Writing lexer and parser in Haskell

Denys Zinoviev 2025

What does the compiler work?

1. Plaintext

```
let fact n = if n <= 1 then n else n * fact (n - 1)
```

What does the compiler work?

1. Plaintext

```
let fact n = if n <= 1 then n else n * fact (n - 1)
```

2. Token Stream

```
Keyword "let",
Identifier "fact",
Identifier "n",
Symbol "=",
Keyword "if",
. . .
```

How do we get the token stream?

- Trivial for simple languages i.e BF or calculator

How do we get the token stream?

- Trivial for simple languages i.e BF or calculator
- Hard for real languages

How do we get the token stream?

- Trivial for simple languages i.e BF or calculator
- Can be a bit tricky for the real languages
    - `if8` is identifier or two separate tokens?
    - Comments (especially nested)
    - String literals

How do we get the token stream?

- Let's represent the tokens using the Regular Expressions

How do we get the token stream?

- Let's represent the tokens using the **Regular Expressions**
- Int: `-?[0-9]+`
- Keywords: `if | then | else | let | in | while | return | do`
- Identifier: `[a-zA-Z_][a-zA-Z0-9_]*`
- And so on

How do we get the token stream?

- Then, translate the Regular Expressions into **Nondeterministic Finite Automata**.

How do we get the token stream?

- Then, translate the Regular Expressions into **Nondeterministic Finite Automata**.
- And finally, translate the **Nondeterministic Finite Automata** into **Deterministic Finite Automata**.

A bit of implementation details

```
data Regex
  = Empty
  | Cat Regex Regex
  | Alt Regex Regex
  | Lit Char
  | Star Regex
  | Plus Regex
  | Range Char Char
```

A bit of implementation details

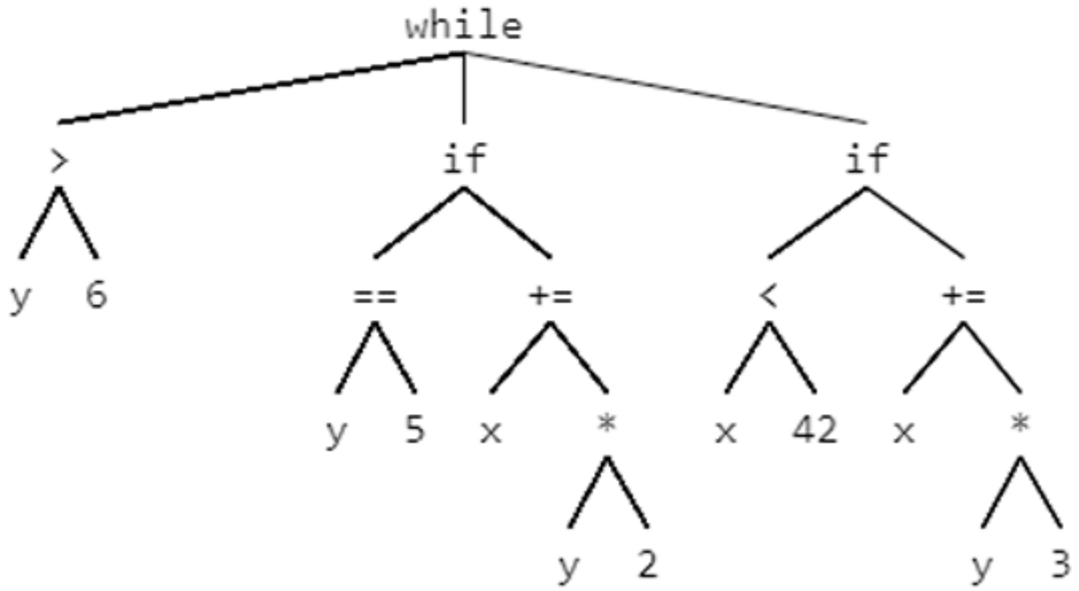
```
data Node = Node Int deriving (Eq, Ord, Show)

data Transition = Eps | Transition Char

data NFA = NFA
  { initial :: Node,
    terminal :: [Node],
    transitions :: Map.Map Node [(Transition, Node)]
  }
```

A bit of implementation details

* Each node should be assigned a unique index - let's use the **State Monad**

```
translate :: Regex -> State Int NFA
```

What is AST?



```
while y < 6:

  if y == 5:
    x += y * 2

  if x < 42:
    x += y * 3
```

Why it's hard to get AST ?

- Operator precedence: (`a / b / c` **->** `a / (b / c)` or `(a / b) / c`)
- Function Calls vs. Identifiers
- Proper Error handling
- `if a then b if x then y else z` - `else z` corresponds to the inner or outer if?

How to represent AST? Context Free Grammars

- Tokens from the lexer : `ID, THEN, IF, (, ), +`

```
Expr ->
  | ID
  | ID `(` Args `)`
  | ID `(` `)`
  | ID `+` ID
  | IF Expr THEN Expr

Args ->
  | ID
  | Args `,` ID
```

How to get AST from the grammars?

- Recursive Descent Parsing `Can become O(2^n)`
- LL parser (Similar to the braces sequence validity check) `O(n) + preprocessing`
- LR parser `O(n) + preprocessing`

So, what is the plan?

- Represent the Regular Expressions
- Represent the NFA
- Implement the RegExpr -> NFA
- Represent the DFA
- Implement the NFA -> DFA
- Tokenizer
- Represent the AST
- Represent the CFG
- Implement the Recursive Descent Parser