CSCI 3366 Programming Languages

Spring 2022

R. Muller, **J. Tassarotti**

Lecture Notes: Week 4

This Week

1. Syntax: Grammars & Derivations
2. Ambiguity
3. Precendence & Associativty

NOTE: This file uses LaTex source. View the README.pdf file instead.

---

1. Syntax: Grammars & Derivations

Context:

- We're interested in the design, specification and implementation of programming languages;
- We're more or less following the slogan:

  `Language = Syntax + Semantics`

- As far as syntax goes there is *concrete syntax* and *abstract syntax.*
  - Concrete syntax describes the textual forms that the programmer can actually type;
  - Abstract syntax refers to the essential structure of the code without all of the syntactic bells and whistles.
- We don't have much to say about the design of concrete syntax other than the common sense idea that it shouldn't interfere with understanding and, if you're lucky, it should be in reasonable taste.

Specification of Syntax

Consider a finite set of symbols T = {a, b, c, ...}. We're going to view a language over T as elements of T* (i.e., sequences of symbols drawn from T). T^* has no constraints so we'll employ a device, a context free grammar (CFG) to define a reasonable subset of T*.

A *context free grammar* G is a 4-tuple $ (N, T, P, S)$ where

- N is a finite set of *nonterminal symbols*;
- T is a finite set of *terminal symbols*;
- $P \subseteq N \times (N \cup T)^*$ is a finite set of *productions*, and
- $S \in N$ is a distinguished element of N called the *start symbol.*

We'll follow well-established conventions using:

- uppercase letters A, B, ... to range over N;
- lowercase letters at the front of the alphabet a, b, ... to range over T;
- lowercase letters at the end of the alphabet w, u, v to range over T*;
- Greek letters α, β, γ to range over (N U T)*.

Example

G0 = ({S, A}, {a, b}, {(S, aA), (A, aA), (A, b)}, S)

The elements of P, in fact, the grammar G is usually written thusly:

```
S --> aA                            S ::= aA
A --> aA        or sometimes as     A ::= aA
A --> b                             A ::= b
```

Given our conventions, we can say things like $(A ::= \alpha) \in P$.

Derives in One Step

Let $(A ::= \gamma) \in$ P. Then $\alpha A\beta \Rightarrow \alpha\gamma\beta$.

The double arrow $\Rightarrow$ is pronounced "derives in one step".

Example

For G0 we have that $aaA \Rightarrow aaaA$, using $\alpha = aa$, $\beta = \epsilon$ and $\gamma = aA$.

Derives in Zero or More Steps

The relation $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$.

Example

For G0 we have that aaA ==>* aaA (derives in zero steps) and that aaA ==>* aaaA (derives in one step) and aaA ==>* aaab (derives in two steps).

Sentential Form

Let G = (N, T, P, S) be a CFG. Then the set of **sentential forms** of G is $\{\alpha \mid S \Rightarrow^* \alpha\}$.

Intuitively, α is a string of terminals and nonterminals derivable from the start symbol.

Definition: L(G) – the language of G

Let $G = (N, T, P, S)$ be a CFG. Each nonterminal symbol $A \in N$ defines a language $L(A) \subseteq T^*$. The language of A, written $L(A)$, is the set $\{w \mid A \Rightarrow^* w\}$. (And in a trivial sense, each terminal symbol $a \in T$ defines a singleton language $L(a) = \{a\}$.) The language defined by CFG G is $L(G) = \{w \mid S \Rightarrow^* w\}$.

> **Heads up!**
>
> It is standard practice to abuse grammar notation in the following way. A nonterminal symbol A will be used as a nonterminal **and** sometimes as a symbol for the defined language $L(A)$ **and** sometimes as a meta-variable ranging over $L(A)$. Which is which should be clear from the context.

Definition: parser for L(G):

A **parser** for $L(G)$ is a function **parse**$(G,\ w)$ that returns true if $w \in L(G)$ and false otherwise.

Comments

1. L(G) is considered as set of *sentences*. So from this perspective, programs are sentences.
2. It's easy to see that $L(G) \subseteq T^*$ — we've thrown out a lot (but definitely not all!) of the nonsensical stuff.
3. Given a grammar G and a string of symbols w, a derivation $S \Rightarrow^* w$ can be seen as a proof that $w \in L(G)$. (Technically speaking, it's called a "witness".)

Conventions:

1. If $(A ::= \alpha) \in P$ and $(A ::= \beta) \in P$ we usually write $A ::= \alpha \mid \beta$ where the vertical bar is pronounced "or".
2. Given a set of productions P, the grammar G = (N, T, P, S) is easy to infer. So here to fore, we'll write grammars by writing the productions.

Leftmost/Rightmost Derivations

A derivation step in which the leftmost nonterminal is replaced:

$$wA\beta \Rightarrow w\alpha\beta \tag{1}$$

is called a *leftmost derivation step*. A derivation:

$$\alpha \Rightarrow^* \beta \tag{2}$$

is leftmost if each step in the derivation is leftmost. And likewise for rightmost.

Derivations and Trees

Let X_i range over $N \cup T$, let $A ::= X_1 X_2 \dots X_n$ be a production such that $X_i \Rightarrow *\beta_i$ with $\beta = \beta_1 \dots \beta_n$ and let D be the derivation $A \Rightarrow^* \beta$. Then **Tree**(D) is the tree structure with root A and with subtrees

Tree$(X_1 \Rightarrow^* \beta_1)$ ... **Tree**$(X_n \Rightarrow^* \beta_n)$.

