**컴파일러의 기초 project 4 보고서**

전기정보공학부 2016-10769 이권형

**1. 구현 방법**

semantic check을 위해 각 reduce에서 embedded action을 이용한다. decl이라는 struct를 사용하여, variable, constant, function, type별로 decl struct를 만들고, linked list와 같은 pointer를 이용하여 각 token에서의 decl struct를 통해 semantic check을 시행한다. 또한, lexer와는 달리 중괄호로 표현되는 scope를 위해 scope stack을 통해 symbol table을 구현한다. scope를 나타내는 stack과 각 scope에서 ste stack이 있어 id에 따른 token의 scope를 알 수 있다.

semantic check에서는 크게 variable declaration에 관련된 것과 type check, pointer operation, struct, function, operation checking이 있다. 각각에 대한 사항을 2번에서 서술할 것이다.

**2. 구현 내용**

1) variable declaration

variable을 비롯하여 scope에 따라 id를 선언할 때는, declare function을 사용한다. declare function은 subc.y에 void type으로 선언되어 있으며, argument로 id와 decl을 받는다. stack을 현재 scope에서 top down search하여 중복되는 id를 확인하고, argument로 받은 id에 대해서는 redeclaration error message를 출력한다.

(1) Undeclared variable & function

variable을 사용할 때마다, unary -> ID reduce에서, scope stack에서 해당 ID에 해당하는 decl을 찾고, 만약 NULL을 return하면 match되는 decl이 없는 것이므로 변수가 선언되지 않은 것이다. 따라서 이 경우 not declared message를 print한다. 따라서 모든 ID에 대해 stack에 없으면 decl되지 않은 것이고, stack에 있으면 decl된 것임을 확인할 수 있다.

(2) Redeclaration in same scope

위에서 서술한 declare 함수에서 이를 검사하도록 하였다. declare 함수는 일단 주어진 id와 decl에 대해, scope stack에서 현재 scope 내에서 같은 id가 있는지 top down searching한다. 이는 scope stack과 ste stack에서, top 이전의 node가 가리키는 node 바로 위의 ste node까지 searching하도록 하여 구현하였다. struct의 경우 type과 같이 정의시에 ste stack의 바닥에 추가되도록 하였으므로 decl의 declclass를 확인하여 4이외의 경우에만 redeclaration message를 출력한다. 이후 redeclaration이 아니면 insert를 통해 stack에 새로운 decl과 id를 삽입한다.

2) pointer operation

\*, & operation에 대해서 구현해야 한다. pointer와 array를 type으로 지정하고, typeclass를 각각 할당하였다. pointer의 경우 \*이 있으면 pointer로, 없으면 일반 variable로 한다. pointer이면 ID에 해당하는 variable decl의 type이 pointer decl을 가리키게 된다. 이 pointer decl에서의 ptrto를 통해 int, char, void, struct의 type을 알 수 있다.

pointer decl는 makepointerdecl을 통해 만들어진다. pointer는 ==과 !=의 equation operation만 허용된다. 따라서 equation operation에 해당하는 binary EQUOP binary 에서 type의 typeclass가 3인 경우 pointer이다. 이경우 ptrto를 통해 원래 type을 비교하여 not comparable message를 출력한다.

\*는 pointer를 선언하는데 사용되고, pointer variable의 값을 dereference할 때 사용된다. reference\_ptr()함수를 통해 이를 구현하였다. reference\_ptr함수는 새로운 VAR의 decl을 만들어 type을 지정하고 return한다.

&는 variable의 주소를 참조할 때 사용된다. addpointer 함수를 통해 이를 구현하였다. addpointer는 variable에 해당하는 type에 맞는 pointer type을 새로 만든 후, variable의 type을 pointer type으로 지정하여 return한다. 이때 declclass를 1로하여 CONST로 지정하고, variable이 아니면 not variable을 출력한다.

3) struct

(1) struct declaration

struct는 declaration할 때 type과 같이 stack의 바닥에 추가되도록 하였다. struct\_specifier -> STRUCT ID reduction에서, declare\_struct 함수를 이용해 구현하였다. declare\_struct 함수는 stack의 bottom node 다음 node로 추가하여 stack 바닥에 struct type을 추가하도록 하였다. redeclaration의 경우 struct declare에서 검사하는데, 이는 check\_struct\_isdefined를 통해 구현하였다. top down searching으로 matching되는 struct를 찾는다. int type아래에 있는 ID만 searching함으로써 struct의 redeclaration만을 확인할 수 있다. 이후 declare되는 struct는 모두 bottom에 추가되기 때문에 나중에 추가되는 redecl된 struct는 사용될 일이 없다. fieldlist에 pop\_scope를 통해 field를 추가한다. pop\_scope는 ppt와 같이 scope를 pop하면서 나온 decl의 linked list로 reverse되어 return된다.

