



Writing a Simple DSL Compiler with Delphi

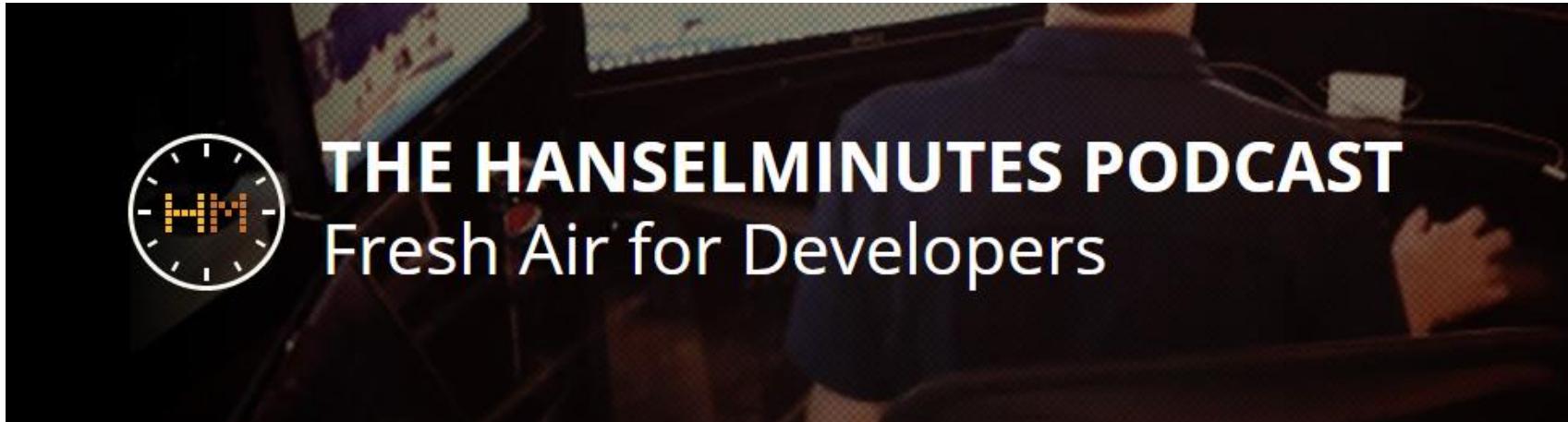
Primož Gabrijelčič / primoz.gabrijelcic.org

About me

- Primož Gabrijelčič
- <http://primoz.gabrijelcic.org>
- programmer, MVP, writer, blogger, consultant, speaker
- Blog <http://thedelphigeek.com>
- Twitter [@thedelphigeek](https://twitter.com/thedelphigeek)
- Skype [gabr42](https://skype.com/gabr42)
- LinkedIn [gabr42](https://www.linkedin.com/in/gabr42)
- GitHub [gabr42](https://github.com/gabr42)
- SO [gabr](https://stackoverflow.com/users/117077/gabr)
- Google+ [Primož Gabrijelčič](https://plus.google.com/+PrimožGabrijelčič)

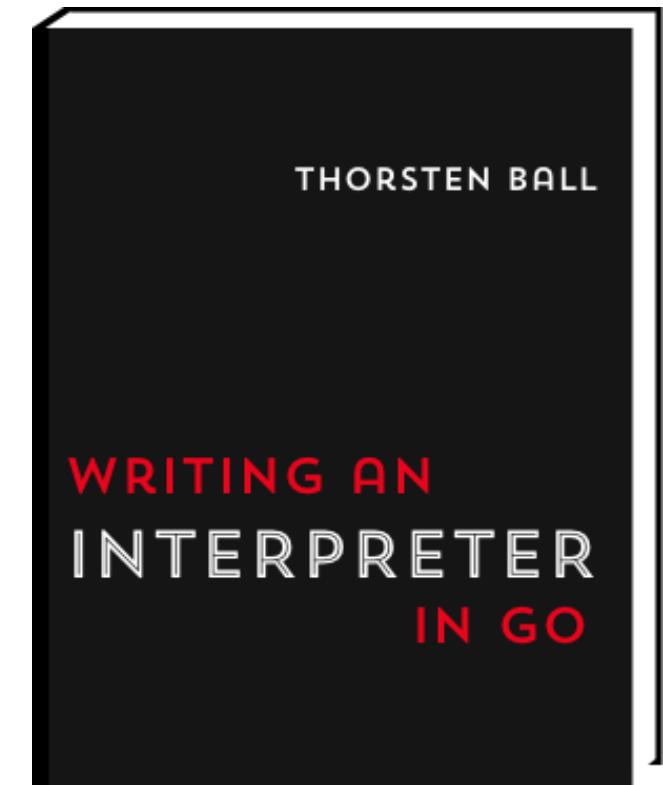
WHY AM I HERE?

