1 Introduction

1.1 What is a compiler?

The first thing that springs to mind when most people think of a compiler is a programming-language translator. For example, the C compiler takes a C program text as input, and generates binary or assembly language machine code that performs the specified computation.

It is important to realize that compiler technology is useful for a more general class of applications. Many programs share the basic properties of compilers: they read textual input, organize it into a hierarchical structure, and then process the structure. An understanding of how programming language compilers are designed and organized can make it easier to implement these compiler-like applications. More importantly, tools designed for compiler writing, such as lexical analyzer generators and parser generators, can make it vastly easier to implement such applications.

Document processing programs such as Tex have to break a document into hierarchical structures (e.g. sections, paragraphs, sentences, words, list items, etc.) and produce output suitable for printers. For example, many programs produce PostScript\textsuperscript{1} programs which are then sent to a printer.

Silicon compilers are translators from some hardware description language to a circuit description that can be implemented in VLSI. For example, a logic synthesis program might take a description of a circuit in Verilog HDL and produce a network of logic gates and latches implementing the description. Silicon compilers draw on programming language technology.
to interpret their input languages, but also (sometimes) for optimizations that are used in the resulting circuits.

Compiler compilers are programs that take a description of a computer language, such as a programming language, and generate parts of compilers, automatically. We will be using some compiler compilers as part of this course.

Intermediate representations Lots of programs generate intermediate files in some structured format. There is a constant need to translate between formats. Compiler technology can be useful for this task.

Unorthodox applications One of my friends once wrote an email processor that used a parser generator to deal with mail headers. Once you know about context-free grammars and parser generators, you may see unexpected problems to which they can be applied.

1.2 The structure of a compiler

Many compilers are organized as a stream of phases which communicate by intermediate representations. Even when the actual structure of the compiler doesn’t follow this organization, the ideas are still often useful for understanding how the compiler works.
Figure 2: A clean separation between the front end and back end makes it easier to support many source and target languages.

At the highest level, a compiler has a front end and a back end (see Figure 1). It is desirable for the front end to deal with aspects of the input language, but to keep it as independent of the machine (or, more generally, specific output format) as possible. Symmetrically, the back end should concentrate on dealing with the specifics of the output language, and try to remain independent of the input.

This structure is very important for retargetability. Think about the problem of generating a suite of compilers for $n$ different languages to $m$ different machines. The above organization would require $m$ front ends and $n$ back ends, for a total of $m+n$ new programs. If input-specific and output-specific issues were all mixed together (as they often are, unfortunately), each new input and output would require $m \times n$ rewrites of the whole system. This situation is shown in Figure 2.

Of course, separating language and machine issues is not usually so simple. There is often a requirement for a “middle” that is both language and machine dependent. I call the middle the representations phase. It depends on the answers to questions, such as: “Where are global variables stored?” “How are arrays indexed?”, and “How are procedure calls and return values handled?”

In reality, each new language/machine combination will require a certain amount of customization of the compiler. One goal of the compiler structure is to minimize the amount of work that needs to be done, and to isolate the dependencies so that they are easy to find.
Figure 3: The front end of a compiler generally consists of three phases: lexical analysis (lexing), syntactic analysis (parsing), and semantic analysis.

### 1.3 Structure of the front end

A typical compiler front end consists of three phases: *lexical analysis*, *syntactic analysis* (generally called parsing), and *semantic analysis*.

The lexical analyzer (“lexer” for short) is the first phase. It takes a stream of characters (the textual input file) and breaks it up into *lexemes* (i.e., “words”). Roughly speaking, lexemes are the smallest program units that are individually meaningful. In Java and C, there are several types of lexemes: identifiers (*abc*), reserved words of various types (e.g., `if` and `return`), integer literals (e.g. `42`), floating point literals (e.g. `1.2E-3`), various operators (e.g. `+`, `<=`), punctuation (e.g. `{`, `}`, `;`), string literals (e.g. "a string"), and comments (e.g. `/* a comment */`). In most cases, the lexer does some processing for each lexeme, returns a lexical type representing the kind of lexeme discovered, along with a data structure with additional information about the lexeme. For example, an identifier may have the name of the identifier associated with it. I’ll call the values returned by the lexer *tokens*.

