Lecture 11

COP3402 FALL 2015 – DR. MATTHEW GERBER – 10/12/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- Parsing
- Context-Free Grammars

The Limits of Regular Languages

We already know how to scan input with regular languages.

The problem is, for parsing, regular languages aren't enough.

Consider this set of strings:

$$\{ \alpha^n \beta^n \mid n \text{ is an integer; } \alpha, \beta \in \Sigma \}$$

There's no way to write a regular expression for that.

- You can't iterate—let alone recurse—with regular expressions
- When parsing a language, you'd *better* be able to recurse
- Otherwise, you can't express nested structures

Breaking the Limits

So far we've stuck with three rules:

Concatenation → (s r)

Alternation (choice) → (s | r)

Kleene closure (repetition) → (s)*

- (There are other rules, but they're all just compound expressions of those three that are more convenient for certain cases)

- For scanning they're fine
- For parsing they're not enough
- *We need a new rule for recursion*
- But we'll also need a new way to write it

Replacement Rules

Consider the following three *rewriting rules* (or *syntax rules*, *productions* or *syntactic equations*):

- sentence ::= subject predicate
- subject ::= “John” | “Mary”
- predicate ::= “eats” | “talks”

The notation is fairly obvious; we’ll get to it more clearly in a few slides, but for now:

- “Literal strings” are enclosed in quotes
- | still means “or”
- ::= means *is defined as* or, equivalently, *is replaced by*

Then we have three valid sentences:

- “John eats”, “John talks”, “Mary eats”, and “Mary talks”.

Replacement rules are allowed to reference themselves.

- So we have concatenation, alternation, closure (we still have the * operator) *and* recursion

Context Free Grammar

Any language that can be defined using rules like those from the previous slide is a *Context Free Language*.

- It's called "context free" because only *one thing* is on the left of each replacement rule
- In other words, each replacement rule can operate without further *context*

Direct analogues to all three important incarnations of regular languages exist:

- Regular languages are generated by *regular expressions* and can be recognized by DFAs
- CFLs are generated by *Context Free Grammars* and can be recognized by *Pushdown Automata*

Components of a CFL

Let's take a look at that little language again:

- sentence ::= subject predicate
- subject ::= "John" | "Mary"
- predicate ::= "eats" | "talks"

We have four important pieces here

- "John", "Mary", "eats" and "talks" are *terminal symbols (*collectively referred to as the *vocabulary*)
  - They cannot be substituted for other symbols – substitution *terminates* when you reach them
- sentence, subject and predicate are *non-terminal symbols* (or *syntactic classes* or *categories*)
  - They can be substituted for other symbols
- The set of *rewriting rules*, also known as the *grammar* proper
  - One rule must be specified for each non-terminal symbol
- The *start symbol*, in this case sentence

These pieces form a tuple (T, N, R, S) that describes a context-free language.

Languages and Meta-languages

A *meta-language* is a language used to describe another language.

- Regular expressions are *sort of* a meta-language
- “Perl 5 Regular Expressions” is a better example of a meta-language

We use *Extended Backus-Naur Form* to describe context-free languages.

- You’ve already seen what rewrite rules look like
- As well as alternation and string literals
- Closure uses curly braces
- Optional parts use square brackets

```
sentence ::= subject predicate
subject ::= “John” | “Mary”
number ::= digit {digit}
expression ::= [“-”] term
```

A Very Small Language

Example of a grammar for a trivial language:

program	::= stmt-list “end”
stmt-list	::= stmt | stmt “;” stmt-list
stmt	::= var “=“ expression
expression	::= var “+” var | var “-” var | var | number
var	::= letter { letter | digit }
number	::= digit { digit }

Grammars in Operation

A statement generation – that is, the process of moving from the start symbol to terminal symbols – is called a *derivation*.

A *left most derivation* is a derivation in which *only the left-most non-terminal symbol is replaced*.

Let's look at an example of a left most derivation for a really simple assignment statement:

a := b * (a + c)

Left Most Derivation Example

GRAMMAR

R1 assgn	::= id “:=“ expr
R2 id	::= “a” | “b” | “c”
R3 expr	::= id “+” expr
R4	| id “*” expr
R5	| “(“ expr “)”
R6	| id

DERIVATION

<assgn>	→ <id> := <expr>	R1
	→ a := <expr>	R2
	→ a := <id> * <expr>	R4
	→ a := b * <expr>	R2
	→ a := b * (<expr>)	R5
	→ a := b * (<id> + <expr>)	R3
	→ a := b * (a + <expr>)	R2
	→ a := b * (a + <id>)	R6
	→ a := b * (a + c)	R2

Derivation and the Parse Tree

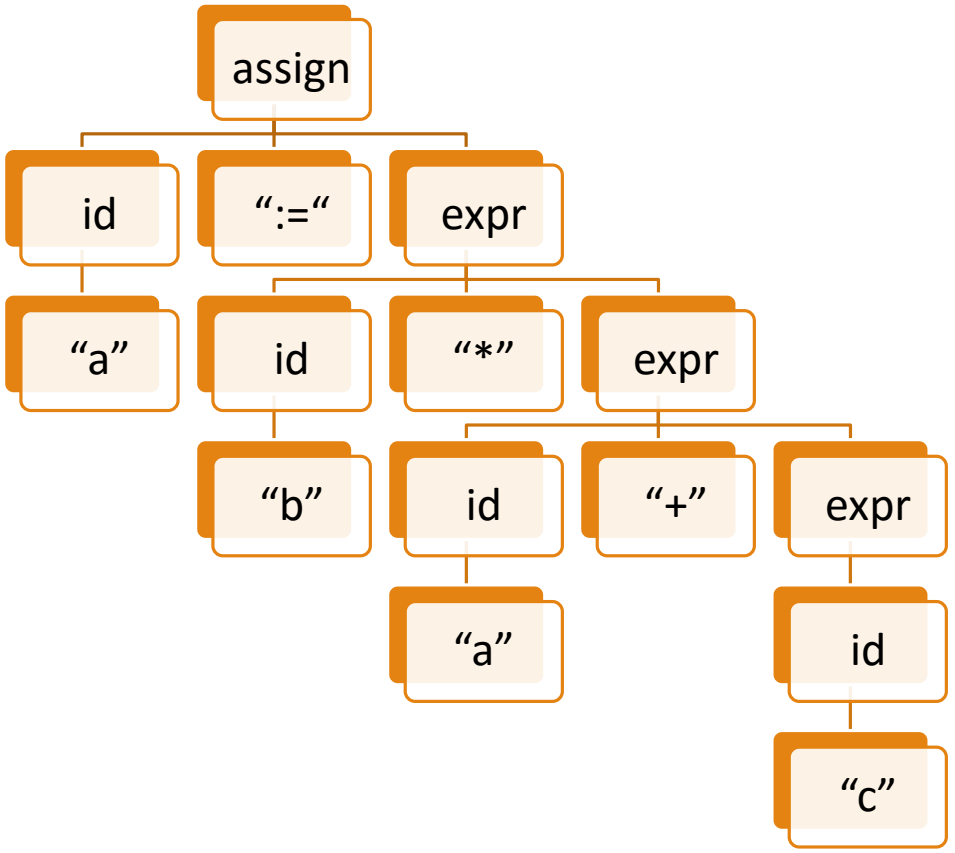
Remember that the whole point of the parser is to generate parse trees.

A parse tree is just a graphical representation of a derivation!

- Every *internal* node is a *non-terminal symbol*.
- Every *leaf* node is a *terminal symbol*.

Here's the parse tree for that same statement:

a := b * (a + c)



Next Time: Parser Problems