A Note on Parsing and Parser Generators*

Michael J.A. Smith

1 Parsers and Compilers

Writing a program is a joyful experience. We sit at the computer, type in a sequence of symbols that encodes what we want the computer to do, and then, as if by magic, the compiler comes along and converts it into something the computer can understand! The computer has no knowledge of variables, or functions, or Java objects. Neither does it know what a `while` loop, or a `for` loop is. And things like semicolons and spaces are as alien as a visitor from Mars.

Processors execute only the simplest of instructions—things like adding two numbers together, or testing whether a number is greater than zero. These instructions operate on *registers* rather than variables, and data structures need to be loaded from and stored into memory, as they are operated upon. The compiler is the tool that converts a high-level programming language into such low-level instructions, and the process of doing this is quite complicated. Luckily, we can break it down into smaller, more manageable stages.

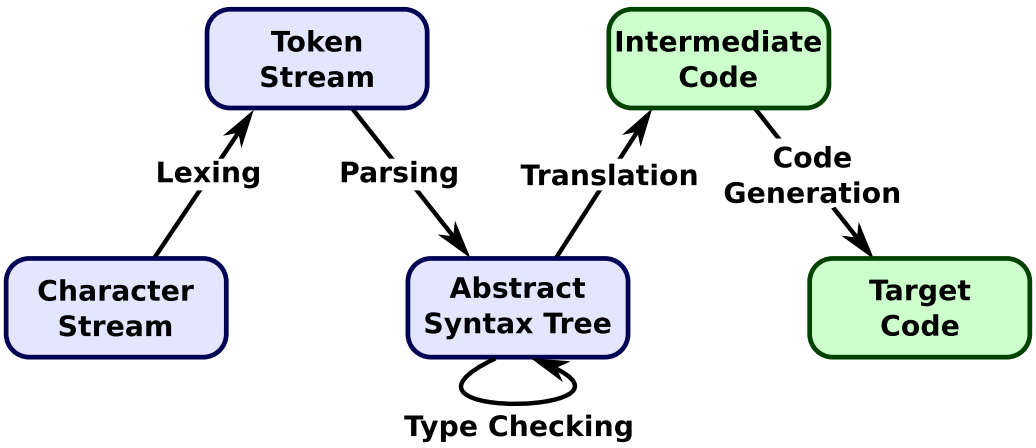


Figure 1: The basic structure of a compiler

Figure 1 shows the general structure of a compiler. Most of these steps are beyond the scope of this course, but our knowledge of regular and context-free languages can help us to solve the first two steps—namely, how do we get from a stream of characters into a representation that describes the syntax of the language? These steps are *syntactic*, in that they concern the structure of the program, rather than what it means, or its *semantics*.

*Adapted for *02141 Computer Science Modelling* 2017

1.1 Lexing

Imagine that you have just written a program in a familiar language, such as Java. Part of your program might look like the following:

```
if (x == 42) { y = "42 is the answer"; }
```

To us, the meaning of this is very clear, but to the computer it looks like the following:

```
i f ␣ ( x ␣ = = ␣ 4 2 ) ␣ { ␣ y ␣ = ␣ " 4 2 ␣ i s ␣ t h e ␣ a n s w e r " ; ␣ }
```

In other words, the computer just sees a sequence of characters. For example, if it sees the character `2`, how does it know whether it is part of a number, or an identifier, or even a string literal?

The job of the *lexer* is to convert such a stream of characters into a stream of *tokens*, which more effectively capture the syntactic elements of the programming language. This means identifying sequences of characters which together form a keyword, an identifier, or a number (i.e. a single syntactic element), and representing this as a single token. The lexer also removes parts of the character stream that are irrelevant to the program, such as comments and whitespace.

The above example would result in a token stream such as the following, after being lexed:

```
IF LPAREN ID("x") EQ NUM(42) RPAREN LBRACE ID("y") ASSIGN STRING("42 is the
                                    answer") SEMICOLON RBRACE
```

Note that some tokens have data associated with them—an identifier has a string attached to it, for example. The important point, however, is that each token is a single entity, rather than a sequence of separate characters.

So how does the lexer work? We can take advantage of the fact that most tokens can be described very simply—in particular, we can write *regular expressions* that describe them. Here are a few examples, written in the Java syntax for regular expressions (see the slides from lecture 4):

- Positive integers: `[0-9]+`
- Identifiers: `[A-Za-z][A-Za-z0-9]*`
- Keywords: `if`, `while`, etc.
- Whitespace: `[ \t\r\n]+`

Exercise 1 *Write a regular expression that describes floating point numbers, using the Java syntax for regular expressions. Your expression should be able to recognise numbers such as* `1.345E-6`.