It all had started with a podcast ...



<https://hanselminutes.com>

<https://interpreterbook.com>



DSL

DSL?

- ~~Damn Small Linux~~
- ~~Danish Sign Language~~
- ~~Dictionary of the Scots Language~~
- ~~Dominican Summer League~~
- ~~Domestic Substances List~~
- Domain Specific Language
 - A (computer) language designed for a specific problem domain
 - In short ... a programming language

[https://en.wikipedia.org/wiki/DSL_\(disambiguation\)](https://en.wikipedia.org/wiki/DSL_(disambiguation))

When?

- When presenting a special syntax helps certain class of users
- Most popular DSLs: SQL, html, LaTeX, BNF, VHDL

VHDL

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity signed_adder is
6   port
7   (
8     aclr : in  std_logic;
9     clk  : in  std_logic;
10    a    : in  std_logic_vector;
11    b    : in  std_logic_vector;
12    q    : out std_logic_vector
13  );
14 end signed_adder;
15
16 architecture signed_adder_arch of signed_adder is
17   signal q_s : signed(a'high+1 downto 0); -- extra bit wide
18
19 begin -- architecture
20   assert(a'length >= b'length)
21     report "Port A must be the longer vector if different sizes!"
22     severity FAILURE;
23   q <= std_logic_vector(q_s);
24
25   adding_proc:
26   process (aclr, clk)
