15-150

Principles of Functional Programming

Slides for Lecture 24

Context-Free Grammars and Parsing

April 23, 2020

Michael Erdmann

Lessons:

- Context-Free Grammar
  - Derivation
  - Context-Free Language
- Abstract Syntax Tree (AST)
- Recursive-Descent Parsing
- Awareness of some subtleties

Language Hierarchy

Class of Languages	Recognizers	Applications
Unrestricted	Turing Machines	General Computation
Context-Sensitive	Linear-bounded automata	Some simple type-checking
Context-Free	Nondeterministic automata with one stack	Syntax checking
Regular	Finite Automata	Tokenization

Big Picture

Big Picture

(atp)

user keystrokes

chars (stream/list) **f a c t (3) ...**

Big Picture

user keystrokes

(atp)

chars (stream/list) **f a c t (3) ...**

tokens

token is some datatype defined within a compiler.

Maybe something like:

```
datatype token = LAMBDA | LPAREN | RPAREN
                 | ID of string | INT of int | ...
```

Big Picture

user keystrokes

(atp)

chars (stream/list)   **f a c t (3) …**

tokenizer

tokens   **(ID "fact")   LPAREN   (INT 3)   RPAREN …**

`token` is some datatype defined within a compiler.

A tokenizer groups characters together into meaningful tokens, perhaps using a regular expression matcher.

Big Picture

user keystrokes

(atp)

chars (stream/list) **f a c t (3) …**

tokenizer

Regular Expressions can be useful

tokens **(ID “fact”) LPAREN (INT 3) RPAREN …**

Big Picture

user keystrokes

(atp)

chars (stream/list) **f a c t (3) …**

tokenizer

Regular Expressions can be useful

tokens **(ID "fact") LPAREN (INT 3) RPAREN …**

expressions

The compiler has an internal datatype to represent "expressions", perhaps called **exp**, maybe like this:

```
datatype exp =   Var of string | Int of int
               | App of exp * exp | ...
```

Big Picture

user keystrokes

(atp)

chars (stream/list) **f a c t (3) …**

tokenizer

Regular Expressions can be useful

tokens **(ID “fact”) LPAREN (INT 3) RPAREN …**

parser

expressions **App(Var “fact”, Int 3) …**

The compiler has an internal datatype to represent “expressions”. A parser assembles tokens into meaningful expressions, generally with the aid of a context-free grammar (creating parsers can be automated, similarly as we could create regular expression matchers automatically).

Abstract Syntax Tree (AST)

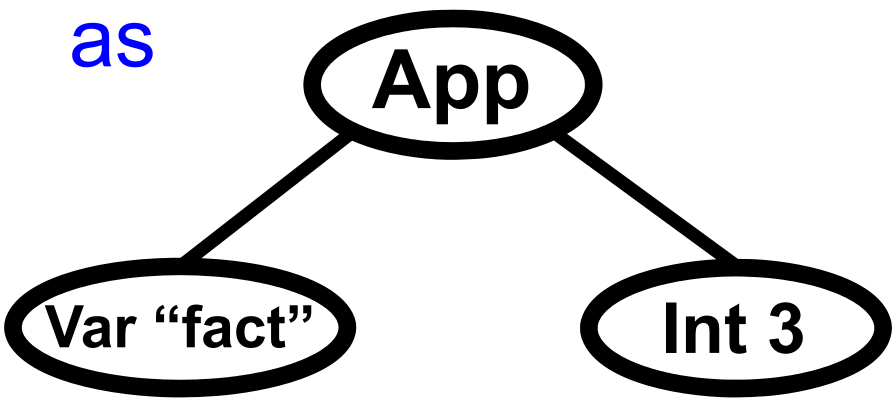
We can think of the declaration

```
datatype exp =   Var of string | Int of int
               | App of exp * exp | ...
```

as defining operator-operand trees.

They are called *abstract syntax trees*.

For instance, we can visualize **App(Var “fact”, Int 3)** as



A parser produces ASTs.

A typechecker and evaluator can then traverse them.