The lexer does not just recognise that a string matches a regular expression—it also *outputs a token* when it finds a match. We can think of the lexer as a finite automaton that reads the character stream until it finds a match, at which point it outputs a token[1]. Because of this, the iterative operators (`*` and `+`) are interpreted *greedily*—they match as many characters as possible before outputting a token.

[1] In the case of matching whitespace, or a comment, the lexer does not output a token, but instead discards the characters it has read. It then starts again, trying to read a token from the next part of the input stream

Exercise 2 *Given that keywords often look similar to identifiers (i.e. they are a sequence of alphanumeric characters), how does the lexer know when to output a keyword token, and when to output an identifier token?*

We will assume from now on that we have a function `lex()` that returns the next token in the input stream—this will help us when building a *parser*, since it means we can take the tokens as input, rather than the individual characters.

1.2 Parsing

After lexing, we have a stream of tokens, but this still does not capture the structure of the program. The job of the parser is to convert this into an *abstract syntax tree*—a data structure that describes the syntactic structure of a program. As an example, consider the following fragment of a grammar, where N represents a number and I represents an identifier:

$$E \to E + E \mid E * E \mid N \mid I$$

How can we build a data structure for such arithmetic expressions? The specific answer to this question depends on the programming language we are using, so let us restrict our attention to Java. We can begin by creating an abstract class corresponding to arithmetic expressions E:

```
public abstract class ArithExpr { }
```

We can now notice that we have four types of arithmetic expression, according to the grammar. For each of these, we can create a subclass of `ArithExpr`. For example, we can create a class `PlusExpr`, describing the addition of two expressions ($E + E$), as follows:

```
public class PlusExpr extends ArithExpr {

    private ArithExpr expression1;
    private ArithExpr expression2;

    public PlusExpr(ArithExpr e1, ArithExpr e2) {
        expression1 = e1;
        expression2 = e2;
    }

}
```

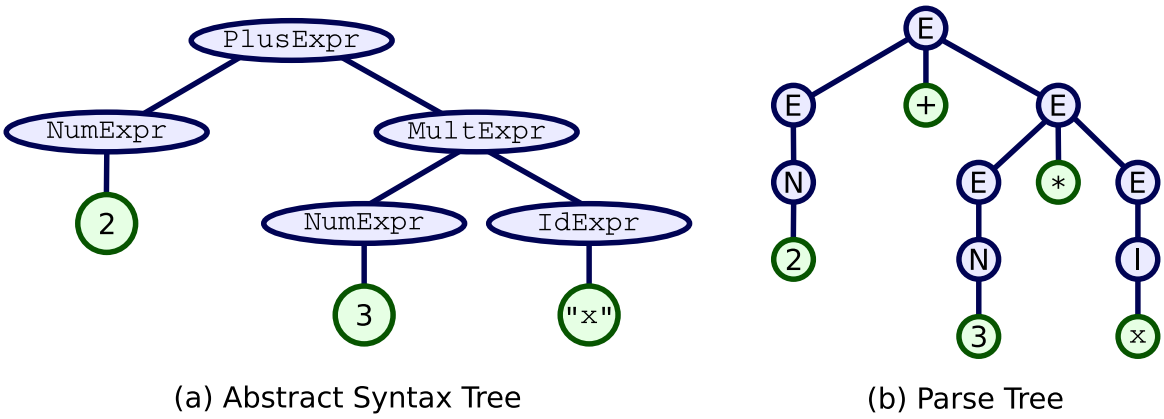
Exercise 3 *Write down a Java class* `IdExpr` *for the identifiers I. You can assume that it has just one field, corresponding to the name of the identifier.*

If we want to represent an expression $2 + 3 * x$, we can construct an object in the abstract syntax as follows:

```
ArithExpr e =  new PlusExpr(new NumExpr(2),
                   new MultExpr(new NumExpr(3), new IdExpr("x")));
```

Figure 2 shows this expression graphically—as an abstract syntax tree—with the corresponding parse tree shown alongside it[2]. These two data structures are related, but are clearly not the

[2]You should notice that the parse tree is not unique, since the grammar used in our example is ambiguous. We have picked the parse tree that corresponds to the structure of the abstract syntax tree. However, if we want to build a parser that generates an abstract syntax tree from an expression, we will have to make sure that our grammar is unambiguous.



