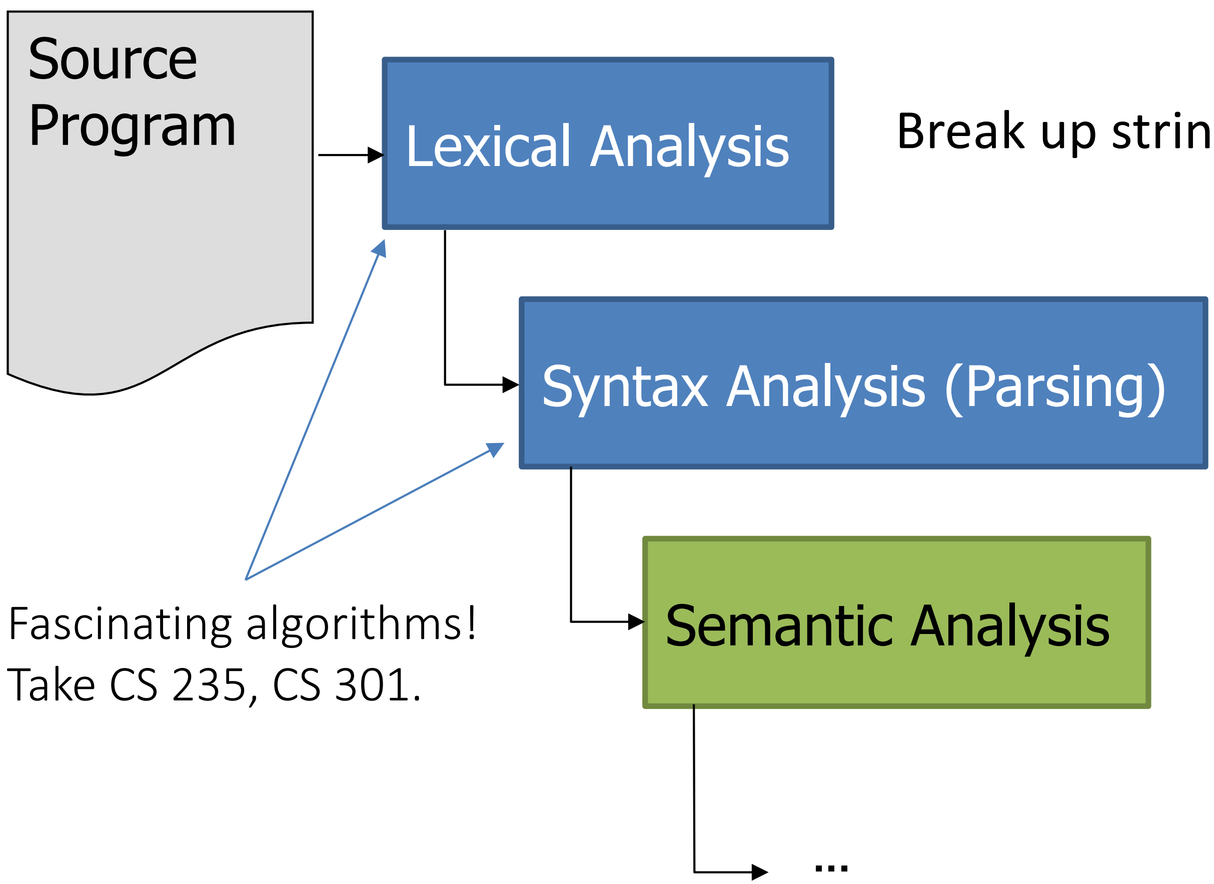
Patterns Everywhere

Reading programs



Break up string input into symbols.

Parse stream of symbols into structured representation of program.

Syntax:

(context-free)

Backus-Naur Form (BNF) notation for grammars

```
<expr> ::= <num>
         | <expr> + <expr>
         | <expr> * <expr>
<num>  ::= 0 | 1 | 2 | ...
```

productions of <expr>

Non-terminals

Terminals
(lexical tokens)

Start symbol: <expr>
designates "root"

```
<expr> ::= <num>
         | <expr> + <expr>
         | <expr> * <expr>
<num>  ::= 0 | 1 | 2 | ...
```