Example

```
E ::= (E + E) | a
```

Let $w = (a + a)$, $D = E \Rightarrow (E + E) \Rightarrow (a + E) \Rightarrow (a + a)$. Then **Tree**$(D)$ is

```
     E
  / / | \ \
 ( E  +  E )
   |     |
   a     a
```

The tree of a derivation is called a *concrete syntax tree*.

Abstract Syntax

The phrase *abstract syntax tree* (AST) was coined by John McCarthy, the inventor of LISP. The abstract syntax tree has only the essential information contained in the concrete syntax tree. For the example above, the AST is:

```
  +
 / \
a   a
```

In a language implementation, the parser usually passes an AST to subsequent phases of the implementation.

The Chomsky Hierarchy

We mentioned that CFGs play a key role in the Chomsky Hierarchy and in the specification of the syntax of programming languages.

```
The Chomsky Hierarchy -- DCFLs can be parsed efficently

+---------------------------------------------------------------+
| Unrestricted Grammars                                         |
| +---------------------------------------------------------+   |
| | Context Sensitive                                       |   |
| | +---------------------------------------------------+   |   |
| | | Context Free                                      |   |   |
| | | +---------------------------------------------+   |   |   |
| | | | Deterministic Context Free                  |   |   |   |
| | | | +-------------------------------------------+  |  |  |  |
| | | | | Regular                                   |  |  |  |  |
| | | | +-------------------------------------------+  |  |  |  |
| | | +------------------------------------------------+  |  |  |
| | +-----------------------------------------------------+  |  |
| +----------------------------------------------------------+  |
+---------------------------------------------------------------+
```

2. Ambiguity

A grammar G is *ambiguous* if there exists a $w \in L(G)$ such that there are two different leftmost (rightmost) derivations of w.

Reviewing our simple grammar for logical expressions

```
S ::= t | f | S || S | S && S | !S | ( S )
```

We found that there are sentences in L(S) with multiple left-most derivations.

```
w = t || t && t

S => S && S => S || S && S => t || S && S => t || t && S => t || t && t
S => S || S => t || S => t || S && S => t || t && S =>      t || t && t
```

Of the two parses above, we prefer the latter because it ascribes higher precedence to the && operator.

```
            &&                    ||
           /  \                  /  \
         ||    t                t    &&
        /  \                        /  \
       t    t                      t    t
```

3. Precedence & Associativity

It's important for programming language syntax to adhere to the standard mathematical conventions relating to operator precedence and associativity. In mathematics, when we write $3x + 2$, we mean $(3 \cdot x) + 2$ rather than $3 \cdot (x + 2)$. When we write $4 - 1 - 1$ we mean $(4 - 1) - 1$; most operators are left-associative. On the other hand, exponentiation is right-associative.

One can impose precedence and associativity preferences directly in the grammar for the concrete syntax.

- In order to give operator **A** higher precedence that operator **B**, introduce it further away from the start symbol;
- In order to make an operator left-associative: **x A x A x** means **(x A x) A x** use left-recursion, and likewise for right.

We rewrite our concrete grammar.

```
E ::= E || T | T
T ::= T && F | F
F ::= t | f | !E | ( E )
```

Now to left-derive $w = t \mathbin{//} t \mathbin{\&\&} t$

```
E => E || T => T || T => F || T => t || T
  => t || T && F => t || F && F => t || t && F => t || t && t
```

The concrete and abstract syntaxes are

```
   E                        ||
 / | \                     /  \
E  ||  T                  t     &&
|    / | \                     /  \
T   T  &&  F                  t    t
|   |      |
F   F      t
|   |
t   t
```

Parsing

There are many ways to write parsers for a language. Some start from the top of the syntax tree and build their way down to the leaves; others start from the leaves and build their way up to the root. A very popular and longstanding top-down technique is called *recursive-descent.* The idea is to define a function for each nonterminal symbol. The job of that function is to recognize the language defined by that particular nonterminal. The collection of these functions can parse the whole language. An example for the little language **righty** is included in the `src` directory.

Left-Factoring

Recursive-descent parsing is in some ways inconsistent with the practice of using left-recursion in a grammar to enforce left-associativity of operators. If we have a right-recursive rule

```
A ::= b A
```

the right side says, find a `b` then call the function parsing `A` recursively. But if the rule is left-recursive

```
A ::= A b
```

the right hand side leads to an infinite series of recursive calls to the function parsing `A` without ever consuming a `b`. This problem can be ameliorated by re-writing the grammar to replace left-recursion with right-recursion.

$$A ::= A\alpha \mid \beta$$

Let's say that β doesn't start with A. Then $L(A) = \{\beta, \beta\alpha, \beta\alpha\alpha, \ldots\}$. We can replace this rule with two rules

$$A ::= \beta A'$$

$$A' ::= \alpha A' \mid \epsilon$$

Then $L(A)$ is unchanged but the repetition in the grammar rule is recursing to the right rather than the left.

Example

```
D ::= a | D b
```

By left-factoring we arrive at an equivalent grammar:

```
D ::= a D'
D' ::= b D' | empty
```