Programming Language Syntax

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

August 26, 2020

Table of Contents

Table of Contents

Syntax and Semantics

All programming languages have syntax and semantics.

Syntax and Semantics

All programming languages have syntax and semantics.

Definition (Syntax)

"Syntax refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language" [Slonneger and Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1995].

Syntax and Semantics

All programming languages have syntax and semantics.

Definition (Syntax)

"Syntax refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language" [Slonneger and Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1995].

Definition (Semantics)

"Semantics reveals the meaning of syntactically valid strings in a language" [Slonneger and Kurtz 1995].

Syntax and Semantics

All programming languages have syntax and semantics.

Definition (Syntax)

"Syntax refers to the ways symbols may be combined to create well-formed sentences (or programs) in the language" [Slonneger and Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, 1995].

Definition (Semantics)

"Semantics reveals the meaning of syntactically valid strings in a language" [Slonneger and Kurtz 1995].

This lecture will focus on syntax. Monday's lecture will focus on semantics.

How do we define the syntax of a programming language?

How do we define the syntax of a programming language? We define the syntax of a programming language by a formal grammar.

Table of Contents

Definition (Grammar (from Slonneger and Kurtz 1995))

A grammar $< \Sigma, N, P, S >$ consists of four parts:

1. A finite set Σ of terminal symbols, the alphabet of the language, that are assembled to make up the sentences in the language.
2. A finite set N of nonterminal symbols or syntactic categories, each of which represents some collection of subphrases of the sentences.
3. A finite set P of productions or rules that describe how each nonterminal is defined in terms of terminal symbols and nonterminals. The choice of nonterminals determines the phrases of the language to which we ascribe meaning.
4. A distinguished nonterminal S, the start symbol, that specifies the principal category being defined—for example, sentence or program.

Chomsky Hierarchy

Noam Chomsky, famous linguist and political writer, developed the Chomsky hierarchy of formal languages to classify them based on how restrictive their production rules P are.

Chomsky Hierarchy

Noam Chomsky, famous linguist and political writer, developed the Chomsky hierarchy of formal languages to classify them based on how restrictive their production rules P are.

Type	Grammar
Type 0	Unrestricted grammar
Type 1	Context-sensitive grammar
Type 2	Context-free grammar
Type 3	Regular grammar

Chomsky Hierarchy

Noam Chomsky, famous linguist and political writer, developed the Chomsky hierarchy of formal languages to classify them based on how restrictive their production rules P are.

Type	Grammar
Type 0	Unrestricted grammar
Type 1	Context-sensitive grammar
Type 2	Context-free grammar
Type 3	Regular grammar

- The higher the type, the more restrictive the grammar's production rules are.

Chomsky Hierarchy

Noam Chomsky, famous linguist and political writer, developed the Chomsky hierarchy of formal languages to classify them based on how restrictive their production rules P are.

Type	Grammar
Type 0	Unrestricted grammar
Type 1	Context-sensitive grammar
Type 2	Context-free grammar
Type 3	Regular grammar

- The higher the type, the more restrictive the grammar's production rules are.
- Also, less restrictive grammatical types encompass more restrictive grammatical types (e.g., the set of Type 1 grammars includes Type 2 and Type 3 grammars).

Backus-Naur Form

Backus-Naur Form is a widely-used notation for specifying programming language grammars.

Backus-Naur Form

```
<declaration> ::= var <variable list> : <type>;
```

Backus-Naur Form

```
<declaration> ::= var <variable list> : <type>;
```

- `declaration` is an example of a production rule in P.
- `var`, `:`, and `;` are terminal symbols in the set Σ.
- `<variable list>` and `<type>` refer to production rules of that name.
- Definitions can be recursive, and production rules can have multiple parts, with each part delimited by |.

