Syntax & Semantics

Readings for this section: PLP, Section 2.1 (required).

In this unit, we are going to start looking at how compilers and interpreters work. Recall the main stages of compilation (this is a review from unit 1!):

1) Scanning: Turning source code into a token stream. This stage removes white space and comments, and identifies what kind of token each piece of the code is.
2) Parsing: Turning a token stream into a parse tree. This stage checks that the sequence of tokens is grammatically correct and can be grouped together according to the specifications of how the language works.
3) Semantic analysis: Turning a parse tree into an abstract syntax tree. This stage cuts out any unnecessary or redundant information from the parse tree to form a concise representation of what the program means (i.e., the AST).
4) Code generation: Turning that AST into executable machine code.

As you might of guessed, this unit is about steps (1) and (2), scanning and parsing. These are the compilation stages that focus on the syntax of the language, i.e., the way the program is allowed to look. Later we will focus more on semantics, i.e., what the program actually means and is supposed to be doing.

In class, we looked at some examples of English sentences that were syntactically or semantically valid/invalid. Hopefully you are starting to get a good feel for what syntax and semantics are. This might also be a good time to go back and review the problem from Homework 1 on syntax errors and semantics errors.

1.1 Programming Language Specification

A computer program is a way of communicating something to the computer. In particular, the program is telling the computer what to do; it’s communicating an algorithm. The programming language is the medium or vehicle for that communication; it specifies a way to communicate algorithms.

Now we’re thinking about compilers and interpreters, and we have this same problem at the next level. How do we communicate what a programming language is? If we can specify programming languages in some well-defined way, then we can more easily write compilers and interpreters for new programming languages.

As it turns out, the syntax of programming languages is relatively easy to specify. This is because scanners and parsers (the parts of a compiler that identify syntax) correspond exactly to the kinds of simple machines you learned about in Theory class:

- The tokens for a language are specified with regular expressions. These can be used to create a scanner, which is implemented as a finite automaton (FA).
• The rules for how tokens can come together are specified with context-free grammars (CFGs). These can be used to create a parser, which is implemented as a push-down automaton (PDA).

So now you see, all that stuff you learned about in SI 340 wasn’t just for the incredible joy of proving interesting things about computation; those things actually exist inside every compiler and interpreter you’ve ever used!

1.2 Some tactical syntactical considerations

In Theory class, you wrote regular expressions and grammars to carefully define a language. The concerns there were mostly about mathematical correctness, that is, does your regular expression or CFG actually define the correct strings that are in this language.

But now we’re using these as tools to write compilers, so there are more concerns. For one, we want our compilers to be fast. This will affect some of our choices in how to write the regular expressions and grammars for our language, as we will see.

Another concern is how the information generated by one stage of compilation affects the next stage. For example, consider a simple calculator language with numbers and plus/minus/times/divides. We could specify the tokens as:

\[ \text{OP} = +|\-|\ast|/ \quad \text{NUM} = (+|\-)|[0-9]+ \quad \text{STOP} = ; \]

and the grammar as:

\[ S \rightarrow \text{exp STOP} \quad \text{exp} \rightarrow \text{exp OP exp} \quad \text{exp} \rightarrow \text{exp OPM factor} \quad \text{term} \rightarrow \text{term OPM factor} \quad \text{factor} \rightarrow \text{NUM} \quad \text{LP} \]

This correctly defines the language we want. Any valid calculator program matches this specification. But this is a really bad way to specify this grammar, for at least two reasons. First, the grammar is ambiguous, meaning that we could get two different parse trees for the same program. For example, if we try to parse \(5 + 3 \ast 2;\) with this grammar, it could group the \(5 + 3\) part together as a single \(\text{exp}\), or the \(3 \ast 2\) part. This will ultimately affect the semantics of the language - since one way of parsing will probably end up with 16 being computed, while the other way will end up with 11! This grammar is also bad for another reason, which has to do with the way parsers are implemented. More on that later...

2 Scanning

Readings for this section: PLP, Section 2.2 intro, 2.2.2, and 2.2.3, (required). All of Section 2.2 (recommended).