The job of the parser is to recover the hierarchical structure of the program from the stream of tokens received from the lexer. The output of the parser depends on the implementation of the compiler. It may produce a *tree* representing the hierarchical structure of the input, or it may perform actions as it parses. In the latter case, all of the subsequent compilation may be performed during parsing. However, even if the parser does all the work, it is helpful to keep the parse tree in mind in order to understand what is happening.

The semantic analyzer is the first phase that deals with the *meanings* of programming language constructs. For example, it is the semantic analyzer that reports whether an identifier is properly declared, because the meaning of the identifier is established in the declaration. The specifics of semantic analysis depend crucially on the semantics of the language being processed, which vary greatly from language to language. However, in most cases, the semantic analyzer processes declarations of various kinds, decides what the types of expressions are (while checking for type errors), makes sure that procedures are called with the right number of parameters, and so on.

A *symbol table* is frequently connected to the semantic analyzer. When the semantic analyzer processes a declaration, it will store information about the declared entity in the symbol table. This information will be looked up later when the entity is encountered again. In addition
to keeping track of definitions, the symbol table has to keep track of scoping information: for example, a local variable is visible in the body of the procedure in which it is declared, but not outside the procedure.

## 2 Lexical analysis

Lexical analysis is one of the simplest phases of compilation. If not implemented carefully, it can also be the slowest, at least in simple compilers, because the lexer handles more data than other phases — it is the only phase that must read each input character individually.

Although it is not very difficult to write a lexer by hand, there are good tools that make it even easier. These tools are based on the theory of regular languages. They compile patterns, which are regular expressions describing the lexemes, into finite automata, which are then stored in tables or compiled directly into code.

The history of these tools follows a pattern that has been repeated for other compiler generation tools. First, the theory of formal languages is applied to define aspects of a language precisely. Then, the formal notation is computerized, and a compiler is built that translates a language definition into compiler parts. Perhaps it is not surprising that compiler people end up writing compilers to help them write compilers.

Although it is helpful to think of the lexical analyzer as a process that receives a stream of characters and produces a stream of tokens, the usual implementation is a function (say, getlex) which is called by the parser only when it needs the next token. The lexer maintains some global variables, including some pointers into the input stream and its current state, which is updated on each call to getlex.

### 2.1 Regular expressions

Since regular expressions are used for the patterns in the definition of a lexical analyzer, we review that notation here.

#### 2.1.1 Preliminaries

An alphabet is a finite set of characters. For many compilers the ASCII character set is the alphabet of interest. For Java, the alphabet consists of Unicode characters. A string is a finite sequence of symbols from the alphabet. A formal language (or just language) is a set of strings. The set may be finite, but is usually infinite. The empty string (the string of
zero length) is written \( \epsilon \). It is important to distinguish the empty string from the empty language, which is a set of no strings.

Two strings can be concatenated, yielding another string. The concatenation of string \( x \) and \( y \) is written \( xy \). The concatenated string consists of the elements of the first string followed by the elements of the second.

A string can be exponentiated by repeatedly concatenating it with itself. Exponentiation can be defined inductively: the basis is \( x^0 = \epsilon \) and the induction is \( x^{i+1} = xx^i \).

The definition of regular expression uses some operations on languages. Let \( X \) and \( Y \) be languages. The concatenation of \( X \) and \( Y \) is written \( XY \), and it is defined to be \( \{xy \mid x \in X \land y \in Y \} \) — the set of strings that can be generated by concatenating some string in \( X \) with some string in \( Y \). For example, if \( X = \{ab, ba\} \) and \( Y = \{cd, dc\} \), then \( XY = \{abcd, bacd, abdc, badc\} \).