Example: Defining Numbers in BNF

```
<number>      ::= <prefix><digit list>
                | <prefix><digit list>.<digit list>
                | <digit list>
                | <digit list>.<digit list>
<prefix>      ::= + | -
<digit list> ::= <digit>
                | <digit><digit list>
<digit>       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Example: Defining Numbers in BNF

This is an alternative definition if we define an empty string as a valid prefix for the number (e.g., for representing non-negative numbers).

```
<number>     ::= <prefix><digit list>
               | <prefix><digit list>.<digit list>
<prefix>     ::= + | - | EMPTY
<digit list> ::= <digit>
               | <digit><digit list>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Exercise: How would you extend the definition of a number to specify hexadecimal integers (e.g., `0xFA`)?

Here is a possibility:

```
<number>       ::= 0x<hex-digits>
<hex-digits>   ::= <hex-digit> | <hex-digit><hex-digits>
<hex-digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                 | A | B | C | D | E | F
```

Note that a hexadecimal numeral is unsigned and thus cannot be preceded by a $-$.

Regular Grammars

Definition (Regular Grammar)

A regular grammar only allows production rules in the form of the following:

1. Terminals (e.g., `<prefix> ::= + | -`)
2. Terminal followed immediately by a single non-terminal (e.g., `<binary> ::= 0 | 1 | 0 <binary> | 1 <binary>` is a *right-terminal* rule)

Note: Grammars where all production rules have a non-terminal plus a terminal to its right are also regular (these are *left-terminal* rules). However, P cannot have both left-terminal and right-terminal rules.

Regular Languages

Definition (Regular Language)

A regular language is a language that can be defined by a regular grammar.

Regular Expressions

We can use regular expressions to describe regular languages, as an alternative to Backus-Naur form.

Context-Free Grammars

Definition (Context-Free Grammar)

A context-free grammar is one where all rules have a non-terminal on the left-hand side of `::=` in the rule definition.

Context-Free Grammars

Definition (Context-Free Grammar)

A context-free grammar is one where all rules have a non-terminal on the left-hand side of `::=` in the rule definition.

The number example is an example of a context-free grammar. Also, all regular grammars are context-free.

Example of a Grammar that Is Not Context Free

```
<thing> b ::= b <thing>.
```

Extended Backus-Naur Form

There are some recursive patterns that frequently occur when defining grammars in BNF:

Extended Backus-Naur Form

There are some recursive patterns that frequently occur when defining grammars in BNF:

```
<digit list> ::= <digit> | <digit><digit list>
```

Extended Backus-Naur Form

There are some recursive patterns that frequently occur when defining grammars in BNF:

```
<digit list> ::= <digit> | <digit><digit list>
```

Extended Backus-Naur Form (EBNF) borrows from the syntax of regular expressions to simplify BNF production rules.

```
<digit list> ::= <digit>+
```

Does defining a programming language using a context-free grammar mean that the language doesn't have ambiguities?

Does defining a programming language using a context-free grammar mean that the language doesn't have ambiguities? No. Ambiguities can still happen, which can be a problem when parsing the code.

The Dangling `else` Problem

Dating all the way back to ALGOL 60, many programming languages suffer from ambiguous `else` statements due to how `if...else....` statements are defined.

Definition of `if` Statement 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 a `<selection-statement>` is a `<statement>`. Also, note that many "bracket-style" languages influenced by C, such as C++, Java, and JavaScript, define `if` statements exactly the same as C.

Dangling else in C

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

Dangling else in C

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

Possible interpretations of the above code:

```
if (condition) {
    if (condition2)
        printf("Yay!");
    else
        printf("No!");
}
```

```
if (condition) {
    if (condition2)
        printf("Yay!");
} else
    printf("No!");
```

Exercise: If you were to redesign C, how would you solve the dangling else problem?

Operator Precedence

Another "gotcha" that comes up with defining a grammar for a programming language is operator precedence, which is the order in which operations are to be evaluated.