Big Picture

user keystrokes

(atp)

chars (stream/list)   **f a c t (3) …**

tokenizer

Regular Expressions can be useful

tokens **(ID "fact") LPAREN (INT 3) RPAREN …**

parser

Context-Free Grammars can be useful

expressions   **App(Var "fact", Int 3) …**

Big Picture

user keystrokes

(atp)

chars (stream/list) **f a c t (3) …**

tokenizer

Regular Expressions can be useful

tokens **(ID "fact") LPAREN (INT 3) RPAREN …**

parser

Context-Free Grammars can be useful

expressions **App(Var "fact", Int 3) …**

type checker

typed expressions **App(...) : int …**

Big Picture

user keystrokes

(atp)

chars (stream/list) **f a c t (3) …**

tokenizer

Regular Expressions can be useful

tokens **(ID "fact") LPAREN (INT 3) RPAREN …**

parser

Context-Free Grammars can be useful

expressions **App(Var "fact", Int 3) …**

type checker

typed expressions **App(…) : int …**

evaluator

values **6 …**

Syntax Charts for Programming Languages

Let’s use the following abbreviations:

`P`	stands for	Program
`E`	stands for	Expression
`M`	stands for	Match
`Q`	stands for	Pattern

(of course, there are lots more …)

Syntax for SML (partial)

$$\mathbf{P} \;\rightarrow\; \varepsilon \;\mid\; \mathbf{E;}\;\; \mathbf{P}$$

This means: A “program” is either (i) empty or (ii) an expression, followed by a semi-colon, followed by a program (recursive!).

Syntax for SML (partial)

```
P → ε | E; P

E → E + E | E * E | ...
      | E andalso E | ...
      | case E of M | ...
```

This means: An "expression" could be an arithmetic expression composed of two subexpression, or similarly a logical expression, or a case expression involving an expression and a match, or …

Comment: This description is **only** syntax, not type-checking.

Syntax for SML (partial)

“|” in description of possibilities

$$P \rightarrow \varepsilon \mid E;\ P$$

$$E \rightarrow E + E \mid E * E \mid \ldots$$
$$\mid E \texttt{ andalso } E \mid \ldots$$
$$\mid \texttt{case } E \texttt{ of } M \mid \ldots$$

“**|**” in SML

$$M \rightarrow Q \Rightarrow E \mid Q \Rightarrow E \ \textbf{|}\ M$$

This means: A “match” consists of one or more instances of
Q ⇒ E separated by SML’s **|** bar
(recall that **Q** stands for “pattern”).

Syntax for SML (partial)

$$P \rightarrow \varepsilon \mid \texttt{E; P}$$

$$\begin{aligned} \texttt{E} \rightarrow\ & \texttt{E + E} \mid \texttt{E * E} \mid \texttt{...} \\ & \mid \texttt{E andalso E} \mid \texttt{...} \\ & \mid \texttt{case E of M} \mid \texttt{...} \end{aligned}$$

$$\texttt{M} \rightarrow \texttt{Q} \Rightarrow \texttt{E} \mid \texttt{Q} \Rightarrow \texttt{E | M}$$

Alternate Notation:
Backus Naur Form (BNF)

```
P ::=  ε | E; P

E ::=  E + E | E * E | ...
       | E andalso E | ...
       | case E of M | ...

M ::=  Q ⇒ E | Q ⇒ E | M
```

Alternate Notation:
Backus Naur Form (BNF)

