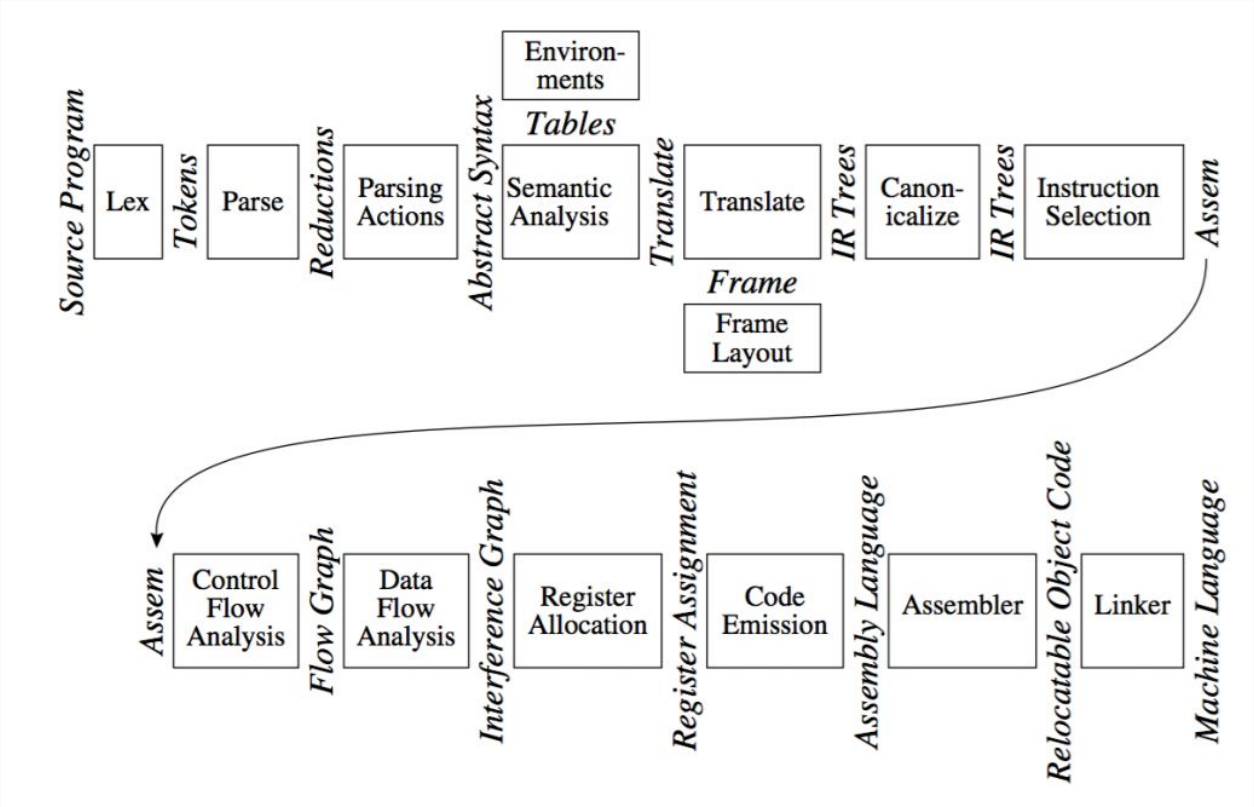
Resumão Compiladores

Um pouco de teoria + PLY

Introdução



Front-End:
Análise

Back-End:
Síntese

Nosso Compilador será mais simples (pelo que eu entendi)

Focaremos então no Front-End, mais especificamente:

- Lexer (separa a entrada em tokens)
- Parser (verifica a estrutura das frases)
- Análise Semântica (analisa o significado das frases)

Lexer

```
float match0(char *s) /* find a zero */
  {if (!strncmp(s, "0.0", 3))
    return 0.;
  }
```

Retorno do analisador léxico:

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s)
COMMA STRING(0.0) COMMA NUM(3) RPAREN RPAREN
RETURN REAL(0.0) SEMI RBRACE EOF

O conjunto de tokens deve ser finito.

Alguns tokens possuem um valor semântico associado.

Definições

Antes de falarmos de regex precisamos de algumas definições:

- Linguagem: conjunto de strings
- String: sequence de símbolos
- Símbolos: elementos de um alfabeto finito

Um regex então especifica uma linguagem, queremos saber se uma string pertence ou não a essa linguagem.