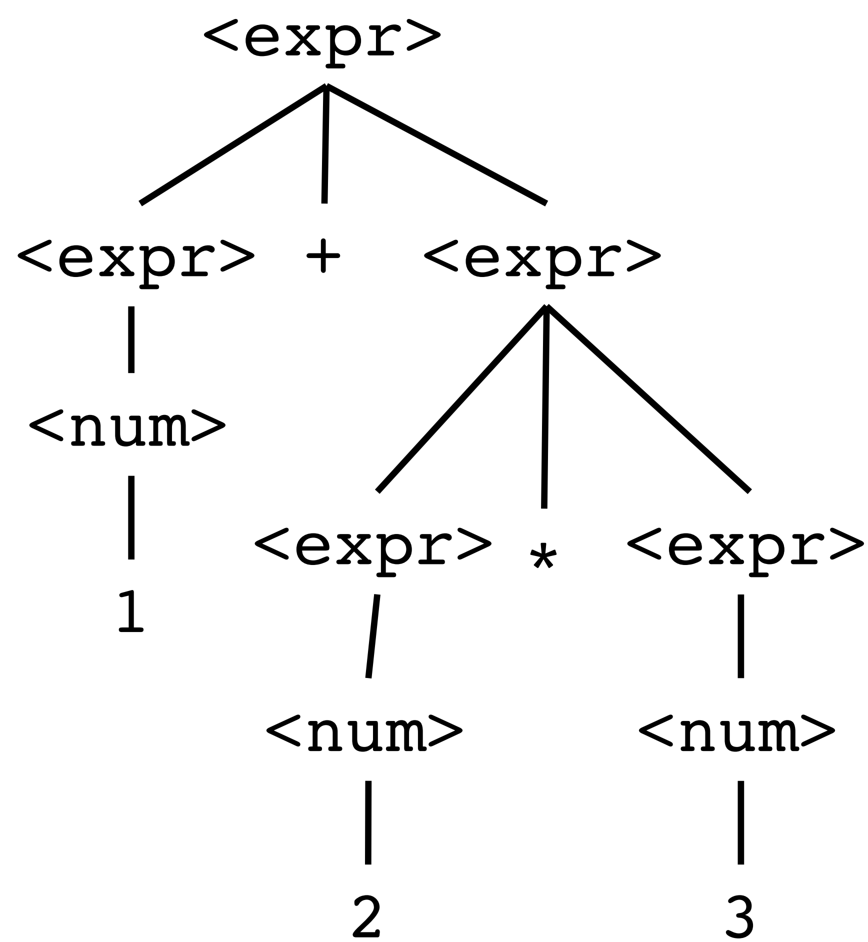
Derivations

```
<expr> → <num>
       → 5
```

```
<expr> → <expr> + <expr>
       → <num> + <expr>
       → 1 + <expr>
       → 1 + <expr> * <expr>
       → 1 + <num> * <expr>
       → 1 + 2 * <expr>
       → 1 + 2 * <num>
       → 1 + 2 * 3
```

(parse tree)

