Syntax in Programming Languages

Michael McThrow
San José State University
CS 152 – Programming Paradigms

August 28, 2021

1 Syntax and Semantics

There are hundreds, if not thousands of programming languages in existence, and of the set of programming languages that have been created, a few dozen of them are in common use. We recognize them by names such as C, Java, Python, and Swift, among many others. Each language has different characteristics and may belong to one or more paradigms (e.g., procedural, object-oriented, functional, logic, domain-specific, etc.).

From the point of view regarding the design and implementation of programming languages, the chief characteristics that distinguishes one programming language from another are its *syntax* and its *semantics*. The New Oxford American Dictionary defines syntax as "the arrangement of words and phrases to create well-formed sentences in a language." When applied to programming languages, *syntax* refers to how strings representing concepts such as symbols, numbers, and names are arranged to create well-formed programs. A programming language's *semantics* define the meaning of well-formed elements of programs.

Definition 1.1. Syntax defines how strings are arranged to create well-formed programs in a specific programming language.

Definition 1.2. Semantics defines the behavior of well-formed programs in a specific programming language.

Users of a programming language need to know its syntax and semantics in order to write programs that will be accepted by the interpreter or compiler and will behave as intended. Implementers of a programming language need to know the language's syntax and semantics in order to accept well-formed programs, reject programs that are not well-formed, and run well-formed programs that conform to the language's semantics.

Example 1.1. In the Java programming language, the following program is well-formed:

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Question to Ponder. Can a program be well-formed yet semantically incorrect? (Note that logic errors are not the same thing as being semantically incorrect.)

This lesson will cover the basic aspects of programming language syntax and how programming language implementers deal with them. *Grammars* allow programming language designers and implementers to formally describe the language's syntax. By using a grammar, an interpreter or compiler can convert a string containing a well-formed program into an *abstract syntax tree* via the processes of *lexical analysis* and *parsing*. Once an abstract syntax tree is generated, the next step is either interpretation or compilation. An interpreter will traverse the abstract syntax tree to execute the program per the programming language's semantic rules. A compiler will traverse the abstract syntax tree to convert it to a representation in a different language, usually machine code or bytecode.

2 Grammar in Programming Languages

We can formally (i.e., mathematically) specify the syntax of a programming language by defining its grammar.

Definition 2.1. According to Slonneger and Kurtz's excellent textbook *Formal Syntax and Semantics of Programming Languages* (1995), a *grammar* G is a four-element tuple (Σ, N, P, S), where each element is described as follows:

1. Σ is a finite set defining the *alphabet* of the language, which is a collection of *terminal symbols* that make up the symbols accepted by the language. For example, the alphabet of the written English language consists of the letters of the Latin alphabet (uppercase A-Z and lowercase a-z), Arabic numerals (0-9), and various punctuation symbols.

2. N is a finite set of *non-terminal symbols* that describe collections of subphrases that make up the language. Some textbooks such as Sundkamp's *Languages and Machines: An Introduction to the Theory of Computer Science* (2006) define this *collection of subphrases* as *variables*.

3. P is a finite set of *production rules* (also known simply as *rules*) that define each non-terminal symbol in N in terms of non-terminal symbols and terminal symbols. We can think of P as a function that maps a sequence of symbols (terminal or non-terminal) to a sequence of terminal and non-terminal symbols. Such rules can be defined recursively.

4. S is a non-terminal symbol in N that is known as the *start symbol*, which is an entry point for describing a program in terms of its grammar.

Let's make this very mathematical definition of what a grammar is more understandable by describing an example of a language that accepts arithmetic expressions such as `2 + 3` and `10 - 5 + 7`:

2.1 Simple Arithmetic Example

Our language will accept basic arithmetic expressions of the following types:

1. Integers with one or more digits such as `3`, `132`, `0`, and `-32`.

2. Addition (e.g., `2 + 3`)

3. Subtraction (e.g., `8 - 4`)

In addition, we are not limited to simple expressions with just one operator. Our language will also accept more complex expressions such as `3 + 10 - -5 - 2 + -2 - 4 + 6`.

Let's define the grammar $G_{expr} = (\Sigma, N, P, S)$ for this language. Σ, N, and S have the following definitions:

- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$
- $N = \{E, I, D, \Omega\}$
- $S = E$

P is harder to express using strict set notation, and so let's describe it more intuitively. Our grammar G_{expr} accepts four non-terminal symbols: E (which represents expressions), I (which represents integers), D (which represents a digit), and Ω (which represents the operators $+$ and $-$). Recall that P must map E, I, D, and Ω to a sequence where each element is in $\Sigma \cup N$.

