Aula Teórica 10 (guião)

Semana de 17 a 21 de Novembro de 2025

José Carlos Ramalho

Sinopsis:

- Parsers Bottom-Up: LR0, SLR1 e outros;
- Geração de código para a VM.

- Imagens, alguns exemplos e outro conteúdo retirados da sebenta "Processamento de Linguagens: Reconhecedores Sintáticos" de José João Almeida e José Bernardo Barros, editada em Abril de 2022.

Geração de código para a VM

Considere a seguinte gramática retirada da sebenta e trabalhada na última aula:

Gramática:

```
p1: S      --> Exp '.'
p2: Exp    --> INT
p3:          | '(' Funcao ')'
p4: Funcao --> '+' Lista
p5:          | '*' Lista
p6: Lista  --> Lista Exp
p7:          |
```

Analisador Léxico (em anexo)

```
import ply.lex as lex

literals = ['(',')','*','+', '.']
tokens = ['INT']

def t_INT(t):
    r'\d+'
    t.value = int(t.value)
    return t

t_ignore = " \t\n"

def t_error(t):
    print('Caráter ilegal: ', t.value[0])
    t.lexer.skip(1)
```

```
lexer = lex.lex()
```

Analisador Sintático (com cálculo do valor)

- Com `debug=True`, o `ply.yacc` vai gerar o autómato num ficheiro de texto.

```
import ply.yacc as yacc
from sexp_lex import tokens, literals
from functools import reduce

def p_Sexp(p):
    "Sexp : Exp '.'"
    p[0] = p[1]
    print(f"Valor: {p[0]}")

def p_Exp_INT(p):
    "Exp : INT"
    p[0] = p[1]

def p_Exp_Funcao(p):
    "Exp : '(' Funcao ')'"
    p[0] = p[2]

def p_Funcao_add(p):
    "Funcao : '+' Lista"
    p[0] = sum(p[2])

def p_Funcao_mul(p):
    "Funcao : '*' Lista"
    p[0] = reduce(lambda x, y: x * y, p[2])

def p_Lista_lista(p):
    "Lista : Lista Exp"
    p[0] = p[1] + [p[2]]

def p_Lista_elem(p):
    "Lista : "
    p[0] = []

def p_error(p):
    print('Erro sintático: ', p)
    parser.success = False

# Build the parser
parser = yacc.yacc(debug=True)
```

Programa exemplo

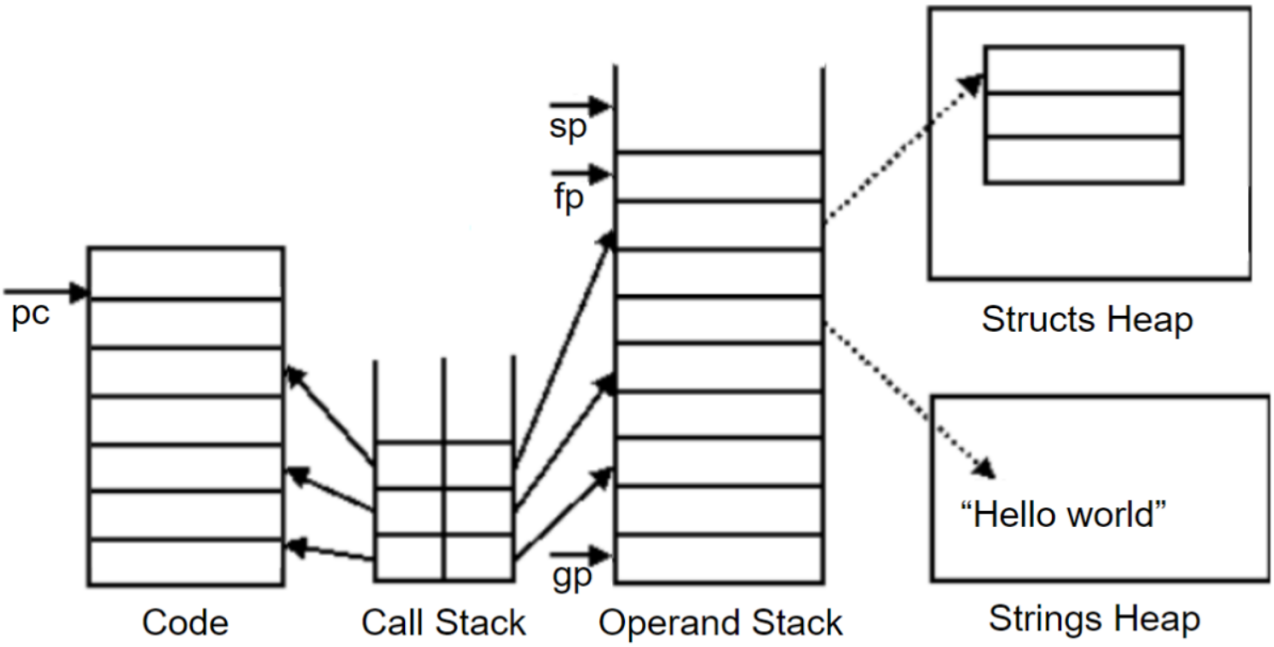
- O programa usa a variável interna do parser `success` para determinar o sucesso da análise sintática;

- Neste exemplo, está a processar-se uma linha de cada vez.

