1 Designing a Context-Sensitive Analyzer

The purpose of context-sensitive analysis in a compiler is to catch all remaining compile-time errors in the input programs that were not caught by the scanner or parser.

In WLP4, these errors fall into two categories – name errors and type errors. Name errors are things like duplicate declarations, or use of declared identifiers outside the proper scope. The process of finding name errors is called name resolution, and it involves building a symbol table containing data about all procedure and variable declarations. Type errors are things like returning the incorrect variable type from a procedure, or using the addition operator with types that can’t be added (e.g., two pointers). The process of finding type errors is called type checking.

Both name resolution and type checking are done by traversing a parse tree representing the input program. The parser tree is the output of the parser you built in Assignment 7.

1.1 The Parse Tree

In Assignment 7, you built a parse tree for the input WLP4 program and output a representation of the parse tree called a .wlp4i file. Your first task in Assignment 8 is to read in the .wlp4i file and reconstruct the parse tree. It may seem silly to have the parser build a parse tree and have the context-sensitive analyzer immediately rebuild the parse tree, but this allows us to keep Assignment 8 independent of Assignment 7.

If you solved Assignment 3 Problem 1, where you had to build a tree by reading in a preorder traversal of the tree, you can use the same technique here. A .wlp4i file is essentially just a preorder traversal of a parse tree. The main difference is that in A3P1 you were given the number of children of each node; for a .wlp4i file you need to infer the number of children as follows. For a line containing a rule, there is one child for each symbol on the right-hand-side of the rule; lines containing terminal tokens represent leaf nodes and have no children.

Here are some suggestions for fields and methods that your parsetree class should have:

- For nonterminal nodes, the production rule used to expand the node.
  Frequently your context-sensitive analyzer will need to make decisions based on the rule.
- Optionally, a way of looking at specific parts of the rule, so you don’t need to type out the whole rule when doing comparisons.
  For example, you could store the left hand side and right hand side as separate strings, or you could store a sequence-based representation of the rule so you can ask things like “what is the symbol at index 3”. (Or you could do both if you find both useful!)
- For terminal nodes, the kind and lexeme of the token stored at the node.
• A sequence of child subtrees.

The data structure used for the sequence should let you access arbitrary elements quickly, like a vector. It should also keep things in the order you inserted them, because you will need to iterate over the children from left to right when traversing the tree. For example, don't use a map which rearranges the order of things.

• Optional but very useful: a function child for looking up children by name.

If you type \texttt{child(name)} it should return the first occurrence of a child with that name (where the “name” is the left-hand-side of the rule for nonterminal nodes, or the kind of the token for terminal nodes). If you type \texttt{child(name,n)} it should return the \texttt{n}-th occurrence of a child with that name.

This makes things easier to read than if you always access children by numeric index (you can write \texttt{tree.child("expr"}) instead of \texttt{tree.children[7]}). Lookup by string name probably will be slower than lookup by index no matter how you implement it, but the readability improvement is arguably enough to outweigh the small loss in efficiency.

• A function \texttt{getType} that computes the type of the tree, if it has a type (not all trees have a type according to the specification).

You should also cache the type after it is computed, and simply return the cached type if \texttt{getType} is called on the tree again, to avoid redoing work.

1.2 The Symbol Table and Name Resolution

The purpose of the symbol table is to keep track of types of variables and type signatures of procedures, and to help you handle scoping issues. Aside from its use in name resolution, it will also be used to look up variable types and procedure signatures when you do type checking, and it will be extended to include additional information when you write the code generator.

In class, the following data structure was recommended for the symbol table:

\[
\text{map\{string, pair<vector<string>, map<string, string>\}}
\]

This looks complicated. Let's break this down into pieces.

\textbf{VariableTable: map\{string, string\}}

A variable symbol table is a map from strings (variables) to strings (types).

\textbf{Signature: vector<string>}

A procedure signature is a vector of strings (types).

\textbf{ProcedureTable: map\{string, pair<Signature, VariableTable>\}}

A procedure symbol table is a map from strings (procedure names) to pairs (signature and local variable table for the procedure). This is the data structure we're using for the symbol table.

