UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Trabalho Prático 2 – Construção de um Tradutor de JSON para uma Linguagem de Agentes

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISCIPLINA: COMPILADORES

PROF. GLEIFER VAZ ALVES

Data (29/06/2025)

FELIPE SILVEIRA DE SOUZA PINTO

Resumo

Este trabalho descreve o desenvolvimento de um tradutor (ou compilador) projetado para converter definições de agentes, especificadas em um formato JSON declarativo, em código Python executável compatível com a biblioteca MASPY. O objetivo principal é simplificar a criação e a gestão de Sistemas Multiagentes (MAS) baseados no paradigma Crença-Desejo-Intenção (BDI). A metodologia empregada envolve o uso de ferramentas clássicas de construção de compiladores: Flex para a análise léxica e Bison para a análise sintática, gerando código C que é subsequentemente compilado com GCC. O processo de desenvolvimento abordou desafios como a configuração automatizada do ambiente de compilação através de Makefiles (incluindo gerenciamento de dependências, setup de ambiente virtual Python e limpeza de projeto) e a padronização da estrutura dos arquivos JSON de entrada para garantir compatibilidade com o tradutor. O resultado é uma ferramenta que aprimora a modularidade e a clareza na especificação de agentes BDI, facilitando o design e a implantação de sistemas complexos utilizando a biblioteca MASPY.

Palavras-chave: Sistemas Multiagentes; BDI; Compilador; Flex; Bison; MASPY; JSON.

Introdução

A automação da geração de código é um pilar fundamental no desenvolvimento de software, especialmente em domínios complexos como os Sistemas Multiagentes (MAS). O paradigma Crença-Desejo-Intenção (BDI), amplamente utilizado para modelar agentes inteligentes, frequentemente requer linguagens específicas ou frameworks como o MASPY para sua implementação em Python. A especificação de agentes diretamente em código-fonte pode, no entanto, introduzir verbosidade e dificultar a manutenção. Este trabalho descreve o desenvolvimento de um tradutor, fundamentalmente um compilador simplificado, cujo propósito é converter especificações declarativas de agentes BDI, expressas em formato JSON, em código Python executável para a biblioteca MASPY. A elaboração deste tradutor serve como uma aplicação prática dos princípios da construção de compiladores, com ênfase nas fases de análise léxica e sintática.

Objetivo

O objetivo primordial deste projeto foi conceber e implementar um tradutor para a especificação de agentes MASPY, transformando a representação declarativa em JSON para código Python funcional. Este objetivo abrangeu metas específicas ligadas aos conceitos de compiladores:

- Implementar um Analisador Léxico: Desenvolver um componente capaz de tokenizar a entrada JSON, identificando os elementos constituintes da linguagem fonte.
- Implementar um Analisador Sintático: Criar um parser que valide a estrutura gramatical da entrada JSON, construindo uma representação intermediária que guiará a geração do código Python.
- Aplicar Conceitos de Geração de Código: Definir ações semânticas associadas às regras gramaticais para produzir o código Python correspondente aos agentes MASPY.
- Automatizar o Processo de Compilação: Desenvolver um Makefile que orquestre as etapas de análise léxica, sintática e compilação do tradutor, demonstrando o ciclo completo de construção de um compilador.

