CS 241 – Week 10 Tutorial Solutions

Semantic Analysis

Spring 2015

Declaration Errors

Error detection

1. The variable 'y' is declared twice. This is a semantic error.
2. The variable 'idx' is not declared at all. This is a semantic error.
3. In WLP4, all variable declarations must proceed all statements. This is a syntax error, since the WLP4 grammar forces this structure.
4. In WLP4, all procedure declarations must also be procedure definition. This is a syntax error.
5. It may at first appear that there should be a semantic error when assigning the output of a function that returns a float to a variable of type int, however this is perfectly legal in C. In fact, the type of constants '3.0' and '4.4' are actually double, so when you return the result of 'a * 3.0' or pass '4.4' as a parameter to 'triple', there is an implicit narrowing conversion to type float! The error actually occurs in two places: when trying to assign the address of 'a' into 'y', and when trying to dereference 'y'. This is because the type of 'y' is actually int, not int*, as it may first appear. This is a semantic error.
6. The function 'getRandom()' is not declared. This is a semantic error.

In the assembler, we needed to do two passes because labels could be used before they were declared. In WLP4, we require *declaration before use*. That is, we know immediately that it is an error to see an unknown identifier in any given variable use or function application. In fact, the WLP4 grammar takes care of the former for us!

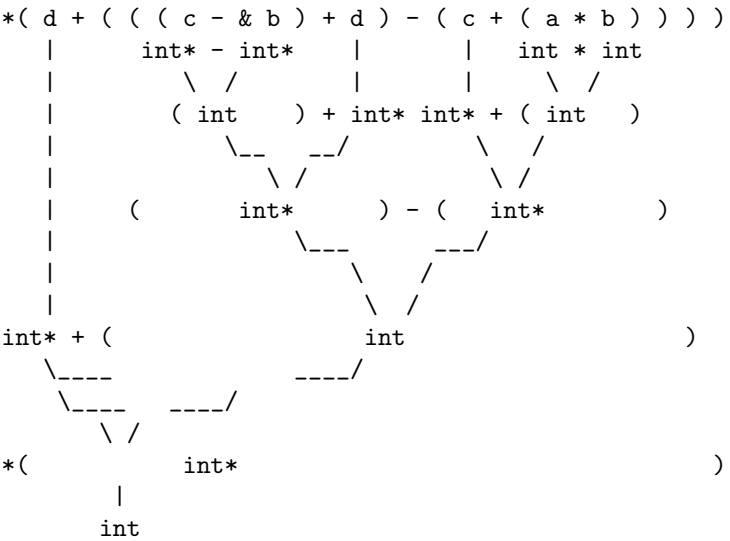
Type Errors

Error Detection

1. For an assignment statement to be well-typed, we need: - LHS and RHS are both well-typed - LHS and RHS both have the same type

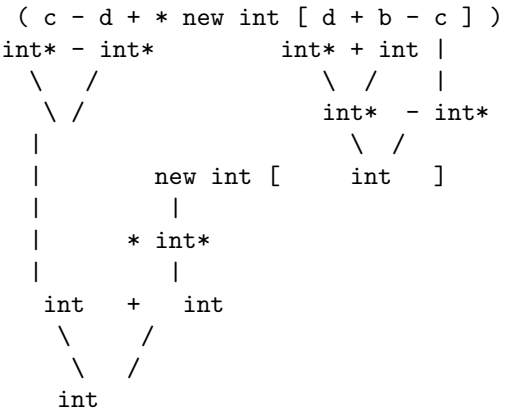
   Let's consider the left-hand side first. `*(d+(((c-&b)+d)-(c+(a*b))))`

We will find the types of the innermost expressions first and use them to build up the type of the full expression. We can draw this as a tree.



So the left hand side is well-typed, and its type is int.

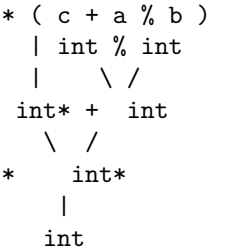
Now consider the right hand side:



The right-hand side is also well-typed, and its type is int. Therefore, the entire statement is well-typed.

2. We need to consider the types of the four expressions, then check if the types are appropriate for the statements they are contained in.

   The type of `*(c+a%b)` is:



   The type of `(&a-&b)` is int* - int* = int.

Therefore we have if (int < int), which is valid because a boolean test requires both things being compared have the same type. So the if statement is well-typed.

The type of `&*&*c-(&b)` is:

```
& * & * c - ( & b )
    * int*    |
     /        |
  & int       |
   /          |
  * int*        |
   /            |
& int            |
 /               |
int*     -    int*
  \__    __/
     \  /
     int
```

println requires an int argument, so the println is well-typed.

The type of `*d+&a-c` is:

```
* d + & a - c
int + int*
   \ /
   int*   - int*
     \   /
    int
```

However, the delete [] operator is for deleting arrays, so it requires an int* type. Thus the statement `delete[]*d+&a-c` is not well-typed.

3. This is relatively straightforward in comparison to the previous two examples.

   The type of a is int.

   So we need to compute the type of `foo(a,&*c + 1)`. But foo is a procedure and procedures always return int (or they should but that would have been caught before we got here). And we know int * int is well-typed. Good, we're done!

   ...Not quite. We need to ensure the types of the parameters passed to foo correspond to the correct types, i.e. we need to ensure the call is well-typed.

   The type of the first parameter, a, is int which correspond to foo's first parameter.

   The type of the second parameter, `(&*c)+1`, is:

```
(& * c) + 1
  |
   int*  +  int
   \       /
     int*
```

   But foo expects an int as the second parameter. So the call is not well-typed.