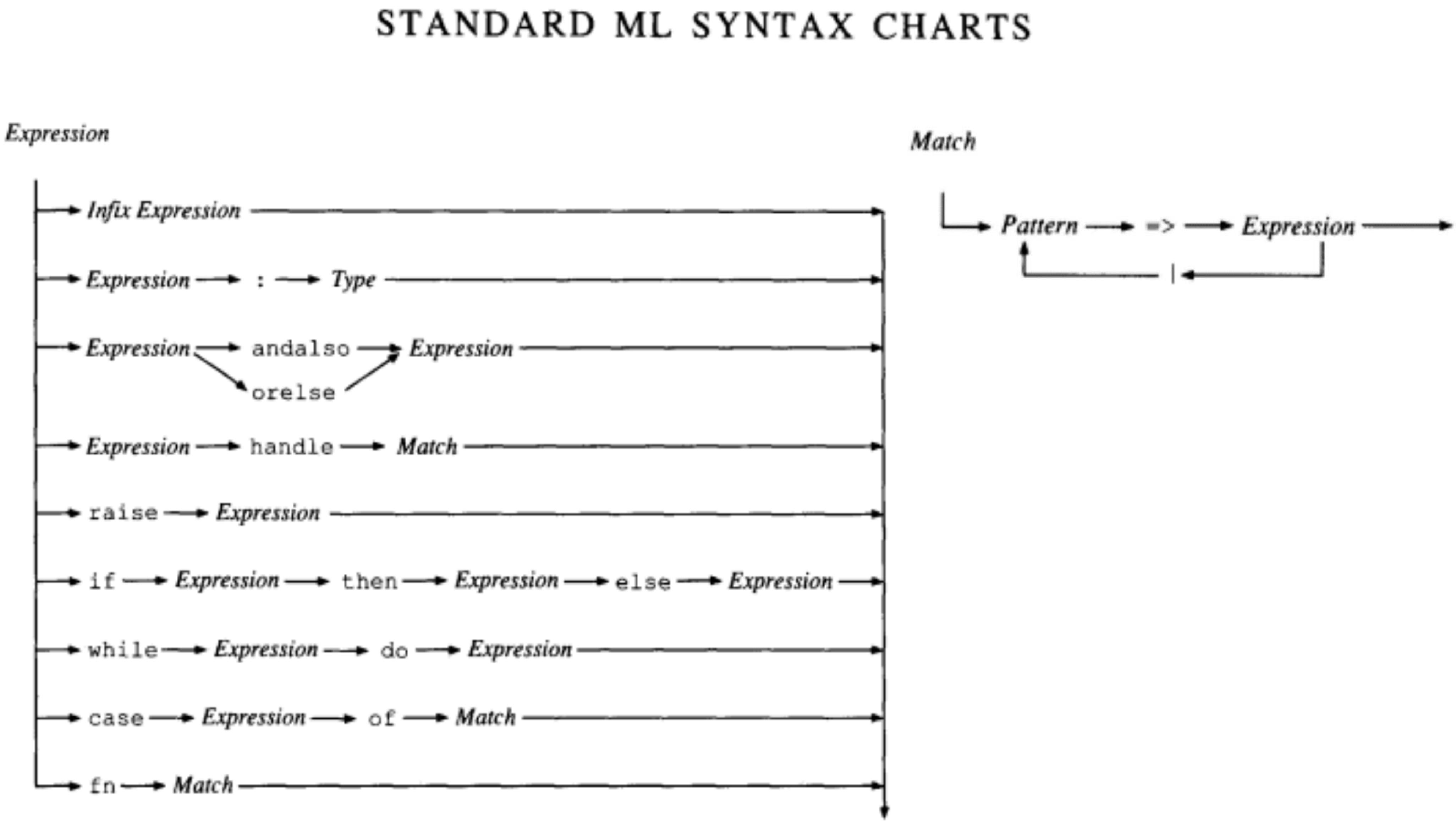
```
P ::=  ε | E; P

E ::=  E + E | E * E | ...
       | E andalso E | ...
       | case E of M | ...

M ::=  Q ⇒ E | Q ⇒ E | M
```

`::=` instead of →.

Or use flow charts. These are from the back of Paulson's *"ML for the Working Programmer"*:



Context Free Grammars

We saw three formats for describing (some of) the syntax of SML:

- Expansion rules (using $\rightarrow$)
- BNF (using **`::=`**)
- Flow charts

These are three different ways of presenting a *context-free grammar* for (some of) the syntax of SML.

The grammar tells us how to *expand* a symbol (such as **E**) in different ways (for instance, as **E + E**).
Each such possibility is called a *rule*.

"context-free" means that one can make an expansion without worrying about the surrounding symbols (e.g., whether and how the original **E** is part of some larger expression).

(would **not** be true for type-checking)

Context-Free Grammars

- Formal definition of context-free grammar.
- Language L(**G**) associated with context-free grammar **G**.
- Examples.
- Abstract syntax trees.
- Parser for a simple grammar.