27   begin
28     if (aclr = '1') then
29       q_s <= (others => '0');
30     elsif rising_edge(clk) then
31       q_s <= ('0'&signed(a)) + ('0'&signed(b));
32     end if; -- clk'd
33   end process;
34
35 end signed_adder_arch;
```

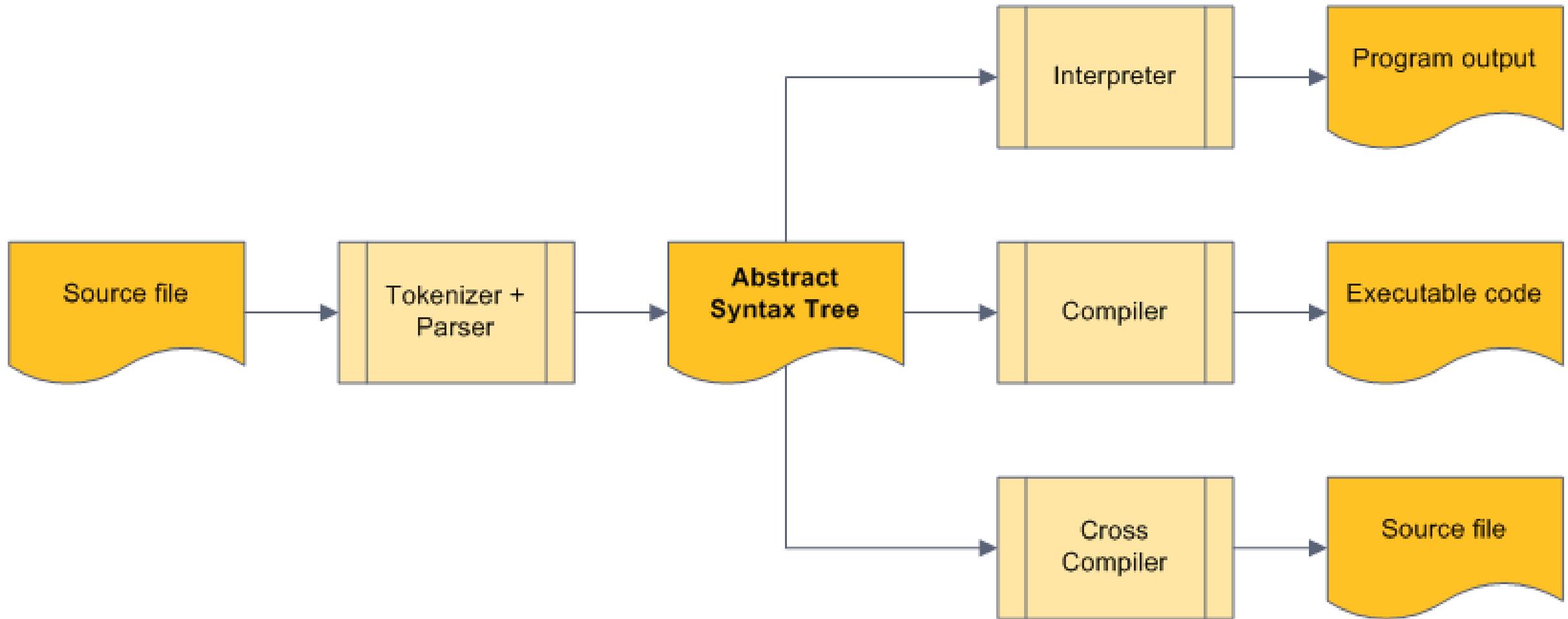
Source: https://en.wikipedia.org/wiki/VHDL#/media/File:Vhdl_signed_adder_source.svg

FROM PROGRAM TO RESULT

From Program to Result

- Program = stream of characters
- Parsing
 - Lexical analysis [lexer/tokenizer]
 - Characters → tokens
 - Defined by regular expressions
 - Syntactical analysis [parser]
 - Tokens → internal representation [AST]
 - Defined by a grammar
- Execution
 - Interpreter: Walk over an AST + execute step by step
 - Cross-compiler: Walk over an AST + rewrite it as an equivalent textual output
 - Compiler: Walk over an AST + generate machine code (for some architecture)
 - [semantical analysis]

From program to result



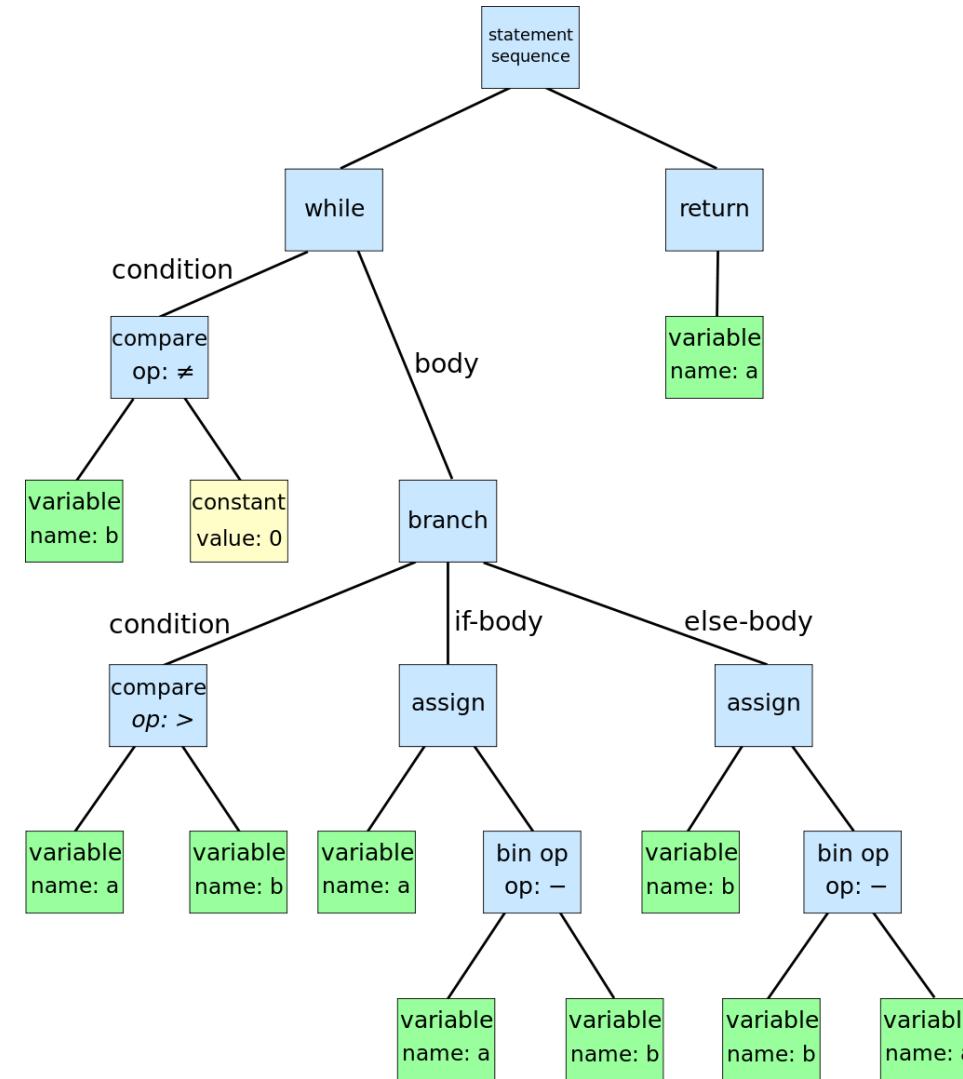
Abstract Syntax Tree (AST)

- An abstract syntactic structure in a tree form
- Inessential stuff is removed
 - Punctuation
 - delimiters
- Can contain extra information
 - position in source code
- Specific for a single language

https://en.wikipedia.org/wiki/Abstract_syntax_tree

AST Example

