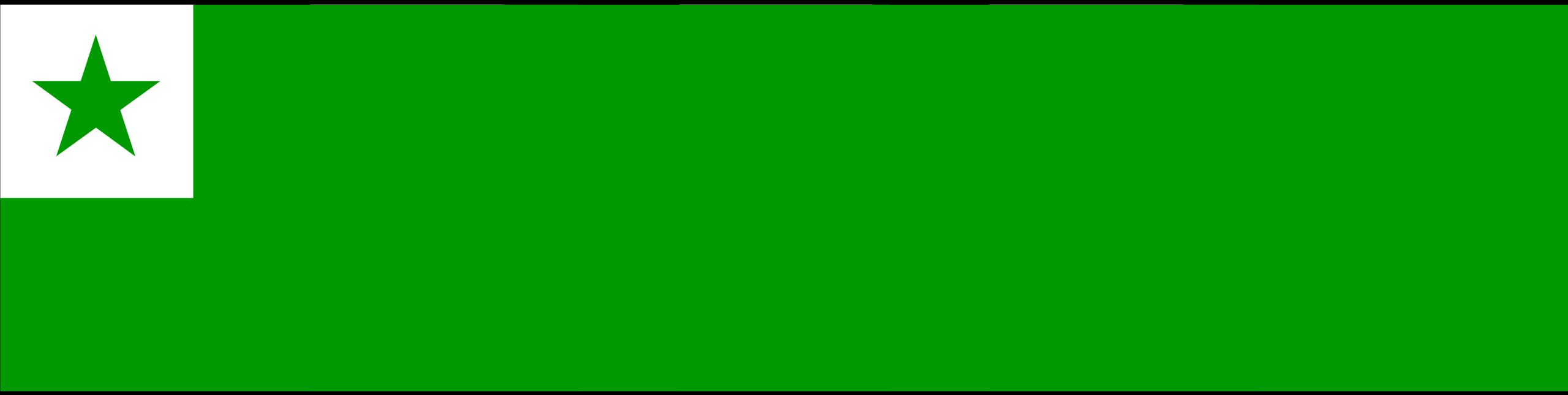
ConLangLang (CLL)

Project Guide and Overview

By Annalise Mariottini



Introduction: What is CLL?

- CLL is a concise and flexible semi-imperative programming language inspired by functional language attributes for the purpose of categorizing and processing language data
- CLL includes user-defined types, complex data structures to contain these types, and standard library functions for manipulating these structures
- CLL also includes function literals, nested functions, built-in regex functionality, automatic garbage collection, and stdin access

Built-In Types

Object Type	Object Definition	Object Literal Example
Int	Base-10 integer	342
Float	Floating point number	5.0
Boolean	1 bit value	true
String	Null-terminated array of characters	‘hello world’
Regex	Compiled regex object (POSIX-based)	“I love (draw)|(runn)ing”
List	Singly-linked-list of similarly-typed nodes	<int>[0,1,2,3]
Dictionary	Hashtable of key-pair values	<string,string>{ ‘a’:’A’, ‘b’:’B’, ‘c’:’C’}
Fun	Function literal	<int x:int>{ x + 10; }

Lists and Dictionaries can hold any of the types above EXCEPT for Funs
Funs can be passed and return any of the types above EXCEPT for Funs

Variable Declaration

VERY SIMPLE → → →

```
myint = 0; #any literal
myfloat = 5.5;
mystr = 'hi';
myre = "hi(!)*";
mylist = <string>{'a'};
mydict = <int,int>{5:10};
myfun = <int x:x>{ x%2; };
```

- Variables are mutable and can be changed in value OR in type (the compiler keeps track of variable type)
- All variable garbage collection is managed by the compiler

User-Defined Types and TypeDefs

- Type = Aesthetic name for a built-in type
- TypeDef = Structure containing predefined child variables

```
type mytype = { Foo<string>, Bar<int> };
typedef $MyTypeDef = {
   Bar barID;
   string name;
};
myfoo = (Foo) 'this is foo';
$MyTypeDef myinstance = {
   barID = (Bar) 212;
   name = 'guy';
};
```

