# Abstract Syntax, Semantic Actions

CS 4100
Gordon Stewart
Ohio University

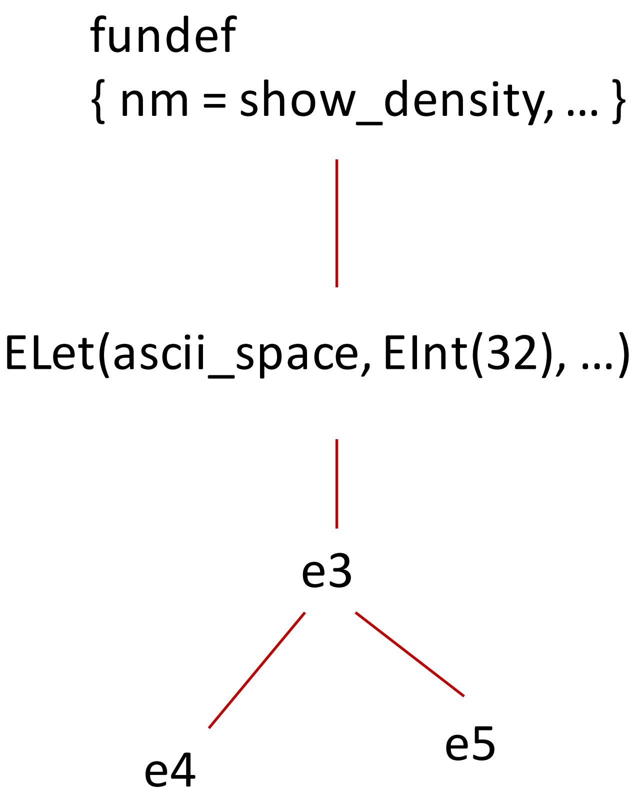
# Abstract Syntax

Lexing and parsing are all about converting ***concrete*** user programs (i.e., the strings of characters in ***.gpy*** files) into

- Streams of ***tokens*** (lexing)
- ***Abstract Syntax Trees (ASTs)*** (parsing)

```
def show_density(i : int) : unit {
  let ascii_space = 32 in
  let ascii_period = 46 in
  let ascii_star = 42 in
  let ascii_plus = 43 in
  let level0 = 2 in
  let level1 = 4 in
  let level2 = 8 in
  if i < level0 then putchar(ascii_star)
  else if i < level1 then putchar(ascii_plus)
       else if i < level2 then
putchar(ascii_period)
            else putchar(ascii_space)
}
```

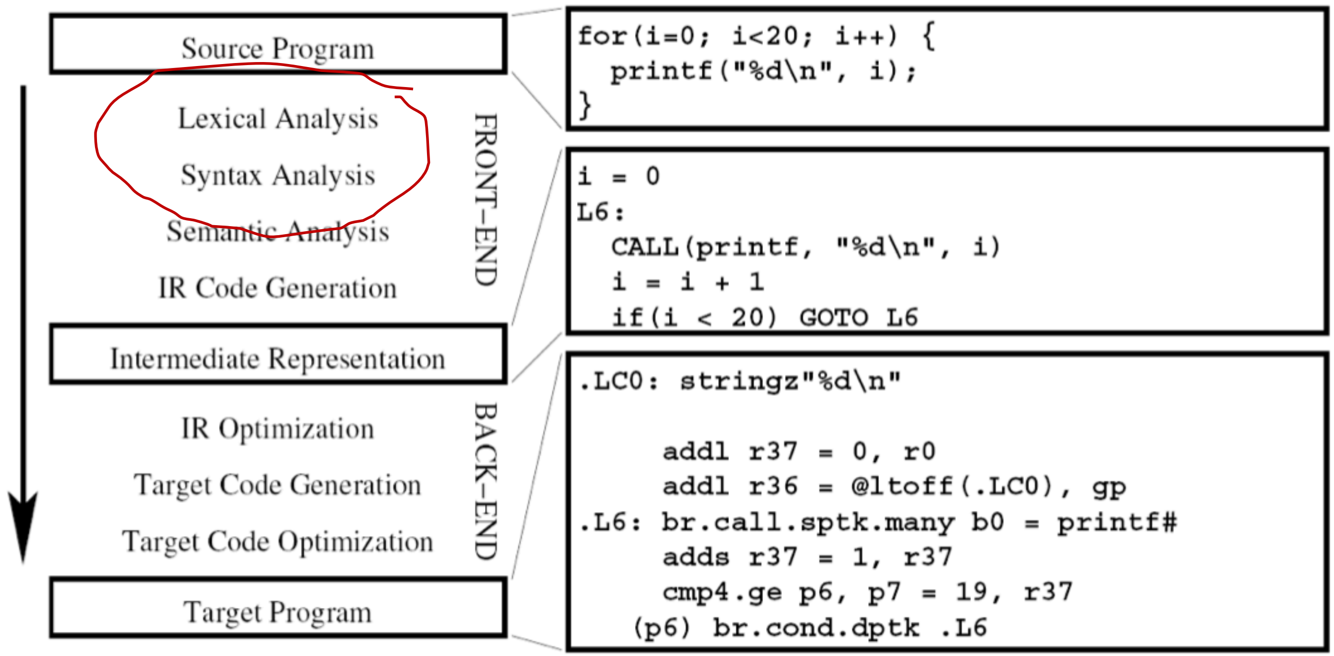
Concrete Syntax (test50-fractal.gpy)




