9 Context-sensitive analysis and types

Although context-free languages are more expressive than regular languages, some requirements of valid programs cannot be expressed even with a context-free language. One example of such a requirement is that every variable that is used in an expression must first be declared. The purpose of *context-sensitive analysis*, also called *semantic analysis*, is to enforce such requirements, as well as to compute information that will be needed by the compiler to generate machine-language code.

For the Lacs language specifically, context-sensitive analysis needs to perform two main tasks:

1. First, context-sensitive analysis needs to resolve identifiers to the variables or procedures that they refer to. This is done by creating a symbol table for each scope in the Lacs program and mapping each name declared in the scope to either a variable declaration or a procedure declaration. Then, whenever an identifier is used in an expression, it is looked up in the symbol table corresponding to the scope in which the expression appears to determine the meaning of the identifier. This analysis may report one of two kinds of errors:

   (a) The analysis detects uses of identifiers that have not been declared in a variable or procedure declaration.

   (b) The analysis detects duplicate declarations, multiple declarations of the same name in the same scope.

2. Second, context-sensitive analysis needs to compute a type for each expression and subexpression in the program and check that the program obeys the Lacs typing rules.

Context-sensitive analyses for languages other than Lacs typically also perform these two tasks, as well as other similar tasks depending on the specific programming language being compiled.

The scoping and typing requirements for Lacs are specified in detail in the Lacs Specification.

9.1 Types

The word *type* has been used in different contexts to mean different concepts, although the various concepts often share common themes. For further reading about what a type *is*, or more accurately the various things that the word "type" *can mean*, see Type Systems by Luca Cardelli, In Search of Types by Stephen Kell, or Against a universal definition of 'type' by Tomas Petricek. For the purpose of this course, we will summarize some of the themes in three possible definitions of a type.

First, a *type* can be defined as a collection of values, such as integers, characters, strings, programs, trees, sounds, images, videos, etc. Note that *values* in this context are the high-level concepts that we express in a programming language, rather than the sequences of bits that represent those concepts in a computer. We often use types to describe the possible

values that could result from evaluating an expression and to ensure that operations are applied only to values for which they make sense.

A second, related but separate view of a *type* is as an interpretation of sequences of bits. For example, the bits 1001 mean the number 9 when interpreted as an unsigned binary integer, but the number -7 when interpreted as a two's-complement integer; the bits 01000001 mean the letter A when interpreted as an ASCII character but the number 65 when interpreted as a binary integer. In this view, we often use types to document and enforce how certain bits in the computer are to be interpreted and to prevent us from accidentally misinterpreting them. We saw an example of this in the discussion of relocation in Module 2: in that context, a word in a machine language file might represent a memory address, in which case it *must* be relocated, or some other type of value such as an integer, in which case it *must not* be relocated.

A third possible definition of a *type* is as a computable property of programs that guarantees some property about their execution. As one example of this, if a compiler determines that some expression has the type `Int`, then the expression always produces integer values when it is evaluated. More general properties are possible. As another example, programs that can be given a type in some programming languages are guaranteed to avoid certain bad behaviours (i.e., crashes) when they run. As a third example, some type systems are sufficiently sophisticated to express functional correctness properties of programs, so, for example, if some function can be typed as a sorting algorithm, it is guaranteed that the function will correctly sort all possible input sequences when it runs. As you will learn in CS 341, even very simple properties of the behaviour of programs are undecidable in general, meaning that there is no algorithm that can look at an arbitrary program and determine whether the program will have the behaviour we want when it executes. Types are one solution to this bleak reality, in that we can algorithmically determine whether a program has a certain type, and if it does, we can guarantee some desirable property of its behaviour. (On the other hand, if the program *does not* have the right type, then we cannot conclude anything about how it will behave when it runs.) In this way, types are an important connection between the static *program text* and its dynamic *runtime behaviour*.

9.2 Types in Lacs

The Lacs programming language has two sorts of types:

1. `Int` is a type whose values are the integers between $-2^{31} = -2147483648$ and $2^{31} - 1 = 2147483647$. Values of this type support operations that implement arithmetic modulo 2^{32}.

