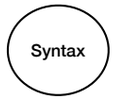
Core Design Concepts Discussed:



Concrete Syntax

The UI of PLs

Harley Eades III

This is the first lecture where we begin learning about general programming language concepts, and the most natural place to begin is the user-interface of the programming language which is called the concrete syntax. This is the the interface the programmer uses when creating programs, and it should be...

Concrete Syntax

Core Design Concepts:

The interface used by the programmer. It should be:

1. Usable
2. Readable
3. Maintainable

usable, readable, and maintainable. Concrete syntax is predominately for humans, and so it should be designed to work well with humans. This isn't necessarily true for most programming languages, and the design of concrete syntax is an understand area of human-computer interaction...

Core Design Concepts:

Syntax

Example of Concrete Syntax

```
let mult m n =
  let rec mult_helper acc n' =
    if m == 0
    then 0
    else if n' == 0
    then acc
    else mult_helper (m + acc) (n' - 1)
  in mult_helper 0 n
```

Here is an example of concrete syntax. The multiplication function we looked at earlier. As we can see the OCaml language had made certain design decisions that some will like and others may not. One of the first choices made about the design of a language is its concrete syntax, but usually designers make choices, and then they change over time.

Parsing Concrete Syntax

Core Design Concepts:

- Checks correctness of concrete syntax.
- Translates concrete syntax into abstract syntax.

More on abstract syntax in a future lecture.

The compiler of a programming language must be able to check the correctness of the concrete syntax. This is known as parsing. A second goal of parsing is to translate the concrete syntax into a form of the syntax that is easier to process by the compiler called abstract syntax. We will discuss this more in a future lecture.

Example

Suppose our "programs" were strings of the form: 0^n1^n

$$S \rightarrow 01 \mid 0S1$$

Parsing corresponds to an algorithm on a concrete syntaxes context-free grammar. This is a tool for specifying exactly how to generate or produce valid programs in a structured way. First, lets look at an example using basic strings of 0's followed by 1's where the number of 0's is the same as the number of 1's. Here we have a context-free grammar that generates such strings using a single...

Example

Suppose our "programs" were strings of the form: 0^n1^n

$$S \rightarrow 01 \mid 0S1$$

↑
Variable

variable called S. Now S can be replaced with one of two....

Example

Suppose our "programs" were strings of the form: 0^n1^n

$$S \rightarrow 01 \mid 0S1$$

Production Production

or

productions. These are the objects we are allowed to replace the variable with. Now as we can see, these productions may be recursive using the variable we are replacing. This how we can form loops to build more complex strings. We use these productions which are sometimes called rules to generate strings...

Example

$$S \rightarrow 01 \mid 0S1$$

$$S \Rightarrow 0S1$$

using a derivation. A derivation must begin with one of the rules of the start variable; here we only have one variable and so it's the start. Then we continously replace variables until no more variables exist in the output string. Then we will obtain the string the derivation generates. So here we start with the second rule. This says that S derives 0S1. Then we can use it again by replacing the middle S....

Example

$$S \rightarrow 01 \mid 0S1$$

$$\begin{aligned} S &\Rightarrow 0S1 \\ &\Rightarrow 00S11 \end{aligned}$$

notice that all of the alphabet symbols, also called terminals, remain, we simply replace S with what the rule produces. If we do it again we obtain...

Example

$$S \rightarrow 01 \mid 0S1$$

$$\begin{aligned} S &\Rightarrow 0S1 \\ &\Rightarrow 00S11 \\ &\Rightarrow 000S111 \end{aligned}$$

another 0 and 1. Now if we instead use the first rule of our grammar, then we can delete the middle S to stop looping...

Example

$$S \rightarrow 01 \mid 0S1$$

$$\begin{aligned} S &\Rightarrow 0S1 \\ &\Rightarrow 00S11 \\ &\Rightarrow 000S111 \\ &\Rightarrow 00001111 \end{aligned}$$

Example

Suppose our "programs" were strings of well-balanced parens:

$$S \rightarrow () \mid (S) \mid SS$$

Here is a grammar for balanced parentheses. This one is an extension of the previous one. Furthermore, this grammar has practical importance. In programming languages we must be sure that if the programmer opens a paren, then they must eventually close the paren. This grammar checks to make sure that's true. Notice also that the last rule calls S twice. Let's derive an example string.

Example

$$S \rightarrow () \mid (S) \mid SS$$

$$S \Rightarrow SS$$

We start with the last rule, and it presents a challenge. In the next step which S do I replace? Currently, it doesn't matter, but later when we talk about ambiguity it will matter, but for now, we can replace either one, but I'll stick with a left-most first approach....

