Week 1: External DSLs

April 3, 2025

Today

External DSLs

- An "external" DSL is implemented as a complete programming language, with its own syntax and semantics.
  - Allows non-standard, specialized syntax
- Although they are not general purpose, they can implement programming constructs found in general purpose languages:
  - variables (common)
  - functions (occasionally)
  - control flow (if, while, etc.) (occasionally)
- They should have concise syntax for their particular domain

External DSLs: CSS

```css
body {
    overflow: hidden;
    background-color: #000000;
    background-image: url(images/bg.gif);
    background-repeat: no-repeat;
    background-position: left top;
}
```

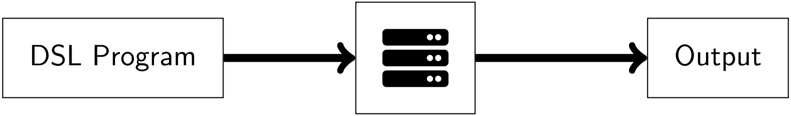
External DSLs: NetLogo

```
to setup
  clear-all
  create-turtles 10
  reset-ticks
end

to go
  ask turtles [
    fd 1              ;; forward 1 step
    rt random 10      ;; turn right
    lt random 10      ;; turn left
  ]
  tick
end
```

External DSLs: Makefile

```
main:
  latexmk main

clean:
  latexmk -C main.tex
  rm -rf project.bbl
```

Goal



Writing an External DSL

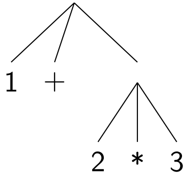
1. Parse: analyze the text and determine its gramatical structure
2. Translate: convert the parse tree into an Abstract Syntax Tree (AST) or other intermediate representation
3. Execute: “run" the program (produce some output, interact with the user, etc.)

Parsing

- Parsing reads in a string, and determines how it matches the **grammar** of a language
- Outputs a **parse tree**: a tree representation of the string
- Checks **syntax**: is the string a correctly structured statement in the language

Parsing

1 + 2 * 3 $\Rightarrow$



Parsing: Tokens and Patterns

Token:

- A string of characters with a label.

Parsing: Tokens and Patterns

Token:

- A string of characters with a label.
- Can be represented as a pair of the token label (or *kind*) and the string.
  - (NUMBER, "1"), (NUMBER, "23")
  - (IF, "if")
  - (IDENTIFIER, "x"), (IDENTIFIER, "y")

Parsing: Tokens and Patterns

Pattern:

- Describes the strings that match a kind of token.

Parsing: Tokens and Patterns

Pattern:

- Describes the strings that match a kind of token.
- For example, using Python Regular Expressions:
  - NUMBER = [0-9]+
    - matches any number
  - IF = "if"
    - only matches "if"
  - IDENTIFIER = [a-zA-Z_][a-zA-Z0-9_]*
    - matches an identifier which starts with a letter

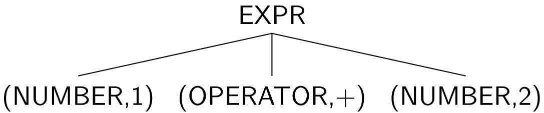
Parsing: Parse Tree

Parse Tree:

- A tree which describes the gramatical structure of an input string
- What does that mean?
  - Each node in the tree represents a rule
  - Each leaf node represents a token
  - The root node represents the entire input string
- The rules which define the tokens and parse tree are called a **grammar**.