Fundamentação Teórica

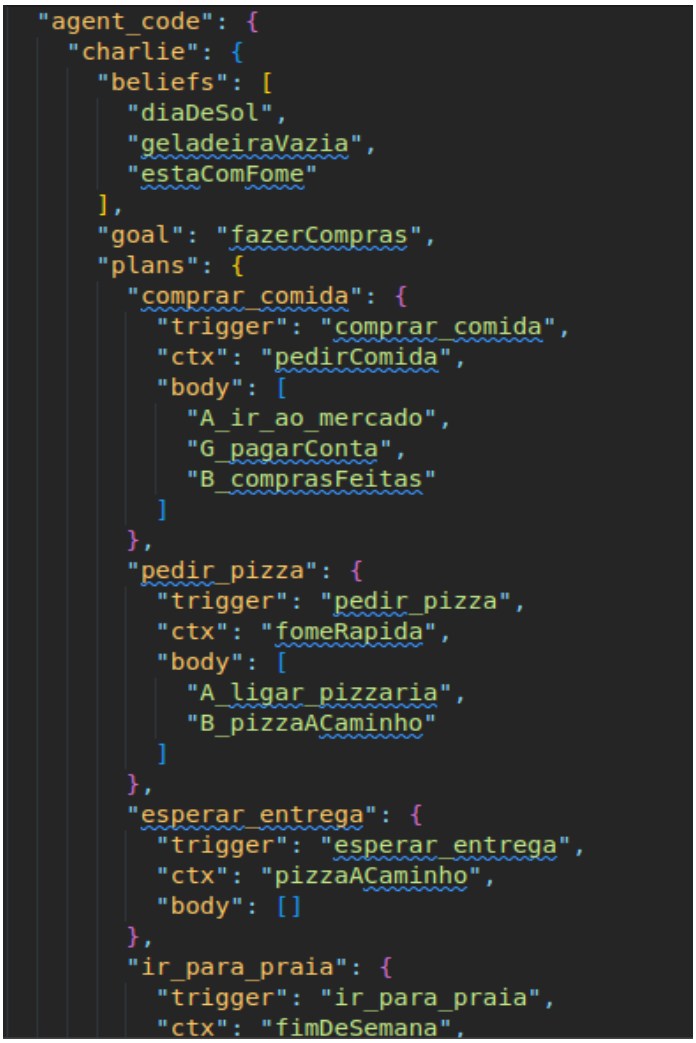
A construção de um compilador envolve fases sequenciais para transformar um código fonte em código alvo. A **análise léxica** é responsável por converter o fluxo de caracteres em tokens, utilizando expressões regulares e autômatos finitos, com ferramentas como o Flex para gerar o scanner. Em seguida, a **análise sintática** verifica a estrutura gramatical dos tokens, seguindo uma Gramática Livre de Contexto (GLC) e construindo uma árvore de análise; ferramentas como o Bison geram parsers que utilizam os conjuntos **FIRST e FOLLOW** para guiar a análise e resolver ambiguidades. A **análise semântica**, muitas vezes integrada à sintática através de ações semânticas, atribui significado às construções e, no contexto deste trabalho, realiza a **geração de código**, traduzindo diretamente a estrutura analisada para a linguagem alvo (Python). Para orquestrar essas etapas, o **GCC** é usado para compilar o código C gerado, e o `make` automatiza todo o processo de build do tradutor.

Procedimento Experimental

O processo de implementação do tradutor foi dividido em etapas que espelham as fases de um compilador, com ênfase na sua aplicação prática:

Modelagem da Linguagem Fonte (JSON): Embora JSON seja um formato de dados, para os propósitos do tradutor, ele foi tratado como uma linguagem fonte. A estrutura esperada para a especificação de agentes MASPY em JSON foi definida, com foco em uma sintaxe que pudesse ser facilmente analisada e que mapeasse diretamente para as construções da MASPY (agentes, crenças, objetivos e planos). A estrutura do 4.json serviu como modelo de referência para a gramática a ser implementada.

Imagem 1: *Arquivo JSON*



```
"agent_code": {
  "charlie": {
    "beliefs": [
      "diaDeSol",
      "geladeiraVazia",
      "estaComFome"
    ],
    "goal": "fazerCompras",
    "plans": {
      "comprar_comida": {
        "trigger": "comprar_comida",
        "ctx": "pedirComida",
        "body": [
          "A_ir_ao_mercado",
          "G_pagarConta",
          "B_comprasFeitas"
        ]
      },
      "pedir_pizza": {
        "trigger": "pedir_pizza",
        "ctx": "fomeRapida",
        "body": [
          "A_ligar_pizzaria",
          "B_pizzaACaminho"
        ]
      },
      "esperar_entrega": {
        "trigger": "esperar_entrega",
        "ctx": "pizzaACaminho",
        "body": []
      },
      "ir_para_praia": {
        "trigger": "ir_para_praia",
        "ctx": "fimDeSemana",
```

Fonte: Autoria própria (2025)

Desenvolvimento do Analisador Léxico (src/lexer.l):

Foram definidas expressões regulares para cada token relevante na sintaxe JSON: chaves de objetos, strings, delimitadores ({, }, [,], ,, :), literais booleanos (true, false), e identificadores para crenças, ações e objetivos dentro do body dos planos (e.g., A_, B_, G_).