2. If $\tau, \tau_1, \tau_2, \ldots, \tau_n$ are types, then $(\tau_1, \tau_2, \ldots, \tau_n)$ `=>` τ is a type whose values are procedures that take n arguments of the corresponding types $\tau_1, \tau_2, \ldots, \tau_n$ and return a value of type τ. A value of such a procedure type supports the procedure call operation: given argument values of the required types, the procedure can be called; the procedure call expression then evaluates to the value returned from the called procedure

A common shorthand notation for a sequence $\tau_1, \tau_2, \ldots, \tau_n$ is $\overline{\tau}$. We will use this shorthand notation throughout this module.

Exercise 20. *Since procedure types are defined recursively in terms of existing types $\tau, \tau_1, \tau_2, \ldots, \tau_n$, it would be natural to define the set of Lacs types using a context-free grammar. It is common to define the sets of types in other programming languages using context-free grammars. As an exercise, you can write such a grammar for Lacs types.*

After we translate Lacs programs to machine language, values of the various Lacs types are all implemented using words of 32 bits in the computer. When debugging your solution to Assignment 6 or earlier assignments, you may have encountered crashes caused by accidentally misinterpreting certain words in memory, for example misinterpreting a static link as an integer or vice versa. Such crashes are challenging to debug. The type system of a high-level language such as Lacs can ensure that values of different types are kept *separate*, that the running program never misinterprets a word that is intended to represent an integer as the address of a closure, or vice versa (assuming, of course, that we implement the type system correctly!) In a high-level language, the type system serves to prevent this class of challenging bugs.

9.3 Type systems

A *type system* is a collection of *inference rules* that define a *typing relation*. A *typing relation* is a set of triples $\langle \Gamma, E, \tau \rangle$. In these notes, we will use the letter $\mathcal{T}$ to denote the typing relation, this set of triples. In each triple $\langle \Gamma, E, \tau \rangle$, the E is typically called a *term* in the terminology of type systems and corresponds to a subtree of the *parse tree* in our compiler implementation. The τ in the triple is a *type*. The Γ in the triple is typically called a *typing context* in the terminology of type systems and corresponds to a *symbol table* in our compiler. In the compiler, a symbol table maps each identifier to its meaning, a declaration of that identifier in the program.

The presence of a triple $\langle \Gamma, E, \tau \rangle$ in the typing relation (i.e., $\langle \Gamma, E, \tau \rangle \in \mathcal{T}$) is intended to mean that the expression represented by the tree E has type τ, using the symbol table Γ to look up the declarations of any identifiers used in E. Since the typing relation contains all possible symbol tables Γ and parse trees E, it is very much infinite and it is usually not helpful to think about enumerating all possible triples in it, but it *is* intuitively helpful to remember that it is a set of triples. It is traditional when discussing type systems to use the notation $\Gamma \vdash E : \tau$ to mean $\langle \Gamma, E, \tau \rangle \in \mathcal{T}$. Do not be intimidated by the symbols "$\vdash$" and ":". Their significance is historical, but in our context, they are just separators like the commas in the triple $\langle \Gamma, E, \tau \rangle$. When you see the perhaps unfamiliar notation $\Gamma \vdash E : \tau$, think $\langle \Gamma, E, \tau \rangle \in \mathcal{T}$.

How do we specify the typing relation $\mathcal{T}$ when it is an infinite set? We will use *inference rules*. An *inference rule* is a collection of zero or more *premises* and a *conclusion*. The premises are written above a horizontal line and the conclusion below the line. The rule asserts that *if* the premises hold, *then* the conclusions holds. As a first example, consider a rule with zero premises:

LITERAL

$$\frac{}{\Gamma \vdash \textbf{NUM} : \textbf{Int}}$$

In this rule, the Γ is a *metavariable* that stands for every possible symbol table. The **NUM** is also a metavariable that stands for every leaf node of the parse tree with a token of kind NUM (corresponding to every possible integer literal in a Lacs program). These are called *metavariables* to distinguish them from the variables in the programming language we are defining (i.e., Lacs in our case); each metavariable stands for many possible typing contexts, parse trees, or types. The metavariables in each rule are implicitly universally quantified: the rule says that *for all* symbol tables Γ and *for all* integer literal tokens **NUM** of kind NUM, the typing relation $\mathcal{T}$ contains the triple $\langle \Gamma, \textbf{NUM}, \textbf{Int} \rangle$. This single rule adds *many* triples to the typing relation $\mathcal{T}$, one for each possible instantiation of the metavariables, including examples such as $\langle \emptyset, \texttt{42}, \textbf{Int} \rangle$, where $\emptyset$ represents the empty symbol table, and $\langle (\texttt{x} \mapsto \textbf{var x : Int}), \texttt{5}, \textbf{Int} \rangle$, where $(\texttt{x} \mapsto \textbf{var x : Int})$ represents a symbol table with a declaration of a variable named `x` declared to be of type `Int`.