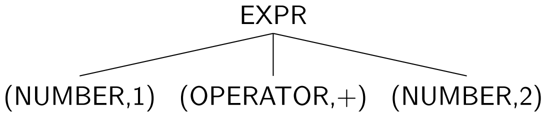
Parsing: Parse Tree

Example Parse Tree Rules:

- Token Patterns:
  - NUMBER := [0-9]+
  - OPERATOR := [+-*/]
- Rules:
  - EXPR := NUMBER OPERATOR NUMBER
    - Means "first a number, then an operator, then a number"
- Example Parse Tree for "1+2":
  - 

Parsing: Parse Tree

Example Parse Tree Rules:

- Token Patterns:
  - NUMBER := [0-9]+
  - OPERATOR := [+-*/]
- Rules:
  - EXPR := NUMBER OPERATOR NUMBER
    - Means "first a number, then an operator, then a number"
- Example Parse Tree for "1+2":
  - 

- Notice that some strings (e.g. "abcd") don't have a valid parse tree.
  - We say a string *matches* a rule if the parser can produce a parse tree for it

Writing a Parser

- We have been seen how to define *patterns* for tokens, and *rules* for interior nodes
- But, we really need to write **code** which takes a string as input, and outputs a parse tree...
- Luckily, there are libraries called *Parser Generators* which take the patterns and rules, and produce parsing code for you!

Parsimonious

For the first assignment, we will use a parser generator for Python called **Parsimonious**

- Parsimonious takes in a description of your grammar in a language called Parsing Expression Grammar (PEG)
- Then, it can parse any input string, or return an error.

Parsing Expression Grammar (PEG)

- PEG is language used to specify the grammar of another language (PEG is a DSL!)

Parsing Expression Grammar (PEG)

- PEG is language used to specify the grammar of another language (PEG is a DSL!)
- PEG consists of a sequence of *rules*
  - `identifier = expression`

Parsing Expression Grammar (PEG)

- PEG is language used to specify the grammar of another language (PEG is a DSL!)
- PEG consists of a sequence of *rules*
  - `identifier = expression`
- Expressions can be patterns for tokens
  - `one = "1"`
  - `plus = "+"`

Parsing Expression Grammar (PEG)

- PEG is language used to specify the grammar of another language (PEG is a DSL!)
- PEG consists of a sequence of *rules*
  - `identifier = expression`
- Expressions can be patterns for tokens
  - `one = "1"`
  - `plus = "+"`
- ...or can be grammar rules
  - `one_plus_one = one plus one`

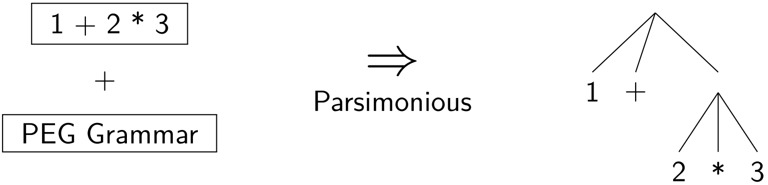
Parsing Expression Grammar (PEG)

- PEG is language used to specify the grammar of another language (PEG is a DSL!)
- PEG consists of a sequence of *rules*
  - `identifier = expression`
- Expressions can be patterns for tokens
  - `one = "1"`
  - `plus = "+"`
- ...or can be grammar rules
  - `one_plus_one = one plus one`
- We say a string *matches* a rule if the parser can produce a parse tree for it
  - "1+1" matches `one_plus_one`

Parsing Expression Grammar (PEG)

- PEG is language used to specify the grammar of another language (PEG is a DSL!)
- PEG consists of a sequence of *rules*
  - `identifier = expression`
- Expressions can be patterns for tokens
  - `one = "1"`
  - `plus = "+"`
- ...or can be grammar rules
  - `one_plus_one = one plus one`
- We say a string *matches* a rule if the parser can produce a parse tree for it
  - "1+1" matches `one_plus_one`
- The first rule is the "starting expression", and is used to match the entire text.

Parsing with PEG



Parsimonious Token Patterns

Type	PEG	PEG Example	Matches
Literal	“ ”	`"abc"`	abc
Python-style Regex	~r“regex”	`~r"[a-z]"`	foobar

Parsimonious Grammar Rules

Let e_1 and e_2 be arbitrary expressions

Type	PEG	PEG Example	Matches
Sequence	$e_1\ e_2$	`"1" "2"`	12
Choice	$e_1 \ / \ e_2$	`"1" / "2"`	2
Grouping	(e_1)	`("1" / "2") "1"`	21
Optional	$e_1?$	`"1"?`	
Zero-or-more	e_1*	`"1"*`	111
One-or-more	e_1+	`"1"+`	1111
Exactly-n	$e_1\{n\}$	`"1"{2}`	11

In Parsimonious, each expression in a rule creates a new parse tree node.