Operators

Precedence	Types	Name	Symbol	Association
0		Child access	.	left
1		Cast	(type)	right
2	Int, Float	Negation	-	right
		Multiplication	*	left
		Division	/	left
		Remainder	%	left
		Addition	+	left
		Subtraction	-	left
	String, List	Concatenation	^	left

Operators (Continued)

Precedence	Types	Name	Symbol	Association
2	Boolean	Logical Not	!	left
		Logical And	&&	left
		Logical Or	||	left
3	Int, Float, Bool	Equals	==	left
		Greater Than	>	left
		Less Than	<	left
		Type Equals	~=	left
4		Assignment	=	right

Control Flow

All control flow blocks are expressions and thus must have a return value. This return value is the last statement of whichever appropriate block is run (or, for do-while, on which iteration).

Fun

```
<string s:string>{
  s ^ '!';
};
```

If-Else

```
if:int ((x % 2) == 0) {
  sprint('x is even');
} else {
  sprint('x is odd');
};
```

Do-While

```
l = <int>[0];
dowhile:list<int> (lget(l,0) < 10) {
  x = lget(l,0);
  ladd(l,x+1); # adds to front
};
```

Match

```
match:string (x) byvalue {
  'hello' { x ^ ' world'; }
  'good' { x ^ ' bye'; }
  default { ''; }
};
```

```
match:string (x) bytype {
  Foo { 'is foo'; }
  Bar { 'is bar'; }
  default { ''; }
};
```

Functions

- Function literals are equivalent to function variables
- Functions can be nested and inherit all the variables defined in scopes that encompass it
- Once a function is assigned to a variable, that variable cannot be redefined

```
outside = 'I get passed';
<:int>{
  inner_f = <:int>{
    sprint(outside);
  };
  inner_f();
}(); # prints 'I get passed'
```

Standard Library

Data Type	Functions
String	sprint(s), sfold(f, a, s), ssize(s)
Regex	rematch(r, s), resub(r, s, s_, n)
List	ladd(l, e), ladd(l, e, n), lfold(f, a, l), lget(l, n), lmap(f, l), lmem(l, e), lremove(l), lremove(l, n), lsize(l)
Dict	dadd(d, k, v), dfold(f, a, d), dget(d, k), dmap(f, d), dmem(d, k), dremove(d, k), dsize(d), dkeys(d)

The most interesting ones:

- rematch = returns true if regex matches string
- resub = returns string with any matches to regex group n replaced by a substitute string
- _fold = iterate function over each element of object and return an accumulated value
- _map = iterate function over each element of object and return new object with those modified elements
    - Actually _fold under the hood

