Week 1: External DSLs

April 3, 2024

Today

What are Domain Specific Languages (DSLs)?

Course Overview

External DSLs

- Parsing
  - Parsing Expression Grammar (PEG)
- Abstract Syntax Trees (ASTs)
- Execution
- Implementing common constructs
- Program Correctness
- Typing

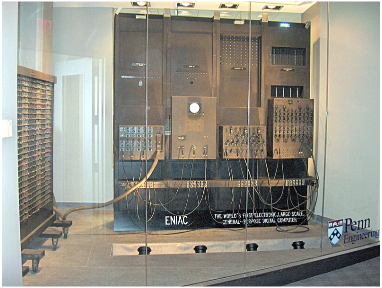
What are Domain Specific Languages (DSLs)?

In the beginning, ‘machine’ *meant* domain-specific

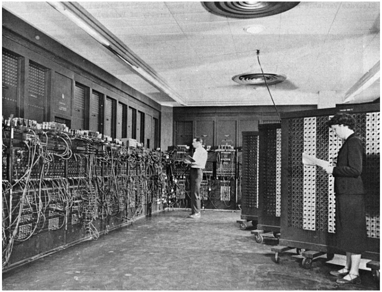
Source: vintagecalculators.com

Source: Smithsonian via aes-media.org

Programmable computers could do anything, . . . if you could wire them



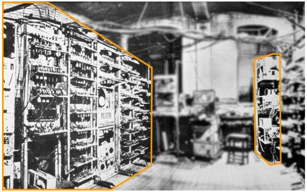
Paul W Shaffer, UPenn, via Wikipedia

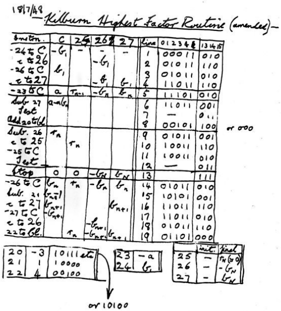


US Army via Wikipedia

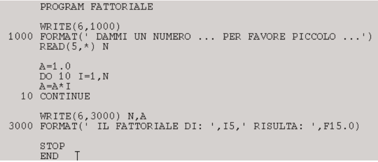
‘Stored Program’ Insight: Programs are Just Data!

Can still do anything; rush towards *general purpose* languages





UManchester via CHM



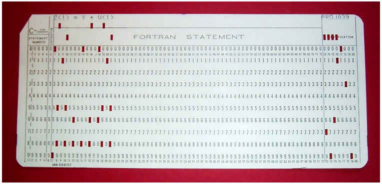
```
      PROGRAM FATTORIALE

      WRITE(6,1000)
1000  FORMAT(' DAMMI UN NUMERO ... PER FAVORE PICCOLO ...')
      READ(5,*) N

      A=1.0
      DO 10 I=1,N
      A=A*I
   10 CONTINUE

      WRITE(6,3000) N,A
3000  FORMAT(' IL FATTORIALE DI: ',I5,' RISULTA: ',F15.0)

      STOP
      END
```



HellDragon.eu and Arnold Reinhold via Wikipedia

Still, generality comes at a price


```
      PROGRAM FATTORIALE

      WRITE(6,1000)
 1000 FORMAT('  DAMMI UN NUMERO ... PER FAVORE PICCOLO ...')
      READ(5,*) N

      A=1.0
      DO 10 I=1,N
      A=A*I
   10 CONTINUE

      WRITE(6,3000) N,A
 3000 FORMAT('  IL FATTORIALE DI: ',I5,' RISULTA: ',F15.0)

      STOP
      END
```

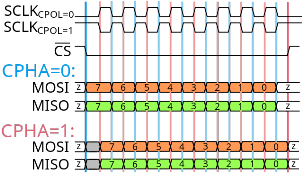
Domain-Specific Languages: Back to Basics

Focusing on a specific domain can enable:

- Better expressiveness
- Better optimizations
- More precise analyses

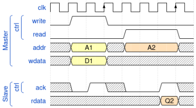
We won't be enforcing a sharp distinction with libraries, GUIs, etc.

In digital design



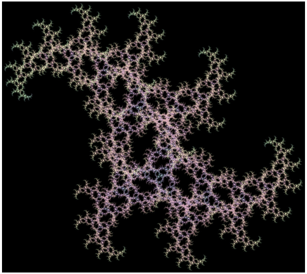
Timing diagram for SPI Bus via Wikipedia

```
{ signal: [
  {    name: 'clk',   wave: 'p..Pp..P'},
  ['Master',
    ['ctrl',
      {name: 'write', wave: '01.0....'},
      {name: 'read',  wave: '0...1..0'}
    ],
    {  name: 'addr',  wave: 'x3.x4..x', data: 'A1 A2'},
    {  name: 'wdata', wave: 'x3.x....', data: 'D1'   },
  ],
  {},
  ['Slave',
    ['ctrl',
      {name: 'ack',   wave: 'x01x0.1x'},
    ],
    {  name: 'rdata', wave: 'x.....4x', data: 'Q2'},
  ]
]}
```