A better syntax for the simple calculator language is as follows. The terminals will be:

\[ \text{OPA} = +|\- \quad \text{OPM} = \ast|/ \quad \text{NUM} = (+|\-)|[0-9]+ \quad \text{LP} = ( \quad \text{RP} = ) \quad \text{STOP} = ; \]

And the grammar is:

\[ S \rightarrow \text{exp STOP} \quad \text{exp} \rightarrow \text{exp OPA term} \quad \text{term} \rightarrow \text{term OPM factor} \quad \text{factor} \rightarrow \text{NUM} \quad \text{LP} \]

\[ \text{exp RP} \]

See what we did? By splitting the operations according to their precedence, we will get an unambiguous grammar that corresponds to the ultimate meaning (semantics) of the language.
2.1 Scanner DFAs

In class, we saw what a DFA for the tokens above looks like. Basically, we have a normal DFA, but each of the final (accepting) states is labeled with the type of that token. There will definitely be at least one final state for each token type in the list, but there can also be more than one final state for the same token type sometimes.

This is all well and good for doing what a DFA does - determining whether a single string is in the language. But when we’re writing a compiler, we have to scan the entire source code into tokens. The big question is, how do we know when to stop scanning this token and move on to the next token?

For example, consider the code fragment `123*54`. Obviously we want this to be a NUM (123), then an OPM (*), then another NUM (54). But what’s to stop us from having two NUMs before the OPM, say 12 and 3? These questions would get even more hairy if the operation were a − instead.

Scanners resolve these questions systematically by using what’s called the maximal munch rule. This rule says basically “make each token as long as possible”. In terms of the DFA, it means that scanning each token proceeds until the next character would transition from an accepting to a non-accepting state. Applying this rule makes scanners very efficient (since they just have to peek ahead to the next character), but also powerful enough to handle pretty much every programming language you could think of. Does it mean there are some ways of defining a programming language that just won’t work anymore? Unfortunately, yes. But programming language designers are more than happy to make this sacrifice to get the speed and efficiency of DFAs.

One nice property of DFAs is that, besides being fast, they’re also simple to program. The file `calc-scanner.cpp` implements the scanner DFA for the tokens listed above. It is used in the bisoncalc program, which you can make using the Makefile and bisoncalc.ypp file linked at the top of this page. We saw what this looks like in class.

2.2 Automatic Scanner Generation

Programs like flex take a list of regular expressions to specify tokens, and then split up a whole source file according to those regular expressions, repeatedly applying maximal munch to get the longest possible tokens.

So far we have seen some examples creating DFAs by hand for simple scanners. But how does this process work in general? This is described in more detail in your assigned reading, but here’s an overview of the steps involved:

1. Turn each regular expression into an NDFA. You saw how this works in Theory class last year.
2. Mark each accepting state of each NDFA with the name of that token.
3. Combine all those NDfas into a single big NDFA using the “or” operation, like you learned in Theory class. What this means is that we will make a new start state, and empty-string ϵ-transitions from that new start state to each of the old start states.
4. Turn this big NDFA into a DFA, using the algorithm you learned in Theory class. Now there is a potential problem, if one of the accepting states in the DFA matches more than one token. In automatic tools like flex, this issue is resolved by giving precedence to whichever token is defined first.
5. Finally, the DFA is minimized using various neat tricks, to make the scanner run as fast as possible.

We looked at some examples of this process in class. See your book for more details and examples.

3 Parsing

Readings for this section: PLP, Section 2.3 intro and 2.3.4 (required), Section 2.4 (recommended).

Parsing is the process of turning a stream of tokens into a parse tree, according to the rules of some grammar. This is the second, and more significant, part of syntax analysis (after scanning).

We are going to see how to write grammars to describe programming languages. You already did this a bunch in Theory class. But in this class, the concern is deeper: we not only need the grammar to properly define the language in a strict mathematical sense; it also needs to make the parser be fast.
How fast? Well, scanning is going to be linear-time; just take one transition for every character in the input. We want our parsers to work in linear-time in the size of the input as well. This is important: if your code gets 10 times as long, you’re OK if it takes 10 times as long to compile. But if it suddenly takes 1000 times longer to compile, that’s going to be a problem! In fact, that’s what the case would be if we allowed any old grammar for our languages; the fastest algorithms for parsing any language have complexity $O(n^3)$.

What this means is, we have to talk about parsers and grammars together. The grammar determines how fast the parser can go, and the choice of parser also affects what kind of grammar we will write!

Parsers are classified into two general categories, and we will look at both kinds in some detail. Here’s a quick overview. We also saw some preliminary examples in class of how to parse a string top-down or bottom-up.

**Top-down parsers** construct the parse tree by starting at the root (the start symbol). The basic algorithm for top-down parsing is:

1) If the first unfinished part of the tree is a token (terminal symbol), *match* it against the next token of input, and discard that token (cross it off and move on).

2) Otherwise, the first unfinished part of the tree is a nonterminal symbol. By peeking ahead at the next unmatched token of input, figure out which right-hand-side production of that nonterminal to take, and expand the tree accordingly.

