Nome: Dante Honorato Navaza **Matrícula:** 2321406
Nome: Rafael Soares Estevão **Matrícula:** 2320470

1. Introdução

Este trabalho tem por objetivo desenvolver um analisador sintático para a linguagem ObsAct capaz de compilar programas escritos nesse dialeto para um código equivalente em outra linguagem de escolha (neste projeto, optou-se por Python). Em termos práticos, o sistema recebe como entrada um arquivo ObsAct, realiza a análise léxica e sintática segundo a gramática dada, aplica checagens semânticas básicas (restrições de identificadores, limites de comprimento, uso de dispositivos e variáveis) e gera como saída um programa em Python que invoca rotinas de runtime (ligar, desligar, alerta), preservando toda a lógica original de automação.

2. Descrição do Código Implementado:

O código entregue implementa um analisador sintático completo para a linguagem ObsAct utilizando a biblioteca PLY (Python Lex-Yacc). Toda a lógica está concentrada em três módulos. No parser.py ficam simultaneamente o analisador léxico, o analisador sintático, a verificação semântica básica e a própria geração de código Python. O léxico define os tokens da linguages, palavras-chave como dispositivo, set e ligar, operadores relacionais, símbolo de ponto, parênteses, chaves, identificadores e strings entre aspas, além de ignorar comentários e sinalizar caracteres inválidos ou nomes excessivamente longos. A parte sintática reproduz a gramática oficial de ObsAct; cada produção não só valida a estrutura como já emite a linha correspondente em Python, a somando no vetor output. Com isso, comandos set viram simples atribuições, ligar e desligar tornam-se chamadas às funcoes de runtime, condições se e entao viram blocos if, e alertas são mapeados para alerta(), inclusive diferentes versoes como o broadcast e com variaveis. Durante esse processo o parser também impede o uso de palavras reservadas como identificadores, controla o tamanho de nomes e mensagens, inicializa variáveis observadas que nunca receberam valor e adverte sobre dispositivos

declarados mas não utilizados. Ao detectar qualquer violação, imprime uma mensagem específica (por exemplo “mensagem não pode ser vazia” ou “parêntese sem par correspondente”) e encerra com exit(1).

O módulo lexer.py traz as quatro rotinas que o Python gerado invoca — ligar, desligar, alerta e alerta (com variaveis) todas registrando a ação em arquivo log para fins de demonstração. Por fim, main.py orquestra tudo: lê o fonte ObsAct, chama o parser, grava saida.py com o cabeçalho from runtime import * seguido das linhas acumuladas em output, e produz um relatorio.txt listando dispositivos, variáveis finais e avisos emitidos. Se o parse concluir sem erro, o programa sai com código 0; qualquer falha léxica, sintática ou semântica provoca saída 1. O projeto acompanha ainda uma pasta tests/ com casos que exercitam todas as construções da linguagem e os principais erros esperados, mostrando na prática a integração de análise léxica, parsing LR, verificação semântica pontual e geração de código que compõem o ciclo de compilação estudado na disciplina.

3. Como executar

1 - Abrir o projeto “Analisador-Sintatico-ObsAct”

2 - Se você quer usar os testes do zip tests, basta extrair o zip e chamar o arquivo desejado na main.py (com o caminho correto).

3 - Usar no console: python main.py

Saída:

Arquivo “saida.py” com com o código traduzido de ObsAct para Python.