Let's describe each rule in P:

- E maps to one of the following sequences:
  - (I) (an integer)
  - (E_0, Ω, E_1) (an expression followed by an operator followed by another expression)

- I maps to one of the following sequences:
  - $(D_0, ..., D_n)$ (one or more of the digits in D)
  - $(-, D_0, ..., D_n)$ (the minus sign $-$, followed by one or more digits in D)
- D maps to 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9
- Ω maps to either $+$ or $-$.

Example 2.1. Given the grammar G_{expr}, is the expression `5 + -12 - -8` well-formed?

In order to check whether this expression is well-formed, we need to begin with the start symbol, which G_{expr} defines as E. Using E, we make the following match:

- $E_0 = 5$
- $\Omega = +$
- $E_1 = -12 - -8$

Note that the matching is done in left-to-right order. Had we performed a right-to-left match, $E_0 = 5 + -12$, $\Omega = -$, and $E_1 = -8$. We will proceed with the left-to-right match.

We then need to check E_0, Ω, and E_1. To check E_0, we make the match $I = 5$. We then check I. We then make the match $D_0 = 5$, and then we check D_0. It turns out that 5 is a valid result for D_0, and thus we have shown that D_0, I, and E_0 are well-formed. We then move on to checking $\Omega = +$. It turns out that $+$ is a valid result for $+$, and thus we have shown that Ω is well-formed. We finally move on to $E_1 = -12 - -8$. We make the following match:

- $E_2 = -12$
- $\Omega_1 = -$
- $E_3 = -8$

To check $E_2 = -12$, we make the match $I_1 = -12$. We then check $I_1 = -12$. We notice that the first element of the sequence -12 is $-$, and thus we have a match with I_1's second rule, which maps D_1 with 1 and D_2 with 2, respectively. 1 and 2 are both valid values for D_1 and D_2 respectively, and thus we have shown that E_2 is well-formed because its descendants I_1, D_1, and D_2 are well-formed.

To check $\Omega_1 = -$, we see whether $-$ is a valid value of Ω_1. Per the definition in the grammar, $-$ is a valid value, and thus Ω_1 is well-formed.

To check $E_3 = -8$, we make the match $I_3 = -8$. We then check $I_3 = -8$. We notice that the first element of the sequence -8 is $-$, and thus we have a match with I_3's second rule, which maps D_3 with 8. 8 is a valid digit, and thus we have shown that E_3 is well-formed because its descendants I_3 and D_3 are well-formed. ■

Notice that we naturally created a tree of derivations during the process of showing that the expression `5 + -12 - -8` is well-formed. Figure 1 shows the resulting derivation tree.

2.2 A Note about Whitespace and Grammars

In the above grammar G_{expr}, whitespace is treated as a sequence delimiter (separator); the input to a routine that checks whether a program is well-formed is a sequence of sequences of symbols. Notice that in the alphabet $\Sigma \in G_{expr}$ there are no whitespace symbols. For example, the expression `2 + -34 - 56` is treated as $((2), (+), (-, 3, 4), (-), (5, 6))$. Notice how in `-34` the symbol `-` is part of the same sequence as the digits `3` and `4`. This is because there is no whitespace between these symbols.

Please keep in mind that not all grammars treat whitespace as sequence delimiters. For example, in languages that support character string constants such as C and Java (e.g., `String a = "This is a string"`), the whitespace characters between the quotes `"` cannot be treated as delimiters; the Java assignment would yield the following sequence:

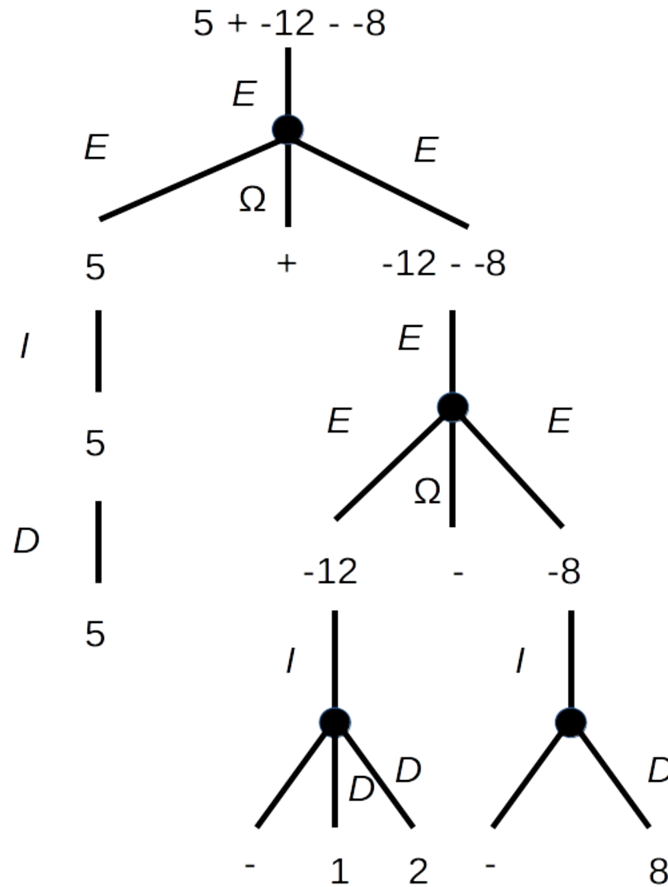

Figure 1: Derivation tree obtained when checking whether $5 + -12 - -8$ is well-formed under grammar G_{expr}. Note that there are no subscripts for each instance of non-terminal symbols.

```
((S,t,r,i,n,g), (a), (=), (",T,h,i,s, ,i,s, ,a, ,s,t,r,i,n,g,"))
```

In the grammars describing these languages, whitespace characters are part of the alphabet Σ.

Question to Ponder. Check to see if the following expressions are well-formed under G_{expr} (pay very close attention to whitespace, which is why these expressions are in a monospace font):

1. `0`
2. `0 + 0 - 0`
3. `-0`
4. `1 * 2 - 3`
5. `-2 - -3 - -4 + -5`
6. `-2 - - 3`
7. `a + b - c`

2.3 Classifying Languages Under the Chomsky Hierarchy

Renowned linguist Noam Chomsky developed a hierarchy of formal languages known as the **Chomsky Hierarchy** in 1956. The Chomsky Hierarchy classifies formal languages based on how restrictive grammatical

rules $P \in G$ are. There are four types, ranging from the least restrictive to the most restrictive: *unrestricted* (type 0), *context-sensitive* (type 1), *context-free* (type 2), and *regular* (type 3).

In this course, we will be focusing on context-free grammars, since many real-world programming languages' grammars are classified as such. However, we will cover the other classifications of grammars since they have important applications.

2.3.1 Type 0 – Unrestricted Grammars

An *unrestricted grammar* $G = (\Sigma, N, P, S)$ is one where for every mapping $(x \to y) \in P$, x consists of a sequence of symbols in $\Sigma \cup N$ where at least one of them is in N.

2.3.2 Type 1 – Context-Sensitive Grammars

Before defining context-sensitive grammars, let's define two terms that we will constantly see in the rest of this lesson: *left-side* and *right-side*. Given a rule $r \in P$ where $r = x \to y$, x refers to the *left side* of the rule r, and y refers to the *right side* of the rule r.

A *context-sensitive grammar* $G = (\Sigma, N, P, S)$ is one where for each rule in P, the amount of symbols on the right side of the rule is greater than or equal to the amount of symbols on the left side. To state this more formally, G is a context-sensitive grammar if every rule $(x \to y) \in P$ meets the following criteria:

1. x consists of a sequence of symbols in $\Sigma \cup N$ where at least one element is in N.
2. Given $x = (\sigma_0, ..., \sigma_m)$ and $y = (\tau_0, ..., \tau_n)$, $m \leq n$.

2.3.3 Type 2 – Context-Free Grammars

A *context-free grammar* $G = (\Sigma, N, P, S)$ is one where for each rule $(x \to y) \in P$, the left side x consists of a single non-terminal symbol. To define context-free grammars more formally, G is a context-free grammar if for every rule $(x \to y) \in P$, x is a sequence (σ_0) with one element $\sigma_0 \in N$.