3) Repeat (1) and (2) until the tree is complete, or we run out of tokens, or another error occurs.

The big question that top-down parsers have to answer is on step (2): *which right-hand side do we take?* This is why top-down parsers are also called *predictive parsers*; they have to predict what is coming next, every time they see a non-terminal symbol.

Top-down parsers work well with a kind of grammar called LL. In fact, sometimes top-down parsers are called LL parsers. We’ll mostly focus on a special case called LL(1) grammars, which will be defined in the next section.

**Bottom-up parsers** construct the parse tree by starting at the leaves of the tree (the tokens), and building up the higher constructs (the non-terminals), until the leaves all form together into a single parse tree. The basic bottom-up parsing algorithm is:

1) If *any suffix* of the current string of non-terminals and terminals matches a right-hand side of a grammar rule, apply that rule in reverse by “building up” the tree, combining the part of the right-hand side that matched. This is called a *reduce* step.

2) Otherwise, if we can’t reduce, add a new leaf into the tree corresponding to the next token of the input. This is called a *shift* step.

3) Repeat (1) and (2) until the tree has been completely formed and is all connected with the start symbol at the top, or until an error occurs.

The big decision for bottom-up parsers is whether to do (1) or (2) every step along the way; that is, whether to shift or to reduce (and how to reduce, if there is more than one option). For this reason, bottom-up parsers are also called *shift-reduce parsers*.

Bottom-up parsers work well with LR grammars, so they’re sometimes called LR parsers. We will focus specifically on SLR(0) and SLR(1) grammars, which will be described below.

### 4 LL Parsers

**Readings for this section**: PLP, Section 2.3.1 and “Writing an LL(1) Grammar” starting on page 82 (required). All of 2.3.2 (recommended).

A grammar is called LL(1) if it can be parsed by a top-down parser that only requires a single token of “look-ahead”. Remember that top-down parsers use look-ahead to *predict* which right-hand side of a non-terminal’s production rules to take. So with an LL(1) grammar, we can always tell which right-hand side to take just by looking at whatever the next token is.
There are two common issues that make a grammar not be $LL(1)$: common prefixes and left recursion. Fortunately, both these issues have somewhat standard fixes. Let’s see what they are.

The following grammar is not $LL(1)$:

\[
X \rightarrow a \ b \ X \rightarrow a \ a
\]

Do you see what the problem is? If we are trying to expand an instance of $X$ in the top-down parse tree, we need to determine which of the two possible rules to apply, based on the next token of look-ahead. But that next token will always be an $a$, which doesn’t give us enough information to distinguish the rules!

The standard fix is to “factor out” the common prefix. First, we make a “tail rule” that has every part of each right-hand side except the common prefix. This should be a new non-terminal in the language, like:

\[
Y \rightarrow b \ Y \rightarrow 'a'
\]

Here, $Y$ gets the part of each right-hand side from $X$, but without the common prefix of $a$. Once we have this tail rule, we can combine all the productions of the original nonterminal into a single rule with the common prefix, followed by the new non-terminal. So the whole grammar becomes:

\[
X \rightarrow a \ Y \ Y \rightarrow b \ Y \rightarrow 'a'
\]

See how that works? It is very important to realize that the language did not change at all! The strings that are defined by this grammar are the same as they were before; the only difference is that the grammar itself looks different. In fact, we can see that this new grammar doesn’t have any common prefixes, and it is definitely $LL(1)$. Hooray!

I should also point out that this process might have to be repeated more than once in more complicated situations. For example, try to make the following grammar $LL(1)$:

\[
X \rightarrow a \ a \ a \ X \rightarrow a \ a \ b \ X \rightarrow a \ b \ b
\]

Left recursion is the other common issue that makes a grammar not be $LL(1)$. By “left recursion”, we mean that the non-terminal on the left hand side of a grammar rule is the same as the first non-terminal on the right-hand side. For example, this grammar defines a language that’s any string with at least one $a$:

\[
X \rightarrow a \ X \rightarrow X \ a
\]

Do you see what the problem is here? If we are trying to expand an $X$, the next token will always be an $a$! In fact, with more complicated kinds of left recursion, it won’t be possible to distinguish between the two possible productions with any amount of look-ahead. Left recursion is pretty much the worst thing that can happen to a top-down parser!

