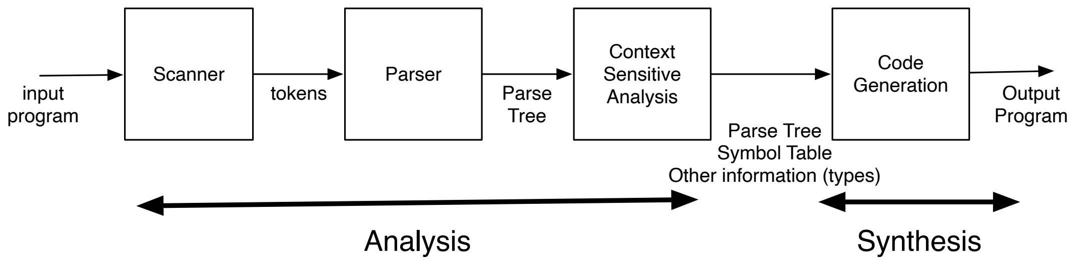
Compiled on 21/10/2022 at 2:04pm

Context-Sensitive Analysis

We begin by situating ourselves with regards to where we are in our journey through writing a complete compiler. We began by learning about regular languages and used that knowledge to implement the scanner. Next, we learned about context-free languages and parsing algorithms that take the sequence of tokens produced by scanners and generate a parse tree representing the structure of the input program. We are now going to learn about the last step in the *analysis* stage of the compiler: the stage that checks high-level language requirements that go beyond syntax. These requirements need awareness of context, i.e., where in the program we currently are, and therefore cannot be checked using a context-free language.



In this module, we discuss the context-sensitive analysis requirements of a small language. Once a program has successfully gone through this stage, it is known to satisfy all requirements of the language. Our next module will discuss the synthesis stage, where the compiler will generate the compiled version of the program.

1 Intro to WLP4

Until now we were successful in discussing the theory and algorithms for scanning and parsing without having to restrict ourselves to any one language. While this approach is preferred, we find ourselves in a situation where we must now begin to talk about specific language features and rules that a compiler must support. Doing this at a high level, without discussing a specific language, is not very useful. Therefore, in this section, we give an introduction to a small language named `WLP4` and then discuss the context-sensitive analyses needed for this language.[1]

`WLP4` has a convoluted history. From what we know, it started as `WL`, Waterloo Language. At some point, pointers were added to the language, and we called this `WLPP`, Waterloo Language Plus Pointers. Later, procedures were added, and the language became `WLP4`, Waterloo Language Plus Pointers Plus Procedures.

`WLP4` programs are a sequence of C++ functions that could be executed if an appropriate `main` function was provided. Instead of a `main` function, valid WLP4 programs must have a special `wain` procedure. This is enforced by the context-free grammar, i.e., a `WLP4` program will not pass through the parsing stage unless an appropriate `wain` function has been defined.

[1]You have likely already been introduced to the language through assignments.

A procedure in WLP4 (except for wain) can have an arbitrary, but fixed, number of parameters. Parameters, and other variables, can have one of two types: `int` or `int*`, i.e., an integer type or a pointer to integer type. The `wain` procedure must always have two parameters, the second of which must be an `int`. This corresponds to our two simulators, `mips.twoints` and `mips.array`, both of which provide two arguments to the program in $1 and $2, and both of which have a number as the second argument.

The language supports `while` loops, an `if` statement (an `else` clause, even if empty, is always required) and a `println` statement to generate output. A specialized `return` statement is also supported which must appear only once in a procedure as the last statement. This requirement is also enforced by the context-free grammar, i.e., an attempt to write a procedure with more than one `return` statement in the procedure or without a `return` statement as the last statement in the procedure will lead to a parser error. The language also supports using heap memory through a `new` expression and a `delete` statement.

We encourage readers to refer to the following two links for more details on the language: a guide to using WLP4 for C/C++ programmers and the WLP4 Language Specification.

2 Context-Sensitive Analyses in WLP4

As discussed earlier, not all language rules can be enforced by a context-free grammar (and therefore a parser). Perhaps the most important of these are rules regarding types; in a statically typed language, we must ensure that type rules are not violated. For instance, if a variable is defined to be an `int`, we cannot store an address in this variable. Since a discussion of type-checking is lengthy, we will leave this till later in the module.