2.3.4 Type 3 – Regular Grammars

A regular grammar $G = (\Sigma, N, P, S)$ is a context-free grammar with an additional restriction: given each rule $(x \to y) \in P$, each right side sequence y must have one of these two forms:

1. A one-element sequence (τ_0) where $\tau_0 \in \Sigma$ (i.e., τ_0 is a terminal symbol).
2. A two-element sequence (τ_0, τ_1) where $\tau_0 \in \Sigma$ and $\tau_1 \in N$ (i.e., a terminal symbol is followed by a non-terminal symbol).

Note that the ordering of the right side sequence $y = (\tau_0, \tau_1)$ matters. If $\tau_0 \in N$ and $\tau_1 \in \Sigma$, then G is not a regular grammar, though it remains a context-free grammar.

2.4 An Aside: Regular Expressions

You may have heard of or perhaps used *regular expressions* in text editors and various Unix command-line tools such as `grep` and `sed`. Regular expressions are very useful for searching for substrings that match patterns defined by those regular expressions. What you might not have known before is that regular expressions are descriptions of *regular languages*, which in turn are languages which syntax are defined by regular grammars.

We won't be covering the intracies of regular expressions in this course; this is in the domain of CS 154 (Formal Languages and Computability), where you will learn more about the theoretical aspects of grammars, formal languages, and other topics related to theoretical computer science. While CS 152 does not ignore theory (just wait until I cover the lambda calculus!), it is more focused on the practical aspects of programming language design and implementation.

2.5 Backus-Naur Form

Using the formal definition of grammars described earlier in this section, we already have the tools necessary to define the grammars of real-world programming languages by specifying our alphabet Σ, developing rules in P that reference Σ and non-terminal symbols N, and defining a starting point S that tools can use to check the syntactical correctness of a given program written in the language defined by $G = (\Sigma, N, P, S)$.

While formally defining grammars in terms of G is useful for the purposes of analyzing grammars and programs in a mathematically rigorous fashion, unfortunately these descriptions are too low level for programming language practitioners, such as users of the language, as well as writers of interpreters and compilers. What would be convenient is another way of expressing grammars that is easier for practitioners to understand while also being capable of being expressed in terms of G for allowing rigorous analysis.

Thankfully, this more convenient way of expressing grammars exists. **Backus-Naur form (BNF)** is a metalanguage used for describing the grammars of programming languages. A grammar description written in Backus-Naur form consists of a sequence of production rules. Each rule has a left side and a right side, delimited by the symbols `::=`. On each side of the rule, it consists of a sequence of terminal and non-terminal symbols. Non-terminal symbols are distinguished from terminal symbols by being enclosed in brackets. For example, in the sequence `<A>B<C>`, A and C are non-terminal symbols, while B is a terminal symbol. Note that we are not limited to single-character names of non-terminal symbols in BNF. This allows us to use names like `<digit>` and `<prefix>` to define non-terminal symbols, which provides semantic meaning that can help those reading the grammar better understand the purposes of the rules using these non-terminal symbols.

Sometimes there may be multiple right side values in a rule. When this happens, each right side possibility is delimited using the pipe (`|`) character. Sometimes a rule with multiple parts may be written in multiple lines, with each part delimited by the `|` character.

Note that production rules may be recursive. These recursive production rules are necessary in traditional BNF in order to represent sequences of arbitrary length. We will see an example of this when we create a BNF representation of the simple arithmetic expression language we formally defined in Section 2.1.

2.6 Simple Arithmetic Example in BNF

In Section 2.1 we defined a formal grammar G for a language that accepts basic arithmetic expressions. Below is the description of that grammar in Backus-Naur form:

```
<operator>    ::= + | -

<digit>       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digits>      ::= <digit>
                | <digit><digits>

<integer>     ::= <digits>
                | -<digits>

<expr>        ::= <integer>
                | <expr> <operator> <expr>
```

Please observe the following correspondences between the above BNF rules and the rules $P \in G$ defined in Section 2.1:

- `<operator>` corresponds to Ω.
- `<digit>` corresponds to D
- `<integer>` corresponds to I. However, to deal with sequences of digits, in the BNF description we created a new rule `<digits>` that expresses sequences of digits.

- <expr> corresponds to E. Note that <expr> is the starting point of the above grammar; unfortunately BNF has no special syntax indicating the starting point.

Isn't the BNF grammar for basic arithmetic expressions much easier to understand than the formal grammar?

Question to Ponder. How could we extend this grammar to support additional operations such as multiplication and division?

