

Grammars, their languages, their strings, their derivations. And ambiguity. Plus syntax-directed translation.

Thursday, September 17, 2009
2:57 PM

In the recitation section, Joel showed you how to construct a *recursive descent parser*. Let's refresh our memory.

Regexes are not enough to parse regexes.

Recall, *parse* a regex = tell if a string is a valid regex. (In general, parsing actually does more, see below)

- Why can't you write a regex that can tell if a string is a valid regex?
- *Regexes have parentheses.*
- So what?
- *I should add that in a regex these parentheses must be balanced. For example $((a)((a)))$ is not ok.*
- I see. This seems important.
- *Yes. Because of these parentheses, regexes are one of those strings that cannot be parsed with a regex. I know because I tried to write a regex that matches only strings with balanced parentheses and I failed.*
- How can you be sure that such a regex does not exist?
- *OK, a regex has the same power as a DFA, and all that a DFA can do is to scan a string left-to-right and remember what it has seen in the prefix of the string scanned so far.*
- Why can't you remember why how many left parentheses remain open (unmatched)? Dedicate one state for having seen zero open parens, one for 1 open paren, etc?
- *That would require the number of states to be a function of the length of the input. Consider the string $(((((...((($. So, the automaton would not be finite-state. A DFA/NFA simply does not have enough memory.*

What subset of regexes that can be parsed with regexes?

- Well, just take parentheses away. This grammar describes these regexes.

$$R ::= c \mid R R \mid R' | R \mid R'^*$$

- Are you sure this language can be described with a regex? This grammar is recursive; regexes are not. They have only repetition (and concatenation and repetition).
- The important thing is that one can write a regex that accepts exactly the same set of strings. And you can do it without recursion. Key idea:

you can rewrite

$$R ::= c \mid R'^* \quad \# \text{ regex is a character or a regex followed by a star}$$

into

$$c'^* \quad \# \text{ a character followed by zero or more '}'$$

See the recitation parser for the rest of this regex, which covers rules $R R$ and $R' | R$.

- Hmm, if parentheses are such a pain when parsing, could we design regex syntax that does not use parentheses and still be able to express everything that regexes express?
- *I need to think about it before the exam. It will probably result in syntax considered by some to*

be unfriendly, like that of Scheme.

We need grammars: a somewhat mild extension to finite automata, but more powerful.

- What does powerful mean?
- Can tell apart strings that regexes cannot, eg whether it has balanced parens. But can also discover program structure (precedence) that regexes cannot. See below.

Grammar: a recursive definition of a language

Example:

- base case: any character is *regular expression*
- inductive case:
 - if e_1, e_2 are regular expression then $e_1 | e_2$ is also a regular expression
 - if e_1, e_2 are regular expression then $e_1 e_2$ is also a regular expression
 - if e is a regular expression then e^* is also a regular expression
 - if e is a regular expression then (e) is also a regular expression

- It's tedious to write in English. Can we invent a handy notation?
- We have been using it already. It's called a context-free grammar.

$$R ::= c \mid R R \mid R' | R \mid R'^* \mid (R)$$

- What does this notation describe?
- A language (set of strings), just like it does with a regex.
(Remember, when it comes to strings, *language is defined as a set of strings*)
- Is there a recipe to obtain this set of strings?
- Depends how long you are willing to wait. This set can be infinite.
- OK, how about a procedure that prints some string from the language.
- Let's first see how we would obtain such a procedure for regexes.

Generating a string described by a regex.

- i. convert the regex to an NFA.
- ii. perform a random walk from the start state
- iii. if you reach a final state, print the string traversed along the walk.
- iv. if you reach a state without outgoing transitions, goto ii.

random walk = randomly decide which outgoing transition to take.

(properly, we should say that the choice is not done randomly but non-deterministically.)

Generating a string described by a grammar.

- How is this random walk related to generating a string from a grammar?
- We don't walk along the automaton but do something recursive.

The string generation recipe. Given a grammar

$$E ::= n \mid E + E \mid E * E$$

we want to generate this (infinite) language

$\text{language}(E) = \{ n, n+n, n*n, n*n+n, \dots \}$

This recursive procedure can print any string described by the grammar

```
def printSomeString():
    switch (choice()):
        case 1: print "n"
        case 2: printSomeString(); print "+"; printSomeString()
        case 3: printSomeString(); print "*"; printSomeString()
```

choice is non-deterministic choice. Or think of it as random, if that's seem simpler.