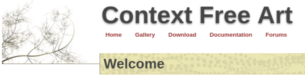


Wavedrom language

In art



A fractal



Context Free language homepage

```
CF::Background = [b -1]

sclngA = 0.7 .. 0.8
sclngB = 0.7 .. 0.8

rA =  (30 .. 180) * (0±1)
rB =  (30 .. 180) * (0±1)

baseCLR = (0..360)

startshape P (17)[h baseCLR sat 0.1 b 0.01]


shape P (natural i){
  if(i < 8) CIRCLE []
    if(i){
    P(i--1) [y 2 s sclngA r rA h 20 b (3*sin(i/3)) sat 0.01 z 1]
    P(i--1) [y 2 s sclngB r rB h -10 z -1 b (3* sin(i/5)) sat 0.01]

  }

}
```

CF program for said fractal

In software development

circ / .github / workflows / **ci.yml**

edwjchen Replaced third party dependencies with binaries to reduce CI build ti...

Code Blame 36 lines (32 loc) · 1.05 KB

```yaml
name: Build & Test

on:
    push:
        branches: [master, ci]
    pull_request:
        branches: [master, ci]

env:
    CARGO_TERM_COLOR: always

jobs:
    build:
        runs-on: ubuntu-latest

        steps:
            - uses: actions/checkout@v3
            - name: Install dependencies
              if: runner.os == 'Linux'
              run: sudo apt-get update; sudo apt-get install zsh cvc4 libboost-
            - uses: actions-rs/toolchain@v1
              with:
                  toolchain: stable
            - uses: Swatinem/rust-cache@v2
            - name: Set all features on
              run: python3 driver.py --all_features
```

continuous integration testing (YAML)

Course Parts

Three parts:

1. Technical skills: external DSLs, design, internal DSLs
   - in-class: lectures, at-home: closed-ended assignments
2. Clinics: somewhat open-ended assignments
   - work on them in-class and at-home
3. Independent project
   - open-ended, focus of the course
   - in-class: feedback and work time
   - starts early

Assignments

Assignments:

1. External Lab
2. Internal Lab
3. Clinics (2-3)
4. Project
   - brainstorming
   - proposal
   - demo and feedback
   - presentation
   - final implementation

Policies

- attendance required (studio class, participation grade)
- Communication:
  - **website**: `cs343s.stanford.edu`
  - announcements, Q&A: Ed (sign up!)
  - instructors mailing list: `cs343s@cs.stanford.edu`
  - anonymous feedback form
- assignments
  - submissions: Gradescope (sign up!)
  - individual submissions, collaboration encouraged
  - three (integer) late days
- office hours on website

External DSLs

- An "external" DSL is implemented as a complete 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)
- On the other hand, they should have concise syntax for their particular domain

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: CSS

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

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 input text, and determines it can be derived from a set of grammar rules (if at all)
- Generally outputs a *parse tree*: a tree representation of the rules used to produce the text
- Used to check *syntax*: is the string a correctly structured statement in the language

Parsing Expression Grammar (PEG)

- PEG is language used to specify the grammar of a language (PEG is a DSL!)
- PEG consists of a sequence of definitions (non-terminals)
  - `identifier = expression`
- At their most basic, expressions can consist of a *terminal* ("abc", ~r"b.*"), or another definitions
  - `one = "1"`
  - `eleven = one one`
- A terminal *matches* the exact text, a definition *matches* if its expression matches
- The first definition is the "starting expression", and is used to match the entire text.

Parsimonious PEG Expressions

Let e_1 and e_2 be arbitrary expressions

- Literal: " " (`"1"`)
- Python-style Regex: ~r"regex"ilmsuxa (`~r"[a-z]"i`)
- Sequence: e_1 e_2 (`"1" "1"`)
- Choice: e_1 / e_2 (`"1" / "2"`)
- Grouping: (e_1) (`("1" / "2") "1"` vs `"1" / ("2" "1")`)
- Optional: e_1? (`"1"?`)
- Zero-or-more: e_1* (`"1"*`)
- One-or-more: e_1+ (`"1"+`)
- Exactly-n: e_1{n} (`"1"{n}`)
- Lookahead: &e_1 (`&"1"`)
- Negative Lookahead: !e_1 (`!"1"`)

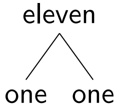
Parse Trees

The parser (e.g. parsimonious) outputs a *parse tree*: a tree representation of the rules which matched the string