Another set of rules that must be enforced are those surrounding variables. Two rules that are common in many languages (including WLP4) are:

1. We cannot declare more than one variable with the same name in the same scope.
2. A variable cannot be used before it has been declared.

There is a class of formal languages called context-sensitive languages, which are more powerful than context-free languages. In theory, we could use these languages and the corresponding context-sensitive grammars to formally define such rules. However, a much easier and more practical option is to use a code-based solution to traverse the parse tree to gather information and enforce rules that require context-sensitive information. This means that it is worth taking some time to set up infrastructure to enable easy traversal of the parse tree. The following is a sketch of what this could look like in C++.

```
class Tree{
  public:
    string rule; // e.g. expr expr PLUS term
    vector<Tree *> children;
};
```

A traversal routine would look like:

```
void doSomething(const Tree *t){
  // do something before visiting children

  for(Tree *child: t->children){
    // visit the children
    doSomething(child);
  }

  // do something after visiting children
}
```

The `Tree` class is rudimentary, e.g., notice that finding the type of a node requires string comparisons over the `rule` field. A self-respecting compiler written in any object-oriented language would at the bare minimum leverage inheritance to create different subclasses of the `Tree` class (e.g., a different class for each construct such as `while` and `if`) and also leverage design patterns such as the *Visitor* design pattern to create mechanisms to visit tree nodes in a specific order. We also intentionally omit discussing where the `Tree` nodes are located (stack vs. heap) since that discussion is orthogonal to the topic at hand. Suffice it to say that if `Tree` nodes are heap allocated you would need appropriate memory management to ensure the program does not leak memory.

As an aside, the parse tree created by a parser is often called a *concrete syntax tree*. Often, this parse tree is passed through a tree transformation stage before applying context-sensitive analyses[2]. During this stage, the tree is pruned by removing useless nodes such as those needed to ensure that the grammar is unambiguous and to satisfy the requirements of a specific parsing algorithm. This transformed tree is often called an *abstract syntax tree*, or AST for short.

2.1 Duplicate Identifiers in `WLP4`

There are two types of identifiers in `WLP4` programs: variable and procedure names. The rules surrounding both types of identifiers are the same. We talk about variables first and then discuss procedures.

For variables in `WLP4`, we need to check that there are no duplicate variables within any procedure and that all variables that are ever used have been declared. This should sound familiar; we did something similar when implementing the MIPS assembler. Our solution is also the same: we need a symbol table for variables! In the MIPS assembler, we mapped each label to a corresponding memory address. In `WLP4`, we will map each variable to its type (since we know we will need type information later). In C++, this data structure could be as simple as a `map` defined as `map<string, string> symbolTable;` where the key is the variable name and the value the type defined for this variable stored as a string, but a better implementation would avoid string comparison for types and thus use something other than a `string` for them.

[2]Alternatively, hand-written parsers can typically bypass the concrete syntax tree stage entirely and output a transformed tree directly.

Before we go on to discuss more details, let's look at some special examples. Should the WLP4 compiler generate an error for the code shown below on the left?

```
int f() {
  int x = 0;
  return x;
}
int wain(int a, int b) {
  int x = 0;
  return x;
}
```

The answer to the above question is no. The compiler should not produce an error since this code is legal. In particular, even though variable x is defined twice in this program, it is **defined only once within each procedure**. The rule regarding variables says that we cannot have duplicates **within** a procedure. This means that we will need to create a separate symbol table for each procedure since, while MIPS labels were global, variables in WLP4 are local to the scope they are declared in.

The context-sensitive analysis (our traversal) can compute this as a single data structure by mapping each procedure name (the key) to its symbol table (the value), e.g.,

```
map<string, map<string, string>> tables;
```

Again, ideally using a better datatype than string for storing variable information.