Example

$$S \rightarrow () \mid (S) \mid SS$$

$$S \Rightarrow SS$$
$$\Rightarrow (S)S$$

the second rule says to replace the S with parens with a new S in the middle. We are only allowed to replace a single variable at a time, so we can't get in a hurry. Next we have....

Example

$S \rightarrow () \mid (S) \mid SS$

$S \Rightarrow SS$
$\Rightarrow (S)S$
$\Rightarrow (())S$

All the terminals stay the same and we only replace a single variable at a time. We keep going until we have no varables left. Next we obtain...

Example

$S \to () \mid (S) \mid SS$

$S \Rightarrow SS$
$\Rightarrow (S)S$
$\Rightarrow (())S$
$\Rightarrow (())SS$

Then...

Example

$S \rightarrow () \mid (S) \mid SS$

$S \Rightarrow SS$
$\Rightarrow (S)S$
$\Rightarrow (())S$
$\Rightarrow (())SS$
$\Rightarrow (())()S$

And...

Example

$S \rightarrow () \mid (S) \mid SS$

$S \Rightarrow SS$
$\Rightarrow (S)S$
$\Rightarrow (())S$
$\Rightarrow (())SS$
$\Rightarrow (())()S$
$\Rightarrow (())()()$

And finally we obtain a string with only balance parens. Now...

Example

$$
\begin{aligned}
E &\to T + T \mid T * T \\
T &\to T + F \mid T * F \mid F \\
F &\to 0 \mid 1
\end{aligned}
$$

$$
\begin{aligned}
E &\Rightarrow T + T \\
&\Rightarrow T * F + T \\
&\Rightarrow T + F * F + T \\
&\Rightarrow F + F * F + T \\
&\Rightarrow 0 + F * F + T \\
&\Rightarrow 0 + 1 * F + T \\
&\Rightarrow 0 + 1 * 1 + T \\
&\Rightarrow 0 + 1 * 1 + F \\
&\Rightarrow 0 + 1 * 1 + 1
\end{aligned}
$$

let's take a look at a more interesting example. Here is a grammar for arithmetic operations, and I give a derivation of the program 0 + 1 * 1 + 1. I used the same technique as I did the others, we just have more options, but as long as you followed the rules you can't go wrong. So what's the formal definition of a context-free grammar? I'm glad you asked...

Context-Free Grammar

A context-free grammar (CFG) is a 4-tuple (V, Σ, S, R)

- A finite set of variables (or non-terminals) V
- A finite set of terminals (alphabet symbols) Σ
- A start variable $S \in V$
- A set of rules $R \subseteq V \times (V \cup \Sigma)^*$

A context-free grammar is a 4-tuple consisting of a set of variables, also called non-terminals and are usually written in uppercase but this is not strictly required, and then a set of terminals, these are the characters of the strings the grammar generates, then a start variable is designated, informally we always choose the variable in the first rule of the grammar, and finally, a grammar is a set of rules. A rule is a pair of a variable and its production which is a string over variables and terminals. Notice that variables are allowed to produce the just variables, just terminals, and a mixture. The language of a context-free grammar...

Context-Free Grammar

The language of a context-free grammar $G = (V, \Sigma, S, R)$:

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

consists of the set of all the derivable (generated) strings starting with the start variable. Now, derivations are actually very important to parsing, because they correspond to...

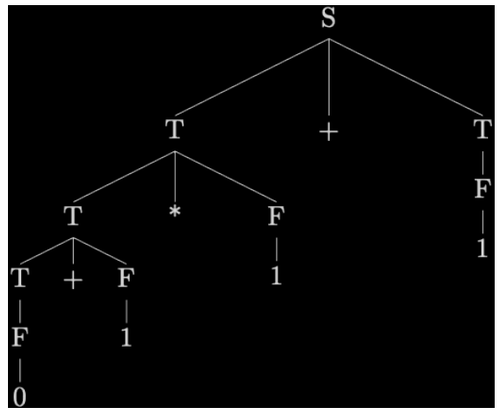
Context-Free Grammar

$E \rightarrow T + T \mid T * T$

$T \rightarrow T + F \mid T * F \mid F$

$F \rightarrow 0 \mid 1$

$$
\begin{aligned}
S &\Rightarrow T + T \\
&\Rightarrow T * F + T \\
&\Rightarrow T + F * F + T \\
&\Rightarrow F + F * F + T \\
&\Rightarrow 0 + F * F + T \\
&\Rightarrow 0 + 1 * F + T \\
&\Rightarrow 0 + 1 * 1 + T \\
&\Rightarrow 0 + 1 * 1 + F \\
&\Rightarrow 0 + 1 * 1 + 1
\end{aligned}
$$