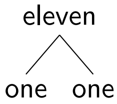
Parsimonious Parse Trees

Example Grammar:

```
eleven = one one
one = "1"
```

Example: Parsing 11

11 $\Rightarrow$



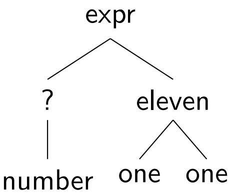
Parsimonious Parse Trees

Example Grammar:

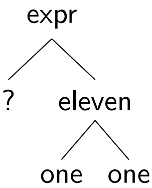
```
expr = number? eleven
number = "\#"
eleven = one one
one = "1"
```

Example: Parsing #11

#11 $\Rightarrow$



11 $\Rightarrow$



Recursion

- Rules may be recursive, meaning they reference themselves within their definitions
  - Example: `ones = one ones?`
- However, PEG does NOT allow the left-most expression in a sequence to be recursive (e.g. no left recursion)
  - Example: `ones = ones one` is NOT allowed

Live Coding: Arithmetic Parsing

Abstract Syntax Trees (ASTs)

- Parse trees are not nice to work with:
  1. they contain many useless nodes (e.g. whitespace)
  2. may not be the exact structure you want
- Instead, we convert the parse tree into an Abstract Syntax Tree (AST)
- AST: a tree where interior nodes represent operators, and their children represent their operands

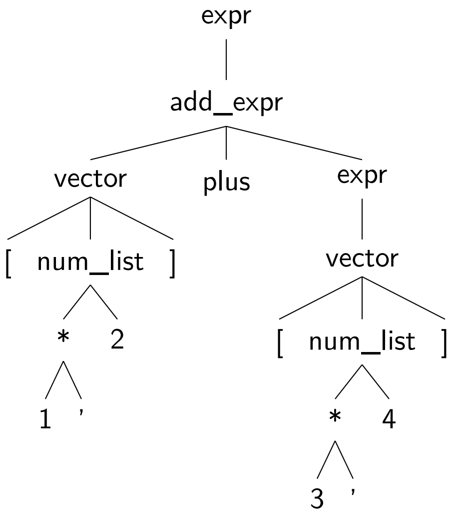
Example: Vector Addition

Example: [1, 2] + [3, 4]

```
expr = add_expr / vector
add_expr = vector plus expr
vector = "[" num_list "]"
num_list = (number comma)* number
number = ~r"[0-9]+" ws
comma = "," ws
ws = ~r"\s*"
```

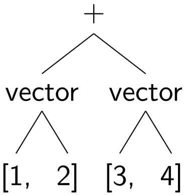
Example: Vector Addition

Example: [1, 2] + [3, 4]



Example: AST

Example: $[1, 2] + [3, 4]$



Converting Parse Trees to ASTs in Parsimonious

- General idea: perform a depth first traversal of the parse tree and convert each node into AST nodes
- Parsimonious steps:
  1. Sub-class the `NodeVistor` class
  2. Implement visitor methods for each grammar rule (corresponding to each kind of interior node in the parse tree)
  3. Call `visit` on the parse tree

```
class VectorVisitor(NodeVisitor):
    def visit_expr(self, node: Node, visited_children: list[Any]):
        ...
```

Converting Parse Trees to ASTs in Parsimonious

- General idea: perform a depth first traversal of the parse tree and convert each node into AST nodes
- Parsimonious steps:
  1. Sub-class the `NodeVistor` class
  2. Implement visitor methods for each grammar rule (corresponding to each kind of interior node in the parse tree)
  3. Call `visit` on the parse tree

```
class VectorVisitor(NodeVisitor):
    def visit_expr(self, node: Node, visited_children: list[Any]):
        ...
```