2.7 Chomsky Hierarchy and BNF

Suppose you wanted to know if a grammar described in BNF is either unrestricted, context-sensitive, context-free, or regular. It is possible to determine whether a grammar fits one of these categories without having to convert the BNF description to a formal $G = (\Sigma, N, P, S)$ representation. Below are the rules for each type:

0. The grammar is unrestricted if for each production rule, the left side of the rule has at least one non-terminal symbol.
1. The grammar is context-sensitive if it meets the criterion for unrestricted grammars, and for each production rule, the amount of symbols on the right side of the rule is greater than or equal to the amount of symbols on the left side. For example, a <beta> := <gamma> de is context-sensitive because the left side has two symbols a and <beta> while the right side has three symbols: <gamma>, d, and e.
2. The grammar is context-free if for each production rule, the left side of the rule has only one symbol, a non-terminal symbol.
3. The grammar is regular if it is context-free and is of one of the following forms:
   (a) <a> ::= a (the right side consists of a single terminal symbol).
   (b) <a> ::= b<gamma> (the right side consists of a terminal symbol followed by a non-terminal symbol).

Note that for regular grammars, multi-part rules where each part conforms to the criteria listed above are still regular. For example, the following grammar is regular:

```
<a> ::= a | b | c | d
<b> ::= e<a> | f<a>
```

2.8 Extended Backus-Naur Form

There are some recursive patterns that frequently occur in BNF definitions of grammars, such as when defining digits using the <digits> rule in the grammar described in Section 2.6. **Extended Backus-Naur Form (EBNF)** borrows from the syntax of regular expressions to make such definitions more compact.

While we will not cover all extensions to BNF, the most important additions we will cover are parentheses, * (known as the Kleene star), +, and ?. While the parentheses may appear around one or more symbols, the operators may appear either after a symbol (to modify the symbol before the operator) or after a closing parentheses (to modify the sequence of symbols inside the parentheses). The Kleene star operator * is used to represent a repeating sequence of zero or more symbols (e.g., <digit>* is a sequence of zero or more digits). The + operator is used to represent a repeating sequence of one or more symbols (e.g., <digit>+ is equivalent to <digit><digit>*). The ? operator is used to represent a zero- or one-time occurrence of a symbol (e.g., -?<digit>+ can be used to represent negative and positive integers since - may appear zero or one times). To show how parentheses are used in EBNF, (ab)+ can be used to match ab, abab, and abababababababab; however, it cannot match a, ba, or abba because the grammar expects repeating sequences of ab.

To convert the BNF grammar describing simple arithmetic expressions in Section 2.6 to EBNF, we remove the <digits> rule and we redefine <integer>'s right side as -?<digit>+.

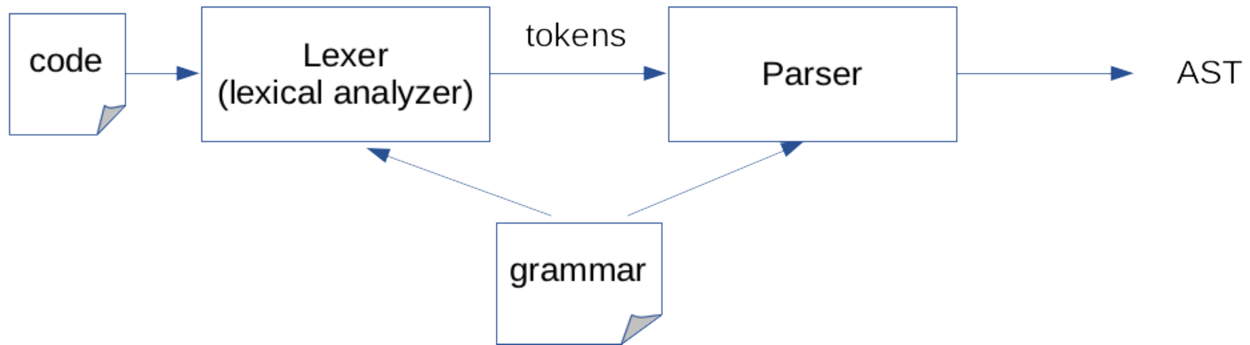

Figure 2: A flowchart of the parsing pipeline, which converts source code to an abstract syntax tree (AST)

Question to Ponder. How would you convert an EBNF representation of a language to BNF? What about the other way around?