To specify the type of a use of a variable, the corresponding inference rule needs a premise that looks up the variable in the symbol table to determine the type that it was declared with:

$$\textsc{Variable}\quad \frac{\Gamma(\textbf{ID}) = \textbf{var ID} : \tau}{\Gamma \vdash \textbf{ID} : \tau}$$

In this rule, Γ, **ID**, and τ are all metavariables: the rule says that *for all* symbol tables Γ, identifier tokens **ID**, and types τ, $\langle \Gamma, \textbf{ID}, \tau \rangle$ is in the typing relation $\mathcal{T}$, but *only if* the premise $\Gamma(\textbf{ID}) = \textbf{var ID} : \tau$ holds: if looking up the identifier in the symbol table returns a variable declaration with declared type τ. Thus, although the rule is universally quantified for all types τ, the condition imposed by the premise ensures that for a given symbol table Γ and identifier **ID**, the rule applies only when the metavariable τ happens to designate the specific type of the variable that is obtained by looking up the identifier **ID** in the symbol table Γ.

If some particular identifier **ID** is not in the symbol table Γ because it has not been declared in the program, this rule will not apply because its premise $\Gamma(\textbf{ID}) = \textbf{var ID} : \tau$ will never be satisfied. In general, every subtree of a correct Lacs program will have some corresponding triple in the typing relation $\mathcal{T}$ to give it a type. The type system designates a program as invalid, that is, it signals an error in the program that should be reported by the compiler, by not giving the program any type at all. For a program with a use of an undeclared identifier, none of the Lacs type inference rules apply to the subtree with the undeclared identifier, so that subtree will not have any type in the typing relation.

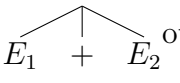
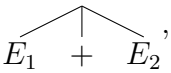
An identifier in Lacs can refer not only to a variable, but also to a procedure, so another rule is needed to look up procedures in the symbol table:

$$\textsc{Procedure}\quad \frac{\Gamma(\textbf{ID}) = \textbf{def ID}(\overline{\textbf{ID} : \tau}) : \tau}{\Gamma \vdash \textbf{ID} : (\overline{\tau}) \Rightarrow \tau}$$

If looking up the identifier **ID** in the symbol table Γ results in a procedure declaration with parameters of types $\overline{\tau}$ and return type τ, then any use of the identifier **ID** is specified to have the procedure type $(\overline{\tau}) \Rightarrow \tau$.

We can write similar inference rules for arithmetic expressions that build larger trees out of smaller trees:

$$\textsc{Arithmetic}\quad \frac{\Gamma \vdash E_1 : \textbf{Int} \quad \Gamma \vdash E_2 : \textbf{Int}}{\Gamma \vdash E_1 \ + \ E_2 : \textbf{Int}}$$

This rule says that for all symbol tables Γ and for all parse trees E_1 and E_2, if we create a larger parse tree  out of E_1 and E_2, then $\langle \Gamma,$  $, \textbf{Int} \rangle \in \mathcal{T}$, but only if the premises hold, that is, if $\langle \Gamma, E_1, \textbf{Int} \rangle \in \mathcal{T}$ and $\langle \Gamma, E_2, \textbf{Int} \rangle \in \mathcal{T}$. If the parse trees E_1 and E_2 are expressions of type **Int**, then this rule specifies that the larger expression $E_1 + E_2$ is also of type **Int**. We could write similar inference rules for the other arithmetic operators $-, *, /,$ and %. Since the premises of these rules would be the same, as a shorthand, we can write them as a single inference rule with multiple conclusions, one for each arithmetic operator:

$$\textsc{Arithmetic}\quad \frac{\Gamma \vdash E_1 : \textbf{Int} \quad \Gamma \vdash E_2 : \textbf{Int}}{\Gamma \vdash E_1 \ + \ E_2 : \textbf{Int} \qquad \Gamma \vdash E_1 \ - \ E_2 : \textbf{Int} \qquad \Gamma \vdash E_1 \ * \ E_2 : \textbf{Int} \qquad \Gamma \vdash E_1 \ / \ E_2 : \textbf{Int} \qquad \Gamma \vdash E_1 \ \% \ E_2 : \textbf{Int}}$$