(2) struct operation

‘.’과 ‘->’의 경우 struct operation이다. ‘.’은 struct variable의 field list를 search하여 해당 ID를 가진 variable이 있는지 확인한다. 이후 variable이 발견되지 않으면 error message를 출력한다. ‘->’의 경우 pointer에만 사용될 수 있으며, pointer type의 ptrto를 통해 원래의 struct를 알아낸다. 이후의 과정은 ‘.’과 같다. 이는 reference\_ptr, reference\_struct를 통해 구현하였다.

4) function

(1) formal argument and actual argument

function의 declare시에, push scope와 pop scope를 통해 function decl의 formals에 linked list로 formal을 추가한다. add\_formals 함수를 통해 이를 구현하였다. 이번 project에서는 declare와 define을 구분하지 않으므로, formal을 그대로 추가하였다. 이때 pop scope에서의 첫번째 formal은 return 변수로 저장하고, 이후의 formal을 저장한다.

function call에서, unary ‘(‘ args ‘)’에서 args를 unary의 formals와 비교한다. 이때 각각의 formal에 대해 type check을 하고, 만약 type이 다르거나 개수가 다르면 error message를 표시한다. args는 args ‘,’ expr에서 list로 연결되어 있다.

(2) return type checking

lookup\_func 함수를 통해 stack을 top down search하여 가장 처음 찾아진 function이 해당 return에 대응되는 function이다. 이는 nested function에서 return을 사용할 수 없기 때문이다. 따라서 해당 function의 return과 실제 return type의 compatibility를 check한다. 둘다 variable인 경우, const인 경우, pointer인 경우, 한쪽만 array pointer인 경우로 case를 나누어 비교하고, 모두 충족하지 않으면 error message를 출력한다.

5) Lvalue check, operand check

Lvalue와 operand check은 모두 type checking이다. Lvalue의 경우 assign이 발생할 때, ‘=’ token이 사용될 때 검사한다. LHS와 RHS에서 RHS가 variable이거나 const인지 확인하고, LHS가 variable인지 확인한다. 또한 LHS와 RHS의 type을 확인하는데, 이 경우도 function의 return type checking과 같이 case를 나누어 확인한다.

operand checking의 경우 각 operand에 대해 제한된 type에 해당하지 않으면 error message를 출력하도록 하였다.

**3. 문제점**

1) declaration

Definition의 경우 grammar 특성상 function의 시작에서만 variable을 정의할 수 있다. Struct는 이와 별개로 항상 global하게 정의된다. 따라서 subc의 경우 제한적인 source code만 parsing할 수 있다. Function의 경우 nested function 정의가 허용되는 것은 block statement가 없는 declaration만 있는 function이다. 따라서 local function은 쓸모가 없게 된다. 따라서 prototype을 미리 정의하고, 이후에 definition을 하는 일반적은 function을 parsing할 수 없다. 이 경우 redeclaration error가 발생하기 때문이다.

Declaration에서 stack을 사용하는데, 이때 stack의 자료구조 특성상 search에 시간이 크게 소모된다. 또한 variable이나 struct 등이 많아지면 stack이 overflow되어 compiler 자체에 error가 발생할 수 있다.

2) pointer

본 프로젝트에서는 pointer를 하나만 사용하였지만, 여러 개를 사용하여도 구현할 수 있다. Makepointerdecl 함수에서 pointers token에서의 int value, 즉 pointer의 개수에 따라 type을 선언해 주면 된다.

3) struct and function

struct와 function의 경우 제한사항에 대한 문제점이 크게 없다. function의 1)에서 서술했듯 nested declaration이 불가능한 것과 declaration과 definition을 분리할 수 없는 문제가 있다.

4) type checking

본 프로젝트에서는 각 operator에 대해, type의 제한이 주어졌다. 예를 들어 ++나 --는 char와 int type만 허용되고, binary +, -는 int만 허용된다. 실제 C에서는 pointer에 대한 operation이 자주 발생하는데, pointer는 equation operator에서만 사용 가능하다. 또한 const address에 대한 operation도 불가하기 때문에, array에서 index를 참조하기 위해서는 []를 이용해야만 한다. C에서의 pointer를 제대로 활용하지 못하는 문제점이 있다.