Imagem 2: *lexer.l*

```
%option noyywrap
%option yylineno

%%

[ \t\r\n]+

"\"agent_code\""    { return AGENT_CODE; }
"\"beliefs\""       { return BELIEFS; }
"\"goal\""          { return GOAL; }
"\"plans\""         { return PLANS; }
"\"trigger\""       { return TRIGGER; }
"\"ctx\""           { return CTX; }
"\"body\""          { return BODY; }
"\"name\""          { return NAME; }

\"([^\"\\\\]|\\\\.)*\"   {
    yylval.str_val = strip_quotes(yytext);
    return STRING_LITERAL;
}

"{"                 { return OBRACE; }
"}"                 { return CBRACE; }
"["                 { return OBRACKET; }
"]"                 { return CBRACKET; }
","                 { return COMMA; }
":"                 { return COLON; }

[a-zA-Z][a-zA-Z0-9_]* {
    yylval.str_val = strdup(yytext);
    register_global_allocated_string(yylval.str_val);
    return IDENTIFIER;
}
```

Fonte: Autoria própria (2025)

Flex foi utilizado para gerar o código C do analisador léxico (lex.yy.c).

Desenvolvimento do Analisador Sintático (src/parser.y):

A gramática livre de contexto (GLC) para a especificação de agentes JSON foi escrita em notação BNF. Esta gramática especificou como os tokens (produzidos pelo lexer) deveriam ser combinados para formar as estruturas sintáticas válidas de um agente MASPY.

Caixa de texto: Notação BNF da gramática

```
<programa> ::= <objeto_json>
<objeto_json> ::= OBRACE AGENT_CODE COLON OBRACE <lista_agentes> CBRACE CBRACE
<lista_agentes> ::= <definicao_agente>
                  | <lista_agentes> COMMA <definicao_agente>
                  | (* Vazio *)
<definicao_agente> ::= STRING_LITERAL COLON STRING_LITERAL COMMA
                       <lista_beliefs> COMMA
                       GOAL COLON STRING_LITERAL COMMA
                       PLANS COLON <mapa_planos>
<lista_beliefs> ::= BELIEFS COLON OBRACKET <elementos_beliefs> CBRACKET
<elementos_beliefs> ::= <item_belief_inicial>
                  | <elementos_beliefs> COMMA <item_belief_inicial>
                  | (* Vazio *)
<item_belief_inicial> ::= STRING_LITERAL
<mapa_planos> ::= OBRACE <lista_planos> CBRACE
                  | OBRACE CBRACE
<lista_planos> ::= <definicao_plano>
                  | <lista_planos> COMMA <definicao_plano>
 <definicao_plano> ::= STRING_LITERAL COLON OBRACE
                    TRIGGER COLON STRING_LITERAL COMMA
                    CTX COLON STRING_LITERAL COMMA
                     BODY COLON OBRACKET <corpo_plano> CBRACKET CBRACE
<corpo_plano> ::= <elemento_corpo>
                  | <corpo_plano> COMMA <elemento_corpo>
                  | (* Vazio (Plano vazio) *)
<elemento_corpo> ::= STRING_LITERAL
Tokens: OBRACE, CBRACE, OBRACKET, CBRACKET, COMMA, COLON AGENT_CODE, BELIEFS,
GOAL, PLANS, TRIGGER, CTX, BODY, NAME IDENTIFIER, STRING_LITERAL, NUMBER
```

Fonte: Autoria própria (2025)

As regras gramaticais incluíram produções para objetos, arrays, pares chave-valor, e as estruturas específicas de agentes, crenças, objetivos e planos.

Ações Semânticas: Associadas a cada regra de produção, foram definidas ações em C que, ao serem executadas pelo parser, geram as strings de código Python correspondentes. Por exemplo, ao reconhecer uma definição de agente, o código para instanciar a classe Agent da MASPY é gerado. Ao reconhecer um plano, a estrutura def <nome_plano>_plan(): e as chamadas self.add_plan() são criadas.

Definição de Tokens: A seção %token em parser.y foi utilizada para declarar os tokens que seriam reconhecidos pelo lexer. Bison gera automaticamente o arquivo parser.tab.h contendo estas definições simbólicas de tokens, que é então incluído em src/lexer.l, estabelecendo a comunicação entre as fases léxica e sintática.

Integração e Compilação com Makefile:

Um Makefile foi elaborado para orquestrar as etapas de compilação.

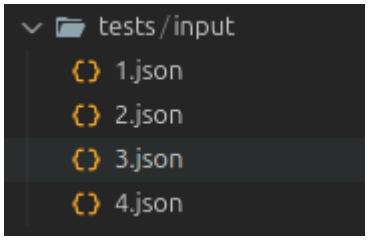
A regra para temp/json2maspy garantiu que o bison fosse executado primeiro (para gerar parser.tab.h), seguido pelo flex (que depende de parser.tab.h), e finalmente o gcc para compilar os arquivos C resultantes.