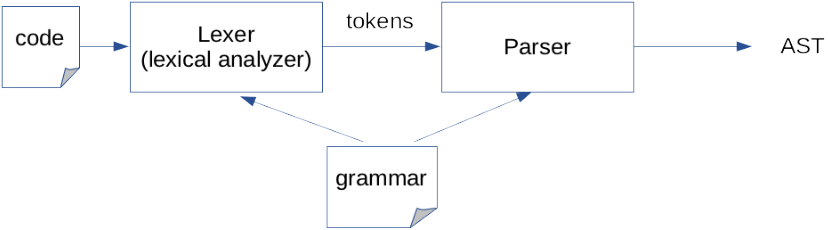
Summary

- Syntax and semantics define a programming language.
- Grammar defines the syntax of a programming language.
- We can define programming languages using context-free grammars (but not all programming languages are defined by them).
- We can use BNF to define programming language grammars.
- Grammars may have semantic ambiguities, such as the dangling `else` problem.

Table of Contents

So, suppose you have source code and a grammar expressed in BNF. How does a program "interpret" the source code using the grammar?

The Parsing Pipeline



Lexical Analyzer

The lexical analyzer, also known as the lexer, splits a string into symbols called *tokens*, and then annotates them based on their syntactical meaning.

Example of Lexical Analysis

Input:

```
var fofx = a*x**2 + b*x + c;
```

Output:

```
[("keyword", "var"), ("identifier", "fofx"),
("operator", "="), ("identifier", "a"),
("operator", "*"), ("identifier", "x"),
("operator", "**"), ("literal", "2"),
("operator", "+"), ("identifier", "b"),
("operator", "*"), ("identifier", "x"),
("operator", "+"), ("identifier", "c"),
("symbol", ";")]
```

Whitespace and Lexical Analysis

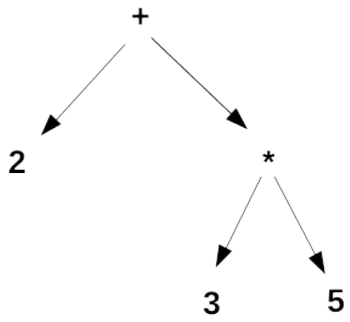
- In some languages, such as C, whitepace that is not inside of a string constant is insignificant as long as keywords are separated.
- There are some languages such as Python where whitespace is syntactically important.
- Takeaway: Dealing with whitespace is not as simple as calling `strtok()` or `split()` on the string.

Parsing

- Parsing is a complex topic in its own regard.
- The textbook *Parsing Techniques, A Practical Guide (2nd Edition)* by Grune and Jacobs (2008) is 622 pages long.
- We won't be focusing on the intracies of parsing algorithms in this course. Instead, we will be enlisting the help of parser generators to parse context-free grammars.
- A parser outputs a parse tree. If we remove extraneous information from the tree, then we end up with an abstract syntax tree, which we can traverse for the purposes of either evaluating the tree (i.e., performing the operations described in the tree) or generating executable code (which a compiler does).

Visual Representation of an Abstract Syntax Tree

2 + 3 * 5



Programmatic Representation of an Abstract Syntax Tree

```
AST ast =
new AddExpr(new Number(2),
            new MultExpr(new Number(3),
                         new Number(5)));
```

Note that for this to compile, the `AddExpr`, `MultExpr`, and `Number` classes need to implement the `AST` interface.

Evaluating an Abstract Syntax Tree

Calling `ast.eval()` will evaluate the abstract syntax tree by recursively evaluating all of its subtrees, eventually resulting in the answer 17.

Summary of the Parsing Pipeline

- Lexical analyzer splits a string containing source code into tokens.
- Tokens are then fed to a parser to generate a parse tree.
- Parse tree is then converted to an abstract syntax tree.
- We can then use the abstract syntax tree to either evaluate the program (interpreter) or to generate code (compiler).
- Parser generators like ANTLR are very convenient tools for parsing grammars.

Table of Contents

Preview of Monday's Lecture

On Monday we will be covering operational semantics and the lambda calculus. Both of these are very handy theoretical tools for programming language analysis.