Let's now look at some other examples. Is the following program a valid WLP4 program?

```
int f() {
  int x = 0;
  return x;
}
int wain(int a, int b){
  return x;
}
```

No, this program is not a valid program and should cause an error; the variable x is **not** defined in procedure wain. This means that while detecting duplicates we will need to ensure that we are looking for variables using the symbol table for **that** procedure (things would be a little complicated if WLP4 allowed global variables).

Let's look at the following example. Should this code compile?

```
int f() {
  int x = 0;
  return x;
}
int f() {
  int x = 0;
  return x;
}
int wain(int x, int y){
  return f() + x;
}
```

While the variables are fine (only one x is defined in each scope), there is a different problem. There are two procedures with the same name, f. The program to the left is **not** a valid WLP4 program since it has a duplicate function. WLP4 requires that all procedure names be unique. This means that the analysis needs to check that all functions in the program have unique names.

Now let's consider the following code. Is the following program a valid `WLP4` program?

```
int f() {
 int f = 1;
 return f + 1;
}
int g(int g) {
  return g - 1;
}
int wain(int a, int b){
  return a;
}
```

This is indeed a valid `WLP4` program. The interesting thing to note is that it is legal to have variables that have the same name as *any* procedure. In fact, the `WLP4` specification clearly addresses this and states that if a variable `x` is declared in a procedure `p`, all occurrences of `x` within `p` refer to variable `x`, even if a procedure named `x` has been declared. The code on the left is the extreme version, where the procedure `f` has a local variable also named `f` and procedure `g` has a parameter named `g`. The specification talks about this special case and states that all occurrences of the identifier refer to the variable and not the procedure.

This leads to the following interesting scenario:

```
int p(int p) {
  return p(p);
}
```

As we just discussed, it is legal for a program to use the same name for a procedure and a variable. However, consider the return expression, `p(p)`. As per the `WLP4` specification, all occurrences of the identifier `p` are treated as a variable and not a procedure even though a procedure with that name exists. This means that the code above should not compile.

Having looked at these examples, let's now discuss how we can implement the context-sensitive rules for `WLP4` using tree traversals. We begin with a traversal that starts at the root of the parse tree. We must determine when we have encountered a tree node that represents a procedure. Given our rudimentary structure (a single `Tree` class) this means we must determine when we are at a tree node whose `rule` field matches a rule for a procedure. Referring to the context-free grammar for WLP4, a tree node that has one of the following rules represents a new procedure:

$$\text{procedure} \rightarrow \text{INT ID LPAREN} \ldots\ldots$$
$$\text{main} \rightarrow \text{INT WAIN} \ldots\ldots$$

At this point, we can check for duplicate procedures; if our master map of all symbol tables (we called it `tables` above) already contains a mapping with the same name as the name of the procedure we just discovered, we know we have encountered a duplicate procedure and can generate an appropriate error.

If this procedure is not a duplicate, we need to create a new symbol table for this procedure and store it in our master map. We can then continue with the traversal inside the procedure. It is a good idea to make a note of the *currently active* symbol table, i.e., the symbol table for the procedure we are traversing. Ideally, the compiler would contain some extra infrastructure to house these symbol tables. Perhaps a class that aggregates all the symbol tables and also maintains a note of the currently active symbol table. However, since `WLP4` is simple enough, a global variable storing the currently active symbol table would also suffice.

Within procedures, variables are either declared in the parameter list of a procedure or at the start of the procedure's body. Referring to the context-free grammar for WLP4, we can observe that we only need to look for parse tree nodes where the rule is `dcl` $\rightarrow$ `type ID`. Populating the symbol table for each such declaration is simply a matter of retrieving the `lexeme` for the ID token (child 2) and the actual type from the `type` child (child 1).

Much like what we did for the MIPS assembler, we can ensure that any duplicate variables that are declared are caught at this stage; when inserting a new variable into the symbol table for a procedure, ensure that a variable with the same name has not already been added.