Comandos como make setup e make clean foram implementados para gerenciar o ambiente de desenvolvimento, incluindo a criação de um ambiente virtual Python e a remoção de arquivos intermediários.

Testes e Validação:

Um conjunto de arquivos JSON (ex: 1.json a 4.json) foi utilizado para testar o tradutor.

Imagem 3: *Arquivos json teste*



Fonte: Autoria própria (2025)

A diferença de comportamento entre 4.json (que funcionava) e os demais (que não funcionavam) foi crucial. Isso levou a uma análise aprofundada da gramática implementada

versus a estrutura real dos JSONs problemáticos. A correção envolveu a reestruturação dos JSONs 1.json, 2.json, e 3.json para aderirem estritamente à gramática que o tradutor foi projetado para aceitar (modelada a partir de 4.json), em vez de tentar adaptar o parser a uma gramática mais genérica que não havia sido completamente implementada.

Resultados e Discussão

O principal resultado deste trabalho é um tradutor funcional capaz de converter especificações de agentes BDI de um formato JSON padronizado para código Python executável na plataforma MASPY. O tradutor gerou com sucesso arquivos .py a partir do 4.json e, após a padronização, dos demais arquivos JSON corrigidos.

Durante o processo, diversos desafios foram encontrados e superados:

Importância da Gramática: A discrepância inicial entre os arquivos JSON e o 4.json revelou que a gramática implementada em parser.y era específica para um subconjunto da estrutura JSON. A falha dos primeiros JSONs indicou que suas estruturas (e.g., array de agentes, goals plurais, plans como arrays com triggers complexos) não eram reconhecidas pelas produções da gramática. Isso sublinha a necessidade de uma gramática clara e abrangente que capture todas as variações da linguagem fonte esperada. A correção dos JSONs de entrada para aderir à gramática existente foi um resultado prático dessa limitação.

Conflitos de Parsing (Implícitos): Embora o Bison geralmente reporte conflitos de shift/reduce ou reduce/reduce, a falha do tradutor em processar certos JSONs, mesmo com gramáticas aparentemente válidas (mas diferentes da implementada), sugere que a gramática do parser.y não era genérica o suficiente para todas as variações de JSON ou que estava otimizada para uma forma específica (a do 4.json). A ausência de erros explícitos de flex ou bison na compilação do próprio tradutor, mas sim no tempo de execução com certas entradas, reforça que a questão era de linguagem aceita versus linguagem esperada.

Ações Semânticas: A eficácia do tradutor dependeu diretamente das ações semânticas implementadas em C dentro do parser.y. A conversão de estruturas JSON para chamadas de função MASPY (e.g., self.add_belief, @pl) exigiu um mapeamento preciso e cuidadoso.

Depuração Integrada: A persistência de erros como "flex: não foi possível abrir -o" ou erros do m4, mesmo com o Makefile revisado, indicou que, às vezes, problemas sutis de ambiente ou de interpretação da linha de comando podem obscurecer a raiz de um erro, que, neste caso, poderia estar em uma pequena falha na definição de lexer.l ou em caracteres invisíveis, mesmo quando a compilação manual parecia funcionar.

O tradutor desenvolvido cumpre seu objetivo de simplificar a especificação de agentes BDI, validando a aplicação prática dos conceitos de análise léxica e sintática em um cenário real de engenharia de software.