(a) Abstract Syntax Tree (b) Parse Tree

Figure 2: An abstract syntax tree and a parse tree for the expression $2 + 3 * x$

same, as we can see from the figure. The purpose of a parser is to convert a token stream into an abstract syntax tree, which is then passed on to the later stages of the compiler. We will now turn our attention to how we can actually write such a parser, given a context-free grammar for the language!

2 Building a Parser—by Hand

If we have a context-free grammar, there are two ways in which we can build a parser for the language described by that grammar:

1. We can build the parser **by hand**, looking at each variable, and building a function that allows us to read a string in the language of that variable.

2. We can use a tool, called a **parser generator**, where we input the grammar, and it automatically generates the code for a parser for us!

This section describes in detail how to build a parser by hand, since it helps to understand how automated parser generators work—and how to use them. In the following section we will then use such an automated parser generator, *ANTLR*, to build a parser for a more complex language automatically. We focus here on a *top-down approach* to parsing, which is known as *recursive descent*.

To start with something straightforward, let us take a look at the following grammar, where E and A are variables, corresponding to expressions and atoms (numbers or identifiers) respectively:

$$E \to A + A \mid A - A$$

We will assume that '+' and '−' respectively produce the tokens `PLUS` and `MINUS` from the lexer. We want to write a function `readE()`, which checks whether the stream of tokens from the lexer is a valid string in the above grammar. We assume here that we have already a function `readA()` that reads an atom. It is natural to write something like the following code fragment:

```
Token token;

void readE() {
    readA();
    if (token == PLUS)       { token = lex(); readA(); }
    else if (token == MINUS) { token = lex(); readA(); }
    else                     { error("expecting + or -"); }
}

void readA() {
    ⋮
}
```

This simply tries to first read an A. (The function `readA()` shall throw an error if it is unable to do so.) Then we continue to check whether the next token is a '+' or a '−'. If it is, it discards the token by calling `lex()`, which replaces it with the next token in the stream, and then tries to read another A. If it encounters an unexpected token, it throws an error. Note that global variable `token` always contains the next token to parse.

This seems straightforward enough, but the grammar we were looking at is very restrictive—it allows expressions such as '`x + 5`', but not of the form '`x + 5 - y`'. To be a bit more realistic, let us extend our grammar to allow a sequence of additions and subtractions:

$$E \to E + A \mid E - A \mid A$$

Exercise 4 *Is this grammar left associative or right associative? Consider the structure of its parse trees.*

Let us try to adapt our function `readE()` accordingly:

```
void readE() {
    readE();
    if (token == PLUS)       { token = lex(); readA(); }
    else if (token == MINUS) { token = lex(); readA(); }
}
```

Here we have a problem to write the last alternative for E, namely a single atom A without plus or minus. But this is part of yet a bigger problem: `readE()` as its first action will call `readE()` which directly gives in an infinite loop. This problem is called *left recursion*, and we cannot write a recursive-descent parser for such a grammar[3]. We could instead change the grammar as follows, so that the recursion occurs on the right instead:

$$E \to A + E \mid A - E \mid A$$

We can now write a recursive-descent parser that actually terminates!

[3]More in detail, one needs a particular form of grammars, called *LL grammars*, in order to build a recursive-descent parser. The same restriction applies to ANTLR, because it uses essentially the same technique that we describe here. Other parser generators such as YACC, however, use a different form of grammars known as *LALR*, and can handle grammars with left-recursion.

```
void readE() {
    readA();
    if (token == PLUS)       { token = lex(); readE(); }
    else if (token == MINUS) { token = lex(); readE(); }
}
```

This is a correct parser for determining whether a string is in the language of our grammar, but we have introduced a bug—we had to change the associativity of our operators. This is a problem when we construct our abstract syntax tree, since we would like to be able to choose their associativity.

Luckily, we can correct for the bug that we introduced, using a simple trick. We can notice that our grammar for E just describes an atom, followed by a sequence of one or more additions or subtractions of atoms. We can therefore rewrite the function `readE()` as follows, converting it from a recursive to an iterative function:

```
void readE() {
    readA();
    while (token == PLUS || token == MINUS) { token = lex(); readA(); }
}
```

This corresponds to a direct implementation of the following grammar, where we use a regular expression in the body of the production:

$$E \to A\ (+\ A \mid -\ A)^*$$

We can easily express this as a grammar without a regular expression as follows:

$$\begin{aligned} E &\to A\ E' \\ E' &\to +A\ E' \mid -A\ E' \mid \epsilon \end{aligned}$$