2.2 Checking Variable Use

In the MIPS assembler, once we had created the symbol table in pass one, we needed a separate pass which went through the code again and checked that all uses of labels referred to labels that were defined somewhere in the code. Recall that two passes were required since labels could be used *before* they were defined. This is *not* true of variables. In `WLP4`, the context-free grammar enforces that all variables must be declared at the top of the procedure, before any statements for the procedure. This means that two passes are not needed. By the time our first traversal reaches the `statements` child of the following two rules, the symbol table for this procedure is already complete, since the `params` and `dcls` children of the tree node have already been visited.

$$\begin{aligned}\text{procedure} &\rightarrow \text{INT ID LPAREN params RPAREN}\\ &\ \text{LBRACE dcls statements RETURN expr SEMI RBRACE}\\ \text{main} &\rightarrow \text{INT WAIN LPAREN dcl COMMA dcl RPAREN}\\ &\ \text{LBRACE dcls statements RETURN expr SEMI RBRACE}\end{aligned}$$

This means that we can check our "declaration before use" rule by simply traversing the `statements` subtree and the return expression looking for rules that use variables (we will talk about function calls later). The two rules of interest are `factor` $\rightarrow$ `ID` and `lvalue` $\rightarrow$ `ID`. The actual check is straightforward; when we encounter a node that represents one of these rules, retrieve the lexeme for the ID token and check that an entry with this name exists in the symbol table for the current procedure.

Notice that the restriction that all variables be declared before any other statements is intentional in WLP4 and is meant to make writing the analysis easier. In C for example, the following code is legal[3]:

```
void foo(){
  int x = 5;
  printf("%d",x);
  int y = 10;
  printf("%d",y);
}
```

Even this simple function would require a more complicated setup for tracking which variables have been declared before use. Notice that the variable `y` is declared after the first print statement. This means that had the first print statement used the variable `y`, this would have caused the C compiler to generate an error. Think about what this means for the context-sensitive analysis stage; we

[3]In fact, C used to have the same restriction as WLP4, that all variables are declared at the beginning of a scope; really, we borrowed this restriction from C. This restriction was lifted in C89 (in 1989), so is largely forgotten, but it's not unusual to find older C code that still has all its declarations at the top!

cannot create a single symbol table for the entire function first and then go checking that symbol table later to see that all the variables used were in fact defined. To further complicate things, C even allows defining variables with the same name within a function as long as their scope is different. For example, the following code is legal in C:

```
void foo(){
  int x = 5;
  printf("x in outer scope %d\n",x);
  {
    int x = 7;
    printf("x in inner scope %d\n",x);
  }
  printf("x %d in outer scope\n",x);
}
```

Notice that the variable `x` is defined twice: once in the outermost scope of `foo` and once within a short-lived inner scope. One symbol table for the entire function would certainly not work in this case; each nested scope requires its own symbol table, sort of like a stack of symbol tables. Fortunately, this too is disallowed in `WLP4`.

2.3 Checking Procedure Calls

Like variables, procedures must also be declared before use. This means that it is in fact incorrect to first collect a list of all procedures that were declared in a program and then check that each procedure call matches the name of a declared procedure; `WLP4` explicitly disallows calling a procedure that is declared later in the code. This means that the analysis must validate the names of procedures being called **during** the creation of the top-level symbol table (that we have been calling `tables`). The rules of interest are: `factor` → `ID LPAREN RPAREN` and `factor` → `ID LPAREN arglist RPAREN`. During the traversal, when a parse tree node with one of these rules is encountered, the name of the procedure being called is checked against the procedures that are known to have been declared until then. If the name is not found, the procedure has not (yet) been declared which is an error.

Calling a procedure also requires passing the correct number of arguments to the procedure. Additionally, the types of the arguments must match the parameter types. This means that when a new procedure declaration is discovered during the traversal we should store the procedure's *signature*, which is the information needed to (correctly) call it. Note that all procedures in `WLP4` return an integer value, i.e., procedures cannot return a pointer and there are no `void` types in `WLP4`. Therefore, the signature is simply a sequence representing the types for the parameters of that procedure. Continuing our simplification, we'll store the types as strings, but again, it would be more practical to use something better suited. We can store signature information in a couple of different ways. One way could be to create a different map which maps the procedure name to a vector of strings where the strings represent the types for the parameters; the size of the vector will tell us how many parameters this procedure expects. An alternate way is to update the definition of `tables`, the data structure which we had defined to map each procedure to its symbol table. We can now map the procedure name to a pair that contains a vector of strings as the first element, i.e., the signature for the procedure and the symbol table for the procedure as the second value. In C++, the definition would look like:

```
map<string, pair<vector<string>, map<string, string>>> tables;
```

This data structure is quite cumbersome, though. A better alternative would be to create a `class Procedure` that stores the signature and table for a procedure and have a map of those.