To get rid of it, we have to reorder things somehow. Usually this means flipping around whichever rule is left-recursive to make it right-recursive instead. In this case it works like this:

\[
X \rightarrow a \ X \rightarrow a \ X
\]

Again, it’s important to emphasize that the language has not changed, only the grammar. But what happened? We got rid of one problem (left recursion), only to get another one (common prefix). But now apply the fix we already know for common prefixes, and the solution emerges:

\[
X \rightarrow a \ Y \ Y \rightarrow a \ Y \ Y \rightarrow \epsilon
\]
That’s it! Now we have a properly LL(1) grammar for the same language, that will make top-down parsing nice and fast. Here, Y is called a “tail rule”, and this kind of thing shows up in LL(1) grammars quite frequently.

The full details of how a top-down parser works are detailed in the book, and you’ll also see it in lab. Here are the main points:

- **Recursive descent** parsers do top-down parsing by making recursive calls for every non-terminal in the language. Each function has to use the look-ahead token to figure out which right-hand side to apply. This is what hand-written parsers usually look like. They are the simplest kind of parsers and usually give the most helpful error messages.

- **Table-driven** top-down parsers do the same thing as recursive descent parsers, but instead of using recursive calls to keep track of what is going on, they store the currently-expanded nonterminals explicitly in a stack. This is what an auto-generated top-down parser will usually look like.

- Both of these kinds of parsers rely on PREDICT and FOLLOW sets to determine which look-ahead tokens mean to apply which right-hand side. Specifically, the PREDICT set for each grammar production says which tokens could come first when that production appears in the grammar. The look-ahead tokens in a top-down parser are compared against those in the PREDICT sets to see which rule to apply.

5 LR Parsers

**Readings for this section:** PLP, Section 2.3.3 (required).

Bottom-up parsers are the counterpoint to top-down parsers. They create the parse tree by alternatively adding the next token as a leaf (shifting) or combining some partial parse trees according to a grammar rule (reducing). The grammars for bottom-up parsers are called LR grammars; the second “R” has to do with the fact that things tend to be grouped from the right instead of from the left.

Just like with top-down parsers, the way a bottom-up parser is actually implemented is with a stack. Now with a top-down parser, each “prediction” step involves popping a single nonterminal from the top of the stack, and replacing it with one of its right-hand sides. A bottom-up parser works in the opposite way: a reduce in LR parsing involves popping an entire right-hand side off the top of the stack and replacing it with the corresponding nonterminal.

The bottom-up parse is complete when the stack consists just of the start symbol and nothing else, and there are no more tokens left on the input stream.

Again, the way we write grammars for LR parsers is influenced by the goal of making parsing faster. For example, consider this grammar for a bunch of NUMs added/subtracted together:

\[
\text{exp} \rightarrow \text{NUM OPA exp} \\
\text{exp} \rightarrow \text{NUM}
\]

Now consider the bottom-up parse for a token stream like NUM OPA NUM OPA NUM using this grammar. As we saw in class, it can work, but it’s not going to be very efficient. We noticed that all of the tokens have to be shifted onto the stack before any reducing can occur. Besides that, the parser also has to constantly look ahead to see if the next token is an OPA or not, to know whether to shift or reduce.

You see, right recursion is great for top-down parsers, but it’s not so great for bottom-up parsers. Here’s the same language, rewritten to use left recursion instead.

\[
\text{exp} \rightarrow \text{exp OPA NUM} \\
\text{exp} \rightarrow \text{NUM}
\]

Now when we parse a string of tokens like NUM OPA NUM OPA NUM with a bottom-up parser using this grammar, it works out great! The stack never has more than three symbols on it, and no look-ahead at all is required.

Here you see two strong indications of why LR parsers are a bit more powerful than LL parsers: First, they can handle both left recursion and right recursion (although left recursion will make them run much more efficiently). Second, they can parse some pretty interesting grammars, like the one above, **without requiring any look-ahead tokens**. Now that’s going to be fast!
The greater power of LR parsers is why most automatic parser generation tools - including bison - generate bottom-up parsers. It’s easier to take a grammar specification and make a reasonably fast LR parser from it. There are disadvantages, however, and they mostly have to do with nice error messages. The problem with LR parsers for compilers is that they try so hard to parse the language, that sometimes they shift everything onto the stack before realizing the parse is never going to work. In such cases, a recursive-descent parser probably would give a much nicer error message, right at the source of the mistake in the code. But as we discussed, the LL(1) grammars required to make recursive-descent parsers work quickly are more restrictive. This is the trade-off that must be made!

5.1 CFSMs

OK, back to LR parsers. How do they work exactly? How are those difficult shift/reduce decisions made, when look-ahead is necessary? Believe it or not, they use a DFA to make these decisions - specifically, the DFA that accepts strings consisting of valid stack contents for the parser itself! This process is detailed in your reading, and we went over it in class. Here’s the gist. See the slides for a complete example.