Derivation Tree

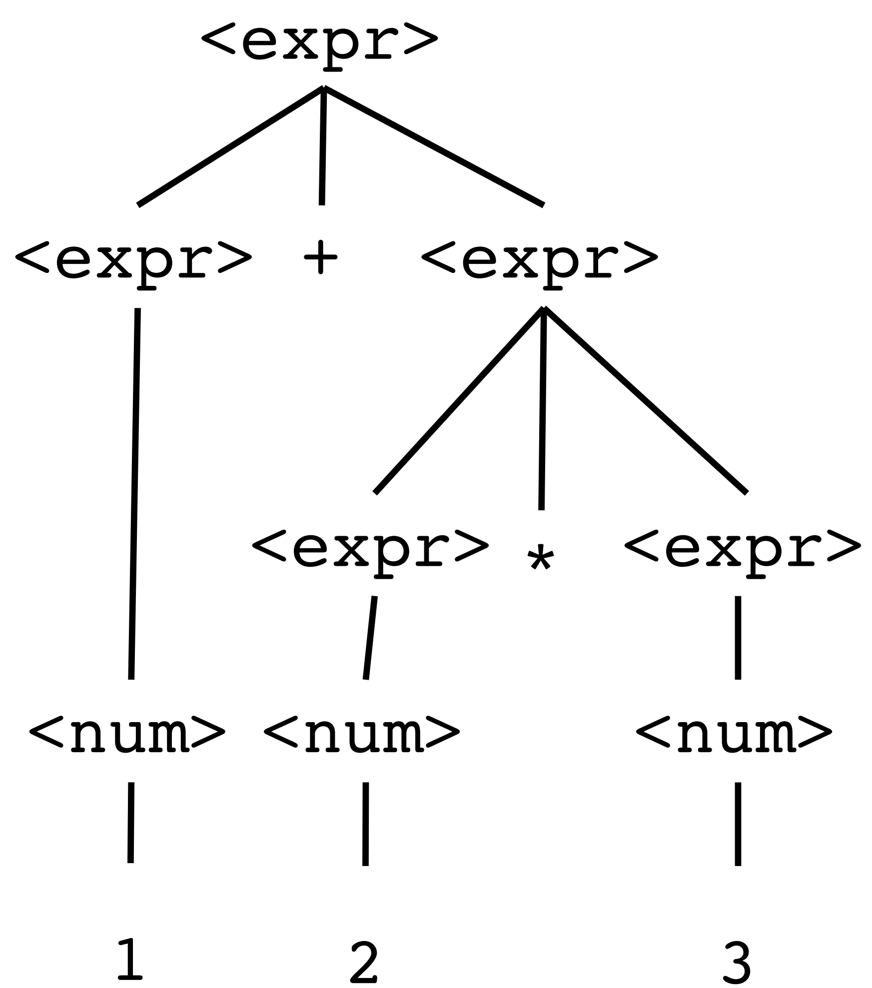


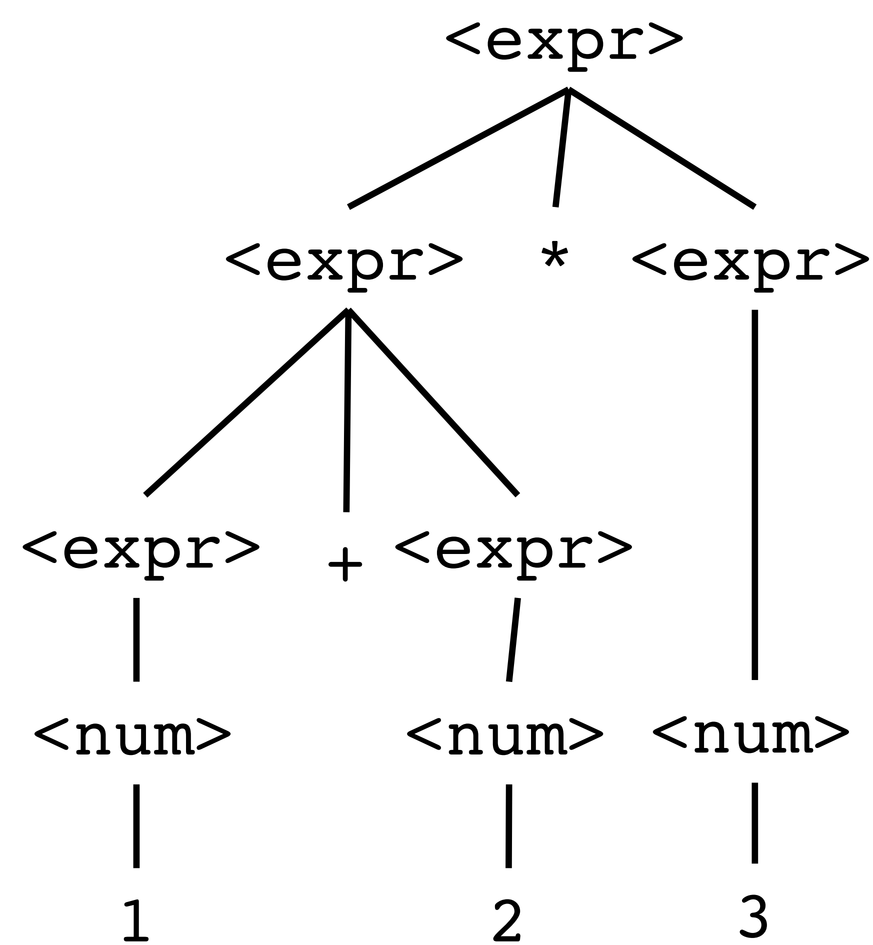
Ambiguity:

>1 derivation of expression

```
<expr> ::=  <num>
         |  <expr> + <expr>
         |  <expr> * <expr>
<num>  ::=  0 | 1 | 2 | ...
```

```
<expr> → <expr> * <expr>
       → <expr> * <num>
       → <expr> * 3
       → <expr> + <expr> * 3
       → <num> + <expr> * 3
       → 1 + <expr> * 3
       → 1 + <num> * 3
       → 1 + 2 * 3
```





Dealing with Ambiguity

Prohibit it.