Obtaining the signature information is straightforward. The relevant subtree must either have the rule `params` $\rightarrow$, which indicates that this procedure has no parameters, or the rule `params` $\rightarrow$ `paramlist`. If it is the latter, we must traverse the children which will have the rules `paramlist` $\rightarrow$ `dcl` and `paramlist` $\rightarrow$ `dcl COMMA paramlist` to retrieve the declarations of the parameters.

As a concrete example of what the final data structure would look like, consider the following `WLP4` program:

```
int f() {
  int *a = NULL;
  return 9;
}
int wain(int a, int b){
  int x = 10;
  return x+a+b;
}
```

The symbol table for `f` would contain a mapping of $a \rightarrow$ `int *`. Similarly, the symbol table for the `wain` procedure would contain the following: $a \rightarrow$ `int`, $b \rightarrow$ `int` and $x \rightarrow$ `int`. The signature for `f` is the empty vector, written as [ ] whereas the signature for `wain` is [`int`, `int`]. The overall data structure would look like this, where $(\ldots)$ is a map, $<\ldots>$ is a pair, and $[\ldots]$ is a vector.

$$\begin{aligned}&(\ \texttt{f} \rightarrow <[\,],\ (\ a \rightarrow \texttt{int *}\ )>,\\ &\ \ \texttt{wain} \rightarrow <[\ \texttt{int},\ \texttt{int}\ ],\ (\ a \rightarrow \texttt{int},\ b \rightarrow \texttt{int},\ x \rightarrow \texttt{int}\ )>\ )\end{aligned}$$

With this added information, when our traversal reaches a use of an identifier that is a procedure, apart from checking that the procedure had already been declared, we can check the number of arguments and their types and compare them to the number and types of parameters as available in the symbol table.

3 Catching Type Errors

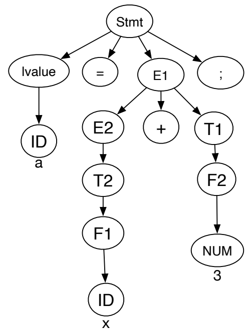
Recall the theme from Module 1; given a sequence of bits, it is not possible to say what these bits represent. The interpretation of a sequence of bits depends on the type represented. In Module 1, we discussed how the same sequence of bits could represent an unsigned integer, a signed integer in two's complement representation, or an ASCII character. Knowing the type of some data is therefore crucial. Programming languages endeavour to define sound type systems that ensure that we will never try to interpret bits incorrectly. For example, consider the following WLP4/C/C++ snippet:

```
int *a = NULL;
a = 7;
```

The type system prevents the assignment of the integer value 7 to the variable a since a is supposed to store addresses. Not preventing this would lead to disaster, e.g., at some point we might try to read from address 7.

`WLP4` only has two types: `int` and `int*`. This enables us to get a taste of what type checking entails without being overwhelmed by a huge type system. In larger programming languages, with multiple types (and implicit conversions from one type to another) the type system is more complicated. Object-oriented programming, and in particular inheritance, creates further complications. In CS 444, students implement a type system for a rather large subset of Java.

Recall that we have already collected type information for variables in each procedure. The question now is, how does one catch a type error? The diagram on the right shows the parse tree that would be generated by a `WLP4` parser for the statement `a = x + 3;`. The type system requires that the type for the left-hand side be the same as the type for the right-hand side. We must, therefore, determine the types for the left and right expressions.



We have created the video Using Parse Trees for Checking Type Correctness that demonstrates using the parse tree to infer types and check for type correctness.