Assignment expressions in Lacs are typed by the following inference rule:

$$\textsc{Assignment}\quad \frac{\Gamma(\textbf{ID}) = \textbf{var ID} : \tau \qquad \Gamma \vdash E : \tau}{\Gamma \vdash \textbf{ID} = E : \tau}$$

The variable **ID** to which the assignment is assigning must be declared in the symbol table with some declared type τ and the parse tree of the expression E whose value will be assigned to the variable must have the same type τ. Then the assignment expression itself is given the same type τ. Notice that this rule does not apply if looking up the identifier **ID** in the symbol table Γ results in a procedure declaration rather than a variable declaration. In such a case, there is no rule in the type system that would apply, so a program that tries to assign to the name of a procedure declaration cannot be typed and the compiler should report an error.

Sequences of expressions derived using the production rule `expras -> expra SEMI expras` in the Lacs grammar are typed by the following inference rule:

$$\textsc{Sequencing}\quad \frac{\Gamma \vdash E_1 : \tau' \qquad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1; E_2 : \tau}$$

The rule requires *both* expressions E_1 and E_2 to have some types τ' and τ. In the conclusion, the overall sequence of expressions is given the type τ of E_2, since the value of a block of expressions is always the value of the last expression in the block. Although the specific type τ' of E_1 is not used anywhere in the rule other than in the first premise, this premise is important to ensure that the first expression E_1 has *some* type. Without this premise, this

rule would give a type to an invalid block such as `{ p = 5; 42 }` when the variable `p` is not declared or is declared with a type other than **Int**. We do not care what the specific type of `p = 5` is, but we do care that it has some type.

The if-statement in Lacs is typed by the following inference rule:

IF STATEMENT

$$\frac{\Gamma \vdash E_1 : \textbf{Int} \qquad \Gamma \vdash E_2 : \textbf{Int} \qquad \Gamma \vdash E_3 : \tau \qquad \Gamma \vdash E_4 : \tau}{\begin{array}{ll} \Gamma \vdash \textbf{if}(E_1 == E_2)\ E_3\ \textbf{else}\ E_4 : \tau & \Gamma \vdash \textbf{if}(E_1\ != E_2)\ E_3\ \textbf{else}\ E_4 : \tau \\ \Gamma \vdash \textbf{if}(E_1 <= E_2)\ E_3\ \textbf{else}\ E_4 : \tau & \Gamma \vdash \textbf{if}(E_1 => E_2)\ E_3\ \textbf{else}\ E_4 : \tau \\ \Gamma \vdash \textbf{if}(E_1 < E_2)\ E_3\ \textbf{else}\ E_4 : \tau & \Gamma \vdash \textbf{if}(E_1 > E_2)\ E_3\ \textbf{else}\ E_4 : \tau \end{array}}$$

Both expressions E_1 and E_2 that are being compared are required to be of type **Int**. The then-clause E_3 and else-clause E_4 must both have the same type τ, and τ also becomes the type of the if-statement as a whole.

The following typing rule applies to both closure calls and direct calls in Lacs programs, both of which are parsed using the same production rule in the Lacs grammar, `factor -> factor LPAREN argsopt RPAREN`.

PROCEDURE CALL

$$\frac{\Gamma \vdash E' : (\overline{\tau}) \Rightarrow \tau' \quad \forall i.\ \Gamma \vdash E_i : \tau_i}{\Gamma \vdash E'(\overline{E}) : \tau'}$$

The expression E' could be a simple identifier naming a specific procedure to be called directly or an expression evaluating to a closure. Either way, the first premise requires this expression to have a procedure type $(\overline{\tau}) \Rightarrow \tau'$. For each parameter type τ_i in the list of parameter types $\overline{\tau}$, the corresponding argument expression E_i in the list of arguments $\overline{E}$ is required to have the parameter type τ_i. If these premises are satisfied, the inference rule gives the return type τ' of the procedure type to the overall call expression.