Regex

- **a** An ordinary character stands for itself.
- ∈ The empty string.
- Another way to write the empty string.
- *M* | *N* Alternation, choosing from *M* or *N*.
- *M* · *N* Concatenation, an *M* followed by an *N*.
- *MN* Another way to write concatenation.
- *M** Repetition (zero or more times).
- *M*+ Repetition, one or more times.
- *M*? Optional, zero or one occurrence of *M*.
- **[a − zA − Z]** Character set alternation.
- . A period stands for any single character except newline.
- "a.+*" Quotation, a string in quotes stands for itself literally.

Regras de Desempate

- Longest Match
- Ordem das Regras (rule priority)

A especificação léxica contém todas as linguagens de todos os tokens aceitos. Ela deve ser completa, isto é, qualquer string deve ser aceita por alguma regra. Por isso colocamos uma regra ponto (.) ao fim, que vai dar match com qualquer string.

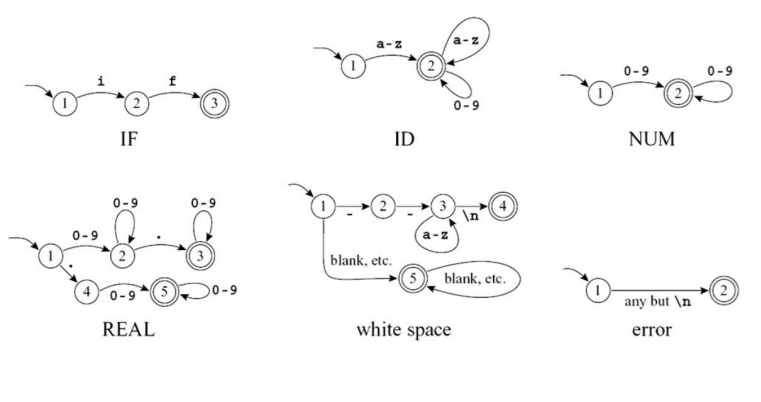
```
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

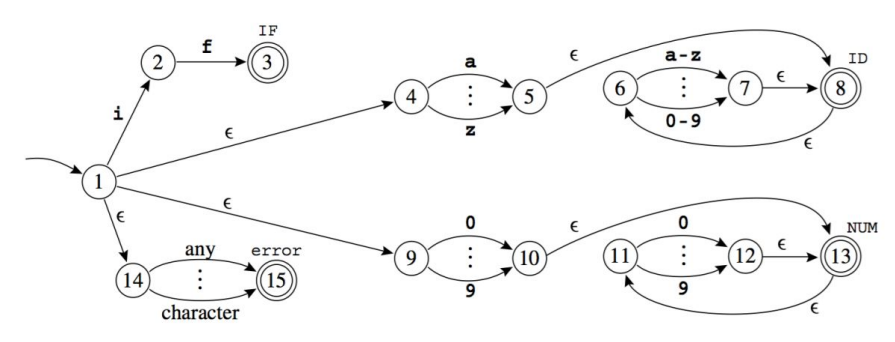
Como fazer o match de string com regex?

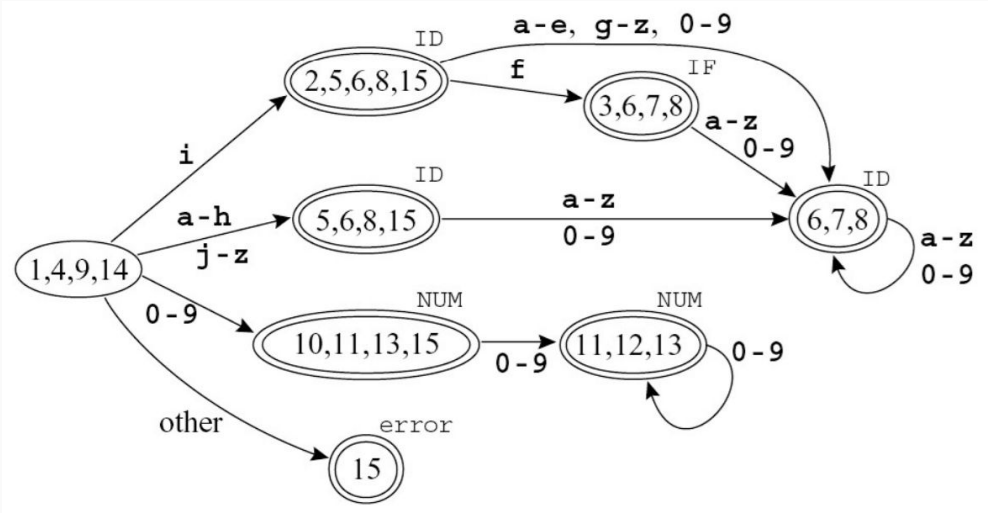
- Crie uma DFA para cada um dos regex.
- Junte as DFA’s em uma NFA
- Utilize o closure dos nós na NFA para converter em uma DFA.

Bazuca para matar uma mosca, mas ok.

DFA -> NFA -> DFA


IF
ID
NUM
REAL
white space
error
blank, etc.
any but \n


IF
ID
NUM
any
error
character


a-e, g-z, 0-9
2,5,6,8,15
3,6,7,8
5,6,8,15
6,7,8
1,4,9,14
10,11,13,15
11,12,13
15
other
error

Parser

Para expressarmos a estrutura de uma frase regex não são o suficiente pois elas são implementadas com DFA’s, e portanto, não possuem memória.

Precisamos de um formalismo mais poderoso chamado de gramáticas livres de contexto. Descreveremos a linguagem através de um conjunto de produções da forma:

Symbol -> Symbol Symbol Symbol...

Context-Free Grammars

1. $S \rightarrow S; S$
2. $S \rightarrow \text{id} := E$
3. $S \rightarrow \text{print}\ (L)$