The following is an admittedly hard-to-follow textual description of how we can determine the types for the left and right expressions. The type of the left-hand side can be determined by traversing the left most child of `Stmt`, i.e., by determining the type for the `lvalue`. Since `lvalue` has one child, `ID`, the type for `lvalue` will be the type for `ID`. We would have already stored the type for this `ID`, the variable a, in the symbol table. Therefore, we can infer that the left-hand side has the type that we obtain for variable a from the symbol table.

We must also compute the type of the right-hand side. This type can be determined by traversing the third child, written as E1 in the diagram. To compute this type, we must compute the type of the `Expr` labelled E2, the `Term` labelled T1 and then apply the type system rule for addition. The type of E2 is the type of its one child, the `Term` labelled T2. The type of T2 is the type of its child, the Factor labelled F1. The type of F1 is the type of its child, an ID. The type of this ID, the variable x, should already be in the symbol table. We have inferred that the type of E2 is the type for the variable x. We must also similarly infer the type of T1. Using a similar approach, we determine that the type of T1 is the type of F2, which is the type of NUM that, by definition, is `int`.

We now know what the types of the left and right expressions for addition are. At this point, we would use the type system rule for addition to determine the resulting type for adding variable x to an `int`. Notice that this would produce `int` if x is an `int` and `int*` if x is a pointer, i.e., pointer addition. This would give us the type for E1. At this point, we would be in a position to confirm that the left-hand side of the assignment has the same type as the right-hand side. If that is not the case, a type error would be generated.

The above should convince the reader that to determine type correctness, we need a tree traversal. At any given node, we can recursively infer the types of child nodes. Then, by looking at the grammar rule for the node currently at, we can determine which type rule from the type system is applicable. Using the inferred types of the children we determine if the rule has been violated.

```
void typeOf(Tree &tree){
  for each child c of tree {
     //recursively compute type of each child subtree
      typeOf(c);
   }
```

```
    // refer to the relevant type system rule for this tree node
    // use the computed types of children to determine if the rule is violated
    // if it is not violated, store the computed type in the tree node
}
```

3.1 Type Inference Rules

All that is left to do is specify the type system for WLP4 that would be used by the recursive type inference algorithm we just discussed. The WLP4 Language Specification page gives a detailed description of the type system in English. Here, we use a standard notation that most would be familiar with from a course such as CS245; premises for a conclusion go above a horizontal bar, the conclusion below the bar, and empty premises means that the conclusion always holds. This is the Post System of notating deductive logic.

If an ID is declared with type τ then it has this type.

$$\frac{\langle \text{ID.name}, \tau\rangle \in \text{declarations}}{\text{ID.name} : \tau}$$

Numbers, by definition, have type int.

$$\frac{}{\text{NUM} : \texttt{int}}$$

NULL, by definition, has type int*.

$$\frac{}{\text{NULL} : \texttt{int*}}$$

Parentheses do not change the type.

$$\frac{E : \tau}{(E) : \tau}$$

Taking the address of an int produces an int*.

$$\frac{E : \texttt{int}}{\&E : \texttt{int*}}$$

Dereferencing an int* produces an int.

$$\frac{E : \texttt{int*}}{*E : \texttt{int}}$$

The expression new int[E] takes an int value and produces an int*.

$$\frac{E : \texttt{int}}{\texttt{new int[E]:int*}}$$

Multiplying two integer values produces an integer value.

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 * E_2 : \texttt{int}}$$

An important point to be aware of is that the absence of type rules for particular types also has an implication. For example, the only rule for multiplication was shown above and states that expressions being multiplied must both be integers. The absence of any rule for multiplying expressions where at least one expression is a pointer implicitly disallows the multiplication of two pointers and the multiplication of a pointer to an integer. That is, we specify only what is allowed; what is not allowed is everything not specified.

Dividing an integer with another integer produces an integer.

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 / E_2 : \texttt{int}}$$

The modulo operator can be applied to two integers and produces an integer.

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 \% E_2 : \texttt{int}}$$

The addition of two integers produces an integer.

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 + E_2 : \texttt{int}}$$

The addition of a pointer to an integer or vice versa produces a pointer. Notice that the absence of a rule that adds two pointers implies the operation to be illegal.

$$\frac{E_1 : \texttt{int*} \quad E_2 : \texttt{int}}{E_1 + E_2 : \texttt{int*}}$$

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int*}}{E_1 + E_2 : \texttt{int*}}$$

The subtraction of two integers produces an integer.