This may not look like a substantial change, but it allows us to construct a parse tree with the appropriate associativity. Let us assume that we have Java classes `PlusExpr` and `MinusExpr` that both extend an abstract class `ArithExpr` that represents all arithmetic expressions. We can then modify our parser as follows, so that it returns an `ArithExpr` object that describes the token stream—in other words, it constructs an abstract syntax tree!

```
ArithExpr readE() {
    ArithExpr ast = readA();
    while (token == PLUS || token == MINUS) {
        if (token == PLUS) {
            token = lex();
            ast = new PlusExpr(ast, readA());
        } else {
            token = lex();
            ast = new MinusExpr(ast, readA());
        }
    }
    return ast;
}
```

Exercise 5 *Convince yourself that the above parser generates a left-associative abstract syntax tree.*

3 ANTLR: An Automated Parser Generator

We have seen how to write a recursive-descent parser by hand, but this is very fiddly, and we can easily make mistakes when translating a grammar into a parser. Luckily, there are a large number of tools available, which can *automatically* build a parser from a grammar. We will look at one such tool, called ANTLR, which actually uses the same sort of approach as we have just seen!

Unlike some parser generators, which have separate tools for generating a lexer and a parser (for example, LEX and YACC), ANTLR allows us to describe both the lexer and the parser in the same grammar file. It even provides a tool called ANTLRworks, where we can easily debug grammars and generate parse trees, among other things.

Before we start to use ANTLR, let us take a look at a simple programming language that we will be working with. We will call this the *while language*. The while language has three syntactic categories—statements S, Boolean expressions (or conditions) C, and arithmetic expressions E. Assuming that we already have tokens for identifiers I and numbers N, the grammar is as follows (including additions you are supposed to implement):

			Intuitive Meaning:
S	$\rightarrow$	I `:=` E	evaluate E and store result in variable named I
	|	`skip`	do nothing
	|	S `;` S	execute statements in sequence
	|	`{` S `}`	(parenthesis for building statement blocks)
	|	`if` C `then` S `else` S	conditional...
	|	`while` C `do` S	while-loop...
C	$\rightarrow$	`true`	
	|	`false`	
	|	E `=` E	
	|	E `<=` E	less or equal
	|	`!` C	logical *not*
	|	C `&&` C	logical *and*
	|	C `||` C	logical *or*
E	$\rightarrow$	N	natural number
	|	I	identifier (variable name)
	|	E `*` E	
	|	E `+` E	
	|	E `-` E	
	|	`(` E `)`	

Note that the variables in this language represent only integers. The above grammar is of course ambiguous—you should, by now, be able to spot the ambiguities due to operator precedence and associativity. In the ANTLR grammar for this language, the ambiguities have been resolved, so you should compare the grammar with the one above, once you have the tool up and running.

Installation and Use All files we need are found in a zip-archive on campusnet, including the version 3.5.2 of ANTLR (don't use any other version from the Internet!):

```
antlr-3.5.2-complete.jar
```

We first describe the installation for command line, below some hints for using Eclipse and the ANTLRworks GUI.

On Linux or MacOS, you can put the ANTLR jar file into a directory like `/usr/local/lib/`. Add it to your java `$CLASSPATH`, by adding to your `~/.bashrc` the line:

```
export CLASSPATH=".:/usr/local/lib/antlr-3.5.2-complete.jar:$CLASSPATH"
```

also you can define yourself a shorthand for invoking ANTLR:

```
alias antlr='java -jar /usr/local/lib/antlr-3.5.2-complete.jar'
```

For Windows you need to find the system setting for environment variables and either create a variable named `CLASSPATH` or append to it if it exists, the ANTLR jar file. (Note that Windows uses backslash for paths.) To get the abbreviated `antlr` command similar to the Linux/Mac, you can create a Windows batch file named `antlr.bat` that contains the text (replace `<ANTLR>` with the location of your ANTLR jar):

```
java -jar <ANTLR> %*
```

This batch file must be somewhere where Windows will find it (the `PATH` environment variable).

One can now compile grammars:

```
antlr <GRAMMAR.g>
```

where `<GRAMMAR.g>` is the grammar you want ANTLR to generate a parser for. The resulting files are the called `GRAMMARLexer.java` and `GRAMMARParser.java`, which can then be translated using `javac`.