If you wrap this data structure in a class with various helper functions for working with the symbol table, it will make your code easier to read and write.

Here are some suggestions for operations your symbol table class should support. In the descriptions below, the “current procedure” refers to a global variable which is updated as you traverse the tree and build the symbol table. If you wish you can make your methods take an extra procedure argument to make them more generic, rather than making them always use the “current procedure”, but this isn’t really necessary due to the simplicity of WLP4.
• \texttt{isAccessibleVariable(name)}: Returns true or false based on whether variable \texttt{name} is accessible in the scope of the current procedure.

• \texttt{isAccessibleProcedure(name)}: Returns true or false based on whether procedure \texttt{name} is accessible in the scope of the current procedure.

This is a bit trickier to implement correctly than you may think – we’ll talk about this later.

• \texttt{addProcedure(tree)}: Given a tree representing a procedure, extract the name and signature of the procedure and add it to the procedure table. Also, update the current procedure to this procedure.

• \texttt{addVariable(tree)}: Given a tree representing a variable declaration, extract the name and type of the variable and add it to the variable table of the current procedure.

• \texttt{lookupVariable(name)}: Returns the type of the variable \texttt{name} in the context of the current procedure. This should only be called if you know the variable is accessible in the current procedure.

• \texttt{lookupProcedure(name)}: Returns the signature of the procedure \texttt{name}. This should only be called if you know the procedure has been added to the procedure table.

• \texttt{printProcedure(name)}: Print the procedure name, signature, and variable table for a particular procedure.

• \texttt{printTable()}: Print the entire procedure table by calling \texttt{printProcedure} for each procedure in the table.

To fill out the symbol table, traverse the parse tree (processing children from left to right) and proceed as follows:

• When you encounter a rule with \texttt{procedure} or \texttt{main} as the left hand side, this is a procedure declaration. Pass this subtree into \texttt{addProcedure}.

• When you encounter a rule \texttt{dcl type ID}, this is a variable declaration. Pass this subtree into \texttt{addVariable}.

• When you encounter a rule \texttt{factor ID} or \texttt{lvalue ID}, this is a variable use. Pass the lexeme of the ID into \texttt{isAccessibleVariable}. If it is not accessible, this is an error.

• When you encounter a rule \texttt{factor ID LPAREN RPAREN} or \texttt{factor ID LPAREN arglist RPAREN}, this is a procedure call. Pass the lexeme of the ID into \texttt{isAccessibleProcedure}. If it is not accessible, this is an error.

You must do this in a single pass through the parse tree. If you do multiple passes, it will be difficult to catch errors involving things being used before they are declared. With a single pass, these errors will be caught naturally, because you won’t reach variable uses/procedure calls until after processing declarations.

Finally, here are some hints for implementing the methods in the symbol table.

• For \texttt{isAccessibleProcedure}, the tricky part is handling programs like the following:

```c
// Valid program
int sum(int* a, int n) {
    int sum = 0;
    while(n > 0) {
        n = n-1;
        sum = sum + *(a+n);
    }
    return sum;
}
int wain(int* a, int n) { return sum(a,n); }
```
// Invalid program
int sum(int* a, int n) {
    int sum = 0;
    if(n > 0) {
        sum = *a + sum(a+1,n-1);
    } else {}
    return sum;
}
int wain(int* a, int n) { return sum(a,n); }

These two programs share a common feature: they use a local variable which has the same name as the procedure containing it! This is allowed, and sometimes it even sort of makes sense, like having a local sum variable in a procedure called sum.

The first program is perfectly fine, but the second one is invalid because of this line:

\[
\text{sum} = *a + \text{sum}(a+1,n-1);
\]

The WLP4 specification states that when a local variable has the same name as its enclosing procedure, all occurrences of the name refer to the local variable, rather than a procedure. So this line is incorrect because we are trying to call \text{sum} but \text{sum} refers to a variable rather than a procedure in this context.