Context-Free Grammar (Definition)

A *context-free grammar* **G** is specified by:

1. An alphabet Σ of *terminals*.
2. A set $\boldsymbol{V}$ of *non-terminals*. (Σ and $\boldsymbol{V}$ are disjoint.)
3. A *start symbol* in $\boldsymbol{V}$ (often it is the symbol S).
4. A set of *expansion rules*, each of the form:

$$N \to \omega,$$

with $N \in \boldsymbol{V}$ and $\omega \in (\Sigma \cup \boldsymbol{V})^*$.

(In other words, N is a single non-terminal, and ω consists of 0 or more terminals and non-terminals.)

Derivations (1 step)

Suppose α and β are two strings of terminals and non-terminals, i.e., $\alpha, \beta \in (\Sigma \cup \boldsymbol{V})^*$.

We say that β is *derivable* from α *in one step*, and write $\alpha \Rightarrow^1 \beta$ if the following holds:

There exist strings $\gamma, \delta \in (\Sigma \cup \boldsymbol{V})^*$ and a rule $N \rightarrow \omega$ in the grammar, such that

$$\alpha = \gamma\, N\, \delta \quad \text{and} \quad \beta = \gamma\, \omega\, \delta\,.$$

(In other words, β may be obtained from α by using a single expansion rule on one non-terminal N appearing in α.)

Derivations (0 or more steps)

Again, suppose $\alpha, \beta \in (\Sigma \cup \boldsymbol{V})^*$.

We say β is *derivable* from α *in zero or more steps*, and write $\alpha \Rightarrow \beta$, if either $\alpha = \beta$ or there is a sequence of 1-step derivations from α to β:

$$\alpha \Rightarrow^1 \sigma_1 \Rightarrow^1 \sigma_2 \Rightarrow^1 \cdots \sigma_n \Rightarrow^1 \beta$$

(Notation: Many authors write $\Rightarrow$ to mean $\Rightarrow^1$ and $\Rightarrow^*$ to mean $\Rightarrow$, but the notation here is more consistent with what you are used to.)

Language of a Context-Free Grammar

Let **G** be a grammar, with terminal alphabet Σ, non-terminals ***V***, and start symbol S.

The *language* L(**G**) consists of all finite-length strings over the alphabet Σ that are derivable from the start symbol S:

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

Example #1

G: $\Sigma = \{a, b\}$

$\boldsymbol{V} = \{S, A\}$

rules: $S \rightarrow AbA$

$A \rightarrow \varepsilon$ (empty string)

$A \rightarrow a$

$A \rightarrow aA$

Example #1

G: $\Sigma = \{a, b\}$

$\boldsymbol{V} = \{S, A\}$

rules: $S \rightarrow AbA$

$A \rightarrow \varepsilon$ (empty string)

$A \rightarrow a$

$A \rightarrow aA$

It is usually enough to write the rules with “or bars”, and specify Σ and S. The rest is implicit.

Example #1 (abbreviated)

G: $\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid a \mid aA$

(It is implicit that ***V*** $= \{S, A\}$.)

Example #1 (abbreviated)

G: $\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid a \mid aA$

(It is implicit that $\boldsymbol{V} = \{S, A\}$.)

Here is a sample derivation of a string in L(**G**):

$$S \Rightarrow^1 AbA \Rightarrow^1 abA \Rightarrow^1 abaA \Rightarrow^1 aba.$$

Called a *leftmost derivation* since each step expands the current leftmost non-terminal.

Here is a rightmost derivation: $S \Rightarrow^1 AbA \Rightarrow^1 Aba \Rightarrow^1 aba.$

Here is a different leftmost derivation: $S \Rightarrow^1 AbA \Rightarrow^1 abA \Rightarrow^1 aba.$

Example #1 (abbreviated)

G: $\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid a \mid aA$

(It is implicit that ***V*** = $\{S, A\}$.)

What is L(**G**)?

(We have seen that aba $\in$ L(**G**).)

Example #1 (abbreviated)

G: $\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid a \mid aA$

(It is implicit that ***V*** = $\{S, A\}$.)

What is L(**G**)?

(We have seen that $aba \in L(\mathbf{G})$.)

Answer: Set of all finite strings over Σ containing exactly one b.

