decl을 가리킨다. 이 value decl은 다시 자신의 type decl을 가리킨다. (e.g. int array의 type decl는 int 변수에 대한 value decl을 가리키며, 이는 다시 inttype을 가리킨다.)
- STRUCT: 각 struct 변수마다 type decl entry가 할당되며, 멤버변수 fieldlist가 해당 struct의 멤버변수들에 대한 value decl들이 저장된 stack을 가리킨다. 각각의 value decl들은 저마다의 type decl을 가리키고 있다.
- STRING: type decl의 경우 기본적으로 char pointer와 동등하다. (단, string과 char pointer은 value decl이 전자는 'expression', 후자는 'variable'이라는 점에서 차이를 둔다. 이는 char* p = "string"은 가능하지만 "string" = p에 대해서는 에러를 발생하기 위한 용도이다.)

타입 비교는 binary/unary 연산 시 호환성을 검사하고, 함수의 argument 및 return type의 오류를 검사하며 특히 assign 연산을 처리하는 데에 있어서 핵심적이다. 이는 check_type_compat()함수에서 진행되며, 두 type decl을 입력 받아서 근본적으로 같은 type임을 확인해주는 함수이다. 구현의 편의성을 위해서 재귀함수로 작성되었다. 1) 두 type decl이 동일한 primitive인 경우 type decl 포인터는 같은 곳을 가리키므로 true를 출력한다. 2) 그렇지 않은 경우 두 type이 둘 다 pointer type인지 또는 둘 다 array type인지를 확인해 준다. 그렇다면 pointer 또는 array를 벗겨낸 후 다시 check_type_compat()의 입력으로 넣어줌으로써 재귀적인 검사를 한다. 3) 둘 다 아니면 false를 출력한다. 예외적으로, array를 pointer에 assign하는 것이 가능하기 때문에 좌변의 pointer와 우변의 array를 벗겨낸 후 check_type_compat()을 재귀 호출하는 mode를 따로 만들었다.

2.4 Basic Operation Check 관련 에러 처리

Check.c의 여러 함수들로 각각의 상황에 맞는 validity check를 진행하였으며, type checking에는 직전에 언급되었던 check_type_compat() 함수를 활용하였다.

- INCOP, DECOP: check_inc_dec() 함수 사용. operand의 type decl이 INT 또는 CHAR가 아니거나 value decl이 VAR이 아니면 error.
- +, −: check_add_sub() 함수 사용. operand의 type decl이 둘 다 INT가 아니면 error

- !, &&, ||: check_and_or() 함수 사용. operand의 type decl이 모두 INT가 아니면 error
- RELOP, EQUOP: check_rel_equ() 함수 사용. 두 operand중 어느 하나라도 type decl이 INT나 CHAR가 아니면 무조건 error. RELOP의 경우 두 operand가 모두 pointer지만 서로 다른 종류의 pointer인 경우 error

2.5 Pointer와 Array 생성 및 관련 에러 처리

Pointer의 경우 해당 pointer가 가리키는 변수의 type decl을 POINTER type decl에 encapsulate 하여서 value decl의 type으로 지정해 주었다. 즉 int pointer의 value decl의 경우 POINTER type decl을 가리키고 있으며, 이는 다시 (멤버변수 ptrto를 통해서) inttype을 가리키고 있는 형태이다. Array의 경우 element의 value decl을 ARRAY type decl에 encapsulate하여서 value decl의 type으로 지정해 주었다. 예를 들어 int a[2]와 같은 integer array a의 경우 a의 value decl는 ARRAY type decl을 가리키고 있으며, 이는 멤버변수 elementvar을 통해 value decl을 가리킨다. 이는 곧 a의 원소(e.g. a[0], a[1])의 value decl이 되며, inttype을 가리킨다.

Pointer의 경우 2.4에서와 같은 방법으로 RELOP, EQUOP를 처리하였으며, assign에서 우변에 NULL이 왔을 경우 유일하게 error를 피해가도록 설정하였다. 또한 *의 경우 type decl이 POINTER인지 확인한 후 POINTER type decl을 decapsulate한 (즉, type decl의 ptrto와 동일한 type을 갖는) 변수(VAR)을 출력하였다. &의 경우 type decl이 VAR인지 확인한 후 이 type decl을 반대로 POINTER type decl에 encapsulate한 식(EXP)을 출력하였다. 전자의 경우 결과값에 *p=1과 같은 assign이 가능하기 때문에 EXP가 아닌 VAR이 되는 것이 자연스러우며, 반대로 후자는 결과값에 &a = p와 같은 assign이 불가능하기 때문에 VAR이 아닌 EXP가 되는 것이 자연스럽다.

Array의 경우 ID[expr]라는 문법에 대해서 ID의 type decl이 ARRAY인지, 그리고 expr의 type decl이 inttype인지 확인해주는 것으로 충분하다. 그 결과로는 비슷하게 ARRAY type decl을 decapsulate한 변수(VAR)을 출력해주면 된다. 역시 a[0]=10과 같은 assign이 가능하므로 EXP가 아니라 VAR이다.