Node object for the parse tree node

Converting Parse Trees to ASTs in Parsimonious

- General idea: perform a depth first traversal of the parse tree and convert each node into AST nodes
- Parsimonious steps:
  1. Sub-class the `NodeVistor` class
  2. Implement visitor methods for each grammar rule (corresponding to each kind of interior node in the parse tree)
  3. Call `visit` on the parse tree

```
class VectorVisitor(NodeVisitor):
    def visit_expr(self, node: Node, visited_children: list[Any]):
        ...
```

List of results from visiting this nodes children

ASTs: What now

Now we have an AST... but what can we do with it?

1. Analyze and/or optimize it...
2. Translate it into a different AST / IR...
3. Execute it...

Execution

There are three main ways to execute a DSL:

1. Compilation: Convert the AST into machine code, which can be executed
2. Transpilation: Convert the AST into an equivalent program in a different language (e.g. C)
3. Interpretation: Write a program which executes over the AST directly

Note that we mean execution in a broad sense (e.g. producing an output, interacting with the user, etc.)

Execution

There are three main ways to execute a DSL:

1. Compilation: Convert the AST into machine code, which can be executed
2. Transpilation: Convert the AST into an equivalent program in a different language (e.g. C)
3. Interpretation: Write a program which executes over the AST directly

Note that we mean execution in a broad sense (e.g. producing an output, interacting with the user, etc.)

Why Interpreters

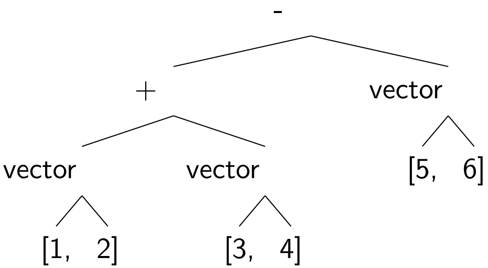
- Fairly straightfoward to write (in comparison to a compiler or transpiler)
- Often easier to debug
- Many DSLs aren't performance critical
- Can use features of the "host" language (e.g. memory management)

Writing a Tree-Walking Interpreter

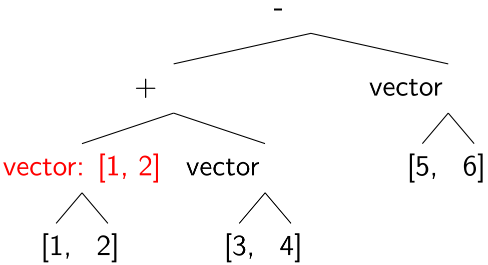
Tree-Walking Intrepreter: Traverse the AST, executing as you go.

- Perform some depth-first traversal of the AST
- When visiting a node, perform the correct computation using its computed children

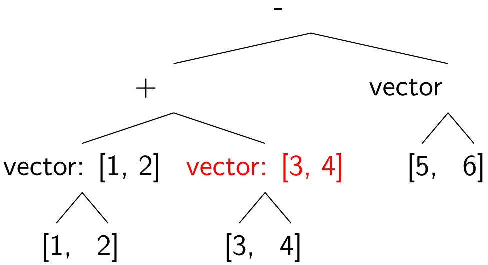
Example: [1, 2] + [3, 4] - [5, 6]



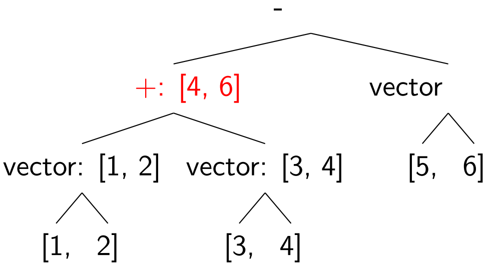
Example: [1, 2] + [3, 4] - [5, 6]