To handle this case in your implementation of isAccessibleProcedure, implement it as follows:

- First use isAccessibleVariable to check if the current procedure has a local variable with the same name as the passed-in name. If so, return false; the procedure is not accessible in this context because it has been overridden by the local variable.
- If there is no local variable with the passed-in name, simply check if the procedure exists in the procedure table.

As long as you follow the advice mentioned earlier of doing only one pass over the parse tree, this will be enough to detect all errors involving calling undeclared or out-of-scope procedures. If you do multiple passes then implementing isAccessibleProcedure as described above might fail to detect use-before-declaration errors.

- When implementing addVariable and addProcedure, check if the variable or procedure already exists in the table – if it does, produce a duplicate declaration error. Don’t use isAccessibleProcedure to check if a procedure exists in the table, due to the extra logic described above – do a direct check. Using isAccessibleVariable to check if a variable already exists should be okay.
- The addProcedure method will probably require two cases: one for wain, and one for other procedures. Also, addProcedure is a good place to update the “current procedure” global variable, since you should only call it when you start processing a new procedure.

1.3 Type Checking

Type checking will probably require more lines of code to write than name resolution, but it is arguably more straightforward. There are two components to type checking:

- Computing the type of subtrees that have a type according to the specification, such as expression subtrees, and producing an error if the type cannot be computed.
- Verifying that subtrees which do not have a type, such as statement subtrees, are well-typed according to the specification, and producing an error if they are not.

Computing the types of subtrees with a type is done using the getType function that was suggested for your parse tree class.
According to the specification, instances of the tokens ID, NUM, NULL and the nonterminals factor, term, expr, and lvalue have a type. So any subtree consisting of one of those terminal tokens, or a subtree with one of those nonterminals on the left hand side of its rule, will have a type.

The WLP4 specification gives a big list of type rules in English. Your getType function should basically have a case for each of the type rules involving the terminals and nonterminals listed above. Let us look at a few examples.

- The type of an ID is int if the dcl in which the ID is declared derives a sequence containing a type that derives INT.
- The type of an ID is int* if the dcl in which the ID is declared derives a sequence containing a type that derives INT STAR.

We would implement this in getType by looking up the ID in our symbol table:

```java
getype(tree) {
    if(tree.type != "") { return tree.type; }
    type = "none";

    if(tree.kind == "ID") {
        type = symbolTable.lookupVariable(tree.lexeme);
    }

    tree.type = type;
    return type;
}
```

Aside from the symbol table lookup, take note of the start and end of the getType function. At the end, we store the computed type in the tree before returning it. Then at the start, if the type has already been computed, we simply return it immediately. Caching the type like this makes sure we don’t redo our work if we ever call getType on a tree whose type has already been computed.

Next let’s look at a nonterminal type rule:

- The type of a factor deriving AMP lvalue is int*. The type of the derived lvalue (i.e. the one preceded by AMP) must be int.

Here’s how we would add this rule to getType:

```java
getype(tree) {
    if(tree.type != "") { return tree.type; }
    type = "none";

    if(tree.kind == "ID") {
        type = symbolTable.lookupVariable(tree.lexeme);
    } else if (tree.rule == "factor AMP lvalue") {
        if(tree.child("lvalue").getType() != "int") {
            output "ERROR: operator & expects int operand" to standard error
        }
        type = "int*";
    }

    tree.type = type;
    return type;
}
```

5
We check for the production rule described in the type rule. Then we recursively call \texttt{getType} to enforce the condition that the type of the operand is \texttt{int}. This recursive call will fail right now because there are no cases for handling \texttt{lvalue} nodes. But once we implement all the type rules here, the recursive call will compute the type of the \texttt{lvalue}, and we can use the computed type to determine if the \texttt{factor \&\& lvalue} node is well-typed. If it is well-typed, we return \texttt{int*}.

Once you get the general idea it’s just a matter of translating the English type rules into code. Probably the most difficult one is \texttt{factor ID LPAREN arglist RPAREN}. You need to figure out the sequence of types specified by \texttt{arglist}, and compare it with the signature of the procedure specified by \texttt{ID}. The sequence of types must exactly match the signature: it must have the same length, and all the types must match up. The \texttt{expr expr PLUS term} and \texttt{expr expr MINUS term} are also a little complicated because they have several different cases to deal with, but they follow the same principle as the example above.

