Main topics of the week:
- Ambiguity in Grammars
- Attribute Grammars
- Dynamic Semantics
- Parsing

Grammar Ambiguity
Consider the following small grammar for assignment statements:

\[
\begin{align*}
<\text{assign}> & ::= <\text{id}> = <\text{expr}> \\
<\text{id}> & ::= A | B | C \\
<\text{expr}> & ::= <\text{expr}> + <\text{expr}> | <\text{expr}> * <\text{expr}> | ( <\text{expr}> ) | <\text{id}>
\end{align*}
\]

The sentence \( A = B + C * A \) has two different parse trees:

- Left parse tree:
  \[
  \begin{array}{c}
  <\text{assign}> \\
  <\text{id}> \\
  <\text{expr}> \\
  <\text{expr}> \\
  <\text{id}> \\
  <\text{expr}> \\
  <\text{id}> \\
  A \quad = \\
  B \\
  + \\
  C \\
  * \\
  A
  \end{array}
  \]

- Right parse tree:
  \[
  \begin{array}{c}
  <\text{assign}> \\
  <\text{id}> \\
  <\text{expr}> \\
  <\text{expr}> \\
  <\text{id}> \\
  <\text{expr}> \\
  <\text{id}> \\
  A \quad = \\
  B \\
  + \\
  C \\
  * \\
  A
  \end{array}
  \]

This happens since the grammar allows \(<\text{expr}>\) to grow on the left or the right. This ambiguity can be a problem for semantic analysis if the semantic analysis is based on the parse tree, which it often is. This particular example is a case where we want the grammar to reflect operator precedence. That is, multiplication should bind more tightly than addition to follow the usual rules in algebra. We can cause this to happen in the grammar by introducing more symbols:

\[
\begin{align*}
<\text{assign}> & ::= <\text{id}> = <\text{expr}> \\
<\text{id}> & ::= A | B | C \\
<\text{expr}> & ::= <\text{expr}> + <\text{term}> | <\text{term}> \\
<\text{term}> & ::= <\text{term}> * <\text{factor}> | <\text{factor}> \\
<\text{factor}> & ::= ( <\text{expr}> ) | <\text{id}>
\end{align*}
\]

Then we will have a unique parse tree for \( A = B + C * A \), and the parse tree reflects the higher precedence of multiplication. Inserting parentheses alters this precedence as expected, because our grammar has parentheses enclosing just an \(<\text{expr}>\), bringing us back to an expression subtree. This grammar also gives us left associativity of the operators since the parse trees expand to the left for \(+\) or \(*\). Generally, when a production rule has its LHS also appearing at the beginning of the RHS, the rule is called \textbf{left}
recursive, and this captures the idea of left associativity. If the LHS appears at the end of
the RHS, then we have a right recursive rule, which implements right associativity.

Ambiguity can cause problems for statements as well, and the classic example is the
dangling else. Suppose we have the grammar:

<if-stmt> ::= if <expr> <stmt>| if <expr> else <stmt>
<stmt> ::= <if-stmt> | S1 | S2

For the sentence “if <expr> if <expr> S1 else S2” we would get two distinct trees:

and this clearly poses a problem for semantic analysis: who does the else belong to? It
can clearly make a difference in the operation of the program since a false first
expression and true second expression results in the execution of S1 with the first tree,
but in the execution of no statement with the second tree. To fix this, we would need
additional non-terminals to distinguish between else-less if’s and if-else’s or change the
language to include delimiters. For example, some languages use delimiters as in the
following grammar:

<if-stmt> ::= if <expr> then <stmt> fi | if <expr> then <stmt> else <stmt> fi
<stmt> ::= <if-stmt> | S1 | S2

Alternatively, we could use more non-terminals to “catch” the dangling else:

<matched> ::= if <expr> <matched> else <matched> | S1 | S2
<unmatched> ::= if <expr> <stmt> | if <expr> <matched> else <unmatched>
<stmt> ::= <matched> | <unmatched>

Here we distinguish between matched if’s (where there is a matching else or the statement
does not involve if at all) and if’s without an else.

It is not always possible to remove an ambiguity by restructuring the grammar. A
language for which there is no unambiguous grammar is said to be inherently
ambiguous. However, there is no algorithm that can tell if a given context free grammar
is ambiguous – this is an undecidable problem.

Attribute Grammars
A context free grammar as we have seen is sufficient to describe the pure syntax of a language, but correct programs (i.e., accepted by a compiler or interpreter) have stricter requirements than the pure syntax. For example, in the C language, variables must be declared before they are used, and this can be hard (actually impossible) to express with a context free grammar. This language rule is an example of static semantics. It is a rule we can verify by examining the program, i.e., by a static analysis. Rules like this can be expressed formally with an attribute grammar, where we associate attributes with symbols (e.g., the type of an identifier grammar symbol), using attribute functions to calculate these values. Attribute values can be synthesized or inherited, indicating whether they are obtained by passing information down the syntax tree or up, or whether they are determined outside of the parse tree as intrinsic attributes.