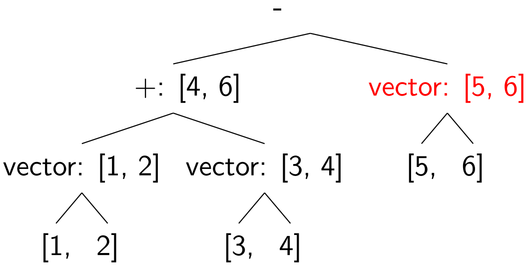
Example: [1, 2] + [3, 4] - [5, 6]



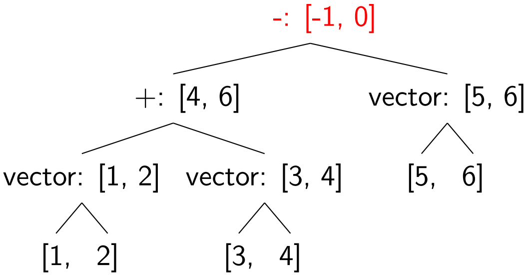
Example: [1, 2] + [3, 4] - [5, 6]



Example: [1, 2] + [3, 4] - [5, 6]



Example: [1, 2] + [3, 4] - [5, 6]



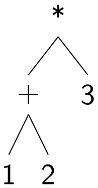
AST Design Decisions: Precedence and Associativity

- **Precedence**: the order in which different operations are executed in an expresion (like PEMDAS in math)
- **Associativity**: the order in which operators of the *same* precedence are executed

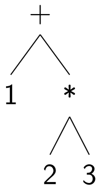
Precedence

Example: 1 + 2 * 3

(1 + 2) * 3



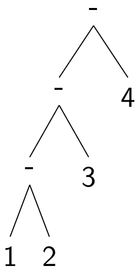
1 + (2 * 3)



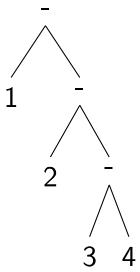
Associativity

Example: 1 - 2 - 3 - 4

(((1 - 2) - 3) - 4)



(1 - (2 - (3 - 4)))



Parse Tree and AST Design Decisions

- Use semantics to guide your parsing and AST (e.g. don't want a right-leaning parse tree for left-associative operations)
  - Stage 1: Design the AST from the semantics
  - Stage 2: Design the parser from the AST
- Think about whether or not evaluation ordering is defined: (e.g. foo(print(1), print(2)))
- Keep it lean: don't implement constructs that aren't necessary for your domain

Live Coding: AST and Evaluating Arithmetic

Expressions vs Statements

Many languages differentiate between *expressions*, pieces of code which return a value, and *statements*, pieces of code which do not.

For example, in python:

- `x = 5` is a statement
  - $y = (x = 5) + 2$?
- `5 + 2` is an expression

In many languages, all expressions are statements, but not all statements are expressions.

Variables

Example: `let x = 5`

Use a dictionary to track “bindings”:

```
class Let(Stmt):
    name: str
    value: expr

class Variable(Stmt):
    name: str

```

```
def interpret_let(ast_node, bindings):
    result = interpret(node.value)
    bindings[ast_node.name] = result

def interpret_var(ast_node, bindings):
    return bindings[ast_node.name]

```

Function Declarations

Example:

```
func foo(arg1, arg2, arg3) {
    body
    return arg1;
}

```

Implementation:

```
class Function(Stmt):
    name: str
    params: list[str]
    body: list[Stmt]

```

```
def interpret_func_declaration(ast_node, bindings,
    declarations):
    declarations[ast_node.name] = ast_node

```

Function Calls

Example:

```
foo(1, 2, 3)
```

Implementation:

```
class FunctionCall(Expr):
    name: str
    args: list[Expr]
```

```
def interpret_func_call(ast_node, bindings,
                        declarations):
    func = declarations[ast_node.name]

    for (param_name, arg) in
            zip(func.params, ast_node.args):
        arg_value = interpret(arg, bindings,
                              declarations)
        bindings[param_name] = arg_value

    for stmt in func.body:
        interpret(stmt, bindings, declarations)
```