```
while b ≠ 0
    if a > b
        a := a - b
    else
        b := b - a
return a
```



Source: https://en.wikipedia.org/wiki/Abstract_syntax_tree#/media/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg

DelphiAST

- <https://github.com/RomanYankovsky/DelphiAST>
 - One unit at a time
- <https://github.com/gabr42/DelphiAST>
 - Project indexer
- <https://github.com/gabr42/DelphiLens>
 - Research project

GRAMMARS FOR DUMMIES

Grammar

- Set of production rules
 - Left hand side → right hand side
- Symbols
 - Nonterminal [can be expanded]
 - Terminal [stays as it is]
 - Start
- Can be recursive or non-recursive
 - Non-recursive → not interesting

https://en.wikipedia.org/wiki/Recursive_grammar

Grammar Example

- Example
 - Teminals: {a,b}
 - Nonterminals: {S, A, B}
 - Rules:
 - $S \rightarrow AB$
 - $S \rightarrow \epsilon$
 - $A \rightarrow aS$
 - $B \rightarrow b$
- Simpler version
 - $S \rightarrow aSb$
 - $S \rightarrow \epsilon$
- Language
 - $a^n b^n$
- Example
 - $S \rightarrow AB \rightarrow aSB \rightarrow aSb \rightarrow aABb \rightarrow aAbb \rightarrow aaSbb \rightarrow aabb$
- Example
 - $S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb$

https://en.wikipedia.org/wiki/Formal_grammar

Chomsky Hierarchy

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ $A \rightarrow aB$



https://en.wikipedia.org/wiki/Noam_Chomsky

https://en.wikipedia.org/wiki/Chomsky_hierarchy

Context-free Grammars

- Base of program language design
 - Typically cannot satisfy all needs
 - Indentation-based languages
 - Macro- and template-based languages
 - Attribute grammar
 - Compiler = definition

https://en.wikipedia.org/wiki/Context-free_grammar
https://en.wikipedia.org/wiki/Attribute_grammar

Syntax vs. semantics

- Not all syntactically correct programs compile!
 - Most of them don't!

```
program Test;  
begin  
  a := 1;  
end.
```

- Set of **syntactically** correct programs = CFG (possibly)
- Set of **semantically** correct programs \neq CFG (= CSG)

Documenting the grammar

- Backus-Naur form (BNF)
- Extended Backus-Naur form (EBNF)

https://en.wikipedia.org/wiki/Backus-Naur_form

https://en.wikipedia.org/wiki/Extended_Backus-Naur_form

Example – Pascal-like Language

```
program = 'PROGRAM', white space, identifier, white space,
          'BEGIN', white space,
          { assignment, ";" , white space },
          'END.' ;

identifier = alphabetic character, { alphabetic character | digit } ;

number = [ "-" ], digit, { digit } ;

string = ''' , { all characters - ''' }, ''' ;

assignment = identifier , ":=" , ( number | identifier | string ) ;

alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                      | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                      | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                      | "V" | "W" | "X" | "Y" | "Z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

white space = ? white space characters ? ;

all characters = ? all visible characters ? ;
```

Example – Delphi 5 EBNF (partial)

```
start= program | unit | library | package .  
  