3.6 Structure 생성 및 관련 에러 처리

Structure의 경우 1) struct type이 생성되는 경우는 STRUCT ID { def_list } 문법을 통해 생성되며 2) 생성된 struct type으로 struct variable을 생성하는 경우는 STRUCT ID 문법을 통해 생성된다. 전자의 경우 'STRUCT ID {' 직후에 ID에 대해서 'redeclaration error'를 잡아주어야 한다. Redeclaration이 아닌 경우 push_scope() 함수를 통해서 def_list의 원소들을 받아들일 fake scope을 잡아준다. 이 때 fake scope 역시 local scope으로 간주가 되기 때문에 이미 사용된 변수 이름들이 struct의 멤버 변수로 사용되어도 error가 잡히지 않게 된다 (e.g. int x; struct st{int x;}와 같은 상황) 마지막으로 'deflist }'까지 끝마치면 pop_scope() 함수를 통해서 원래의 definition stack을 복원시키고, fake scope에 있던 definition들은 STRUCT type decl의 fieldlist에 저장한다. 이렇게 생성된 새로운 STRUCT type은 declare_struct_type()이라는 함수를 통해서 definition stack의 가장 바닥에 append 시켜서 'global declaration'으로 만들어 준다. Definition stack의 바닥은 2.1에서 언급된 것과 같이 globaldef이라는 전역 변수를 통해서 접근할 수 있다.

후자의 경우 반대로 ID에 대한 undeclaration error를 잡아주어야 한다. 만약 해당 struct type이 이미

선언이 된 상태라면 definition stack의 가장 바닥에 global declaration으로써 존재하기 때문에 find() 함수를 통해서 반드시 찾을 수가 있다. Find() 함수를 통해 찾아지지 않거나, 찾아졌으나 struct type이 아닌 경우 (e.g. int x; struct x y;)는 모두 error를 띄워준다.

Struct operation은 .과 STRUCTOP(->)이 있으며, 전자는 operand의 type decl이 STRUCT가 아닌 경우와 type decl의 fieldlist에서 field가 찾아지지 않는 경우 error를 띄워준다. 후자는 먼저 operand의 type decl이 POINTER가 아닌 경우 error를 띄워 주어야 한다. Type decl이 POINTER가 맞다면 이를 decapsulate 해준 결과에 대해서 전자와 동일한 작업을 반복하면 된다.

3.7 Function 생성 및 관련 에러 처리

Function의 parameter들은 struct의 definition과 동일하게 'fake scope'을 push하고 pop하는 방법으로 수거된다. 수거된 parameter은 FUNCTION value decl의 'formals' 멤버 변수에 저장되며 함수의 return type에 대한 type decl이 'returntype' 멤버 변수에 저장된다. 이렇게 생성된 FUNCTION value decl은 definition stack에 push가 되어서 이후에 호출될 수 있도록 한다.

만약 함수에 compound statement가 존재하는 경우 다시 한 번 push_scope()을 통해 local scope을 생성한 후에 formals와 return type을 갖는 가상의 변수 'returnid'를 push해준다. Formals을 push해 줌으로써 함수의 compound statement에서 parameter를 접근할 시 undeclaration 에러를 피해주며, parameter와 동일한 이름의 변수를 선언할 시 redeclaration 에러를 발생시킬 수 있다. Returnid는 compound statement에서 RETURN 또는 RETURN expr을 만날 때마다 return type check를 해주기 위한 용도이다. RETURN의 경우 returnid의 type decl이 VOID가 아닌 경우, RETURN expr의 경우 expr의 type decl이 return id의 type decl과 일치하지 않는 경우 error를 띄워준다. 마지막으로 compound statement가 완료되면 pop_scope()을 통해서 함수의 local scope을 모두 제거해준다.

마지막으로, 함수가 호출되는 경우 1) 함수 ID가 존재하지 않는 경우 undeclaration error를 띄워주며 2) 존재하지만 FUNCTION이 아닌 경우 (e.g. int x; x();) error를 띄워준다. 3) 이후에 args 각각을 함수의 type decl의 formals와 1대 1로 type checking을 진행하여 하나라도 어긋나는 경우 error를 띄워준다. 물론 args와 formals의 길이가 다른 경우도 error를 발생시켜주면 된다. 모든 error를 피해갔다면 최종적으로 함수의 returntype을 type decl로 하는 새로운 식(EXP)를 출력해주면 된다.

3. 문제점

이렇게 구현이 된 컴파일러는 스펙에 명시된 대로 c언어에 대한 semantic check를 성공적으로 할 수 있다. 단 기존의 c와 비교해서 다음과 같이 여러 가지 문제점들이 존재한다.

1) Explicit type casting만을 허용하여 int와 char 사이에 assign이나 여러 유용한 operation들이 불가능.
2) Operation에 과도한 제한이 존재. 예를 들어 pointer에 대한 덧셈 뺄셈 등의 연산이 불가능.
3) int; char; struct st;와 같은 의미 없는 문장들을 걸러내지 못함
4) array를 pointer처럼 사용할 수가 없다. E.g. int a[10]에 대해 &a[1]을 a+1과 같이 나타내지 못함
5) struct나 function의 forward declaration이 불가능(이를 모두 redclaration으로 처리함)
6) 함수에 return문이 없거나 return문이 비어있는 branch를 잡아내지 못함. 등등..