CS 241 – Week 9 Tutorial Solutions

Semantic Analysis

Spring 2017

1 Errors in WLP4 and C

1.1 Errors in WLP4

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. We don’t have character literals (in single quotes) in WLP4. This is a syntax error.
5. We have a duplicate definition of sub. Although they have different number of parameter, function overloading is not a feature in WLP4. This is a semantic error.
6. Function f is declared before function g, therefore f has no knowledge about g’s existence. Note that this prevents the use of mutual recursion in WLP4. This is a semantic error.

1.2 Errors in C

1. 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.
2. The function ‘getRandom()’ is not declared. This is a semantic error.

2 Symbol Tables and Type Checking

2.1 Symbol Tables

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 if we see an unknown identifier used as a variable or as a function call. It also means we can build our symbol table in one pass.
2.2 Type Checking

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.

\[
\begin{align*}
\text{int*} & \rightarrow \text{int* int*} \\
\text{int*} & \rightarrow \text{int* + ( int* )} \\
\text{int*} & \rightarrow \text{int* - ( int* )} \\
\text{int*} & \rightarrow \text{int* + ( int* )} \\
\end{align*}
\]

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

Now consider the right hand side:

\[
\begin{align*}
\text{int*} & \rightarrow \text{int* int*} \\
\text{int*} & \rightarrow \text{int* - ( int* )} \\
\text{int*} & \rightarrow \text{int* + ( int* )} \\
\text{int*} & \rightarrow \text{int* - ( int* )} \\
\end{align*}
\]

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:

\[
\star \ ( \ c + a \ % \ b ) \\
| \ \text{int} \ % \ \text{int} \\
| \ \text{\textbackslash} / \\
\text{int}* + \text{int} \\
\text{\textbackslash} / \\
* \ \text{int*} \\
| \\
| \ \text{int}
\]

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:

\[
\& \ * \ & \ * \ c - ( \ &b ) \\
* \ \text{int*} \\
| \\
\& \ \text{int} \\
| \\
\text{\textbackslash} / \\
* \ \text{int*} \\
| \\
\& \ \text{int} \\
| \\
\text{int*} - \text{int*} \\
\text{\textbackslash} \_ \_ / \_ \_ /
\]

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

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

\[
\star \ d + \ &a - c \\
\text{int} + \text{int*} \\
\text{\textbackslash} / \\
\text{int*} - \text{int*} \\
\text{\textbackslash} / \\
\text{int}
\]

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

3. We follow the same strategy as we did in the previous questions, but this time we added in a procedure call. A procedure call is well-typed if the call matches the procedure’s signature (the number and type of arguments are the same). In WLP4, all functions return an int.

The first part we verify is foo(a,b). Since both a and b are int*ns and foo takes as parameters two int*ns, therefore this part is well-typed. The expression itself is of type int.
Next, we take a look at \( \text{foo}(\ast c, \text{foo}(\&a-\&b,\ast(c-a))) \).

\[
\text{foo}( \ast c, \text{foo}( \& a - \& b, \ast (c - a)))
\]

Since both \text{foo} calls have two int arguments, they are well-typed, and thus \( \text{foo}(\ast c, \text{foo}(\&a-\&b,\ast(c-a))) \) is well-typed.

Since this and \text{foo}(a,b) are well-typed int, thus the entire expression is well typed.