AST

# Why Abstract Syntax?

An entire compiler could be built into the lexical (lexer.mll) and syntactic (parser.mly) analysis phases




But doing so requires that the compiler perform ***semantic actions*** in the order in which the program is parsed

- Compiler must be "one-pass"
- Often difficult to maintain and modify

# Semantic Actions

A fancy term for a simple concept:

***Semantic Action***: A value or effect associated with a production in a context-free grammar

***Example Grammar:***

E -> num

E -> E + E

E -> E * E

***Example Semantic Actions:***

"When we parse a number using ***E -> num***, print the number to ***stdout***."

"When we parse ***E -> E * E***, return the value ***EBinop(BTimes, e1, e2)***"

# Abstract Syntax Example: ***cs4100-public/calc***

```
type binop = BPlus | BMinus | BTimes | BDiv

type exp =
  | EInt of Int32.t
  | EBinop of binop * exp * exp
```

exp.mli

# Abstract Syntax Example: *grumpy/src/exp.mli*

```
type 'a raw_exp =
  | EInt of int32                    (** 32-bit integers *)
  | EFloat of float                  (** Double-precision floats *)
  | EId of id                        (** Program identifiers [x, y, z, ...] *)
  | ESeq of ('a exp) list            (** [e1; e2; ...; eN] *)
  | ECall of id * ('a exp) list      (** [f(e1, e2, ..., eM)] *)
  | ERef of 'a exp                   (** Allocate a reference cell *)
  | EUnop of unop * 'a exp                  (** Apply a unary operation, e.g. [-e] *)
  | EBinop of binop * 'a exp * 'a exp  (** Apply a binary operation e.g., [e1+e2] *)
  | EIf of 'a exp * 'a exp * 'a exp     (** Conditional [if e1 then e2 else e3] *)
  | ELet of id * 'a exp * 'a exp        (** [let x = e1 in e2] *)
  | EScope of 'a exp                    (** \{ e \} *)
  | EUnit                               (** The unit value () *)
  | ETrue                               (** true *)
  | EFalse                              (** false *)
  | EWhile of 'a exp * 'a exp           (** while e1 \{ e2 \} *)
and 'a exp =
  { start_of : Lexing.position;        (** The source-file startpos of this [exp] *)
    end_of : Lexing.position;          (** The source-file endpos of this [exp] *)
    exp_of : 'a raw_exp;               (** The [exp] itself *)
    ety_of : 'a }                      (** The "extra" data, typically a type *)
```

# Incorporating Actions into Syntactic Analysis

Exactly how depends on the kind of parser you're building

***Recursive Descent:***

```
https://github.com/gstew5/cs4100-
public/recursive/grammar311Semantic.ml
```

***LR(1)/Menhir:***

```
https://github.com/gstew5/cs4100-public/calc-example/parser.mly
```

# calc-example: Binary Operators

binop -> PLUS | MINUS | TIMES | DIV

Grammar

```
type binop =
   BPlus
 | BMinus
 | BTimes
 | BDiv
```

Abstract Syntax

```
%inline binop:
| PLUS
  { BPlus }
| MINUS
  { BMinus }
| TIMES
  { BTimes }
| DIV
  { BDiv }
```

Semantic Action:

On token **PLUS**, return abstract syntax expression **BPlus**

Menhir CFG w/ Semantic Actions