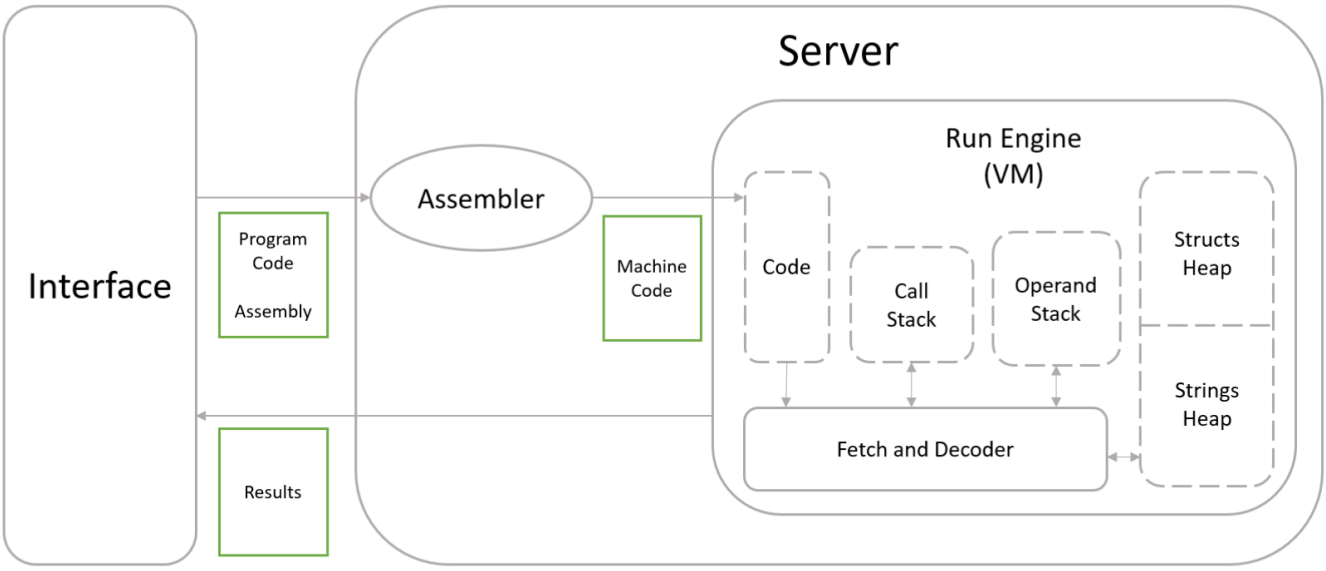
```
import sys
from sexp_sin import parser

for linha in sys.stdin:
    parser.success = True
    parser.parse(linha)
    if parser.success:
        print("Frase válida: ", linha)
    else:
        print("Frase inválida... Corrija e tente novamente!")
```

Desafio: gerar código para o cálculo na VM

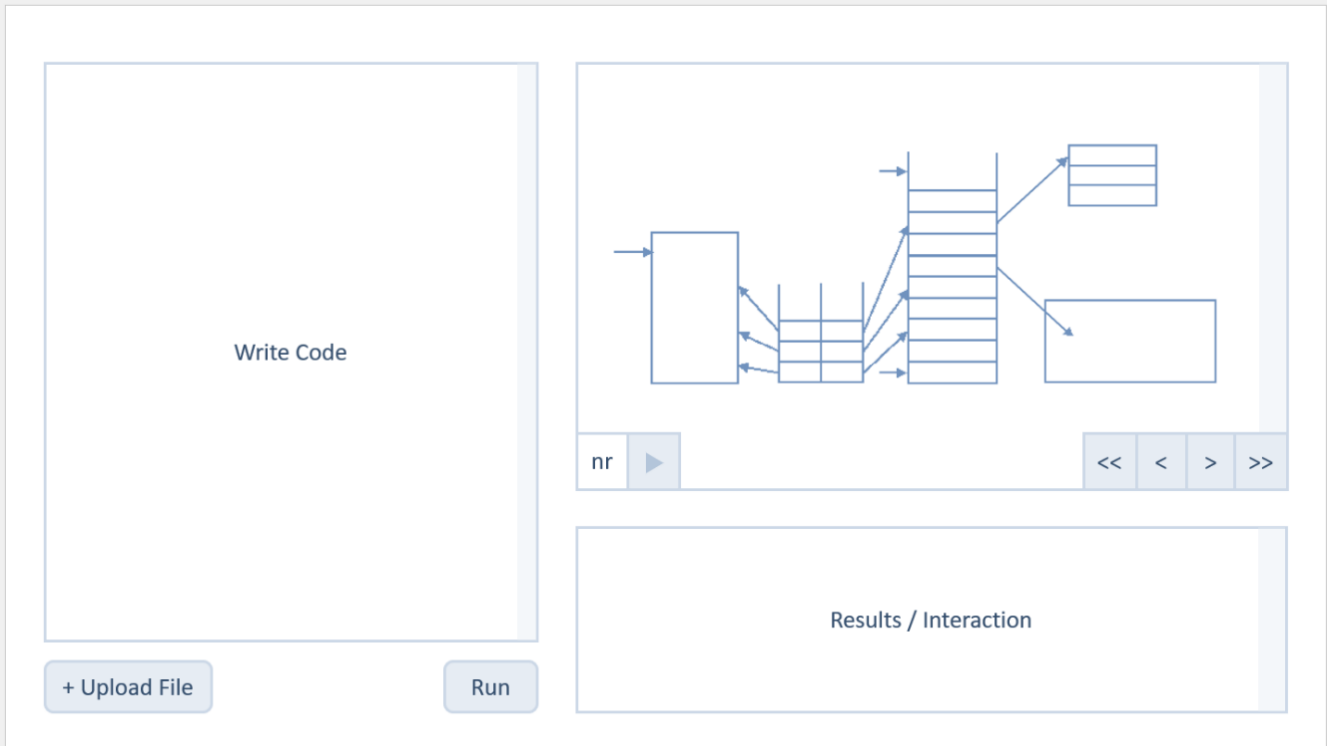
- A VM está disponível em: VM online
- Qualquer dúvida ou erro detetado reportar para: jcr@di.uminho.pt
- A VM foi pensada com a seguinte estrutura funcional:



- Mais recentemente, com a tese de mestrado de Sofia Teixeira, foi modernizada e apresenta agora a seguinte arquitetura aplicacional:



- Cuja interface contem o seguinte dashboard:



Programas exemplo

- Vamos começar por ver alguns programas exemplo

Escrever a soma de dois números, 17 e 23, no monitor

```
start
    pushi 17
    pushi 23
    add
    writei
stop
```

Escrever a soma de dois números lidos no monitor

```
start
    pushi 0
    pushi 0

    pushs "Introduza o primeiro inteiro:"
    writes
    read
    atoi
    storeg 0
    pushs "Introduza o segundo inteiro:"
    writes
    read
    atoi
    storeg 1
    pushs "A soma é: "
    writes
    pushg 1
    pushg 0
    add
    writei
stop
```

O mair de dois números lidos

```
start
    pushi 0
    pushi 0

    pushs "Introduza o primeiro número inteiro:"
    writes
    read
    atoi
    storeg 0
    pushs "Introduza o segundo número inteiro:"
    writes
    read
    atoi
    storeg 1

    pushg 1
    pushg 0
    supeq
    jz maior
    pushs "O maior é: "
    writes
    pushg 1
```

```
    writei
    jump fim
maior:
    pushs "O maior é: "
    writes
    pushg 0
    writei
fim:
    stop
```

Escrever a soma de dez números lidos no monitor

```
start
    pushi 0     // soma
    pushi 1     // n - contador

ciclo:
    pushs "Introduza o inteiro "
    writes
    pushg 1
    writei
    read
    atoi        // inteiro lido
    pushg 0
    add         // soma + lido
    storeg 0    // soma = soma + lido

    pushg 1
    pushi 1
    add
    storeg 1    // n = n + 1

    pushg 1
    pushi 10    // n <= 10
    infeq
    jz fimciclo
    jump ciclo
fimciclo:
    writeln
    pushs "A soma é:"
    writes
    pushg 0
    writei
stop
```

Desafio

- Qual o código a gerar para `(+ 3 4 5 12).`?

```
???
```

- E para `(+ (* 3 4 5) (* 12 2 3 2))`.?

```
???
```

Implementação:

```
import ply.yacc as yacc
from sexp_lex import tokens, literals
from functools import reduce

def p_Sexp(p):
    "Sexp : Exp '.'"
    print(p[1])

def p_Exp_INT(p):
    "Exp : INT"


def p_Exp_Funcao(p):
    "Exp : '(' Funcao ')'"


def p_Funcao_add(p):
    "Funcao : '+' Lista"


def p_Funcao_mul(p):
    "Funcao : '*' Lista"


def p_Lista_lista(p):
    "Lista : Lista Exp"


def p_Lista_elem(p):
    "Lista : "


def p_error(p):
    print('Erro sintático: ', p)
    parser.success = False

# Build the parser
parser = yacc.yacc(debug=True)
```

Problema (se houver tempo)

- Dado um ficheiro fonte do tipo:

```
("pl2022"
    ("aulasTeóricas")
    ("aulasPráticas")
    ("avaliação"
        ("teste"
            ["skeleton.tex" "~/templates/skeleton.tex"])
        ("recurso")
        ("especial"))
    ("web"
        ["index.html" "~/templates/index.html"]
        ("imagens"
            ["prof.jpg" "~/templates/imagens/prof.jpg"]
            ["um.jgp" "~/templates/imagens/um.jpg"])))
```

- Em que `("id" ...` especifica uma pasta a ser criada no File System com nome `id`;
- e `["destino" "origem"]` especifica um ficheiro `origem` que deve ser copiado para o `destino`.
- Especifique um parser que realize estas ações.