Control Flow

```
    if (x == 5) {
        ...
    } else {
        ...
    }

```

```
class If(Stmt):
    condition: Expr
    true_block: list[Stmt]
    false_block: list[Stmt]

```

```
def interpret_if(ast_node, bindings, declarations):
    cond_value = interpret(ast_node.condition, ...)
    if cond_value:
        for stmt in ast_node.true_block:
            interpret(stmt, ...)
    else:
        for stmt in ast_node.false_block:
            interpret(stmt, ...)

```

Program Correctness

- Some programs may not be correct...
- Some errors can be found before running the program (i.e. statically), but others can only be caught during execution (i.e. dynamically)
- We have already seen how parsing can catch some errors:
  - `4 & 8 ( 0`
- But some errors can't be caught by the parser...
  - `let for = 5;`

Turtle DSL

- Let

```
x = 5;
y = "circle";
t = turtle;

```

- Ask

```
ask t {
    shape = y;
    color = "red";
}

```

- ontick

```
ontick t {
    forward(x);
    right(random(50));
}

```

Turtle DSL: Error

- Let

```
x = 5;
y = "circle";
t = turtle;

```

- Ask

```
ask t {
    color = 5; # Error! 5 is not a color!
}

```

Static vs Dynamic error checking

In general, catching errors statically is prefered to catching them dynamically. Why? Consider the following code:

```
for (int i = 0; i < 1,000,000; i++) {
    ... long running code ...
}

int x = "hello";
```

…but sometimes Dyanmic is better

- Sometimes, static isn't possible: we need the actual value to find the error
  - `5 / x  # if x is 0, need to throw an error`
- Sometimes, static is possible, but it is really hard...

```
if (b):
    x = 5;
else:
    x = "hello";

match x:
    case int():
        ...
    case float():
        ...

```

- Communication to the programmer: At runtime, we have concrete values we can give to the programmer!

Typing

A common type of error checking is called *typing*.

Types are *sets of values*, which give information about what operations are permitted on those values.

For example, we might use the type *int* for integers, or the type *Function(int, int)* $\rightarrow$ *int* for functions which take two integers, and return an integer.

A simple type system

Lets consider a small language, with numbers and strings.

```
let x = 5;
let y = "hello";
let z = x * 5 + 3;
```

Type checking

What should the following code do?

```
let x = 5;
let y = "hello";
print(x + y)
```

Type checking

What should the following code do?

```
let x = 5;
let y = "hello";
print(x + y)
```

Some options:

- Define addition over combinations of integers and strings
- Throw an error at
  - compile-time
  - run-time

Static vs Dynamic Typing

- Static Typing: Types are known and checked at compile-time
  - C, C++, Rust, Haskell...
- Dynamic Typing: Types are known and checked at run-time.
  - Python, Javascript...

Static vs Dynamic Typing Advantages

- Static Typing:
  - Checks are done at compile time (no need to run the code)
- Dynamic Typing:
  - More flexible (e.g. python functions can automatically accept any argument, duck typing, etc.)

Implementing a type checker

Very basic type checker: Traverse the AST, and check that the types of function/operator arguments match.

Type checking function calls

```
def add(x: int, y: int) -> int { ... }

add(5, 6)

```

Type checking function calls

Type checking function calls

add: Function(int, int) -> int

5: int 6: int

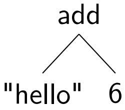
Type checking function calls

add: Function(int, int) -> int

Does 5.ty == int and 6.ty == int?

5: int   6: int

Type checking function calls



Type checking function calls

add: Function(int, int) -> int

"hello": string   6: int

Type checking function calls

add: Function(int, int) -> int

Does "hello".ty == int and 6.ty == int?

"hello": string   6: int

Live Coding: A turtle type-checker

We will live code a type checker for a small turtle language (similar to Logo).