Languages can also be exponentiated by repeatedly concatenating them. The inductive definition is \( X^0 = \{\epsilon\} \) and \( X^{i+1} = XX^i \). For example, if \( X = \{ab, ba\} \), then \( X^2 = \{abab, baab, abba, baba\} \). The Kleene closure, \( X^* \) of a set of strings \( X \) is the union of all of the exponentiated sets, \( \bigcup_{i=0}^{\infty} X^i \). In the above example, \( X^* = \{\epsilon, ab, ba, abab, baab, abba, baba, ababab, \ldots\} \) (the Kleene closure of a nonempty set of strings is infinite, so I can’t write out the whole set).

### 2.1.2 Regular expressions

Regular expressions are a notation for a certain class of languages, called the regular languages. A regular expression stands for a language. If \( R \) is a regular expression, we write \( L(R) \) for the language \( R \) represents. The set of legal regular expressions can be defined inductively, as can the language of each expression.

In the table below, we use | for the “or” of two regular expressions. This is the convention in most uses of regular expressions in computer programs, including lexical analyzer generators. In textbooks on formal language theory, \( \cup \) is often used for this operator.

<table>
<thead>
<tr>
<th>regular expression</th>
<th>language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon )</td>
<td>( L(\epsilon) = {\epsilon} )</td>
</tr>
<tr>
<td>( a(\in \Sigma) )</td>
<td>( L(a) = {a} )</td>
</tr>
<tr>
<td>( (R_1)(R_2) )</td>
<td>( L((R_1)(R_2)) = L(R_1) \cup L(R_2) )</td>
</tr>
<tr>
<td>( (R_1)(R_2) )</td>
<td>( L((R_1)(R_2)) = L(R_1)L(R_2) )</td>
</tr>
<tr>
<td>( (R_1)^* )</td>
<td>( L(R_1)^* )</td>
</tr>
</tbody>
</table>
As a convention, we drop the parentheses when the meaning is clear, using a default grouping that treats $|$ as $+$ in arithmetic, concatenation as $\times$, and Kleene closure as exponentiation (so $a|bc^*$ stands for $(a)||(b)((c)^*)$).

Example:

If the alphabet is $\{a, b\}$, what is the regular expression for the set of all strings that have at least one $a$ and one $b$?

Answer: $(a|b)^{*}a(a|b)^{*}b(a|b)^{*}|(a|b)^{*}b(a|b)^{*}a(a|b)^{*}$.

Another answer: $(a|b)^{*}(ab|ba)(a|b)^{*}$. (Justification: any string that has one $a$ and one $b$ has an $a$ right next to a $b$.)

2.1.3 Non-regular languages

Although regular expressions are very powerful, it is important to note that not all languages are regular. A classic example is the language of balanced parentheses. I will use $a$ to stand for left parenthesis and $b$ to stand for right parenthesis. The language is $\{a^n b^n | n \geq 0\}$; in other words, a sequence of $n$ $a$’s followed by $n$ $b$’s. There is no regular expression standing for exactly this language. There are regular expressions for subsets or supersets, but, to define a language, the regular expression must allow exactly the strings in the language: no more, no fewer.

2.1.4 Extended notation for regular expressions

There are notations for regular languages which make it possible to define languages more concisely, without increasing the languages that can be defined.

The first extension is the $?$ operator. If $R$ is a regular expression, $R?$ means “zero or one occurrence of $R$,” or “$R$ is optional.” To see that $?$ does not allow any new languages to be defined, we need only note that any expression with $?$ in it can be converted to an equivalent expression without $?$ by converting $R?$ to $(R|\epsilon)$.

Another extension is positive closure, written $R^+$. Positive closure is similar to Kleene closure, but stands for “one or more occurrences of $R$” instead of “zero or more . . . .” The positive closure can also be eliminated by replacing it by $RR^*$ or $R^*R$, so positive closure does not increase the range of definable languages.
Another extension is to allow user-defined abbreviations for expressions. Each abbreviation is a name followed by a regular expression; the final definition is the definition of the regular language. If a set of definitions is not recursive, it can be converted into an equivalent single regular expression by replacing each defined name by the corresponding expression until all names have been eliminated (obviously, this can make the expression much larger). A simple policy for preventing recursion is to require that every definition use only names that were defined previously.