É possível executar esse arquivo “saida.py” também, digitando o seguinte comando no console: python saida,py no console

4. Gramática (gerada automaticamente pelo parser.out da biblioteca PLY)

Obs: As regras **sublinhadas e em negrito** são regras novas extras implementadas na gramática modificada

Rule 0 S' -> program

Rule 1 program -> devices commands

Rule 2 devices -> device devices

Rule 3 devices -> empty

Rule 4 empty -> <empty>

Rule 5 device -> DISPOSITIVO DOISPONTOS ABRECHAVE ID VIRG ID FECHACHAVE

Rule 6 device -> DISPOSITIVO DOISPONTOS ABRECHAVE ID FECHACHAVE

Rule 7 commands -> command PONTO commands

Rule 8 commands -> command PONTO

Rule 9 command -> SET ID IGUAL value

Rule 10 command -> LIGAR ID

Rule 11 command -> DESLIGAR ID

Rule 12 command -> SE condition ENTAO action

Rule 13 command -> SE condition ENTAO action SENAO action

Rule 14 condition -> ID logicop value

Rule 15 condition -> ID logicop value logicbool condition

Rule 16 logicop -> MAIOR

Rule 17 logicop -> MENOR

Rule 18 logicop -> IGUALIGUAL

Rule 19 logicop -> DIF

Rule 20 logicop -> MAIORIGUAL

Rule 21 logicop -> MENORIGUAL

Rule 22 value -> NUM

Rule 23 value -> TRUE

Rule 24 value -> FALSE

Rule 25 action -> LIGAR ID

Rule 26 action -> DESLIGAR ID

Rule 27 logicbool -> E

Rule 28 logicbool -> OU

Rule 29 action -> ENVIAR ALERTA alert_content ID

Rule 30 action -> ENVIAR ALERTA alert_content PARA TODOS DOISPONTOS lista_ids

Rule 31 command -> ENVIAR ALERTA alert_content ID

Rule 32 command -> ENVIAR ALERTA alert_content PARA TODOS DOISPONTOS lista_ids

Rule 33 alert_content -> ABREPAREN STRING VIRG ID FECHAPAREN

Rule 34 alert_content -> ABREPAREN STRING FECHAPAREN

Rule 35 lista_ids -> ID VIRG lista_ids

Rule 36 lista_ids -> ID

5. Mudanças e Implementações Novas

Nossa gramática modificada da ObsAct introduz diversas funcionalidades e detalhamentos extras em relação à gramática base do PDF. A primeira mudança consiste da implementação do broadcast ENVIAR ALERTA PARA TODOS, que, conforme solicitado, gera uma linha de código python de alerta para cada dispositivo especificado no comando. Outra mudança realizada conforme solicitado foi a possibilidade de concatenar strings e/ou variáveis nos alertas (as funções de alerta que recebem variáveis como parâmetro foram renomeadas para alerta_var para evitar conflitos/sobreposições). É importante notar que também nossa gramática modificada também para adicionar restrições solicitadas como o programador só pode definir os dispositivos no começo do código, os valores atribuídos no SET podem ser apenas int ou booleano, Os nomes dos dispositivos só podem conter letras, enquanto as observations podem conter letras e números, porém só pode começar com letra, e as variáveis que nao sao sentadas são automaticamente inicializadas com 0, (também adicionamos a limitação de 100 caracteres para os nomes das variáveis e dos dispositivos).

Além disso, para expandir nosso trabalho, foram criados várias features novas extras em nosso parser.py, começando pela possibilidade de adicionar comentários no ObsAct que são levados para o Python. O sistema aceita comentários em linha (começando por #) ou multi-linha, usando três aspas (duplas ou simples), idêntico a sintaxe python. Nosso parser.py também notifica o usuário via print se alguma variável ou dispositivo declarado não foi usado no código e lista todos os dispositivos e observations (variaveis) declaradas no obsact dentro do codigo python no cabecalho os colocando em comentarios. Também adicionamos suporte

para definir os nomes dos dispositivos e observações usando acentos e exibimos uma mensagem de erro específica quando o programador tenta definir o nome de um dispositivo ou observação usando uma palavra reservada (como dispositivo). Por último, no final da execucao, o parser.py gera um relatorio.txt contendo uma lista de todos os dispositivos declarados e das observações declaradas (junto com seus valores e se elas foram declaradas manualmente pelo programador ou se foram automaticamente inicializadas com 0). Nossa gramatica tambem foi estendida para permitir o uso do operador OR em python (usando || no obsact).

