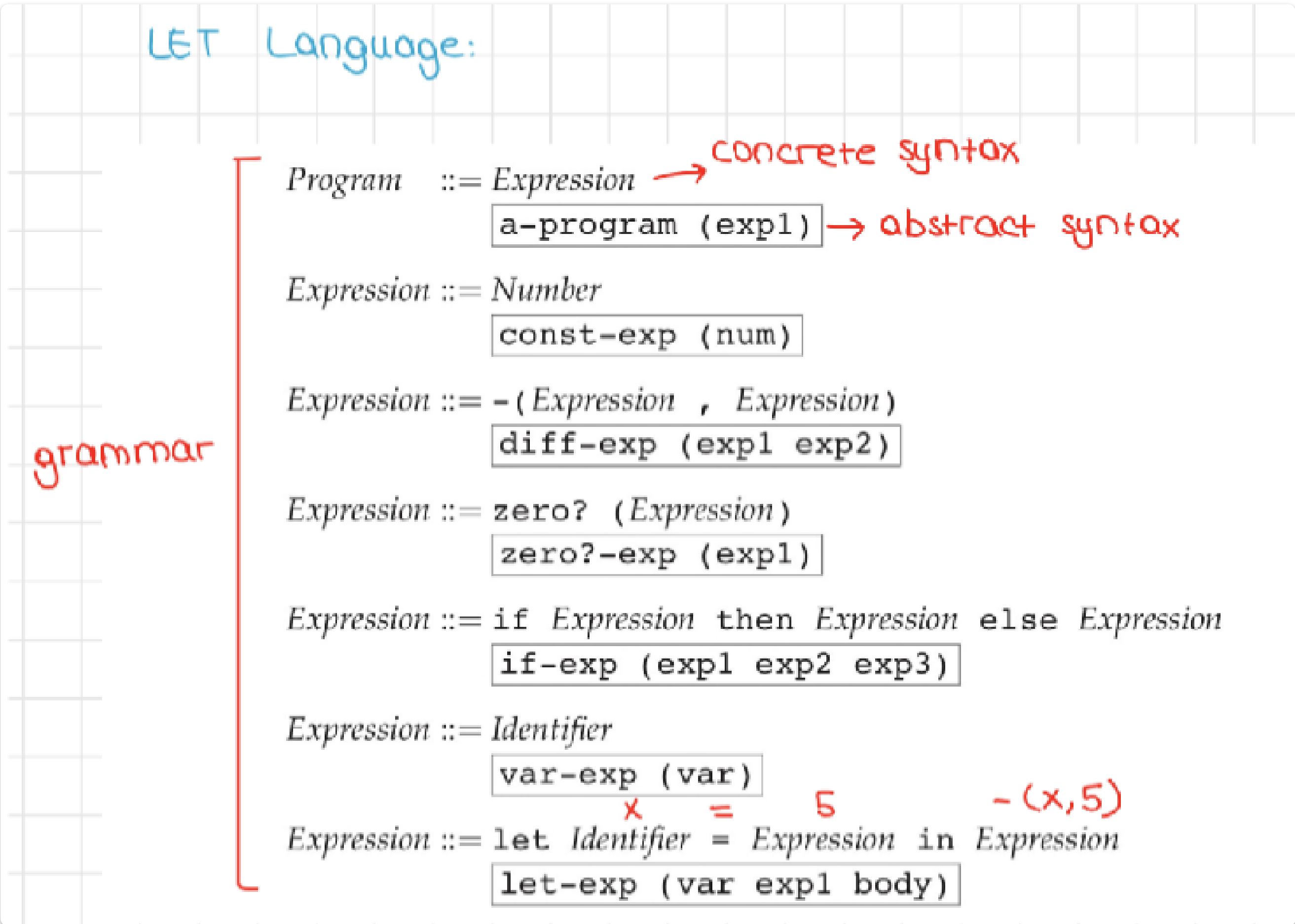
11. LET Behavior Specification

Input:
"-(55, -(x, 11))"
(scan&parse "-(55, -(x,11))")