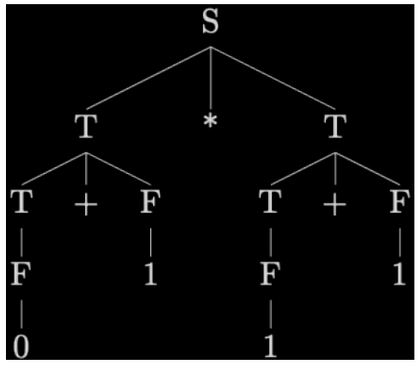


parse trees. The parse tree of the string 0+1*1+1 is on the right. If we start at the root, and traverse down the tree moving left-most first, then we can see that it follows the derivation exactly. Parse trees are used a lot in computer science esp. when parsing the syntax of programming languages. An interesting feature of this example grammar is that the string 0+1*1+1 has a second left-most derivation....

Context-Free Grammar

$$E \rightarrow T + T \mid T * T$$
$$T \rightarrow T + F \mid T * F \mid F$$
$$F \rightarrow 0 \mid 1$$

$$\begin{aligned} S &\Rightarrow T * T \\ &\Rightarrow T + F * T \\ &\Rightarrow F + F * T \\ &\Rightarrow 0 + F * T \\ &\Rightarrow 0 + 1 * T \\ &\Rightarrow 0 + 1 * T + F \\ &\Rightarrow 0 + 1 * F + F \\ &\Rightarrow 0 + 1 * 1 + F \\ &\Rightarrow 0 + 1 * 1 + 1 \end{aligned}$$



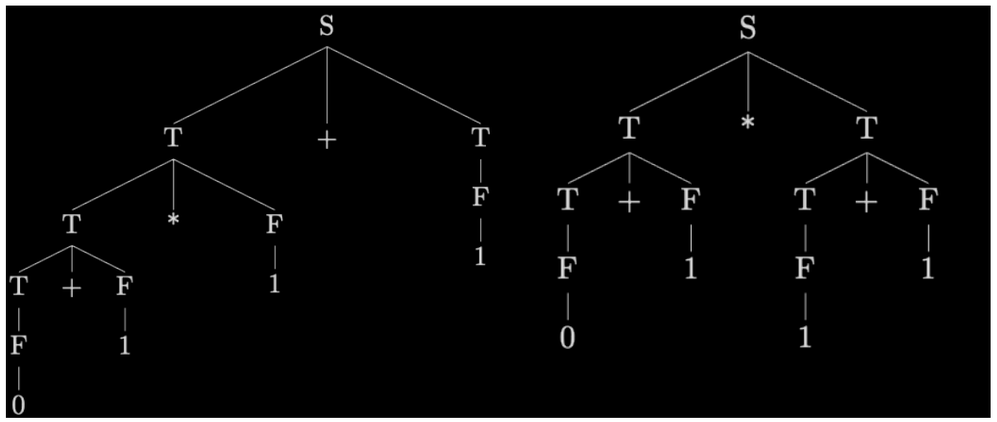
given here. Instead of starting with addition we start with multiplication, and then proceed with a left-most derivation, and the parse tree is significantly different...

Context-Free Grammar

$E \to T + T \mid T * T$

$T \to T + F \mid T * F \mid F$

$F \to 0 \mid 1$



and comparing them shows that they are not equivalent. This is bad, because when more than one parse tree exists we don't know what the proper syntax of the word is supposed to be. For example, this is equivalent to parsing a computer program and then coming up with two different ways to parse it, how would the compiler figure out when a syntax error has been made if there are multiple parse trees? How would it figure out where the syntax error occurred? Which parse tree is the correct one? 👇

If a context-free grammar generates a word with more than one parse tree, that is, has more than one left-most derivation, then we call that grammar....

Ambiguous Grammars

A context-free grammar $G = (V, \Sigma, S, R)$ is ambiguous if and only if there exists a string s such that G can generate s from more than one left-most derivation.

ambiguous. Fixing ambiguity requires one to assign precedences to the operators in the grammar. For example, forcing the grammar to respect the order of operations when deriving expressions would fix our example grammar. I think we will stop here for this lecture, but we still have lots to learn about syntax!