- Force parenthesization or equivalent.
- Racket, S-expressions:
  - `(there is (always an unambiguous) parse tree)`

Allow it with:

- ***Precedence*** by kind of expression (think *order of operations*)
  - `1 + 2 * 3` means `1 + (2 * 3)`
- Directional ***associativity*** (left, right)
  - left-associative function application: `f 2 3` means `((f 2) 3)`

Representing Abstract Syntax Trees (ASTs)

(or expression trees)

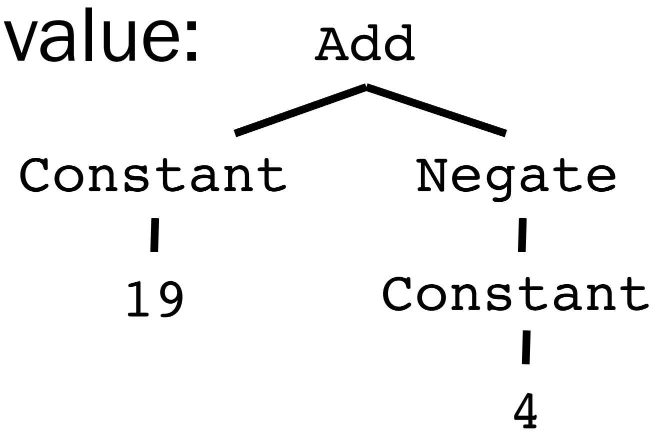
A tiny calculator language:

```
datatype exp = Constant of int
             | Negate   of exp
             | Add      of exp * exp
             | Multiply of exp * exp
```

An ML expression of type `exp`:

```
Add (Constant (10+9), Negate (Constant 4))
```

Structure of resulting value:



Recursive functions for recursive datatypes

Find maximum constant appearing in an `exp`.

```
fun max_constant (e : exp) =
```

Evaluating expressions in the language

Interpreter for tiny calculator language

```
fun eval (e : exp) =
```

Datatype bindings, so far

Syntax:

```
datatype t = C1 of t1 | C2 of t2 | … | Cn of tn
```

Type-checking:

Adds type `t` and constructors `Ci` of type `ti->t` to static environment

Evaluation: nothing!

Omit "`of ti`" for constructors that are just tags, no underlying data

– Such a `Ci` is a value of type `t`

Case expressions, so far

Syntax: `case e of p1 => e1 | p2 => e2 | … | pn => en`

Type-checking:

- Type-check e. Must have same type as all of `p1 ... pn`.
  - Pattern `C(x1,…,xn)` has type `t` if datatype `t` includes a constructor: `C of t1 * ... * tn`
- Type-check each `ei` in current static environment extended with types for any variables bound by `pi`.
  - Pattern `C(x1,…,xn)` gives variables `x1`, …, `xn` types `t1`,…,`tn` if datatype `t` includes a constructor: `C of t1 * ... * tn`
- All `ei` must have the same type `u`, which is the type of the entire case expression.

Case expressions, so far

Syntax: `case e of p1 => e1 | p2 => e2 | … | pn => en`

Evaluation:

- Evaluate `e` to a value `v`
- If `pi` is first *pattern* to *match* `v`, then result is evaluation of `ei` in dynamic environment "extended by the match."
  - Pattern `Ci(x1,…,xn)` matches value `Ci(v1,…,vn)` and extends the environment by binding `x1` to `v1` … `xn` to `vn`
    - For "no data" constructors, pattern `Ci` matches value `Ci`
  - Pattern **`x`** matches and binds to any value of any type.
- Exception if no pattern matches.

Patterns everywhere

Deep truths about ML and patterns.

1. Every `val` / `fun` binding and anonymous `fn` definition uses pattern-matching.

2. Every function in ML takes exactly one argument

First: extend our definition of pattern-matching…

Pattern-match any compound type

Pattern matching also works for records and tuples:

- Pattern `(x1,…,xn)`
  matches any tuple value `(v1,…,vn)`

- Pattern `{f1=x1, …, fn=xn}`
  matches any record value `{f1=v1, …, fn=vn}`
  (and fields can be reordered)

`val` binding patterns

Syntax: a `val` binding can use any pattern `p`, not just a variable

```
val p = e
```

Type checking:

`p` and `e` must have the same type.

Evaluation:

1. Evaluate `e` to value `v`.
2. If `p` matches `v`, then introduce the associated bindings
   Else raise an exception.

Style:

- Get all/some pieces out of a product/each-of type
- Often poor style to use constructor pattern in `val` binding.

Parameter patterns

Rad!!

A function parameter is a pattern.

– Match against the argument in a function call.

```
fun f p = e
```

Examples:

```
fun sum_triple (x, y, z) = x + y + z

fun full_name {first=x, middle=y, last=z} =
    x ^ " " ^ y ^ " " ^ z
```

Convergence!