6. Testes

O conjunto de arquivos .obs que acompanha o projeto cobre todo o espectro de construções aceitas pela gramática de ObsAct, reflete fielmente os exemplos do enunciado e muito mais. Os programas começam com casos elementares que apenas declaram dispositivos e disparam ações diretas (ligar / desligar), passa por atribuições com set, condições simples e compostas que podem combinar comparações tanto com o operador lógico && quanto com o ||, e culmina em diferentes variantes do comando enviar alerta: mensagem literal, mensagem concatenada a variável e broadcast para múltiplos dispositivos. Comentários foram testados nas três formas reconhecidas: linha iniciada por # e blocos multilinha entre ''' … ''' ou """ … """ para confirmar que o analisador ignora texto não executável sem afetar a numeração de linhas.

As declarações de dispositivos exercitam as restrições de nomenclatura: nomes formados só por letras (por exemplo Lampada), nomes com observation associada (Termometro , temperatura) e nomes que utilizam sublinhado e algarismos na parte da observation (temperatura1, umidade_99). Também há um dispositivo propositalmente não utilizado, DispositivoInutil, que dispara o aviso de “dispositivo declarado mas não utilizado”. O tratamento de variáveis é verificado por instruções como set temperatura = 35 ., que criam a variável e ativam a lógica de inicialização automática para qualquer observation lida em condições, mas nunca setada pelo usuário.

Para validar a geração de estruturas de decisão em Python, incluem-se exemplos com SE … ENTAO … e com ramo SENAO, além de casos em que a condição combina duas comparações por && ou por ||. As linhas se temperatura > 30 entao ligar Ventilador . e se temperatura > 30 && potencia >= 90 entao enviar alerta ("Ventilador em potência alta")

Ventilador . demonstram, respectivamente, a tradução para um bloco if simples e para um bloco que utiliza o operador lógico and, já se temp_atual > 40 || umidade < 20 entao … comprova a geração do operador or.

Os comandos de alerta foram testados isolados e em broadcast. Casos como enviar alerta ("Temperatura está alta!") Monitor . verificam a função de alerta simples, enquanto enviar alerta ("Temperatura em", temperatura) Monitor . confirma a concatenação de variáveis na mensagem. Para broadcast, instruções do tipo enviar alerta ("Aviso geral de manutenção") para todos: Monitor, Celular, asseguram que o transpilador gere múltiplas chamadas de função, uma para cada dispositivo listado.

O conjunto fecha com testes de robustez: tentativa de mensagem vazia, string que excede o limite de 100 caracteres, string sem aspas de fechamento, ponto ausente ao fim do comando e nome de dispositivo contendo dígitos (Lampada123). Nesses casos o analisador emite erros explícitos e termina com código diferente de zero, permitindo que a suíte distinga falhas reais de execuções bem-sucedidas.

Com esse leque de cenários, o transpilador foi exercitado em todas as funcionalidades exigidas e nas extensões implementadas, provando que a linguagem ObsAct é reconhecida, validada e convertida de forma robusta no código Python de saída.

7. Conclusão

O desenvolvimento do transpilador para a linguagem ObsAct utilizando PLY foi uma oportunidade valiosa para ganhar conhecimentos em compiladores, gramáticas formais e geração de código. A gramática original definida no enunciado foi ampliada para suportar funcionalidades avançadas, como condições compostas, broadcast de alertas, e validações de nomes de dispositivos e observações, além de para o uso de mensagens específicas alerta.

Este projeto mostrou como ajustes pequenos, mas importantes, na gramática e na lógica de parsing podem permitir à linguagem suportar casos mais complexos e garantir robustez na geração do código Python de saída. Foi uma experiência construtiva e essencial para aplicar, de forma prática, os conceitos de análise léxica e sintática, além de oferecer aprendizados importantes sobre a construção de analisadores sintáticos capazes de traduzir linguagens específicas para linguagens de propósito geral como Python.