Program Structure

```
{# THIS IS A COMPLETE PROGRAM #}
sprint('hello');
```

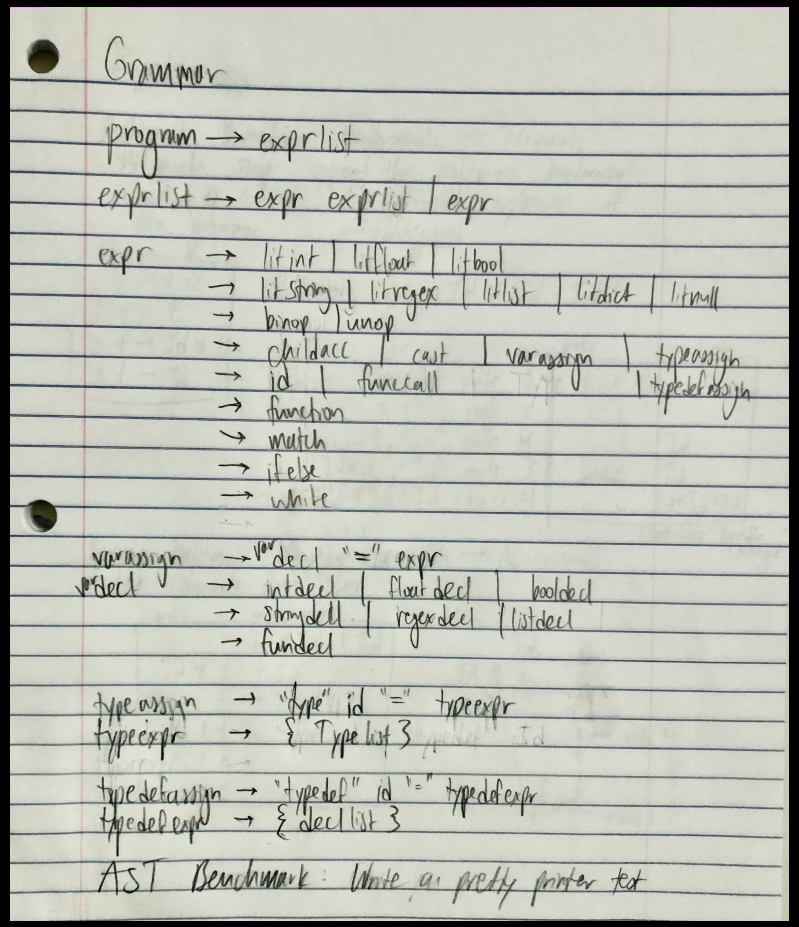
- Program consists of a series of statements
- Statements are either type definitions, typedef definitions, or expressions
- Expressions make up the bulk of programming language elements

```
words = <string>['esti', 'lerni', 'havi', 'pano',
'pilko'];
type verb = { Root<string>, Infinitive<string>,
Present<string>, Past<string>, Future<string> };

typedef $Stems = {
    string inf;
    string pres;
    string past;
    string fut;
};

typedef $VerbDef = {
    Infinitive inf;
    Present pres;
    Past past;
    Future fut;
};

$Stems stems = {
    inf = 'i';
    pres = 'as';
    past = 'is';
    fut = 'os';
};
```

```
conj = <string,regex>{ stems.inf: "[a-z]+(i)$",
stems.pres: "[a-z]+(as)$", stems.past: "[a-z]+(is)$",
stems.fut: "[a-z]+(os)$" };
root = <list<Root> l, string w : list<Root>> {
    w = match:string (true) byvalue {
        rematch(dget(conj, stems.inf), w) {
            resub(dget(conj, stems.inf), w, '', 1);
        }
        rematch(dget(conj, stems.pres), w)  {
            resub(dget(conj, stems.pres), w, '', 1);
        }
        rematch(dget(conj, stems.past), w)  {
            resub(dget(conj, stems.past), w, '', 1);
        }
        rematch(dget(conj, stems.fut), w)  {
            resub(dget(conj, stems.fut), w, '', 1);
        }
        default  {
            '';
        }
    };
```