Now we understand the process of generating a string. Can we invent a more convenient notation (language) for this process? Sure.

Let's write down the *start symbol* of the grammar. Rewrite it until nothing can be rewritten. At that point, we have obtained a string. This is called a derivation.

Example:

$E \rightarrow E + E \rightarrow E * E + E \rightarrow n * E + E \rightarrow n * E + n \rightarrow n * n + n$

\rightarrow is the symbol for one rewrite step

- Can you tell which symbol E was rewritten in each step?
- Yes. But I am curious how the choice of which symbol to rewrite (when there were multiple symbols to choose from) relates to the value of choice() in the above recursive procedure.
- When there was a choice of which symbol to rewrite, would the procedure get stuck if we chose another symbol?
- Let me think about that.

Parsing vs. derivation

- The process shown above is *derivation*. It goes from what to what?
- From start symbol to a string
- Could parsing be related to the derivation?
- Parsing seem the opposite of derivation. Given a string, find a derivation that derives that string.
- What if we cannot find such a derivation?
- Then the string is not from the language. It has a "syntax error".

Let's understand what parsing is used for.

- First, what is input the parser?
- a string, of course.
- And how about the grammar?
- You could say that's an input, too. Although the grammar is typically hardcoded in the parser.
- What is the output of parsing?
- a boolean: whether the string is/is not in the language
- That can hardly be enough knowledge about the program for the compiler to compile it.
- OK, the AST is produced, too.
- Well, not quite. The AST is actually produced after parsing.

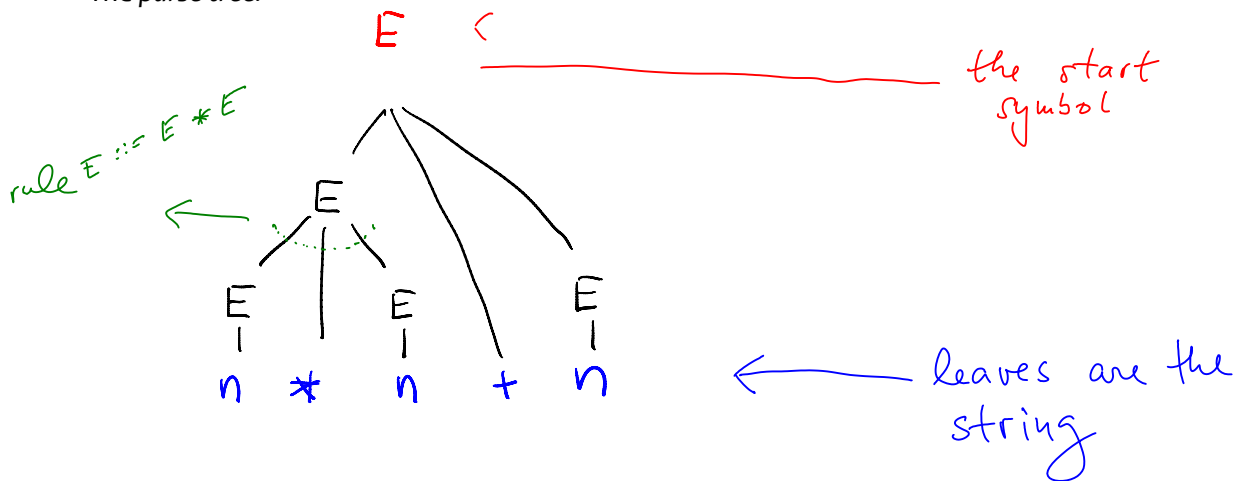
- From what is it produced?
- Could the derivation be the input for constructing the AST?
- Strange, but the derivation is the true putput of parsing, and the AST is computed from that.

It helps to think of the derivation in more convenient way, as a tree.

The derivation:

$E \rightarrow E + E \rightarrow E * E + E \rightarrow n * E + E \rightarrow n * E + n \rightarrow n * n + n$

The parse tree:



How did we built this tree? Given a derivation,

- went top-down
- we decided to visualize each step of the derivation by adding children to a node.
- that node is the symbol being rewritten.
- the children correspond what the symbol was rewritten to.

Interesting facts

- The leaves are the characters of the string
- Multiple derivations can produce the same parse tree! That's ok.