Ambiguity

- The previous grammar **G** is said to be *ambiguous* because a string in its language has more than one leftmost (or rightmost) derivation.
- Ambiguity is undesirable: A parser might want to produce an operator-operand tree for expressions by scanning input and performing a leftmost derivation. Ambiguity means the parse is not inherently unique.
- Deciding whether a grammar is ambiguous is uncomputable in general, but in a specific setting one may be able to design a provably unambiguous grammar.

Example #1 (revisited)

G: $\Sigma = \{a, b\}$

$S \to AbA$

$A \to \varepsilon \mid a \mid aA$

Here is an unambiguous grammar **G'** such that L(**G'**) = L(**G**):

?

(*unambiguous* means each string in L(**G'**) has a unique leftmost derivation)

Example #1 (revisited)

G: $\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid a \mid aA$

Here is an unambiguous grammar **G'** such that L(**G'**) = L(**G**):

G':

$\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid aA$

(The class nicely came up with this.)

(unambiguous means each string in L(**G'**) has a unique leftmost derivation)

Example #1 (revisited)

G: $\Sigma = \{a, b\}$ $S \to AbA$

$A \to \varepsilon \mid a \mid aA$

Here is a different unambiguous grammar **G'** such that L(**G'**) = L(**G**):

G':

$S \to b \mid bA \mid Ab \mid AbA$

$\Sigma = \{a, b\}$ $A \to a \mid aA$

(unambiguous means each string in L(**G'**) has a unique leftmost derivation)

Regular and Context-Free Languages

Let Σ be a given alphabet.

Let L be a subset of Σ^*. (finite strings over Σ)

Recall:

We say that L is *regular* if $L = L(r)$ for some regular expression r.

We now also can define:

We say that L is *context-free* if $L = L(\mathbf{G})$ for some context-free grammar $\mathbf{G}$.

Regular and Context-Free Languages

Let Σ be a given alphabet.

The languages $L = \{\ \}$, $L = \{\varepsilon\}$, and $L = \{a\}$, with $a \in \Sigma$, corresponding to the base cases of regular expressions are context-free.
(Exercise: Exhibit a context-free grammar for each L.)

The class of context-free languages is closed under alternation (union), concatenation, and Kleene Star.
(Exercise: To prove this, exhibit context-free grammars.)

Thus: Every regular language is context-free.

Example #1 (re-revisited)

G: $\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid a \mid aA$

Here is a regular expression **r** such that L(**r**) = L(**G**):

?

Example #1 (re-revisited)

G: $\Sigma = \{a, b\}$

$S \rightarrow AbA$

$A \rightarrow \varepsilon \mid a \mid aA$

Here is a regular expression **r** such that L(**r**) = L(**G**):

$$r = a^*ba^*$$

Some Languages

- Regular:

$$\{a^n \mid n \equiv 0 \bmod 3,\ n \geq 0\}$$

- Context-Free, but not Regular:

$$\{a^n b^n \mid n \geq 0\}$$

- Context-Free, but not the language of any unambiguous context-free grammar:

$$\{a^n b^m c^m d^n \mid n,m \geq 0\} \cup \{a^n b^n c^m d^m \mid n,m \geq 0\}$$

- Not Context-Free:

$$\{a^n b^n c^n \mid n \geq 0\}$$

Pumping Lemmas

- One approach for showing that a language is not regular (or is not context-free) is to use a so-called *pumping lemma*.
- A pumping lemma is an assertion that a (non-finite) language must contain infinitely many strings of a certain form.
- One uses the pumping lemma to show that the form contradicts the language definition (and so cannot be in the class of languages covered by the pumping lemma).

A Pumping Lemma for Regular Languages

Let L be an infinite regular language.

Then there exist strings α, ω, β, such that

- $\omega \neq \varepsilon$ (i.e., ω is not the empty string)
- $\alpha\omega^k\beta \in L$ for every $k \geq 0$.

(The second bullet says language L must contain strings with arbitrarily many repetitions of ω between α and β.)

[There exist stronger pumping lemmas.]

Example #2

G: $\Sigma = \{a, b\}$ $S \to \varepsilon \mid aSb$

(Implicitly $V = \{S\}$.) $L(\mathbf{G}) = \{a^n b^n \mid n \geq 0\}$

Can you use the pumping lemma to show that L(**G**) is not regular?