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 - E_2 : \texttt{int}}$$

Subtracting an integer from a pointer produces a pointer.

$$\frac{E_1 : \texttt{int*} \quad E_2 : \texttt{int}}{E_1 - E_2 : \texttt{int*}}$$

Subtracting two pointers produces an integer. This rule might seem counter-intuitive but is used to determine the number of elements between two addresses. Notice the absence of a rule that allows subtracting a pointer from an integer.

$$\frac{E_1 : \texttt{int*} \quad E_2 : \texttt{int*}}{E_1 - E_2 : \texttt{int}}$$

A procedure call must check that the arguments of the call match the types of the parameters for the procedure. Procedure calls always return integers.

$$\frac{\langle f, \tau_1, ..., \tau_n \rangle \in \text{declarations} \quad E_1 : \tau_1 \quad E_2 : \tau_2 \quad ... \quad E_n : \tau_n}{f(E_1, ..., E_n) : \texttt{int}}$$

3.2 Type Checking Statements

Since expressions produce values, these values have types. We discussed above the way to infer types for expressions. However, programs are comprised of statements which in turn contain expressions. While statements do not produce values (and therefore do not have an inferred type), we must still check the type correctness of statements.

Definition 1 *We say a statement is* `well-typed` *if its components are well-typed.*

An expression is well-typed if a type can be inferred.

$$\frac{E : \tau}{\text{well-typed}(E)}$$

The println statement is well-typed if and only if the expression to print has type `int`, i.e., the language does not support printing pointers.

$$\frac{E : \texttt{int}}{\text{well-typed}(\texttt{println(E);})}$$

Deallocation is well-typed if and only if the parameter has type `int*`, i.e., we can only deallocate pointers.

$$\frac{E : \texttt{int*}}{\text{well-typed}(\texttt{delete[ ] E;})}$$

An assignment is well-typed if and only if the types of the Left and Right side are of the same type.

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 = E_2;)}$$

While the above rule requires that the type of the left-hand side be the same as the type of the right-hand side, that, on its own, is not enough. Another requirement, when assigning a value, is that the left-hand side must represent a storage location, i.e., the left-hand side must be an `lvalue`. In other words, while `x = y` is legal, `3 = y` is not. In our language, lvalues include variables as well as the result of dereferencing an address. It turns out that the way the context-free grammar for `WLP4` is designed, there is no need to check that the left-hand side is an lvalue, i.e., this requirement is already enforced by the grammar.

An empty sequence of statements is well-typed.

A sequence of statements is well-typed if and only if each statement in the sequence is well-typed.

$$\frac{\text{well-typed}(S_1) \quad \text{well-typed}(S_2)}{\text{well-typed}(S_1\ S_2)}$$

To type check the correctness of control flow statements, `if` and `while`, we must have a way to check the correctness of the conditional expression that is evaluated. Typically, these expressions would evaluate to a boolean. However, `WLP4` does not have a bool type. Instead, we will use our notion of well-typed and say that a `test` is well-typed if the operands for the comparison are of the same type.

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 < E_2)} \qquad \frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 <= E_2)}$$

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 > E_2)} \qquad \frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 >= E_2)}$$

$$\frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 == E_2)} \qquad \frac{E_1 : \tau \quad E_2 : \tau}{\text{well-typed}(E_1 != E_2)}$$

An `if` statement is well-typed if and only if the components of the statement are well-typed.

$$\frac{\text{well-typed}(\texttt{test}) \quad \text{well-typed}(S_1) \quad \text{well-typed}(S_2)}{\text{well-typed}(\texttt{if (test) \{S}_1\texttt{\} else \{S}_2\texttt{\}})}$$

A `while` statement is well-typed if and only if the components of the statement are well-typed.

$$\frac{\text{well-typed}(\texttt{test}) \quad \text{well-typed}(S)}{\text{well-typed}(\texttt{while (test) \{S\}})}$$

An empty sequence of declarations is well-typed.