For the while language, you can now compile the whole project if you are in the directory *above* `while_language` by the commands

```
javac while_language/ast/*java
javac while_language/parsing/*java
javac while_language/*java
```

and then run the resulting WhileLanguage interpreter on say test program `program1.while`:

```
java while_language/WhileLanguage while_language/test/program1.while
```

ANTLRworks and Eclipse In case you like GUIs and IDEs we recommend the following free software and installation:

1. *ANTLRworks* for writing the grammar of a parser, and for generating the Java code. On campusnet you find the version of ANTLRworks that we will use. (More on ANTLR on its webpage `www.antlr3.org/works`) You can start ANTLRworks from the command line:

   ```
   java -jar antlrworks-1.5.2.jar
   ```

2. *Eclipse*. You can download the latest version for your own computer from the following link (the version for Java developers will be sufficient):

   ```
   http://www.eclipse.org/downloads/
   ```

Once you have ANTLRworks and Eclipse ready, you will need to carry out the following steps to get started with the exercises. This will allow you to build a parser for the while language using ANTLRworks, and then compile and run an interpreter for the language from Eclipse.

1. Create a new "Java Project" in Eclipse, selecting "Use project folder as root for sources and class files" under "Project layout".
2. Copy the whole directory `while_language` into the folder of the new project created.
3. Create a directory called `lib` in the project folder, and place into it the JAR file `antlr-3.5.2-complete.jar`.
4. Right-click on `antlr-3.5.2-complete.jar` in the `lib` directory. Select `Build Path -> Add to Build Path`.
5. You can now generate the code for the lexer and parser:
   (a) Open ANTLRworks. If this is your first time running it, you will be prompted to create a new grammar—just enter an arbitrary name and continue.
   (b) Select `Preferences` from the main menu. Delete the text "`output`" from the field `Output path`, and click `Apply`.
   (c) Open the file `.../while_language/parsing/WhileLanguage.g` (the version that you just copied into your Eclipse project).
   (d) Select `Generate -> Generate Code` from the menu. This will create three new files in the same directory as the grammar, with the names: `WhileLanguage.tokens`, `WhileLanguageLexer.java`, and `WhileLanguageParser.java`.
6. In Eclipse, refresh the project (right-click it and select `Refresh`).
7. You are now ready to try running the parser and interpreter for the while language[4]:
   (a) Open the file `WhileLanguage.java`.
   (b) Select `Run -> Run As -> Java Application` from the menu.
   (c) This should produce the output "`Error: No program specified`" in the console.
   (d) Select `Run -> Run Configurations...` from the menu, and select `WhileLanguage`.
   (e) In the arguments tab, enter "`while_language/test/program1.while`" under program arguments.
   (f) Click `Run`. You should see the following output in the console:

   ```
   tmp -> 20 y -> 20 x -> 10
   ```

   This gives the values of all the variables in the program, after executing it.

Let us now take a closer look at the ANTLR grammar in the file `WhileLanguage.g`. If we look at the bottom of the file, we find the following:

```
NUM : '0'..'9'+ ;
ID  : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9')* ;

WS  : (' '|'\t'|'\r'|'\n')+ { $channel = HIDDEN; } ;
```

[4]If the project still has an error, you should clean it using `Project -> Clean`.

This is the *lexing* part of the file. Tokens in ANTLR are written in capital letters, and this defines the tokens `NUM` and `ID`, which represent numbers and identifiers respectively. The ANTLR syntax for regular expressions differs a little from that in Java, but the meaning of the above should be clear. The third line matches whitespace, and the command '`$channel = HIDDEN;`' tells ANTLR to ignore such characters, and not to pass them to the parser.

To see how *parser* rules work, let us look at one of the rules:

```
mult_arith_expr returns [ArithExpr value]
    : e=base_arith_expr       { $value = e; }
      ( '*' e=base_arith_expr { $value = new MultExpr($value,e); } )*
    ;
```

The basic form of a rule is just like a production in a context-free grammar, except that we allow regular expressions in the body of the production, and we annotate the production with *Java code*, which tells us how to build the abstract syntax tree. The above rule corresponds to a multiplicative arithmetic expression, which consists of a sequence of base arithmetic expressions that are multiplied together[5].

The variable is annotated with the text '`returns [ArithExpr value]`', which tells us that it returns a variable `value` of type `ArithExpr`. We refer to this variable within the body of the rule as `$value`—the '`$`' sign is to make sure that ANTLR does not confuse variables that stand for Java objects with variables that stand for non-terminals in the grammar. Notice that we construct a new `MultExpr` after reading each base expression, and this ensures the left-associativity of the operator.

You should spend some time looking at the other rules in the grammar, and make sure you understand it—there is detailed documentation for ANTLR on its website, and you can ask the teaching assistants if you have any questions.

[5]You should recall that the reason for separating out arithmetic expressions into different levels is so that we can enforce the correct binding precedence of the operators.