Recall: The pumping lemma says L(**G**) must contain all the strings $\alpha\omega^k\beta$, $k \geq 0$, for some α, ω, β, with $\omega \neq \varepsilon$.

Example #2

G: $\Sigma = \{a, b\}$ $S \rightarrow \varepsilon \mid aSb$

(Implicitly $V = \{S\}$.) $L(\mathbf{G}) = \{a^n b^n \mid n \geq 0\}$

Can you use the pumping lemma to show that L(**G**) is not regular?

Recall: The pumping lemma says L(**G**) must contain all the strings $\alpha\omega^k\beta$, $k \geq 0$, for some α, ω, β, with $\omega \neq \varepsilon$.

Now do a case analysis on how α, ω, β might overlap a string in L(**G**), and you will find that pumping ω creates strings outside the language.

Example #3

G: $\Sigma = \{a, b\}$

$\boldsymbol{V} = \{S\}$

$S \rightarrow \varepsilon \mid aSa \mid bSb$

What is L(**G**)?

Example #3

G: $\Sigma = \{a, b\}$

$\boldsymbol{V} = \{S\}$

$S \rightarrow \varepsilon \mid aSa \mid bSb$

What is L(**G**)?

Answer: $L(\mathbf{G}) = \{\omega\omega^{\mathbf{R}} \mid \omega \in \Sigma^*\}$

= all even length palindromes over Σ.

(ω^R means "reverse of ω")

Comment: L is not regular.

(A stronger pumping lemma is useful to show that.)

Example #3

G: $\Sigma = \{a, b\}$

$\boldsymbol{V} = \{S\}$

$S \rightarrow \varepsilon \mid aSa \mid bSb$

What is L(**G**)?

Answer: $L(\mathbf{G}) = \{\omega\omega^{\mathbf{R}} \mid \omega \in \Sigma^*\}$

= all even length palindromes over Σ.

How to change **G** to include odd length palindromes?

Example #3

G: $\Sigma = \{a, b\}$

$\boldsymbol{V} = \{S\}$

$S \rightarrow \varepsilon \mid aSa \mid bSb$

What is L(**G**)?

Answer: $L(\mathbf{G}) = \{\omega\omega^R \mid \omega \in \Sigma^*\}$

= all even length palindromes over Σ.

How to change **G** to include odd length palindromes?

$$S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$$

Big Picture

user keystrokes

(atp)

chars (stream/list) f a c t (3) …

tokenizer

Regular Expressions can be useful

tokens (ID “fact”) LPAREN (INT 3) RPAREN …

parser

Context-Free Grammars can be useful

expressions App(Var “fact”, Int 3) …

type checker

typed expressions App(…) : int …

evaluator

values 6 …

Parsers

- Top down recursive descent
  - Useful for LL(k) grammars: left-to-right parsing, construct a leftmost derivation with k-character lookahead.
- Bottom up operator precedence shift reduce
  - Useful for some LR(1) grammars: left-to-right parsing, construct rightmost derivation in reverse,1-character lookahead.
- Compiler compilers
- You will learn a lot more in a compilers course

Example #4 (a CFG for $*$/+ precedence)

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

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

$$F \rightarrow n \mid (E)$$ (n means any integer)

Example #4 (a CFG for $*$/+ precedence)

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

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

$$F \to n \mid (E)$$ (n means any integer)

$3+4*(2+5)$ has unique leftmost derivation

$$E \Rightarrow^1 E+T \Rightarrow^1 T+T \Rightarrow^1 F+T \Rightarrow^1 3+T$$

$$\Rightarrow^1 3+T*F \Rightarrow^1 3+F*F \Rightarrow^1 3+4*F \Rightarrow^1 3+4*(E)$$