4. $E \rightarrow \text{id}$
5. $E \rightarrow \text{num}$
6. $E \rightarrow E + E$
7. $E \rightarrow (S, E)$
8. $L \rightarrow E$
9. $L \rightarrow L, E$

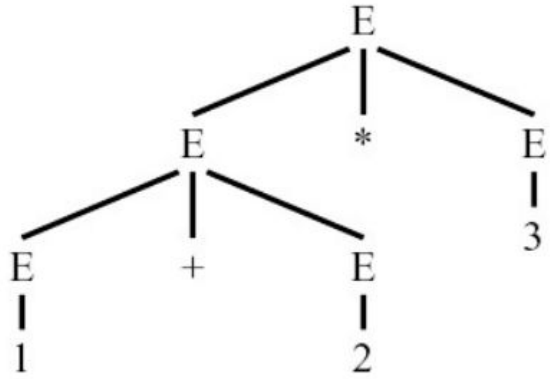
id := num; id := id + (id := num + num, id)

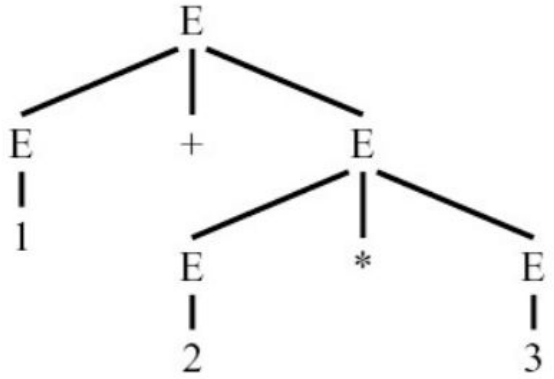
Para checarmos se uma frase pertence a gramática nós podemos começar do símbolo inicial e derivar até chegar (ou não) na frase. Isso é bem ineficiente.

Gramáticas ambíguas

Uma gramática é dita ambígua quando uma frase pode gerar duas parse-trees diferentes.

$E \rightarrow \text{id}$
$E \rightarrow \text{num}$
$E \rightarrow E * E$
$E \rightarrow E / E$
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow (E)$





Ambígua!

(1+2)*3 = 9 e 1+(2*3) = 7

Eliminando ambiguidades

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \text{id}$$
$$E \rightarrow E - T \qquad T \rightarrow T / F \qquad F \rightarrow \text{num}$$
$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (E)$$

A expressão só será avaliada após o termo ser avaliado. O termo só será avaliado após o fator ser avaliado.

Calculadora no PLY

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \text{id}$$
$$E \rightarrow E - T \qquad T \rightarrow T / F \qquad F \rightarrow \text{num}$$
$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (E)$$

Note que sempre que ocorrer um match com alguma derivação, já estamos calculando o valor do resultado (parsing action). No caso, a calculadora é interpretada e não compilada.

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
    try:
        s = raw_input('calc > ')
    except EOFError:
```

Duas técnicas para fazer o Parsing

- Predictive Parsing: utilizado em linguagens LL(1), isso é, pode-se prever a produção que deve ser utilizada olhando para o próximo símbolo.
- LR Parsing: utiliza uma DFA com pilha. Posterga a decisão de qual produção utilizar após ter lido todo o RHS da produção e ler 1 (se for LR(1)) símbolo depois.