3 The Parsing Pipeline

A compiler or an interpreter needs to *parse* a program according to the grammar of the language that the program is written in. Parsing the program allows the compiler/interpreter to work with a data structure that is more amenable to the analysis and eventual compilation or execution of the program.

Figure 2 is a flowchart of the parsing pipeline. The *lexical analyzer*, which is also known as simply the *lexer*, processes the source code, which is usually represented as a string, and produces a sequence of *tokens* that are annotated based on their syntactical meaning. Example 3.1 shows how a lexer may operate.

Example 3.1. Continuing with the EBNF grammar description for our language of arithmetic expressions as defined in Section 2.8, suppose we have the input string `5 + -12 - -8`. The lexer would produce the following sequence of tokens, where each token is a pair (2-tuple) where the first element is the annotation and the second element is the name of the token:

```
("digits", "5"), ("operator", "+"), ("digits", "-12"), ("operator", "-"),
("digits", "-8")
```

Please note that other representations are possible.

While the previous example was simple, lexical analysis can quickly become complex (for example, handling significant whitespace). Indeed, fully implementing lexers for most useful programming languages will require knowledge of finite automata, which is beyond the scope of this course. However, you will learn about finite automata in CS 154, and in CS 153 (Concepts of Compiler Design) you will learn the nuts and bolts of lexical analysis. In this course, being aware of the role that lexers play in the compilation/interpretation process is sufficient.

After the lexer produces a sequence of tokens, the *parser* processes this sequence. It may create a *derivation tree* (as seen in Figure 1), but whether or not it does so, in the end the parser produces an *abstract syntax tree (AST)*. Abstract syntax trees differ from derivation trees in the sense that all parent nodes consist of operators. Figure 3 is an illustration of the abstract syntax tree for the expression `5 + -12 - -8`. Such an abstract tree can be represented programmatically; below is the abstract syntax tree written in Java:

```
// Note that AST is the base class for all abstract syntax tree
// objects.  The classes Number and Expr all derive from AST.
AST ast = new Expr(ExprKind.ADD, new Number(5),
                                 new Expr(ExprKind.SUBTRACT,
                                          new Number(-12),
                                          new Number(-8)));
```

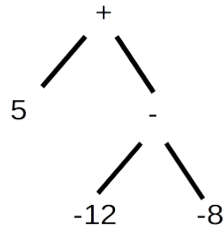

Figure 3: The abstract syntax tree for the expression `5 + -12 - -8`

An interpreter would implement and invoke the `eval()` method on the `AST` object `ast`. It will recursively evaluate all of its subtrees, eventually resulting in the answer `1`.

Like lexing, parsing is a very complex topic in its own regard. In fact, it is possible to spend an entire semester studying parsing techniques. A larger treatment of parsing techniques will have to wait for CS 153.

However, we can leverage the help of *parser generators* such as ANTLR where we specify context-free grammars using ANTLR's custom syntax. Given a grammar, ANTLR will generate Java code that handles our lexing and parsing tasks. It will supply us AST classes that we can extend to perform compilation, interpretation, or other programming language analysis tasks. We will be using ANTLR during the first project of this course.

4 Some Trouble Spots

4.1 Ambiguities in Programming Languages

In the English language, we can produce sentences that are grammatically-correct yet can be ambiguous to parse. For example, let's consider the sentence "Tim told Dave to grab his coat and leave." Is *his* modifying Tim (Tim's coat) or Dave (Dave's coat)? Usually context will disambiguate this sentence, but context is a matter of *semantics* (the meaning of well-formed sentences) rather than syntax (the formation of well-formed sentences).

Similar ambiguities may arise when defining programming languages. One ambiguity that is common is the **dangling else problem**, which dates all the way back to ALGOL 60, the first programming language to introduce now-familiar if/else statements.

Example 4.1. This is code written in C (that can also be written in C++, Java, and JavaScript) that has a dangling `else`:

```
if (condition1)
  if (condition2)
    printf("Yay!");
  else
    printf("No!");
```

Let's play close attention to the `else` statement. The problem is: does `else` modify the first `if` statement (for `condition1`) or the second `if` statement (for `condition2`)? Unfortunately this is not clear from looking at the syntax alone, and recall that in C and similar language, indentation is not semantically meaningful.