Here is an example definition of Java floating point numbers using regular definitions (this definition is for both “floats” and “doubles”).

```
Digit     (0|1|2|3|4|5|6|7|8|9)
Expt      ((e|E)(+|−)?(Digit)+)
Suffix    (f|F|d|D)
Float     (Digit)+ . Digit* Expt? Suffix?
         | . Digit+ Expt? Suffix?
         | Digit+ Expt Suffix?
         | Digit+ Expt? Suffix
```

Writing a regular expression for this language would be even more painful without regular definitions!

3 Deterministic finite automata

One of the profound results of formal language theory is that there are several natural ways to define classes of languages, all of which turn out to be equivalent. The simplest result is that regular expressions and finite automata both define the regular languages. While regular expressions are very good for user-level definitions, it is easier to write efficient code for a lexical analyzer based on an automaton. In particular, a deterministic finite automaton (DFA) is especially appropriate as a basis for generating an automatic lexical analyzer.

An automaton is a recognizer of the strings in a formal language. It takes a string as input, and responds with “accept” if the string is in the language, or “reject” if it is not.

A DFA can be defined mathematically to consist of five components:

- A finite alphabet $\Sigma$. The inputs to the automaton are strings over $\Sigma$.
- A finite set of states $Q$. 

Figure 4: The DFA for Modula 2 identifiers

- A next-state function $\delta : Q \times \Sigma \rightarrow Q$.
- A start state $q_0$, which is a member of $Q$.
- A set of accepting states, $F \subseteq Q$.

In this class, we will consider $\delta$ to be a partial function, meaning that $\delta(q, a)$ is undefined for some $q \in Q$ and $a \in \Sigma$. This is just a small convenience feature; such an automaton can always be converted to a DFA with a total function for $\delta$ by adding one more state.

Figure 4 shows a DFA for Modula 2 identifiers which are described by the regular expression `letter(letter|digit)+` (letter and digit represent sets of characters).

In this automaton, $\Sigma$ is the ASCII character set; $Q = \{s_0, s_1\}$; $\delta$ is defined so $\delta(s_0, a) = s_1$ when $a$ is a letter, $\delta(s_0, a)$ undefined when $a$ is not a letter, $\delta(s_1, a) = s_1$ when $a$ is a letter or digit, and $\delta(s_0, a)$ is undefined when $a$ is not a letter or digit; $q_0 = s_0$; and $F = \{s_1\}$.

A run of a DFA is on an input string $x = a_0a_1 \ldots a_{n-1}$ is a sequence of states $q_0, q_1, \ldots, q_n$ where $q_0$ is the initial state and $q_{i+1} = \delta(q_i, a_i)$. A run is said to be accepting if $q_n$ is a final state ($q_n \in F$). There can be at most one run on any input string. Since we have permitted $\delta$ to be a partial function, there may be no runs when $\delta$ is undefined at one of the steps of the run. The automata accepts an input $x$ if its run is accepting, and rejects if the run is not accepting or if there is no run.

A run of the DFA in Figure 4 on the input “abc” would be $s_0, s_1, s_1, s_1$. This is an accepting run since $s_1$ is an accepting state.

A regular expression can be converted to a DFA automatically, although the process is not trivial (it is better done by a computer than a human), and can result in very large DFAs in some cases. Typically, the DFAs for lexers in programming languages are of reasonable size.

Let’s consider a more complicated example of a lexical definition. Real numbers in Modula 2 are simpler than real numbers in Java. The regular expression is `digit+.digit*(E(+-)?digit+)?`. The DFA is shown in Figure 5. When I ask the class to do this, there are frequently errors in handling the $(+-)?$ at the beginning of the exponent, when someone tries to put an $\epsilon$ transition from $s_4$ to $s_5$. $\epsilon$ transitions are not allowed in DFAs (intuitively, the $\epsilon$ intro-
Figure 5: A DFA for Modula 2 real numbers.

roduces nondeterminism, since there can be several different runs depending on whether the automaton chose to take the $\epsilon$ transition or not).