9.4 Type soundness

The type inference rules define the typing relation $\mathcal{T}$, a set of triples. We could have written an arbitrary set of rules to define an arbitrary relation. In order for the typing relation to be practically useful, it must be connected somehow to the actual execution of Lacs programs. Specifically, we say that a type system is *sound* if whenever the expression tree E has type τ in the typing relation, then during the execution of a program, the expression E evaluates to a value of the type τ. For example, since the Lacs type system gives the type **Int** to the expression `1 + 1`, this expression should evaluate to an integer rather than a procedure, and it does indeed evaluate to the integer 2. The definition of soundness can be extended to include the typing context Γ by relating it to the runtime state of the program that records the current values of variables.

The Lacs typing rules were designed with the *intent* of being sound, and if any unsoundness is discovered in the rules, it should be considered a bug. If we had more time in the course, we should formally *prove* that the typing rules are sound. Before we could do this, we would have to formally specify the runtime behaviour, or *semantics* of Lacs programs, so that our proof could reason precisely about the value that each expression evaluates to

at run time. Since the Lacs language contains mutable variables and requires both a stack and a heap in its implementation, such a semantics and the soundness proof would get quite complicated. Although we do not have time to do this in CS 241E, type system soundness proofs are a central part of the design of programming languages. They are partly covered in CS 442.

9.5 Well-formedness relation

The Lacs type inference rules that we have discussed so far do not yet give types to all Lacs parse trees. They give types only to trees that represent expressions, which evaluate to values. Some Lacs parse trees represent parts of a program that are not expressions and do not evaluate to a value. One example of this is the parse tree of a procedure declaration such as `def main(a: Int, b: Int): Int = { a + b }`. The declaration itself does not evaluate to any value until the procedure is called, and even then, it is the procedure call expression (such as `main(1, 2)`) that evaluates to a value, not the procedure declaration. Since the type of an expression is intended to indicate something about the value to which the expression evaluates, it does not make sense for the typing relation to give any particular type to a procedure declaration. Nevertheless, we would like inference rules to check that the procedure declaration declares a valid Lacs procedure and that it does not contain type errors in its body, for example using an undeclared variable or adding two expressions whose types are not **Int**.

To perform such checks, we define another relation $\mathcal{W}$, called the *well-formedness relation*. The *well-formedness relation* is a set of pairs $\langle \Gamma, E \rangle$, where Γ is a again a typing context (symbol table) and E is again a term (parse tree). The presence of a parse tree E in the well-formedness relation $\mathcal{W}$ means that the tree is considered to be a valid Lacs parse tree with no compile-time errors in it, but that it is not an expression and therefore it does not make sense to assign it any particular type. Like with the typing relation, it is traditional to use the notation $\Gamma \vdash E$ to mean $\langle \Gamma, E \rangle \in \mathcal{W}$. Again, the "$\vdash$" symbol is just a separator. Some people use alternative notations such as $\Gamma \vdash E$ **wf** or $\Gamma \vdash E$ ✓. We can define the well-formedness relation using the same style of inference rules as for the typing relation. These inference rules enforce the compile-time requirements of Lacs programs for parts of the parse tree other than expressions.

To define the well-formedness of the root node of the parse tree of an entire Lacs program, which is expanded using the production rule `defdefs -> defdef defdefs` or `defdefs -> defdef` in the Lacs grammar, we use the following inference rule:

$$\textsc{Program}\ \frac{\forall i.\ \overline{\text{defdef}} \vdash \text{defdef}_i}{\emptyset \vdash \overline{\text{defdef}}}$$

In the conclusion of the rule, the parse tree of a Lacs program is a list $\overline{\text{defdef}}$ of procedure declarations, `defdef` parse tree nodes. We define the well-formedness of the overall Lacs program using the empty symbol table $\emptyset$ because before we start looking inside the program, no identifiers have yet been declared. The premise of the rule requires that each procedure declaration defdef_i in the program must be well-formed, which will be checked by the next inference rule. The symbol table used to check the well-formedness of each procedure is

$\overline{\text{defdef}}$, the list of all (outermost) procedure declarations in the Lacs program. For example, consider the following program:

```
343 def a(): Int = { b() }
344 def b(): Int = { a() }
```

Procedure `a` can call procedure `b` and vice-versa, so each procedure must be checked for well-formedness with a symbol table containing both of the procedure declarations so that the identifiers `a` and `b` are allowed to be used in both procedures.