Takes one `int*int*int` tuple, returns `int` that is their sum:

```
fun sum_triple (x, y, z) = x + y + z
```

Takes three `int` values, returns `int` that is their sum:

```
fun sum_triple (x, y, z) = x + y + z
```

Every ML function takes exactly one argument

"Multi-argument" functions:

- Match a tuple pattern against single argument.
- Elegant, flexible language design

Cute and useful things

```
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left(rotate_left t)
```

“Zero-argument” functions:

- Match the unit pattern `()` against single argument.

Even more pattern-matching

```
fun eval e =
   case e of
        Constant i          => i
      | Negate e2           => ~ (eval e2)
      | Add (e1,e2)         => (eval e1) + (eval e2)
      | Multiply (e1,e2) => (eval e1) * (eval e2)
```

```
fun eval (Constant i)          = i
  | eval (Negate e2)           = ~ (eval e2)
  | eval (Add (e1,e2))         = (eval e1) + (eval e2)
  | eval (Multiply (e1,e2)) = (eval e1) * (eval e2)
```

Critical: added parens around each pattern, replaced => with =.

– If you mix them up, you'll get some weird error messages…

Patterns are deep!

Patterns are recursively structured

- Just like expressions
- Nest as deeply as desired
- Avoid hard-to-read, wordy, nested case expressions

Examples of nested list patterns

Pattern `a::b::c::d` matches
any list with _________ elements

Pattern `a::b::c::[]` matches
any list with _________ elements

Pattern `[a,b,c]` matches
any list with _________ elements

Pattern `((a,b),(c,d))::e` matches
any _________

List checkers (suboptimal style)

```
fun nondec (x::xs) =
      case xs of
          (y::_) => x <= y andalso nondec xs
        | [] => true
  | nondec [] = true

fun nondec [] = true
  | nondec [x] = true
  | nondec (x::xs) =
      let val (y::_) = xs
      in
        x <= y andalso nondec xs
      end
```

List checkers (good style)

```
fun nondec (x::y::zs) = x <= y andalso nondec (y::zs)
  | nondec _ = true

fun allsq (x::y::zs) = x*x = y andalso allsq (y::zs)
  | allsq _ = true

fun checkl (f, x::y::zs) =
        f (x,y) andalso checkl (f, y::zs)
  | checkl _ = true
```

More examples: see code files

Style

Nested patterns: elegant, concise

- Avoid nested case expressions if nested patterns are simpler

  Example: `checkl and friends`

- Common idiom: match against a tuple of datatypes to compare all

  Examples: `zip3` and `multsign`

Wildcards instead of variables when data not needed

- Examples: `len` and `multsign`

(Most of)
The definition of pattern-matching

The semantics for pattern-matching takes a pattern *p* and a value *v* and decides (1) does it match and (2) if so, what variable bindings are introduced.

Definition is **elegantly recursive**, with a separate rule for each kind of pattern. Some of the rules:

- If *p* is a variable *x*, the match succeeds and *x* is bound to *v*
- If *p* is _, the match succeeds and no bindings are introduced
- If *p* is *(p1,…,pn)* and *v* is *(v1,…,vn)*, the match succeeds if and only if *p1* matches *v1*, …, *pn* matches *vn*. The bindings are the union of all bindings from the submatches
- If *p* is *C p1*, the match succeeds if *v* is *C v1* (i.e., the same constructor) and *p1* matches *v1*. The bindings are the bindings from the submatch.
- … (there are several other similar forms of patterns)

Rad!!

```
fun fib n =
    if n = 0 orelse n = 1 then 1
    else (fib (n - 2)) + (fib (n - 1))
```

```
fun fib n =
    case n of
         0 => 1
       | 1 => 1
       | x => (fib (x - 2)) + (fib (x - 1))
```

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = (fib (n - 2)) + (fib (n - 1))
```

`int`uition...

Do you suppose...?

```
datatype int = ... | 0 | 1 | 2 | ...
```

(Efficiency reasons to *implement* `int` specially, but could be a datatype.)

```
datatype nat = Zero | Succ nat

val one = Succ Zero

fun add (Zero,x) = x
  | add (x,Zero) = x
  | add (Succ x,y) = Succ (add (x, y))
```

Rad!!

```
datatype bool = true | false
```

```
if e1 then e2 else e3
```

sugar

```
case e1 of
     true  => e2
   | false => e3
```

poor style

Are you noticing a pattern here?