1. **Make the LR item parser states.**
   An “LR item” sort of represents the hopes and dreams of our bottom-up parser at any given moment in time. It says “I might be expanding THIS right-hand-side at THIS point”.
   Specifically, each LR item consists of some right-hand-side from the grammar, with a “bullet” in some position indicating the current position within that right-hand side that we could be in. For example, the LR item \( E \rightarrow E \bullet OPA T \) indicates that there is an \( E \) on the top of the stack, and if we see an OPA followed by a \( T \) coming up, these can be reduced to a single \( E \) using this grammar rule.
   So we start by making an NDFA with each LR item as a state.

2. **Make the transitions**
   The start state will be the production for the start symbol with the bullet all the way at the beginning, as you might expect. Now how can we transition between states? There are two ways:
   - A *shift* transition means reading the next symbol on the stack and moving the “bullet” past that symbol. **Important**: this only really corresponds to the parser performing a shift from the input stream if we’re at the end of the stack. Otherwise, it just means reading past the tokens and non-terminals that are already on the stack.
   - A *closure* transition is an empty-string transition that happens when the “bullet” is right before some non-terminal in the grammar. In other words, we are expecting to see that non-terminal come up next. What the closure transition means is that, instead of directly shifting that non-terminal symbol, we start to parse that non-terminal symbol instead, by transitioning to an LR item where the bullet is at the beginning of an expansion of that non-terminal.

I think this will be most clear if you review the examples from class, specifically the slide on “Pieces of the CFSM”. More examples can be found in your textbook too!

Since there will never be any \( \epsilon \)-transitions in this construction (unless there is something seriously wrong with your grammar), turning this nonterministic machine into a DFA is pretty easy: it will just involve consolidating multiple states into single states. This leaves us with the actual CFSM, which is a deterministic finite state machine that accepts valid stack contents.

Now here’s why this matters: bottom-up parsers use CFSMs to figure out what actions to take (shift or reduce). Specifically, any time the CFSM comes to a state with an LR item with a bullet all the way at the end, like

\[ E \rightarrow E \text{ OPA } T \bullet \]

this means we can reduce! The reason is, it means that we’ve just read in the right-hand side from the stack, so we can safely pop those symbols off and replace them with the left-hand side. So what a bottom-up parser is really doing is just following the transitions in a CFSM, shifting onto the stack every time it goes over a transition, and reducing whenever it hits a state with a reduce item like the one above. What about when it could do either, you ask? Well, that means we have a . . .
5.2 Conflicts

A conflict arises when there’s a state in the CFSM that contains some kind of ambiguity about what to do. Now the only thing a bottom-up parser really “does” that involves a hard decision is reducing. If we do a reduction at any time, the stack contents at that moment are destroyed and replaced by something else. And if we choose not to do a reduction, we’ll never be able to go back to it later. What this means is that the two kinds of LR parser conflicts both involve reducing:

- A shift-reduce conflict is the most common type and occurs when a single CFSM state has at least one reduce item (where the bullet is at the end of the right-hand side), and at least one shift item (where the bullet is somewhere in the middle). As the name implies, we don’t know whether to shift the next token, or to reduce the stuff we already have.
- A reduce-reduce conflict occurs when a single CFSM state has more than one reduce item (where the bullet is at the end of the right-hand side). Here we know that we need to reduce, but we don’t know which way to do the reduction; i.e., which symbols should we pop off the stack and replace?

SLR parsers (stands for “Simple LR”) resolve these kinds of conflicts by using look-ahead tokens and the FOLLOW sets that we saw earlier. The general rule is, if the next token (the lookahead) is in the follow set of some nonterminal, then we can reduce to that nonterminal. If the lookahead token is an outgoing edge from the current state, then we can shift that token. As long as there is no overlap between these outgoing transitions and the FOLLOW sets in any given state, then the parser can always resolve the conflicts above by using one token of lookahead. In this case, the grammar is SLR(1), and an SLR(1) parser can parse any string in the language in linear time. Hooray!

I will expect you to be able to make CFSMs and determine whether grammars are SLR(1), and this will be our main goal in creating bottom-up parsers. However, you should also be aware that there is another kind of parser called LALR (“look-ahead LR”) that uses a more specific kind of follow set to resolve conflicts. LALR parsers can handle everything that SLR parsers can, and just a little bit more because their FOLLOW sets end up being smaller. For this reason, automatic parser generators such as bison usually create LALR(1) parsers. You can read more about this in your textbook.

As always, be sure to review the slides and your notes from class for more details and examples on these parsing things.