In practice, formal attribute grammars result in such a large set of rules that they are not often used for real programming languages. Rather, compilers implement less formal attribute grammars for issues such as type checking. So, for example, assignment statements could require that the types of the expressions on both sides be the same, and the type of an expression could be determined by rules about how types combine (e.g., the sum of a double and an int is a double). Applying such rules recursively to the parse tree of expressions, where the leaves are atomic elements with defined types, then allows the type of an expression subtree to be determined. This is the same concept as an attribute grammar, but without the formalism.

Dynamic Semantics

So far, we have only talked about static analysis of a program’s correctness, and this only addresses the correct form of the program. We still have yet to address the issue of describing how the program behaves, something any programmer using the language obviously wants to know. One way is through operational semantics, which essentially means we provide an ideal virtual machine to describe how each construct in the language works. That is, we describe its operation by example. There are other more rigorous approaches such as axiomatic and denotational semantics that have the advantage of being able to prove formally the correctness of a program. As with formal attribute grammars, the specification of these semantics often becomes unwieldy for real programming languages.

Axiomatic semantics uses preconditions and postconditions on every statement to indicate constraints on variables in the program. The desired ending state of the program is the postcondition of the last statement, and from that point we work backwards, finding weakest preconditions that can guarantee the postcondition. Finding these preconditions can proceed from axioms or inference. As you might expect, the notation of axiomatic semantics is predicate calculus.

Denotational semantics formally associates functions with the syntactic objects specified by the grammar. This gives a mapping between the objects of the programming language and mathematical objects, which can be more rigorously manipulated. This formalization of the meaning of a program is similar to the informal handling of the abstract syntax tree one sees in practical compilers.
Top-down and bottom-up Parsing

A grammar is a language generator, but a programming language compiler or interpreter needs to have a parser – a parser recognizes the language. It can be shown that for any CFG, we can create a parser that runs in \(O(n^3)\) time (where \(n\) is the length of the input program), but this amount of time is much too slow for large programs. Fortunately, there are large classes of grammars for which it is possible to build linear time parsers. The most important of these classes are called LL (left-to-right scanning, left-most derivation) and LR (left-to-right scanning, right-most derivation).

LL parsers are top-down (or predictive) parsers. They build the parse tree from the root down by predicting at each step which production will be used to expand the current node after looking at the next token of input. LR parsers are bottom-up parsers, and build the parse tree from the leaves up, recognizing when a set of leaves can be combined as the children of a single parent. Let’s look at these parsers for a simple example:

\[
\begin{align*}
\text{csv} &::= \text{val} \ \text{csv}_\text{tail} \\
\text{csv}_\text{tail} &::= \ , \text{val} \ \text{csv}_\text{tail} \mid ;
\end{align*}
\]

This grammar describes a list of comma separated values terminated by a semi-colon. Here val is a terminal token that could be, say A, B, or C. Let’s examine how the different parsers would build the parse tree for the string: A,B,C;

The LL parser starts with the root \text{csv}, predicting it will be replaced by \text{val} \ \text{csv}_\text{tail} (the only rule), and looks to the input for a val token, which it finds (the A). It then predicts (by looking at the input and seeing a comma) that the \text{csv}_\text{tail} will be replaced by \ , \text{val} \ \text{csv}_\text{tail}. It then looks for a val token, and finds B. Again, it predicts, by peeking at the comma, that csv_tail will be replaced by the same rule again, and proceeds in this manner until all input is exhausted, and the tree is built.

The LR parser gets the first token, and sees that it is a val (A), so forms a leaf. The next token is a comma, so is another leaf. It continues in this way until it sees a complete right hand side. This happens when it sees the semi-colon, at which point it can form a \text{csv}_\text{tail} node, into which it reduces the last leaf. It continues working back through the leaves in this way, reducing and building the parse tree from the bottom up.

Although this example grammar could be parsed top-down or bottom-up, we can see that in the bottom-up case all the input has to be read before the tree can be constructed. In a very large program, this would require too much memory, so this example grammar is not very conducive to bottom-up parsing. By shifting the focus to the front of the list, we can get around this problem, e.g.,

\[
\begin{align*}
\text{csv} &::= \text{csv}_\text{prefix} \ ; \\
\text{csv}_\text{prefix} &::= \text{csv}_\text{prefix}, \text{val} \mid \text{val}
\end{align*}
\]

However, this grammar can no longer be parsed top-down since we can’t tell the difference between a \text{csv}_\text{prefix} and a val when we peek ahead, so we don’t know which rule to use. The rule for \text{csv}_\text{prefix} is called left recursive since the symbol of the rule also appears as the left most symbol in the rule itself. This property is exactly
what makes the grammar desirable for bottom up parsers since it allows for incremental reduction.

Yacc is an LALR (Look Ahead LR) parser generator. It allows disambiguating rules to resolve ambiguities. Thus, you can specify a BNF like grammar to yacc with ambiguities, but use its rules to specify left or right associativity for particular tokens, thus avoiding the need to express that directly in the grammar. A bottom-up parser works by maintaining a stack of the tokens that have been seen. When these tokens constitute a right hand side of a rule, it can reduce them to the left hand side of the rule. When the parsing is finished, the stack will have one object on it – the root of the tree.

Although parsers for simple languages can be hand crafted, most implementations will use a parser generator that is table driven. Issues of error recovery in parsers can become complex. However, in general, the area of parsers is well known and techniques and tools exist to implement parsers.