The following inference rule defines the well-formedness of each individual procedure declaration in a Lacs program:

PROCEDURE DECLARATION

$$\frac{\text{all names in } \overline{\text{vardef}},\ \overline{\text{vardef}'},\ \overline{\text{defdef}} \text{ distinct} \qquad \forall i.\ \Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}} \vdash \text{defdef}_i \qquad \Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}} \vdash E : \tau}{\Gamma \vdash \textbf{def ID}(\overline{\text{vardef}}) : \tau = \{\overline{\text{vardef}'}\ \overline{\text{defdef}}\ E\}}$$

In the conclusion of the inference rule, the root node of the parse tree is expanded using the Lacs grammar production rule `defdef -> DEF ID LPAREN parmsopt RPAREN COLON type BECOMES LBRACE vardefsopt defdefsopt expras RBRACE`. Below the `defdef` node of the parse tree, each procedure declaration has a list of parameters $\overline{\text{vardef}}$, a list of variable declarations $\overline{\text{vardef}'}$, a list of nested procedure declarations $\overline{\text{defdef}}$, and a body expression E. The premises of the rule first enforce the requirement that the parameters, variables, and nested procedures must all have distinct names. Then, the symbol table Γ of the procedure that we are checking is *extended* with the declarations of the parameters $\overline{\text{vardef}}$, the variable declarations $\overline{\text{vardef}'}$, and the nested procedure declarations $\overline{\text{defdef}}$. The commas in the notation $\Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}}$ mean that any of the new declarations take precedence over (i.e., replace) declarations of the same name in Γ. This expresses the Lacs requirement that the declaration corresponding to a name is looked up in the *closest* (innermost) scope that declares that name. The body E of the procedure may refer to any of the parameters, variables, and nested procedures declared in that procedure, as well as those declared in any outer enclosing procedures, so the body is type-checked using the premise $\Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}} \vdash E : \tau$ with the extended symbol table $\Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}}$. Notice that this premise is a type-checking premise rather than a well-formedness premise: in order for the procedure declaration to be *well-formed*, that is, for $\left\langle \Gamma, \textbf{def ID}(\overline{\text{vardef}}) : \tau = \{\overline{\text{vardef}'}\ \overline{\text{defdef}}\ E\} \right\rangle \in \mathcal{W}$, the body E of the procedure must *have a type* according to the typing rules, that is, $\left\langle \left(\Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}}\right), E, \tau \right\rangle \in \mathcal{T}$, for some τ. The inference rule also checks that the type τ of the body expression E is the same as the return type τ declared in the parse tree of the procedure declaration. In addition, in order for the procedure declaration to be considered well-formed, each of the declarations defdef_i of procedures that are nested inside it must also be recursively checked for well-formedness. This is done by the premise $\forall i.\ \Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}} \vdash \text{defdef}_i$. Like the body E of the procedure, each nested procedure may refer to the parameters, variables, and nested procedures declared in its outer enclosing procedure, so each defdef_i is checked for well-formedness with the extended symbol table $\Gamma, \overline{\text{vardef}}, \overline{\text{vardef}'}, \overline{\text{defdef}}$.

Overall, to check that a parse tree corresponding to an entire Lacs program is valid, we just use the first well-formedness rule for the whole program. In its premise, this rule recursively checks well-formedness of each outermost procedure in the program using the second well-formedness rule. The second well-formedness rule recurses into nested procedures, and also recurses into the type inference rules to check that the body of the declared procedure has the same type as the declared return type. The type inference rules in turn recurse throughout the entire body of the procedure, checking that all expressions in the body have consistent types and are thus valid.