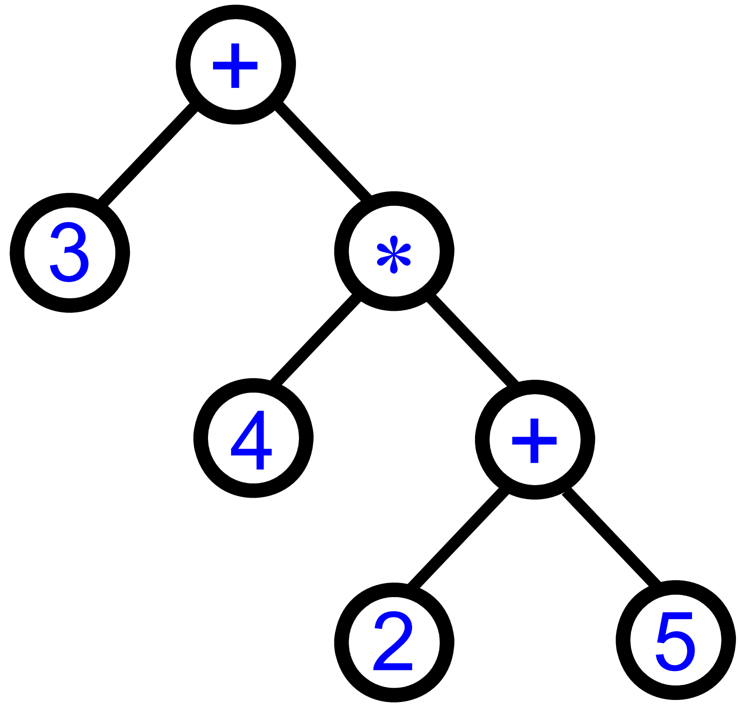
$$\Rightarrow^1 3+4*(E+T) \Rightarrow^1 \cdots \Rightarrow^1 3+4*(2+5)$$

Example #4 (a CFG for $*$/+ precedence)

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

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

$$F \rightarrow n \mid (E)$$



3+4$*$(2+5) derivation

$$E \Rightarrow^1 E+T \Rightarrow^1 T+T \Rightarrow^1 F+T \Rightarrow^1 3+T$$

$$\Rightarrow^1 3+T*F \Rightarrow^1 3+F*F \Rightarrow^1 3+4*F \Rightarrow^1 3+4*(E)$$

$$\Rightarrow^1 3+4*(E+T) \Rightarrow^1 \cdots \Rightarrow^1 3+4*(2+5)$$

Parsers

- Top down recursive descent
  - Useful for LL(k) grammars, left-to-right parsing, construct a leftmost derivation with k-character lookahead.
- Bottom up operator precedence shift reduce
  - Useful for some LR(1) grammars: left-to-right parsing, construct rightmost derivation in reverse,1-character lookahead.
- Compiler compilers
- You will learn a lot more in a compilers course

Recursive Descent Parsing

Basic Idea:

One parsing function for each nonterminal, one clause for each possible expansion rule.

Recursive Descent Parsing

<u>Basic Idea:</u>

One parsing function for each nonterminal, one clause for each possible expansion rule.

<u>Issue: Left Recursion</u>

$$E \rightarrow E + E \mid n$$ (n means any integer)

If we write **`parseE`** for E, then it would instantly call itself recursively, leading to infinite loop.

Eliminate the recursion by rewriting the grammar rules:

$$E \rightarrow nE'$$
$$E' \rightarrow \varepsilon \mid +nE'$$

(changes associativity)

Example #5 (simplified lambda calculus)

G: Σ = (implicit in the rules and tokens below)

$V = \{E, X\}$ (with E as start symbol)

$E \rightarrow \lambda X.E \mid (E\ E) \mid X$

$X \rightarrow$ any token for a nonempty alphanumeric string

```
datatype token = LAMBDA | LPAREN | RPAREN
               | ID of string | DOT

  datatype exp =   Fun of string * exp
                 | App of exp * exp
                 | Var of string
```