Some of you may be wondering why the above code is valid C, since there are four lines below the first `if` statement. The short answer is that the rule for introducing brackets in C is as follows: brackets are required for compound statements (i.e., two or more consecutive statements). The more complete answer lies in the grammatical rule for handling `if`, `else`, and `select` in C (adapted from *The C Programming Language, 2nd Edition* by Brian Kernighan and Dennis Ritchie [1989]):

```
<selection-statement> ::= if ( <expression> ) <statement>
 | if ( <expression> ) <statement> else <statement>
 | switch ( <expression> ) <statement>
```

Note that no brackets are used in this definition; you will need to look at the <statement> rule to see where brackets are used. Here is where the ambiguity is introduced: is the entry point to parsing <selection-statement> the first part (which lacks else) or the second part (which handles else)?

How is this ambiguity resolved? According to Kernighan and Ritchie, "[t]he else ambiguity is resolved by connecting an else with the last encountered else-less if at the same block nesting level" (p. 223). In the above example, the else statement is associated with the second if statement, if (condition2).

Question to Ponder. What are other ways of resolving the dangling else problem in C and silimar languages?

4.2 Dealing with Operator Precedence

Suppose we were to extend our simple arithmetic expression language to support multiplication (*), division (/), and parentheses (()). A naïve attempt would look like the following in EBNF:

```
<operator>    ::= + | - | * | /

<digit>       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<integer>     ::= -?<digits>+

<expr>        ::= <integer>
                | '(' <expr> ')'
                | <expr> <operator> <expr>
```

(Note that the left and right parentheses are enclosed with single quotes because parentheses have syntactical meaning in EBNF.)

A problem arises when it is time to construct the abstract syntax tree: will the resulting AST respect algebraic rules on operator precedence? Unfortunately BNF and EBNF do not specify the order of evaluation of parts of rules. In addition, when considering <expr> <operator> <expr>, a compliant sequence of tokens is generally going to be parsed in either a left-to-right or a right-to-left fashion.

Assuming a left-to-write parsing, suppose we have the expression 2 + (3 * 4 + 5) / 7. If we draw a derivation tree with the expression being the top node of the tree, the second level will have three nodes: 2, +, and (3 * 4 + 5) / 7. The last node would have three child nodes: (3 * 4 + 5), /, and 7. The first of these child nodes would have five child nodes: (, 3, *, 4 + 5, and). Finally, the node 4 + 5 would have three child nodes: 4, +, and 5. After converting the derivation tree to an abstract syntax tree and evaluating it, the result would be 5 (note that 27/7 is truncated to 3 due to our language only supporting integers). However, when applying standard arithmetic evaluation rules to the expression 2 + (3 * 4 + 5) / 7, the result is 4.

How can we specify a grammar that observes standard arithmetic evaluation rules? The answer lies in the fact that while under BNF and EBNF we cannot specify the ordering of how multi-part rules are evaluated, we can impose ordering by naming these rules differently. Here is a version that imposes ordering so that way resulting ASTs adhere to standard arithmetic evaluation rules:

```
<add-operator>    ::= + | -

<mult-operator>   ::= * | /

<digit>           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<integer>         ::= -?<digits>+
```

```
<add-expr>        ::= <expr> <add-operator> <expr>
                    | <mult-expr>

<mult-expr>       ::= <expr> <mult-operator> <expr>
                    | <parens-expr>

<parens-expr>     ::= '(' <expr> ')'
                    | <integer>

<expr>            ::= <add-expr>
```

Pay very close attention to how these rules are ordered. Make sure you study this well, since questions related to how to express operator precedence in BNF and EBNF grammars make great exam questions.

When using parser generator tools, sometimes the tool may have guidelines regarding the ordering of the evaluation of multi-part rules. For example, in ANTLR each part of the multi-part rule is evaluated in order from the first defined to the last.

Question to Ponder. What would the resulting derivation tree look like under this new grammar for `2 + (3 * 4 + 5) / 7`?

Question to Ponder. Why aren't there a `<sub-expr>` for subtraction and a `<div-expr>` for division?

Question to Ponder. How would you add support for exponentiation (`^`) in this grammar?

5 Further Reading

The 1995 textbook *Formal Syntax and Semantics of Programming Languages* by Kenneth Slonneger and Barry Kurtz is an excellent guide to learning syntax and semantics. The programming exercises in this textbook are in Prolog. I highly recommend finding this book and learning from it!