10 Overall compiler structure

In the first half of the course, we developed abstractions on top of machine language to implement high-level programming language constructs. Later, we developed lexical, syntactic, and context-sensitive analyses of high-level programs. We can now combine these two parts to complete our compiler.

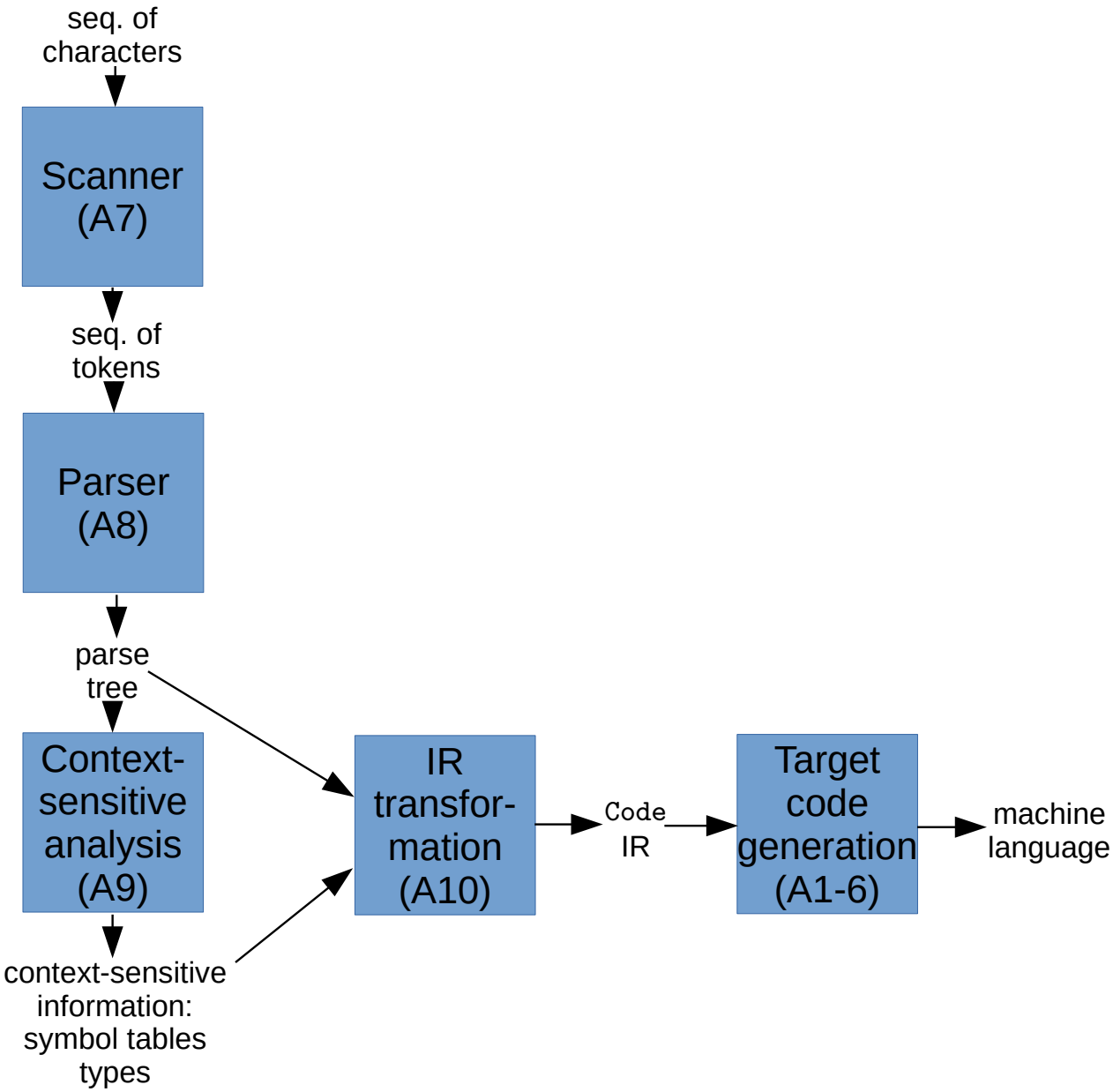
Both the parse tree and the `Code` tree are examples of *intermediate representations*. In a compiler, an *intermediate representation (IR)* is a data structure that represents the program being compiled. Many compilers have multiple intermediate representations and multiple translation passes between them to *gradually* transform a source-level view of the program into the final machine language code as a sequence of multiple transformation steps.

In the Lacs compiler, the translation to the `Code` IR can be defined recursively on the nodes of the parse tree. Specifically, we write a function that takes a parse tree node as an argument and returns the `Code` tree node to implement it. Such a function is typically recursive, in that the `Code` for a parse tree is built up from the `Code`s for subtrees of the parse tree that are returned from recursive calls.

The `Code` to be generated depends not only on the parse tree, but also on the additional information that is computed by context-sensitive analysis. One example of this is the symbol table: it is needed to list the variables to be allocated in each frame (expressed with a `Scope` in our compiler) and to map each appearance of an identifier within the program to the variable or procedure that it designates.

Types associated with parse tree nodes also affect the code to be generated. For example, in a Scala program, the expression `a + b` compiles to addition if the types of `a` and `b` are `Int`, but to concatenation of their types are `String`. Although Lacs does not have strings, a Lacs compiler has a similar need for type information when generating a call to a closure. In such a call, we do not know during compilation which target procedure will be invoked; this is determined by the run-time value of the closure expression. Therefore, when generating code to pass arguments to the closure, the compiler cannot depend on the parameter layout of any specific target procedure. In Assignment 6, we side-stepped the issue by requiring a `CallClosure` to include an extra sequence of parameter variables to specify to the compiler how many parameters the target procedure has. When we translate a parse tree node to a `CallClosure`, it is the *type information* attached to the parse tree node that enables us to generate this sequence of parameter variables. Recall that one of our definitions of a type was a computable property of programs that guarantees some property about their execution. Although we cannot know at compile time which specific procedure a closure call will execute, the type of the closure does tell us the number and types of parameters that this unknown procedure will take. For example, if the type of the closure is `(Int, Int) => Int`, we know the procedure will take two integer parameters. The compiler needs exactly this information to generate the sequence of parameter variables in the `CallClosure`. This sequence of parameter variables is then used to generate code to allocate memory for the parameters at the call site, as you implemented in Assignment 6.

The following diagram summarizes the overall structure of a compiler and of the Lacs compiler in particular:



The scanner turns a sequence of characters into a sequence of tokens. The parser turns a sequence of tokens into a parse tree. The context-sensitive analysis computes context-sensitive information, symbol tables and types, from the parse tree. The intermediate representation consisting of the parse tree, symbol tables, and types is then transformed into another intermediate representation, `Code`. The `Code` intermediate representation is specific to our Lacs compiler. Other compilers have different sets of intermediate representations. Finally, the target machine language code is generated from the intermediate representation.

Since each of the transformations in Assignments 1 to 6 eliminates some kinds of `Code` from the intermediate representation, we could also consider each of these elimination steps to be an IR transformation that turns one IR into another (slightly) different IR. For example, `eliminateClosures` is a transformation from the general `Code` IR to a `Code` IR without `Closures`; `eliminateCalls` is then a transformation from that IR to one without `Closures`, `CallClosures`, and `Calls`; `eliminateIfStmts` is a transformation from that IR to one without

`Closure`s, `CallClosure`s, `Call`s, and `IfStmt`s, and so on. In that sense, the final machine language code that is generated is just the culmination of a long sequence of intermediate representations.