Danger with parse trees.

When we do parsing, we actually don't care about the derivation, but about the parse tree. The parse tree is useful because it encodes th meaning of the programs (it shows in which order the operations are evaluated.)

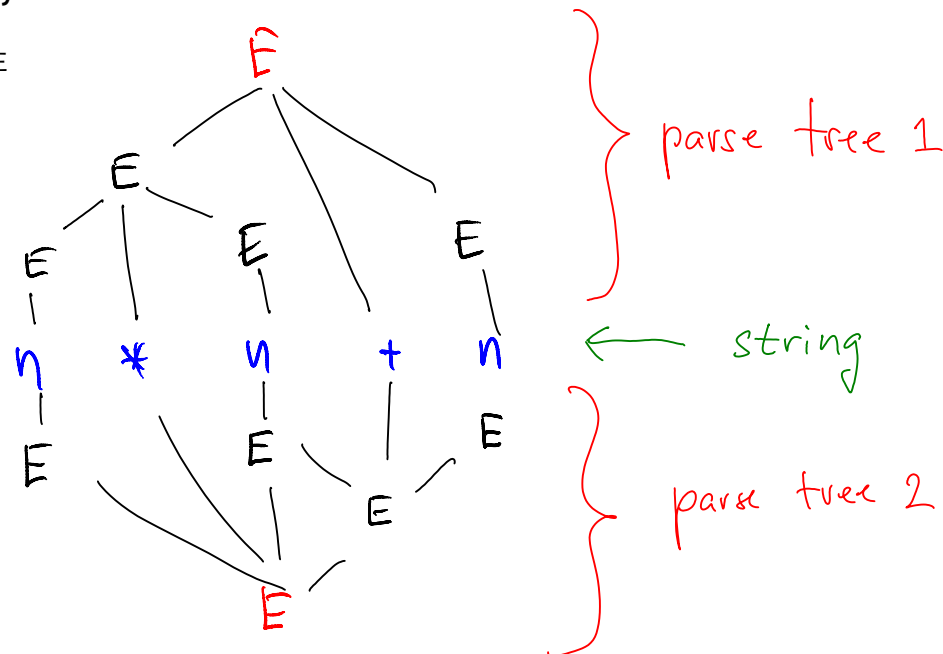
- But what is a string has multiple parse trees?
- Did not we already say that was ok?
- What was ok was that it does not matter what derivation constructed a parse tree. But it matters whether a single string has a unique parse tree.
- Because if it has multiple trees, then it has multiple (amnbiguous) meanings.
- Yes.

Ambiguos grammars.

- We say that a grammar is *ambiguous* if there is a string on which this grammar gives multiple parse trees.
- How do we test if a grammar is ambiguous?
- If we find a string and two parse trees for it, it is ambiguous.
- And we cannot find such a string?
- Then in general we cannot be sure it is non-ambiguous. It is one of those undecidable problems. But there are often easy ways to prove the grammar is non-ambiguous.

Example of ambiguity

$E ::= n \mid E * E \mid E + E$



Resolving Ambiguity.

- Which of the two parse trees do we want?
- Well, if the grammar describes arithmetic expressions (as it seems it does), the tree that "computes" $*$ before $+$ is the right one.
- The parse tree "computes" something?
- No, it just shows the order in which operations are intended to be evaluated according to the grammar.

Solution 1: make the grammar non-ambiguous, via rewriting the grammar.

Example: ambiguous:

$E ::= n \mid E * E \mid E + E$

(manually) rewritten to be non-ambiguous:

$E ::= E + E_2 \mid E_2$
 $E_2 ::= E_2 * n \mid n$

- How do you go about convincing yourself that that the second grammar (i) induces the same language as the original one and (ii) is not ambiguous.

Solution 2:

- What if the grammar is too hard to rewrite?
- Could we indicate which of the multiple trees do we want?