Example: Parsing `11`

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

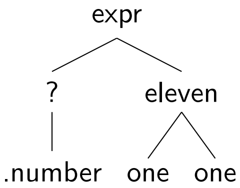
$\Rightarrow$



Example: Parsing `#11`

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

$\Rightarrow$



PEG is unambiguous: every string has exactly 0 or 1 valid parse trees

Recursion

Rules may be recursive, meaning they reference themselves within their definitions

Example: `ones = one ones?`

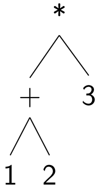
However, PEG does NOT allow the left-most expression in a sequence to be recursive (e.g. no left recursion)

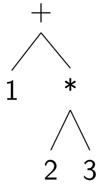
Example: `ones = ones one` is NOT allowed

Live Coding: Arithmetic Parsing

Precedence

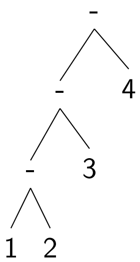
Example: $1 + 2 * 3$

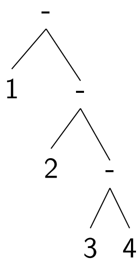




Associativity

Example: 1 - 2 - 3 - 4





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

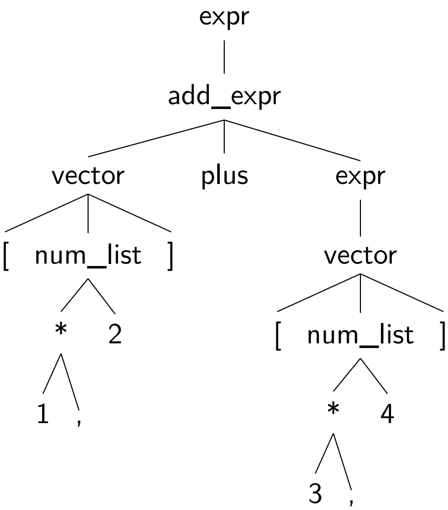
Example: Vector Addition

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*"
```

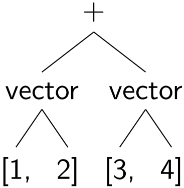
Example: Vector Addition

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 tree and convert each node into AST nodes
- Parsimonious steps:
  1. Sub-class the `NodeVistor` class
  2. Implement visitor methods for each definition
  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 tree and convert each node into AST nodes
- Parsimonious steps:
  1. Sub-class the `NodeVistor` class
  2. Implement visitor methods for each definition
  3. Call `visit` on the parse tree

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

Node object representing the matching definition

Converting Parse Trees to ASTs in Parsimonious

- General idea: perform a depth first traversal of the tree and convert each node into AST nodes
- Parsimonious steps:
  1. Sub-class the `NodeVistor` class
  2. Implement visitor methods for each definition
  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

Live Coding: Parse Tree → AST

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

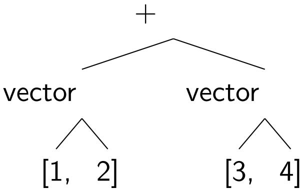
- Fairly straightfoward to write (in comparison to a compiler or transpiler)
- Often easier to debug
- Many DSLs aren't performance critical
- Can use features of the "host" language (e.g. memory management)

Writing a Tree-Walking Interpreter

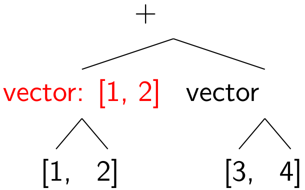
Tree-Walking Intrepreter: Traverse the AST, executing as you go.

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

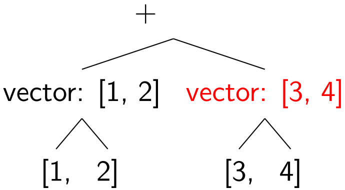
Example: [1, 2] + [3, 4]



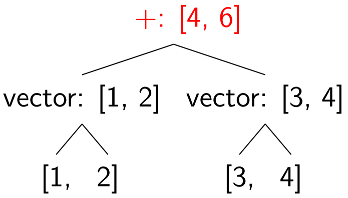
Example: $[1, 2] + [3, 4]$



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



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



Live Coding: Evaluating Arithmetic

Tips and Tricks

- 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

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 {
        ...
    }

```

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

```

```python
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

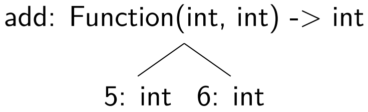
```
def add(x: int, y: int) -> int { ... }

add(5, 6)

```

Type checking function calls

Type checking function calls



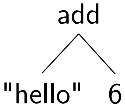
Type checking function calls

add: Function(int, int) -> int

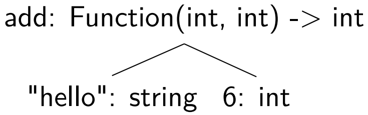
Does 5.ty == int and 6.ty == int?

5: int   6: int

Type checking function calls



Type checking function calls



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).