Nondeterministic finite automata (NFAs) are not used very much in compilers. This may be surprising at first, because we know that NFAs can be much smaller than DFAs for some languages. However, there is usually not much savings for real programming languages (the DFAs and NFAs are almost the same size), and the biggest problem is that a lexical analyzer based directly on an NFA would generally be slower, since it must keep track of many possible runs. It is sufficient to keep a set of states when each symbol is read, but that is still much slower than tracking a single state, as in a DFA.

4 Theory versus practice

There is an almost perfect match between regular expressions to the lexical analysis problem, with two exceptions:

1. There are many different kinds of lexemes that need to be recognized. The lexer treats these differently, so a simple accept/reject answer is not sufficient. There should be a different kind of acceptance for each different kind of lexeme.

2. A DFA reads a string from beginning to end, then accepts or rejects. A lexer must find the end of the lexeme in the input stream. Then, the next time it is called, it must find the next lexeme in the string.
The solution to problem 1 is to build separate DFAs corresponding to the regular expressions for each kind of lexeme, then merge them into a single combined DFA. In order to keep track of which lexeme was recognized in the combined DFA, it can have many different kinds of accepting states, one for each type of lexeme.

As a very simple example, suppose we had a lexer that recognized simple integers with pattern \( \text{digit}+ \) and Modula 2 reals, as given by the pattern and DFA above. The combined DFA is shown in Figure 6. Note that states are labelled with the type of lexeme they accept: if the run stops at a state labelled “INT”, the lexer performs the action for an integer; if it stops at a state labelled “REAL”, it performs the action for a real; otherwise, it rejects the input.

Making a combined DFA is not always this simple. Sometimes, states have to be split in order to keep it deterministic. For example, the DFA in figure 7 recognizes patterns for “if”, “else”, and the general pattern for Modula 2 identifiers above. (By the way, this happens on a much larger scale if you have the lexer recognize a bunch of reserved words using patterns, instead of storing them in a hash table.)

The last example illustrates a further difficulty that can occur in the combined DFA: now that we have different types of accepting states, what do we do when a state has more than one type (which lexeme have we found)? The policy in lexical analyzer generators like Lex and Flex is to choose the type corresponding to the first definition appearing in file the and discard the rest. This can be extremely convenient when there is a simple general case with a collection of exceptions. The patterns for the exceptions can be defined first, followed by the simple pattern for the general case. On the other hand, it is always possible to write a regular expression for the general case with the exceptions removed, but such an expression can be very complicated. There can also be dangers, where the lexer depends on the order
Figure 7: A combined DFA for some reserved words and Module 2 Identifiers. The transition from the start state to the “ID” state should be taken on any letter except \(i\) or \(e\). Other unlabelled transitions should be taken on any letter or digit except those already labelling transitions out of the state.

pattern unbeknownst to the user. The problem here is that lexer does not do what the user expects, and it may be difficult to detect the discrepancy by testing it.

The combined DFA of Figure 6 illustrates the second problem. An input like “1.23” goes through several types of accepting states. When should it decide it is really at the end of the lexeme? The best policy, which is used in most tools, seems to be the “longest lexeme rule.” The lexer reads characters until there is no hope of ever getting to another accepting state, then it backs up to the most recent accepting state it saw, and uses that as the end of the lexeme (and the next input character as the beginning of the next lexeme).

The longest lexeme rule does the right thing almost all of the time. There are some cases where it fails, though. For example, suppose a lexer handled comments in Java by recognizing the lexemes “/\*” and “/\*\*/”, then entering a special comment processing mode. The Java specification says that “/\*\*/” is a complete comment, so, in this case, we want the first lexeme to be “/\*”, not “/\*\*/”. For another example, Modula 2 defines real literals so that “1.” and “.2” are legal. But it also demands that “1..2” be interpreted as a range declaration, consisting of lexemes “1”, “..”, and “2”. Both problems can be solved using the trailing context feature of both Lex and Flex: a pattern can be followed by “\*” and another regular expression, meaning that the first pattern matches only if it is followed by a string matching the second pattern. The Java comment problem can be solved by matching “/\*\*/” only if it is not followed by “/\*/”; the Modula 2 problem can be solved by recognizing “1.” as a real only if it is not followed by another “.”. Another simple solution to the Java comment
problem is to have a special pattern for “/**/” which will match in preference to “/*” and “/**” because it is longer.