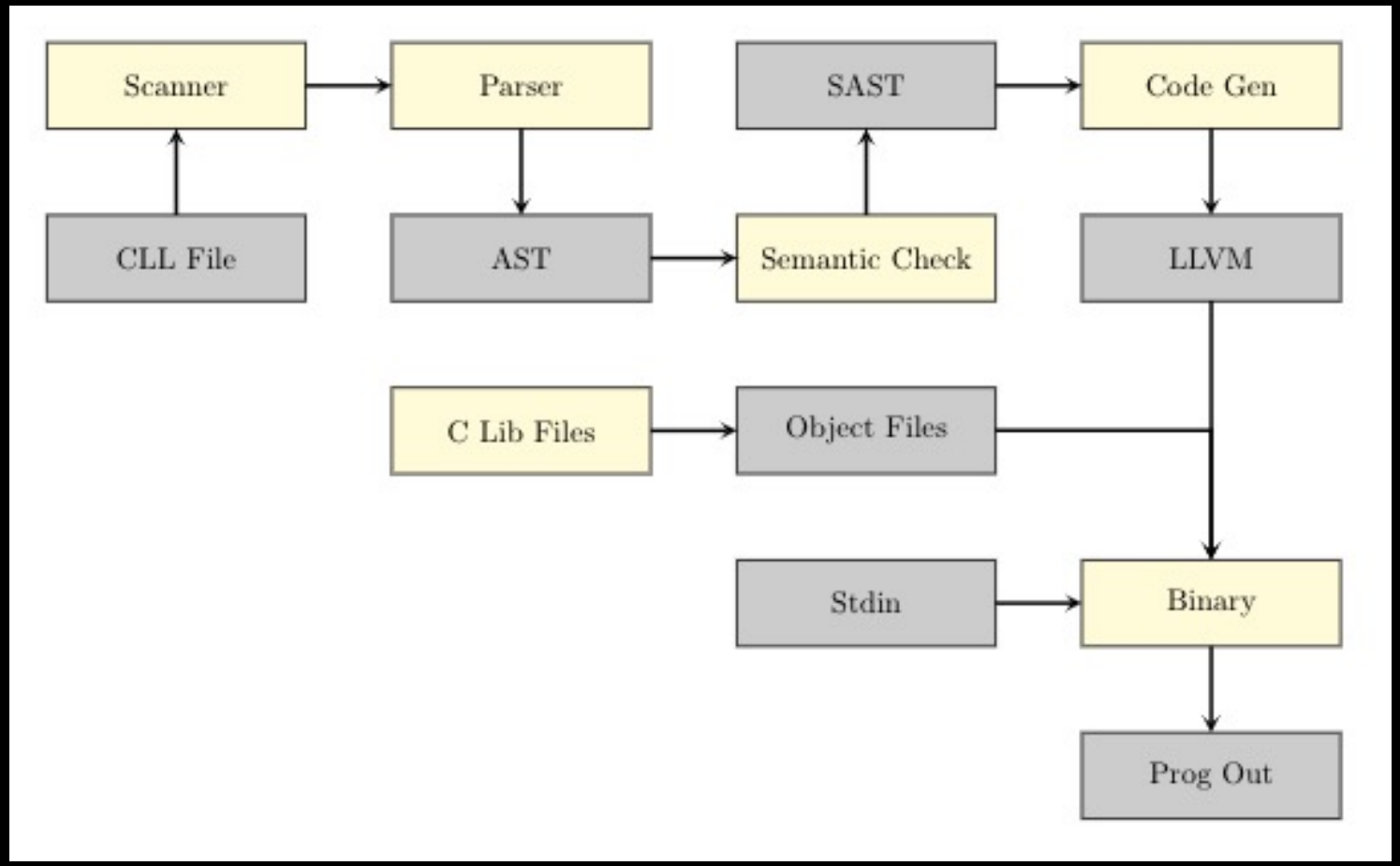
```
 if:list<Root> (ssize(w) > 0) {
        ladd(l,(Root)w);
    } else {
        l;
    };
};


create_verb = <dict<Root,$VerbDef> d, Root w:
dict<Root,$VerbDef>> {
    $VerbDef v = {
        inf = (Infinitive)((string)w ^ stems.inf);
        pres = (Present)((string)w ^ stems.pres);
        past = (Past)((string)w ^ stems.past);
        fut = (Future)((string)w ^ stems.fut);
    };
    dadd(d, w, v);
};

words = lfold(root, <Root>[], words);
verb_dict = lfold(create_verb, <Root,$VerbDef>{},
words);
```

Development Process



- Started with inspirations for language: Conlangs
- Built out scanner, parser, AST, and SAST
- Created semant and started automated testing
  - Created tests for features as features were implemented, including tests for failures
- Implemented and tested codegen features
  - Started checking compiled program output
- Created C file support for more complicated features
  - Linked list, hashtable, regex, malloc manager, stdin
- Tools used:
  - Git
  - GNU C Library
  - Make/clang
  - Bash scripts
- Programming environment:
  - macOS 10.15.7, LLVM 11.1, OCaml 4.12.0

Compiler Pipeline



Takeaways

- Small bites as opposed to big chunks
- When possible, take the laziest route possible
- TDD: Create tests for the features you want and make those tests pass
- Understand the tools you’re using before you use them