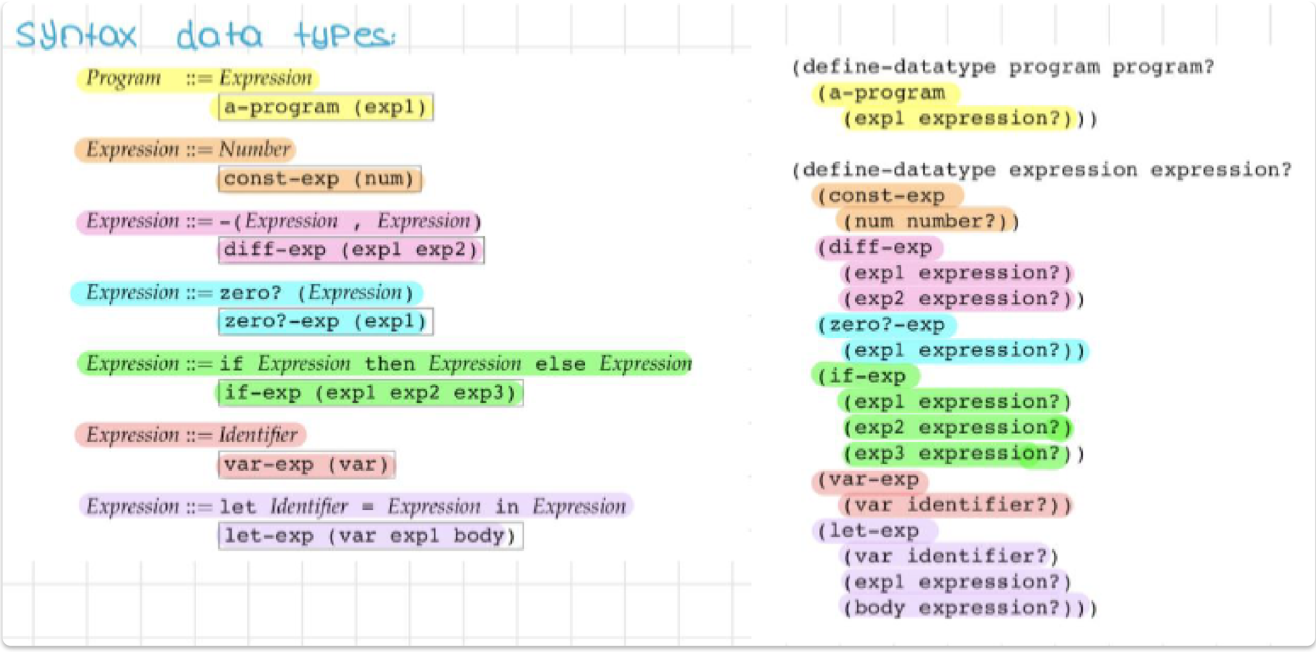
Components of the language

- Syntax and datatypes
- Values
- Environment
- Behavior specification
- Behavior implementation
  - Scanning
  - Parsing
  - Evaluation

Grammar



How do we define grammar structures?



Expressed Values: possible values of expressions: (define x (+ 2 (* 4 5)))
Denoted Values: possible values of variables, are usually the fundamental values in the language, such as strings, booleans, numbers etc.

Expressions:
const-exp: Int -> Exp
zero?-exp: Exp -> Exp
if-exp?: Exp x Exp x Exp -> Exp
diff-exp: Exp x Exp -> Exp
var-exp: Var -> Exp
let-exp -> Var x Exp x Exp x -> Exp

Observer:
value-of : Exp x Env -> ExpVal

Example:
(value-of (const-exp n) p) = (num-val n)
(value-of (var-exp var) p) = (apply-env p var)

(value-of (diff-exp exp1 exp2) p) =
(num-val
(-
(expval->num (value-of exp1 p))
(expvap-num (value-of exp2 p))
)
)

Project2 MyLet

`data-structures.rkt`

Key Components of `data-structures.rkt`

1. Expressed Values: the `expval` datatype represents values that the MyLet language can express. These are the types of values that can be the results of evluating expressions in MyLet.
   `num-val`: Represents numeric values.
   `bool-val`: Represents boolean values.
2. Extractors: Extractor functions are used to retrieve the actual value from an `expval`.
   `expval->num`: Extracts a number from a `num-val` expval.
   `expval->bool`: Extracts a boolean from a `bool-val` expval.
   `expval-extractor-error`: A helper function that throws an error when the wrong type of expval is accessed.
3. Environment Structures: Environment is a function that maps variables to their saved values, examples: 8. Interfaces & Representation. In this implementation, environment is a list of pairs, where each pair consists of a symbol (variable name) and its corresponding value (an expval).
   `empty-env-record`: Creates an empty environment.
   `extended-env-record`: Extends an existing environment with a new symbol-value pair.
   `empty-env-record?`: Checks if an environment is empty.
   `environment?`: Checks if a given structure is a valid environment.
   `extended-env-record->sym`, `extended-env-record->val`, `extended-env-record->old-env`: These functions extract the symbol, value, and the old environment from an extended environment record.

This file forms the backbone of the language's runtime, as it defines how values and environments are represented and manipulated.

`lang.rkt`

Key Components of `lang.rkt`

1. Lexical Specification (`the-lexical-spec`): This part defines the tokens that the language recognizes. Tokens are the smallest units in the language, like keywords, numbers, identifiers, etc.
   - Whitespace and Comments: It's configured to skip whitespace and comments (which start with `%`).
   - Identifiers: Defines the pattern for identifiers (variable names, function names, etc.). They must start with a letter and can include letters, digits, and specific special characters.
   - Numbers: Defines the pattern for recognizing both positive and negative numbers.