A variable that is declared to be an integer is well typed if it is initialized with an integer value.

$$\frac{\text{well-typed}(\texttt{dcls}) \quad \langle \text{ID.name}, \texttt{int}\rangle \in \text{declarations}}{\text{well-typed}(\texttt{dcls int ID = NUM;})}$$

A variable that is declared to be a pointer is well typed if it is initialized with a NULL value.

$$\frac{\text{well-typed}(\texttt{dcls}) \quad \langle \text{ID.name}, \texttt{int*}\rangle \in \textit{declarations}}{\text{well-typed}(\texttt{dcls int* ID = NULL;})}$$

We must also type check the declaration of procedures. A procedure is well-typed if the declarations and statements are well-typed and it returns an integer:

$$\frac{\text{well-typed}(\texttt{dcls}) \quad \text{well-typed}(S) \quad E : \texttt{int}}{\text{well-typed}(\texttt{int ID}(\texttt{dcl}_1, ..., \texttt{dcl}_\texttt{n})\{\texttt{dcls S return E;}\})}$$

We must add an extra restriction on the `wain` procedure. The procedure `wain` is well-typed if the second parameter is an `int`, the declarations and statements are well-typed and the procedure returns an integer:

$$\frac{\texttt{dcl}_2 : \texttt{int} \quad \text{well-typed}(\texttt{dcls}) \quad \text{well-typed}(S) \quad E : \texttt{int}}{\text{well-typed}(\texttt{int wain}(\texttt{dcl}_1, \texttt{dcl}_2)\{\texttt{dcls S return E;}\})}$$

4 Other Context-Sensitive Analyses

High-level languages often have other requirements that must be satisfied. For example, in Java, a variable must be initialized before it is used. Consider the following Java program:

```
public class HelloWorld{
  public static void main(String[] args){
    int x;
    System.out.println(x);
  }
}
```

This program will be rejected by the Java compiler with an error that the variable x has not be initialized. This is part of the Analysis stage of the Java compiler. Notice that such an analysis requires more compiler infrastructure than what we have discussed, as the analysis needs to track which variable has been initialized and which has not (control flow makes this even more complicated). C and C++ do not have such a requirement (which leads to all sorts of bugs!). `WLP4` does require that all variables be initialized before they are used but does so through the context-free grammar; a variable when declared must be initialized to a value.

`WLP4` procedures must return an integer value through the last statement of the procedure. There is a reason that the language was designed this way. If multiple returns were allowed, an analysis would be needed to check that a value is returned on all paths through the program. This is non-trivial. Consider the following C/Java function:

```
int foo(int x) {
   if ( x < 0 )
      return -1;
   else
      return 1;
}
```

The required analysis needs to keep track of all paths through the function and ensure that all paths return an appropriate value. This is not always possible. For example, in Java, the following code will not actually compile:

```
int foo(int x) {
   if ( x < 0 )
      return -1;
   else if (x == 0 )
      return 0;
   else if (x > 0 )
      return 1;
}
```

The error reported by the Java compiler is that the function does not return on all paths. But it does! The analysis is just not smart enough to figure out that the three conditions exhaust all possible values of x and so one of these three code blocks must execute. What about C/C++? The above code does not generate an error (C/C++ have the philosophy of letting programmers shoot themselves!). However, in most compilers, it will generate a warning if the program is compiled with warnings enabled (e.g., `-Wall`).

Java imposes another requirement: there should be no *dead code*, i.e., code that is unreachable. Consider the following function:

```
int foo(){
   return 1;
   int x = 5;
   return x;
}
```

The programmer who wrote this should be fired! Java compilers are required to report an "unreachable statement error". C/C++ do not have a similar requirement. `WLP4`, once again, avoids this since the only way to return from a function is through the last statement of the function.

5 Looking Ahead

Finally! We have now covered the analysis stage of the compiler. Any input program that successfully passes through the context-sensitive analysis stage is a syntactically and semantically correct program. In the next module, we will discuss the synthesis stage; taking our parse tree and gathered type information and producing the intended output. In our case, we will generate MIPS Assembly.