Imagem 1: M*akefile*

```
all: $(EXECUTABLE)
    @echo "--- Compilacao do tradutor concluida. ---"
    @echo "Execute o comando ./$(EXECUTABLE) tests/input/ 1.json"
    @echo "Depois execute python maspy_output.py"

$(EXECUTABLE): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $(EXECUTABLE)

$(TEMP_DIR)/tradutor_utils.o: $(UTILS_C) $(UTILS_H)
    mkdir -p $(TEMP_DIR)
    $(CC) $(CFLAGS) $(INCLUDES) -c $(UTILS_C) -o $(TEMP_DIR)/tradutor_utils.o

$(TEMP_DIR)/main.o: $(MAIN_C) $(UTILS_H)
    $(CC) $(CFLAGS) $(INCLUDES) -c $(MAIN_C) -o $(TEMP_DIR)/main.o

$(TEMP_DIR)/parser.tab.o: $(PARSER_C) $(PARSER_H) $(UTILS_H)
    $(CC) $(CFLAGS) $(INCLUDES) -c $(PARSER_C) -o $(TEMP_DIR)/parser.tab.o

$(TEMP_DIR)/lex.yy.o: $(LEXER_C) $(UTILS_H)
    $(CC) $(CFLAGS) $(INCLUDES) -c $(LEXER_C) -o $(TEMP_DIR)/lex.yy.o

$(PARSER_C) $(PARSER_H): src/parser.y
    mkdir -p $(TEMP_DIR)
    bison -d src/parser.y -o $(PARSER_C)

$(LEXER_C): src/lexer.l
    flex -o $(LEXER_C) src/lexer.l

setup:
    @echo "Configurando ambiente virtual Python..."
    test -d venv || python3 -m venv venv
    @echo "Ambiente virtual criado/existente em venv"
    @echo "AVISO IMPORTANTE: Para usar o ambiente virtual em seu terminal, voce DEVE ativa-lo manualmente:"
    @echo "  source venv/bin/activate"
    @echo "Certifique-se de instalar as dependencias Python (como 'maspy') dentro do ambiente virtual, se ainda nao o fez:"
    @echo "  venv/bin/pip install maspy"

clean:
    rm -rf $(TEMP_DIR)/* $(TEMP_DIR)
    rm -f tests/output/*.py
    rm -f src/parser.tab.c src/parser.tab.h src/lex.yy.c
```

Fonte: Autoria própria (2025)

Observação do conteúdo

A análise dos diferentes arquivos JSON (1.json a 4.json) forneceu uma valiosa oportunidade de observar como a definição da gramática da linguagem fonte em um compilador impacta diretamente sua capacidade de processar entradas variadas. O 4.json, sendo o único inicialmente funcional, revelou implicitamente a gramática exata para a qual o tradutor foi projetado. Este JSON emprega um conjunto específico de produções: agent_code como a raiz de um objeto de agentes, nomes de agentes como chaves diretas (não em um array de objetos com campo name), um único goal por agente, e uma estrutura simplificada de plans onde os gatilhos (trigger) e contextos (ctx) são strings simples, e o body consiste em uma lista de strings com prefixos semânticos (A_, B_, G_). Em contraste, os demais JSONs apresentavam estruturas mais aninhadas e pluralizadas (ex: agents como array, goals como array, trigger como objeto complexo), que não eram contempladas pelas produções da gramática implementada. Essa observação foi crucial para entender as limitações do parser atual e para realizar as adaptações necessárias nos arquivos de entrada, reforçando a ideia de que a linguagem a ser traduzida é definida pela sua gramática aceita, e não apenas pela intenção de seu designer.

Conclusão

Este projeto demonstrou com sucesso a aplicação dos princípios fundamentais da disciplina de Compiladores para a construção de um tradutor de JSON para código Python compatível com a biblioteca MASPY. Através do uso de Flex para análise léxica e Bison para análise sintática, foi possível construir um compilador capaz de interpretar uma especificação declarativa de agentes BDI e gerar código executável. A experiência reforçou a importância da definição precisa da gramática da linguagem fonte, da correta integração entre as fases de compilação e da depuração iterativa de erros, incluindo aqueles mais sutis relacionados à interpretação do ambiente. O tradutor finalizado oferece uma abordagem mais modular e legível para a criação de agentes MASPY, validando a eficácia das ferramentas e teorias de compiladores no desenvolvimento de soluções para sistemas complexos. Futuros trabalhos poderiam focar na expansão da gramática para suportar toda a riqueza da biblioteca MASPY, na implementação de mecanismos robustos de recuperação de erros e na otimização do código Python gerado.

Referências

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. Compilers: Principles, Techniques, and Tools. 2. ed. Boston, MA: Pearson Addison-Wesley, 2007.

CROCKFORD, Douglas. JSON.org. Disponível em: https://www.json.org/json-en.html. Acesso em: 3 jul. 2025.

GNU PROJECT. GNU Bison. Disponível em: https://www.gnu.org/software/bison/manual/. Acesso em: 3 jul. 2025.

LACA-IS. MASPY - A Multi-Agent System for Python. [S. l.]: GitHub, [2022]. Disponível em: https://github.com/laca-is/MASPY. Acesso em: 3 jul. 2025.

PAXSON, Vern et al. Flex: The Fast Lexical Analyzer. [S. l.]: Flex Development Team. Disponível em: https://flex.sourceforge.io/manual/. Acesso em: 3 jul. 2025.