identifier_list= ID_NAME { ',' ID_NAME } .  
unit_qualified_identifier= ID_NAME { '.' ID_NAME } .  
  
type_name= TYPE_NAME | STRING | FILE .  
unit_qualified_type_name= type_name [ '.' type_name ] .  
  
function_result_type= type_name .  
  
constant_expression= F .  
string_expression= ( STRING_NAME | STRING_LITTERAL )  
{ '+' ( STRING_NAME | STRING_LITTERAL ) } .  
  
variable_access= ( ACCESS_NAME | STRING ) { end_access_ } .  
end_access_= { array_access_ | record_access_ | '^' | function_parameters_ } .  
array_access_= '[' constant_expression { ',' constant_expression } ']' .  
record_access_= '.' variable_access .  
function_parameters_= '(' [ constant_expression { ',' constant_expression } ] ')' .  
  
set_factor= '[' [ set_element { ',' set_element } ] ']' .  
set_element= constant_expression [ '..' constant_expression ] .  
  
constant_expression= simble_expression [ ('=' | '<' | '>' | '<=' | '>=' | IN )
```

Source: http://www.felix-colibri.com/papers/compilers/delphi_5_grammar/delphi_5_grammar.html

PARSING

Parsing in Practice

- Lexer
 - Typically DFA (regular expressions)
 - Generator
 - Custom
- Parser
 - Typically LR(0), LR(1), LALR(1), LL(k)
 - L_x top-to-bottom
 - xL Leftmost derivation
 - xR Rightmost derivation
 - (n) lookahead
 - LALR Look-Ahead LR, a special version of LR parser
 - Generator
 - Custom

https://en.wikipedia.org/wiki/Lexical_analysis

https://en.wikipedia.org/wiki/Parsing#Computer_languages

https://en.wikipedia.org/wiki/Comparison_of_parser_generators

LL / LR

Leftmost

$S \rightarrow S + S$ (1)

$\rightarrow 1 + S$ (2)

$\rightarrow 1 + S + S$ (1)

$\rightarrow 1 + 1 + S$ (2)

$\rightarrow 1 + 1 + a$ (3)

1. $S \rightarrow S + S$

2. $S \rightarrow 1$

3. $S \rightarrow a$

Rightmost

Input: $1 + 1 + a$

$S \rightarrow S + S$ (1)

$\rightarrow S + a$ (3)

$\rightarrow S + S + a$ (1)

$\rightarrow S + 1 + a$ (2)

$\rightarrow 1 + 1 + a$ (2)

https://en.wikipedia.org/wiki/LR_parser
https://en.wikipedia.org/wiki/LL_parser
https://en.wikipedia.org/wiki/LALR_parser

A SIMPLE PRIMER

A Simple Primer

- Language
 - Addition of non-negative numbers
 - 1
 - 1 + 2
 - 1 + 2 + 44 + 17 + 1 + 0
- AST
- Tokenizer
- Parser
- Interpreter
- Compiler

MY “TOY LANGUAGE”

My Toy Language

```
fib(i) {  
    if i < 3 {  
        return 1  
    } else {  
        return fib(i-2) + fib(i-1)  
    }  
}
```

Specification

- C-style language
- Spacing is ignored
- One data type - integer
- Three operators: +, -, and <
 - $a < b$ returns 1 if a is smaller than b , 0 otherwise
- Two statements - **if** and **return**
 - If statement executes **then** block if the test expression is not 0. **Else** block is required
 - **Return** statement just sets a return value and doesn't interrupt the control flow
- There is no assignment
- Every function returns an integer
- Parameters are always passed by value
- A function without a **return** statement returns 0
- A function can call other functions (or recursively itself)

Grammar

function ::= identifier "(" [identifier { "," identifier }] ")" block

block ::= "{" statement ";" statement } [";"] "}"

statement ::= if | return

if ::= "if" expression block "else" block

return ::= "return" expression

expression ::= term | term operator term

term ::= numeric_constant | function_call | identifier

operator ::= "+" | "-" | "<"

function_call ::= identifier "(" [expression { "," expression }] ")"

EXTENDING THE LANGUAGE

Attributes

- AST
- Tokenizer
- Parser
- Interpreter
- Compiler

```
fib(i) [memo] {  
    if i < 3 {  
        return 1  
    } else {  
        return fib(i-2) + fib(i-1)  
    }  
}
```

THANK YOU!