For Assignment 8 Problem 5, make sure \texttt{getType} on every tree that has a defined type, or you might miss some type errors. An easy way to ensure this is to just traverse the whole tree and call \texttt{getType} on every subtree. This is obviously inefficient, although if you cache the computed type as suggested it won’t be too bad. A more efficient way is to traverse the tree until you find one of the top-level \texttt{expr} or \texttt{lvalue} nodes occurring in statements, and just call \texttt{getType} on these; then \texttt{getType} will be recursively called on all subexpressions as part of computing the type of the top-level \texttt{expr} or \texttt{lvalue}.

One final note: although trees with the rule \texttt{dcl type ID} technically don’t have a defined type in the specification, you may want to make \texttt{getType} compute the type associated with the declaration anyways. This will help with the next part of type checking, and it will even help with building the symbol table since that requires you to extract type information from \texttt{dcl type ID} trees.

Now we can move onto the second part of type checking: subtrees that don’t have types themselves, but need to be checked for well-typedness. These are described after all the other type rules in the WLP4 specification, starting with:

- The second \texttt{dcl} in the sequence directly derived from \texttt{main} must derive a \texttt{type} that derives \texttt{int}.

To check this, simply do a tree traversal, and when you see a relevant production rule, check the condition using \texttt{getType}. For example, for the above condition you could write something like this:

```cpp
if(tree.lhs == "main") {
    if(tree.child("dcl",2).getType() != "int") {
        output "ERROR: second parameter of main should be int type" to standard error
    }
}
```

This shows how handy the suggested function for looking up children by name in the tree is – otherwise you would have to look at the rule and count tokens and write something like \texttt{tree.children[5].getType()}; and it wouldn’t really be clear what you are getting the type of.

Here is another example:

- Whenever \texttt{test} directly derives a sequence containing two \texttt{exprs}, they must both have the same type.

The code could look something like this:

```cpp
if(tree.lhs == "test") {
    leftType = tree.child("expr",1).getType();
    rightType = tree.child("expr",2).getType();
    if(leftType != rightType) {
        output "ERROR: mismatched types in conditional test" to standard error
    }
}
```
The final difficulty is making sure that every part of the tree which needs to be type-checked actually gets
type-checked. One way handle this is to visit each node, but again this is less efficient than it could be. A
better but more error-prone way is to explicitly recurse on the nodes that need to be checked. For example,
if you are looking at the main rule you could do this:

```c
if(tree.lhs == "main") {
    // check the second dcl and the return expr
    // now recurse on the parts that need checking: the dcls and statements
    typeChecking(tree.child("dcls"));
    typeChecking(tree.child("statements"));
}
```

Or if you are looking at one of the dcls rules you could do this:

```c
if(tree.lhs == "dcls") {
    // check the type of the derived dcl
    // now recurse on the rest of the dcls
    typeChecking(tree.child("dcls"));
}
```

You have to be careful that you actually cover everything if you use this method, but it avoids looking at
unnecessary nodes.

**Appendix: Tree Building and Tree Traversal**

If you did not solve A3P1, you might have difficulty figuring out how to build the parse tree for this problem.
The .wlp4i format is a preorder traversal of the parse tree. There is actually a very simple algorithm for
building a tree from a preorder traversal of the tree on standard input, given below in pseudocode.

```c
buildTree() {
    read the root node from standard input
    n = the number of children the root node should have
    repeat n times {
        child = buildTree()
        add child to children of root node
    }
    return the tree
}
```

For this problem, you determine the number of children by looking at the number of symbols on the right
hand side of each rule for nonterminal nodes. Terminal nodes have no children. Make sure to properly
handle the case of rules with nothing on the right hand side (the node should have no children).

In this tutorial, there are many mentions of “tree traversals”. Here is pseudocode for traversing a parse tree
and performing an action depending on the rule at the root:

```c
traverseTree(tree) {
    if(tree.rule == "some production rule") { do something }
    else {
        for each child subtree {
            traverseTree(child)
        }
    }
}
```