In the case of Lacs, only the well-formedness rules *extend* the symbol table, in that they add new declarations to it. In the well-formedness rules, the symbol table Γ is different in the conclusion than in the premises. In contrast, the type inference rules discussed in Section 9.3 always propagate the symbol table Γ unchanged: the symbol table is the same Γ in their premises as in their conclusions. This is true in Lacs because all new scopes with new declarations (of parameters, variables, and nested procedures) occur only in a procedure declaration. In other programming languages in which new scopes with newly declared names can be opened within expressions, the type inference rules for those expressions would also extend the symbol table with the new declarations in their premises.

9.6 Type checking

The inference rules in the type system declaratively define the typing relation $\mathcal{T}$ and the well-formedness relation $\mathcal{W}$. How do we use these theoretical sets in the implementation of the compiler? In the compiler, we would like to compute a type for each (sub-)expression or report an error if the expression does not have any type (i.e., the expression is not valid). More precisely, assuming the type checker in the compiler has already constructed a symbol table Γ, its task is, given the parse tree E of an expression, to determine whether the typing relation contains any triple of the form $\langle \Gamma, E, \tau \rangle$, and if it does, to return the type τ from that triple as the type for the expression E. In Assignment 9, you will implement the Scala function `def typeOf(tree: Tree): Type`, which also has access to a `symbolTable: SymbolTable` parameter from an outer procedure. Thus, the inputs to the `typeOf` function are a symbol table and a parse tree, and its output is a type.

So, our task is, given Γ and E, to find a τ such that $\langle \Gamma, E, \tau \rangle \in \mathcal{T}$. To do so, we will make use of the observation that the triple $\langle \Gamma, E, \tau \rangle$ can only be in $\mathcal{T}$ if the conclusion of some type inference rule forced it to be there. Examining the Lacs type inference rules, notice that their conclusions all have distinct syntactic forms. That is, for each of the Lacs grammar rules that could be the production rule expanding the root node of a tree E, there is only one type inference rule that could possibly apply. For example, if E is a numeric literal, only the LITERAL rule could possibly give it a type; if E is an arithmetic expression, only the ARITHMETIC rule could possibly give it a type; if E is an if-statement, only the IF STATEMENT rule could possibly give it a type, etc. Therefore, the implementation of our type checker can look at the grammar production rule in the root node of the tree E and implement the inference rule corresponding to that production rule.

In implementing an inference rule, the type checker needs to determine values for the metavariables from the tree and the symbol table. For example, if the tree E is an addition expression, the metavariables E_1 and E_2 must refer to the subtrees of E defining the two

operands of the addition. As another example, if the tree E is a use of an identifier, the premise of the IDENTIFIER inference rule requires that identifier to be looked up in the symbol table, so the result of this lookup becomes the value of the metavariable τ. With values for the metavariables, the type checker checks the premises of the inference rule. In many cases, this involves recursive calls to determine types for subtrees of E. Finally, the type checker returns the type specified by the conclusion of the inference rule.

Remark. The property that the syntactic form of each parse tree node determines which inference rule to apply and suggests the values for the metavariables in the inference rule is specific to Lacs. Other type systems for other programming languages may require various more complicated algorithms to search for ways to apply the inference rules and instantiate the metavariables to find a triple $\Gamma \vdash E : \tau$ in the typing relation for each subtree E.

Putting it more directly, your task in the `typeOf` function in Assignment 9 is to translate the type inference rules that we have discussed in Section 9.3 into Scala code. For each inference rule, the Scala code proceeds from the bottom left, from the syntactic shape of the tree in the conclusion of the rule, to the top of the rule, where the premises are checked, and finally to the bottom right, where the inference rule specifies the type to be returned, as shown in the following diagram.

$$\frac{\Gamma \vdash E_1 : \mathbf{Int} \quad \Gamma \vdash E_2 : \mathbf{Int}}{\Gamma \vdash E_1 + E_2 : \mathbf{Int}}$$

Since `typeOf` is recursive and we will want to know the type associated with each subtree of the parse tree during code generation in Assignment 10, the handout code maintains a memoization map `treeToType` for you that records all the types that `typeOf` has returned for all the parse tree nodes.

The well-formedness rules are implemented using Scala code in the implementation of the `ProcedureScope` class and in the `typeTree` method provided in the handout code that creates the initial outermost symbol table. The most significant issue specified and enforced by these rules is the addition of new declarations to the symbol table to be used to type check the body of each procedure and to be passed as the symbol table for checking well-formedness of nested procedures.