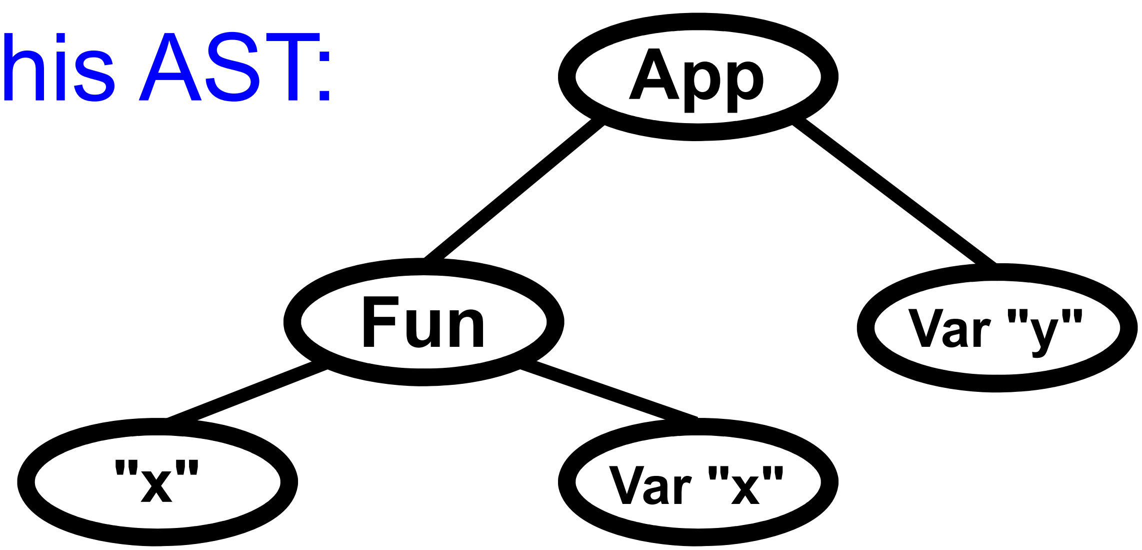
Example #5 (simplified lambda calculus)

For instance: $(\lambda x.x\ \ y)$

Tokenizes to: **`LPAREN, LAMBDA, ID("x"), DOT, ID("x"), ID("y"), RPAREN`**

Parses to: **`App(Fun("x", Var "x"), Var "y")`**

which is this AST:



CPS version of the parser

```
exception ParseError

(* parseExp : token list -> (exp * token list -> 'a) -> 'a
   REQUIRES: true
   ENSURES:  (parseExp T k) ==> k(E,T2) if T ≅ T1@T2
                    such that
                    T1 is derivable in the grammar with
                    abstract syntax E;
                raises ParseError otherwise.
*)

(* parse : token list -> exp
   REQUIRES: true
   ENSURES:  parse(T) returns E if T is derivable in the
                      grammar with abstract syntax E;
                raises ParseError otherwise.
*)
```

CPS version of the parser

```
fun parseExp ((ID x)::ts) k = k(Var x, ts)
```

$E \to X$

CPS version of the parser

```
fun parseExp ((ID x)::ts) k = k(Var x, ts)

  | parseExp (LPAREN::ts) k =
      parseExp ts (fn (e1, t1) =>
        parseExp t1
          (fn (e2, RPAREN::t2) => k(App(e1,e2), t2)
                             | _ => raise ParseError))
```

$E \rightarrow (E\ E)$

CPS version of the parser

```
fun parseExp ((ID x)::ts) k = k(Var x, ts)

  | parseExp (LPAREN::ts) k =
      parseExp ts (fn (e1, t1) =>
        parseExp t1
          (fn (e2, RPAREN::t2) => k(App(e1,e2), t2)
                             | _ => raise ParseError))

  | parseExp (LAMBDA::(ID x)::DOT::ts) k =
      parseExp ts (fn (e, ts') => k(Fun(x,e), ts'))
```

$E \rightarrow \lambda X.E$

CPS version of the parser

```
fun parseExp ((ID x)::ts) k = k(Var x, ts)

  | parseExp (LPAREN::ts) k =
      parseExp ts (fn (e1, t1) =>
        parseExp t1
          (fn (e2, RPAREN::t2) => k(App(e1,e2), t2)
                             | _ => raise ParseError))

  | parseExp (LAMBDA::(ID x)::DOT::ts) k =
      parseExp ts (fn (e, ts') => k(Fun(x,e), ts'))

  | parseExp _ _ = raise ParseError


(* parse : token list -> exp *)

fun parse tokens =
      parseExp tokens (fn (e, nil) => e
                                 | _ => raise ParseError)
```

Direct version of the parser

The continuations are not really doing much, so the direct version looks very similar.

Instead of having values bound to variables as function arguments, one binds them explicitly in **`let`** expressions.

See the code posted online.

That is all.

Please have a good weekend.

See you Tuesday.

We will discuss computability.