The longest lexeme rule raises an efficiency and language design issue. The lexer may have to read some input characters before discovering that it is beyond the end of the longest lexeme, then it must back up. When it tries to match the next lexeme, it will see the same characters again. In the worst case, it may have to read each character almost as many times as there are characters in the file, which would be very slow. It also has to save all of the lookahead characters, which is annoying if it requires the lexer's input buffer to expand or to overflow unnecessarily (most languages have some unbounded constructs, such as strings, which may require grow-able buffers even if there is not a lot of lookahead, but long lookaheads may lead to buffer size problems where there were none before).

Let's think about how a lexer implementing the longest lexeme rule would have to work. First, there would be a pointer that says what the next character to read is (call it `nextchar`). There will also have to be a pointer to the beginning of the lexeme, because the action associated with the lexeme generally needs to know what string in the input file matched the pattern (e.g. to construct numbers or build identifiers); let's call this `startchar`. Finally, the lexer may need to back up to the longest lexeme so far, so it needs to keep a pointer to the end of the lexeme; for convenience, we'll have it point to the character right after the lexeme, and call it `lastchar`.

We can imagine a loop in a lexical analyzer something like the following. The input is treated as an array of characters, and the pointers are integer indices into this array. The $\delta$ function is encoded as an array that has $-1$ entries when the next state is undefined.
/ * initialization */
startchar = lastchar = nextchar = 0;
curstate = start_state;

while (curstate != -1) {
    /* take another step through the DFA */
curstate = delta[curstate, input[nextchar++]];
    if (curstate == -1) {
        /* we have found the longest lexeme. */
        /* string is from input[startchar..lastchar-1] */
        <perform lexical action on current lexeme>
        /* start next lexeme just after this one. */
        startchar = nextchar = lastchar;
    }
    else if (isaccepting(curstate)) {
        /* found a longer lexeme */
        lastchar = nextchar;
    }
    /* otherwise, continue looping */
}

Using this program, we can define the lookahead on an input string \(x\) as \(\text{nextchar} - \text{lastchar}\) then the lexical action is performed. Note that lookahead depends on all of the patterns in the lexer (it depends on the entire combined DFA).

Consider the input “123+5” in the combined DFA for integer and real literals. The \(\delta\) table will return a \(-1\) entry when \(+\) is read, at which point the \(\text{lastchar}\) pointer will be \(3^2\) (the index of \(+\)) and the \(\text{nextchar}\) pointer will be \(4\) (the index of 5 in the input), so the lookahead for this input will be 1.

On the other hand, if the input is “123E+X”, the \(\delta\) table will return \(-1\) after the \(X\) is read, at which time the longest lexeme will be 123. The lookahead will be \(\text{nextchar} - \text{lastchar} = 6 - 3 = 3\).

The maximum lookahead is defined to be the greatest lookahead for all possible strings. For the combined DFA for integers and reals, maximum lookahead is 3, and “123E+X” is an example of an input that causes it. You can determine the maximum lookahead by inspecting the combined DFA. It is the length of the longest sequence of non-accepting states that you can visit to a accepting state from an initial state or another accepting state, plus one (for the character that is read at the end for which \(\delta\) is undefined).

\(^2\)I'm assuming the first char is at index 1 here. It doesn't matter, because we only care about the difference between the \(\text{lastchar}\) and \(\text{nextchar}\) indices.
The maximum lookahead may actually be unbounded for some patterns. The longest path of non-accepting states may be unbounded if there is a cycle along the path.

A final detail: according to the definition above, the lookahead for a string must be at least 1. This doesn’t seem right for examples like strings delimited by double quotes — once the closing double-quote has been read, the lexer doesn’t have to read anything to know it is at the end of the lexeme. We could adjust the code for the lexer, above, to make the definition work right by marking accepting states that have no successors. In such a case, it is clear that no possible next character could ever lead to an accepting state, so the lexer might as well accept the input immediately, with lookahead of zero.