2. Grammar Specification (`the-grammar`): This outlines the syntactic structure of the language. It defines how tokens can be combined to form valid expressions in the language.
   - Program: The top-level structure, which consists of an `expression`.
   - Expression: Defined as different types. For instance:
     - A `number` is an expression (`const-exp`).
     - A `zero?` expression checks if an expression evaluates to zero.
     - A `let` expression for variable binding (typical of let languages).
3. SLLGEN Boilerplate:
   - `sllgen:make-define-datatypes`, `show-the-datatypes`, `scan&parse`, and `just-scan` are part of the SLLGEN library, which is used for generating scanners and parsers based on the lexical and grammar specifications. This library helps in automatically creating the necessary infrastructure to parse the language based on the rules defined.

Understanding the Role of `lang.rkt`

- The lexical specification (`the-lexical-spec`) tells the language how to break the input code into meaningful tokens.
- The grammar specification (`the-grammar`) tells the language how to interpret sequences of these tokens as valid expressions or constructs in your MYLET language.

Usage in the Language Pipeline

In the overall pipeline of the language's implementation:

- `lang.rkt` defines what is syntactically valid in the language.
- Other components like `interp.rkt` would use these definitions to parse and execute the code.

`environments.rkt`

Key components of `environments.rkt`

Environment Constructors and Observers:

`empty-env`: Creates an empty environment.
`empty-env?`: Checks if a given environment is empty.
`extend-env`: Takes a symbol, a value, and an old environment, and extends the old environment with the new symbol-value pair. This is how new bindings are added to the environment.
`apply-env`: This is a lookup function. Given an environment and a symbol, it searches for the value bound to that symbol in the environment. If the symbol is found, it returns its value; if not, it throws an error.

How does `apply-env` work?

- It first checks if the environment is empty. If it is, it means the symbol isn't bound in this environment, and an error is raised.
- If the environment isn't empty, it checks the first binding.
- If the symbol in the current binding matches the searched symbol, it returns the associated value.
- If not, it recurses into the "old environment" (the environment before the current binding was added).

Structure

```
environment: (symbol, value, old-env)
>  (define e (empty-env))
>> ('empty-env)

>  (extend-env 'z (num-val 30) e)
>> ('z 30 (empty-env))

>  (extend-env 'y (num-val 20) e)
>> ('y 20 ('z 30 (empty-env)))

>  (extend-env 'x (num-val 10) e)
>> ('x 10 ('y 20 ('z 30 (empty-env))))

>  (extend-env 'v (num-val 3) e)
>> ('v 3 ('x 10 ('y 20 ('z 30 (empty-env)))))
```

`interp.rkt`

Key Components of `interp.rkt`

Value of a Program (`value-of-program`):

Takes a `program` as an input and calls `value-of` in the program's expression with the initial environment created by `init-env` The datatype " `program` " is generated in the sllgen boilerplate of `lang.rkt`

```
((define-datatype program program? (a-program (a-program7 expression?)))
 (define-datatype
  expression
  expression?
  (const-exp (const-exp8 number?))
  (zero?-exp (zero?-exp9 expression?))
  (let-exp (let-exp10 symbol?) (let-exp11 expression?) (let-exp12 expression?)))
```

Runner function (`run`)

Calls `(value-of-program (scan&parse "string"))`

Value of an Expression (`value-of`):
This function is the heart of the interpreter. It evaluates expressions based on their type. Uses the `cases` construct to pattern-match against different expression types.

Handling Different Expression Types:
Constant Expressions (`const-exp`): Simply wraps a number into a `num-val` expressed value.
Variable Expressions (`var-exp`): Looks up the value of a variable in the given environment using `apply-env`.
Operation Expressions (`op-exp`): Evaluates expressions that involve operations like addition or multiplication. This includes handling different combinations of number and rational types.
Zero? Expressions (`zero?-exp`): Checks if an expression evaluates to zero, returning a boolean value accordingly.
Let Expressions (`let-exp`): Evaluates the body of a let expression in an environment extended with the new binding.

Functionality of `interp.rkt`

The interpreter is essentially a function that takes an expression and an environment, and recursively evaluates the expression based on its type. This evaluation is context-sensitive, meaning it respects the bindings in the current environment. It's a direct implementation of the operational semantics of the MyLet language.

Example: Evaluating an Expression

```
(let x = 5 in (add x 3))
```

The `value-of` function would:

- Evaluate `5` as a `const-exp`, getting `num-val 5`.
- Extend the environment to bind `x` to `num-val 5`.
- Evaluate `(add x 3)` where `x` is replaced by its value from the environment.