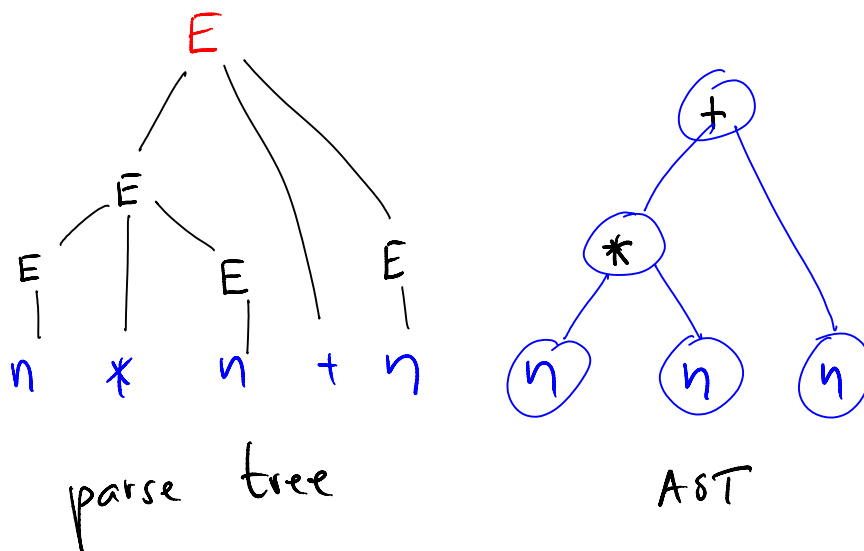
- Sometimes. We could say that $*$ has higher precedence than $+$, and that will prefer the upper of the two trees. This approach works best for rules with binary operations (like $E + E$) on the RHS (right-hand side)

In your project, you will see how to implement this.

How to compute the AST?

We said that the AST is computed from the real result of parsing -- the derivation. But the derivation is best thought of as the parse tree, and indeed we will compute the AST by transforming the parse tree.

First, what is the difference between a parse tree and an AST?



- Clearly, the AST is smaller. Which nodes from the parse tree have been omitted?
- Well, the E nodes are gone. I wonder if any information present in the parse been lost in the translation?
- Clearly, the parse tree is determined by the grammar (its parent-child links correspond to rules in the grammar). Does the AST have a direct relation to the grammar?
- It seems the programmer can design the AST however she wants. It does not even need to be a tree, really, as long as the interpreter understands it.
- In fact, this expression, in reverse polish notation (RPN) is a well-know representation of programs. It shows the precedence structure without using parentheses.

$1+2*3$ is $123*+$ in RPN
 $(1+2)*3$ is $12+3*$ in RPN

Puzzle: Can you decode the meaning of these RPN expressions?

Language design: can RPN ideas be used to come up with syntax of regexes than can be parsed with a regex?

Translating the parse tree.

- Given a string, the parser produces a parse tree, which we then "evaluate".
- Didn't you say we will transform the parse tree (into AST)?
- We will, by evaluating it.
- Strange.
- Let's look at examples.

Our parser:

First let's understand the format accepted by our parser.

```
// characters skipped by the lexer

%ignore /\n+/

%%    // the grammar starts here

R -> 'a'
    | R '*'
    | R R
    | R '|' R
    | '(' R ')'
    ;
```

Now let's add semantic rules, so called because they add semantics to the syntax (parse) tree. These rules built an AST:

```
R -> 'a'           %{ return n1.val           %}
    | R '*'         %{ return ('*', n1.val)     %}
    | R R           %{ return ('.', n1.val, n2.val) %}
    | R '|' R       %{ return ('|', n1.val, n3.val) %}
    | '(' R ')'     %{ return n2.val           %}
    ;
```

The rules work well on some inputs:

`a|aa` correctly produces `('|', 'a', ('.', 'a', 'a'))`

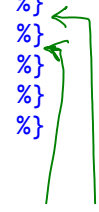
but the input

`aa|a` produces `('.', 'a', ('|', 'a', 'a'))`
when it should produce `('|', ('.', 'a', 'a'), 'a')`

- Why is that?
- The grammar is ambiguous, and the parser happened to select the wrong tree on the string `aa|a`.

We add declaration informing the parser of the precedence of the three rules that induce ambiguity.

```
R -> 'a'           %{ return n1.val           %}
    | R '*'         %dprec 1 %{ return ('*', n1.val)     %}
    | R R           %dprec 2 %{ return ('.', n1.val, n2.val) %}
    | R '|' R       %dprec 3 %{ return ('|', n1.val, n3.val) %}
    | '(' R ')'     %{ return n2.val           %}
    ;
```



The semantic rules specify how the evaluation of the parse happens. Specifically, each rule computes the value of the node (of the parse tree). For example:

