Gramática e Validação da DSL **DroneGeneric**

Este documento técnico explica a definição da gramática **droneGeneric** (um arquivo ANTLR) e a arquitetura de validação implementada no plugin Java **DroneGenericPlugin**. A DSL *droneGeneric* é uma linguagem específica de domínio voltada a missões de drones, com uma sintaxe estruturada em seções (cabeçalho, tipos, variáveis, instruções) e suporte a expressões, vetores e arrays. O plugin **DroneGenericPlugin** utiliza o parser gerado pelo ANTLR para analisar programas nessa linguagem e então aplica vários *visitors* (visitantes) em sequência para verificar regras semânticas, acumular erros e reportá-los ao desenvolvedor. O público-alvo deste documento são desenvolvedores que pretendem **manter ou estender** essa DSL de drones, portanto focaremos nos detalhes relevantes de sintaxe da gramática e nas estratégias de validação semântica no código Java.

Gramática **droneGeneric** (ANTLR)

A gramática **droneGeneric** é definida em um arquivo `.g4` usando ANTLR (ANTLR v4). Ela especifica tanto as regras de parser (sintáticas) quanto as regras léxicas (tokens) da linguagem. Conforme as convenções do ANTLR, as regras de parser começam com letra minúscula e definem a estrutura hierárquica da linguagem, enquanto as regras léxicas (tokens) são escritas em letra maiúscula e definem os símbolos básicos, como identificadores e números. Normalmente, coloca-se as regras de parser primeiro e as léxicas ao final do arquivo. A gramática segue um estilo *top-down*, começando pela estrutura geral do programa e detalhando cada seção; esse método é adequado quando já se conhece os principais blocos da linguagem. Abaixo, explicamos a estrutura geral da gramática e seus principais componentes:

- **Estrutura Geral (Seções do Programa):** Um arquivo/programa da DSL **droneGeneric** é dividido em seções bem definidas, em ordem fixa. A gramática possui uma regra inicial (por exemplo, `program`) que reconhece todo o arquivo e delega para sub-regras correspondentes a cada seção principal. As seções típicas são:

    - **Header (Cabeçalho):** Contém informações do programa, como nome ou versão da linguagem. A sintaxe esperada inclui palavras-chave específicas. Por exemplo, a gramática pode exigir a palavra-chave `PROGRAM` seguida de um identificador (nome do programa), e a indicação da versão da linguagem com `LANGUAGE VERSION` seguido de um número de versão. Uma declaração de cabeçalho válida poderia ser: `PROGRAM MeuPlano LANGUAGE VERSION 1.0`. Essa seção é opcional em alguns casos (a depender se a versão pode defaultar), mas quando presente deve seguir exatamente o formato esperado – essas palavras-chave são tokens reservados da gramática.

    - **TYPES (Tipos):** Secão obrigatória onde são definidos os tipos utilizados no programa. Inicia com a palavra-chave reservada `TYPES` e geralmente contém uma lista de definições de tipo customizado. Cada definição associa um nome de tipo a um *kind* básico da DSL. Por exemplo, pode-se definir tipos escalar, vetor ou array. A sintaxe de definição de tipo pode ser algo como `<NomeTipo> = scalar;` (para definir

um tipo escalar simples), `<NomeTipo> = vector;` (para um tipo vetor) ou `<NomeTipo> = array of <OutroTipo>;` (definindo um tipo array cujos elementos são de `<OutroTipo>`). Assim, o desenvolvedor pode criar aliases de tipos para usar nas variáveis. Exemplo:

```
TYPES:
    Number = scalar;
    Point  = vector;
    Path   = array of Point;
```

Isso define `Number` como um tipo escalar (por exemplo, número simples), `Point` como um vetor (por exemplo, coordenada com vários valores) e `Path` como um array de `Point` (lista de coordenadas formando um caminho).

- **VARIABLES (Variáveis):** Seção opcional que declara variáveis utilizadas nas instruções. Inicia com a palavra-chave `VARIABLES` seguida de declarações de variáveis, uma por linha. Cada declaração especifica o nome da variável, seu tipo e possivelmente um valor inicial. Uma forma comum é `nomeVar: Tipo = <expressão>;`. Por exemplo: `altitude: Number = 100;` declara a variável `altitude` do tipo `Number` (escalar) com valor inicial 100. A gramática permite omitir a parte de inicialização (`= ...`) se a variável não tiver valor inicial, embora na prática possa ser recomendado sempre inicializar. Nessa seção, a sintaxe é validada pela gramática (espera-se um identificador, depois `:` e um tipo válido, depois opcional `=` e uma expressão, terminando com ponto e vírgula). A verificação se o tipo de fato existe ou se o valor é compatível é feita posteriormente na fase de validação semântica (pelo plugin).

- **INSTRUCTIONS (Instruções):** Seção obrigatória contendo a sequência de instruções (comandos e expressões) que definem o comportamento do drone. Inicia com a palavra-chave `INSTRUCTIONS` e inclui uma lista de instruções terminadas por `;`. As instruções podem ser de diversos tipos definidos pela gramática: por exemplo comandos simples (`TAKEOFF;`, `LAND;`), comandos com parâmetros (`MOVE destino;` onde `destino` é uma variável do tipo Point), atribuições (`x = 5+2;` para alterar o valor de uma variável) e possivelmente estruturas de controle simples (condicionais ou loops, se a DSL suportar – não mencionado explicitamente, então assumimos que não há estruturas complexas na versão *generic*). A gramática define cada instrução possível como uma alternativa dentro da regra de instruções. Assim, ela reconhece sintaticamente sequências válidas de comandos. Exemplo de instruções válidas:

```
INSTRUCTIONS:
    TAKEOFF;
    MOVE destination;
    altitude = altitude + 10;
    LAND;
```

Aqui vemos um comando de decolagem, um comando de movimento para um ponto representado pela variável `destination`, uma instrução de atribuição somando 10 à variável `altitude`, e um comando de pouso. A gramática garante que apenas instruções suportadas sejam aceitas (por exemplo, evitar palavras desconhecidas ou sintaxes malformadas nesta seção).

Expressões, Vetores e Arrays: A gramática **droneGeneric** suporta expressões aritméticas e literais estruturados (vetor e array) dentro do código (por exemplo, em valores de variáveis ou parâmetros de instruções). As *expressões* seguem regras de precedência e associatividade definidas na gramática, geralmente usando produção recursiva. Por exemplo, uma estratégia típica é ter uma regra `expr` com sub-regras para operações de soma/subtração e multiplicação/divisão, permitindo associatividade à esquerda. Uma simplificação ilustrativa de regra de expressão poderia ser:

```
expr
  : expr ('/'|'*') expr      # MulDivExpr
  | expr ('+'|'-') expr      # AddSubExpr
  | NUMBER                   # LiteralExpr
  | ID                       # VarRefExpr
  | '(' expr ')'             # ParenExpr
  ;
```

Essa construção (similar à mostrada acima) permite analisar expressões como `a + b * 2` respeitando a precedência (no exemplo acima, `*` e `/` seriam avaliados antes de `+` e `-`). A DSL droneGeneric suporta pelo menos operações aritméticas básicas com números e possivelmente entre vetores (se fizer sentido somar vetores, etc., o que dependeria da definição semântica – a gramática em si apenas reconhece a sintaxe). As *expressões* podem incluir literais numéricos, referências a variáveis (ID) e sub-expressões entre parênteses. A avaliação ou a verificação de tipos dessas expressões não é papel da gramática, mas a estrutura sintática é garantida (por exemplo, a gramática impede algo como `x + * 3`, que seria um erro de sintaxe).

A gramática define também notações literais para **vetores** e **arrays**:

- Um **vetor** na DSL é tratado como um tupla de valores numéricos, delimitada por símbolos específicos (por exemplo, parênteses). Por exemplo, um vetor 3D poderia ser escrito como `(10, 20, 5)`. A gramática provavelmente exige ao menos dois elementos separados por vírgula para distinguir de um escalar isolado. Assim, um *literal de vetor* pode ser reconhecido por uma regra como: `vectorLiteral : '(' expr (',' expr)+ ')' ;` indicando um parêntese aberto, seguido de duas ou mais expressões separadas por vírgulas e um parêntese fechado. Esse literal produz um valor de *kind* vetor.
- Um **array** (lista) é representado por valores entre colchetes `[...]`. A sintaxe de um *literal de array* pode seguir o padrão de listas do JSON, por exemplo: `arrayLiteral : '[' expr (',' expr)* ']' ;`. Isso permite escrever uma lista de valores ou mesmo de vetores. Exemplo: `[1, 2, 3]` seria um array de números escalares, enquanto `[ (0,0,0), (10,20,5) ]` seria um array cujos elementos são vetores (no caso, um array de pontos coordenados). A gramática trata a sintaxe – garantindo que haja vírgulas entre elementos e fechando corretamente os colchetes – mas **não** garante que todos os elementos tenham o tipo correto ou o tamanho consistente (essa verificação é feita pelo plugin na etapa de validação semântica).

Regras Léxicas (ID, NUMBER, etc): No final do arquivo de gramática, definem-se os tokens básicos. Destacam-se:

- **ID (Identificador):** Representa nomes de variáveis, tipos ou do programa. Tipicamente é definido como uma sequência de letras (e possivelmente dígitos e underscore) que não começa com dígito para não conflitar com números. Por exemplo, uma regra comum seria: `ID : [A-Za-z_] [A-Za-z0-9_]* ;` que aceita uma letra ou _ inicial, seguida de qualquer combinação de letras, dígitos ou underscore. Assim, tokens como `altitude`,

`Point1` ou `_temp` seriam reconhecidos como ID. A gramática **droneGeneric** reserva certas palavras-chave (como `PROGRAM`, `TYPES`, `VARIABLES`, `INSTRUCTIONS`, possivelmente nomes de comandos como `TAKEOFF`, etc.), então o lexer deve diferenciá-las de IDs comuns. Em ANTLR, isso é feito definindo tokens literais ou regras específicas antes da regra ID, já que o lexer escolhe a primeira regra que casar com a entrada. Por exemplo, se há uma palavra-chave `TAKEOFF`, ela seria definida explicitamente (`TAKEOFF : 'TAKEOFF';`) antes da definição genérica de ID, garantindo que ao ler "TAKEOFF" o lexer produza um token TAKEOFF de palavra-chave em vez de um ID arbitrário.

- **NUMBER (Número):** Representa literais numéricos na linguagem. A definição abrange números inteiros e possivelmente de ponto flutuante. Uma regra simples poderia ser `NUMBER : [0-9]+ ('.' [0-9]+)? ;` para permitir um inteiro ou um decimal com parte fracionária. Alguns gramáticas também permitem um sinal opcional na frente (`'-'?`), mas é comum tratar o sinal como parte da expressão (operador unário) em vez de embuti-lo no token. Assim, `123` e `3.14` seriam tokens NUMBER válidos. O trecho de gramática a seguir exemplifica como poderia ser definido um número com parte decimal opcional usando fragmentos para dígitos:

  ```
  fragment DIGIT : [0-9] ;
  NUMBER : DIGIT+ ('.' DIGIT+)? ;
  ```

  Isso aproveita um fragmento `DIGIT` para evitar repetir o padrão de dígito e permite números como `10` ou `10.5`. (Obs: A gramática pode optar por usar `','` como separador decimal dependendo do contexto local, mas aqui assumimos ponto como separador decimal padrão.)

- **Outros tokens:** A gramática também define tokens para símbolos de pontuação e operadores: por exemplo, símbolos como `(`, `)`, `[`, `]`, `:`, `;`, `,`, operadores aritméticos `+ - * /` e comparadores (se houver), e possivelmente comentários ou espaços em branco. Espaços em branco e quebras de linha tipicamente são consumidos por uma regra especial no lexer, por exemplo `WS : [ \t\r\n]+ -> skip ;`, que indica para ignorar (skip) qualquer sequência de espaços, tabs ou quebras. Assim, a gramática fica livre de ter que mencionar whitespace entre tokens. Se a DSL suporta comentários (não citado na pergunta, mas comum), haveria tokens para comentário de linha ou bloco com ação `-> skip` para ignoralos também.

Resumo da Gramática: Em suma, a gramática **droneGeneric** estabelece a *sintaxe* da linguagem de forma completa: a ordem e presença das seções (cabeçalho, tipos, variáveis, instruções), as construções válidas dentro de cada seção (por exemplo, declarações de tipo/variável, comandos), e os elementos léxicos básicos (identificadores, números, símbolos). Ela assegura que um arquivo fonte que passe por ela tem uma estrutura sintática válida. Entretanto, muitas regras de consistência da linguagem (por exemplo, usar uma variável não declarada, atribuir um vetor a um tipo escalar, omitir uma seção obrigatória) **não** são verificadas apenas pela gramática para manter a simplicidade e evitar complicações de contexto no parser. Essas regras semânticas são tratadas na fase seguinte, pelo plugin, conforme descrito a seguir. Vale notar a filosofia de projeto aqui: a gramática (parser) cuida **do correto formato** e sequência dos elementos (sintaxe), enquanto a lógica extra implementada em Java cuida **do significado e coerência** (semântica). Isto segue a recomendação comum em compiladores/DSLs de deixar o parser aceitar construções sintaticamente corretas mesmo que potencialmente sem sentido, delegando a validação de sentido ao passo posterior.

Funcionamento do **DroneGenericPlugin** (Validação Semântica)

O **DroneGenericPlugin** é um plugin Java que integra o parser gerado pelo ANTLR para a gramática droneGeneric e implementa a lógica de validação semântica da DSL. Em outras palavras, após o texto do programa ser convertido em uma *parse tree* pela gramática, o plugin verifica se, do ponto de vista de regras de negócio da linguagem, o programa é válido. Isso inclui checar versões, tipos, declarações, uso correto de variáveis e instruções. A arquitetura é composta por *visitors* (visitantes) que percorrem a árvore sintática. Cada visitor foca em um aspecto específico da validação, acumulando erros conforme encontra problemas. Essa abordagem modular em múltiplos passes torna o código de validação mais organizado e facilita futuras extensões (por exemplo, adicionar novos tipos de checagens sem mexer em um monolito único).

Integração com ANTLR (Análise Sintática)

Na inicialização, o plugin utiliza o ANTLR para analisar o código-fonte da DSL:

1. O texto de entrada (arquivo do script de drone) é passado para um lexer gerado (por exemplo, `DroneGenericLexer`), que divide o texto em tokens de acordo com as regras léxicas definidas.
2. Esses tokens alimentam o parser gerado (`DroneGenericParser`), que tenta reconhecer os tokens conforme as regras da gramática. Ele produz uma *árvore de parse* (conhecida também como AST – árvore sintática abstrata) representando a estrutura hierárquica do programa. Se houver erros de sintaxe (violação das regras da gramática), o parser aciona seus mecanismos de erro. O plugin pode fornecer um *ErrorListener* customizado para capturar mensagens de erro sintático e reportá-las de forma mais amigável ao usuário, mas por padrão o ANTLR reporta a linha/coluna e token inesperado. Geralmente, porém, a maior parte da validação de sintaxe já foi garantida pela gramática, então se o parser produziu a árvore, significa que a estrutura básica do programa está correta.
3. Com a parse tree construída (nó raiz representando, por exemplo, `program` e nós filhos para header, types, etc.), o plugin então inicia a fase de validação semântica. Em vez de tentar incorporar todas as regras de validação no próprio parser (o que seria complexo ou impossível, pois ANTLR lida principalmente com sintaxe), optou-se por implementar *visitors*. Um visitor é uma classe Java gerada a partir da gramática (por ANTLR, que gera uma interface `DroneGenericVisitor` e/ou uma classe base `DroneGenericBaseVisitor` com métodos visit para cada regra). O plugin define subclasses desses visitantes para percorrer a árvore e checar condições adicionais. Essa separação segue a prática comum: "*parse first, then validate semantics*".

Visitantes de Validação Semântica

Os visitantes (*Validation Visitors*) percorrem a árvore abstrata do programa e verificam diferentes aspectos. No **DroneGenericPlugin**, existem quatro classes principais de visitantes, cada uma focada em uma parte do arquivo DSL ou de suas regras de consistência:

- **ProgramLanguageVersionVisitor:** Este visitor verifica elementos de alto nível do programa, tipicamente o cabeçalho. Seu foco principal é validar a versão da linguagem e presença dos blocos essenciais. Ao visitar o nó do cabeçalho (caso exista), ele extrai o número de versão após a palavra-chave `LANGUAGE VERSION`. Então compara com a(s) versão(ões) suportada(s) pelo plugin. Por exemplo, se a DSL suporta versão "1.0" e o cabeçalho indicar `VERSION 2.0`, o visitor registra um erro indicando versão de linguagem não suportada. Se o cabeçalho estiver ausente (e se for opcional), o plugin pode assumir um valor padrão ou emitir um aviso, dependendo do design. Além disso, o ProgramLanguageVersionVisitor checa a estrutura global: por exemplo, verifica se as seções obrigatórias **TYPES** e **INSTRUCTIONS** aparecem no parse tree. Caso alguma dessas seções esteja faltando, ele gera um erro de validação (por exemplo: *"Seção TYPES ausente: é necessário definir ao menos um tipo."*). Essa responsabilidade pelos *Required Fields* pode estar aqui ou em um nível geral – de qualquer forma, antes de prosseguir com validações mais detalhadas, o plugin garante que a forma macro do programa faz sentido (versão correta e blocos requeridos presentes). Todos os erros encontrados são armazenados (acumulados) para posterior relatório.

- **TypesValidationVisitor:** Após garantir que a seção de tipos existe, este visitor percorre todas as definições de tipo dentro de `TYPES`. Ele constrói uma espécie de *tabela de símbolos* dos tipos declarados. Para cada definição, ele registra o nome do tipo e a sua classificação *kind*. A classificação refere-se a se o tipo é escalar, vetor ou array. Por exemplo, se encontra `Number = scalar`, ele registra `"Number"` com *kind* = scalar. Se encontra `Point = vector`, registra `"Point"` com kind = vector. Se encontra `Path = array of Point`, registra `"Path"` com kind = array e também guarda que o tipo base de Path é Point. Durante esse processo, o visitor executa validações semânticas importantes:

    - **Tipos duplicados:** se mais de um tipo for declarado com o mesmo nome, isso é um erro. O visitor checa e reporta (*"Tipo 'X' definido mais de uma vez"*).
    - **Referências de tipo desconhecidas:** no caso de tipos array, ele verifica se o tipo base existe. Por exemplo, em `Path = array of Point`, ele verifica se `Point` foi definido em alguma linha (pode ser antes ou depois, dependendo se a gramática permite ordem livre; geralmente permite ordem livre, então o visitor poderia primeiro coletar todos os nomes de tipo em uma passada e depois verificar referências). Se encontrar `array of UndefinedType`, ele registra erro (*"Tipo 'UndefinedType' não definido, referenciado em definição do tipo 'Algo'"*).
    - **Categorias válidas:** se a DSL limitou os *kinds* a exatamente essas três palavras-chave (scalar, vector, array), o visitor garante que não haja nada diferente. Como a gramática provavelmente já restringe isso sintaticamente, essa checagem pode ser trivial. Contudo, se houvesse parâmetros extras (como tamanho de vetor, etc., que não parecem existir aqui), ele os validaria também.
    - **Outras regras:** Poderia também evitar definições recursivas indevidas (ex: `A = array of A` poderia causar dependência circular – se considerado um erro, o visitor identificaria e avisaria).
    - O resultado principal desse visitor é a construção de um repositório de tipos válidos que será usado nos passos seguintes. Ele populará, por exemplo, uma estrutura `Map<String, TypeInfo>` onde `TypeInfo` inclui o *kind* (scalar/vector/array) e possivelmente o tipo base (para arrays) ou dimensão (se vetores tivessem dimensão fixa, mas não parece haver especificação disso na sintaxe – provavelmente vetor é abstrato aqui, representando um tupla de N números, sem amarrar N no tipo).
    - Quaisquer erros são acumulados na lista global de erros. O visitor não interrompe no primeiro problema; ele tenta validar todos os tipos para reportar múltiplos problemas de uma vez.

- **VariablesValidationVisitor:** Este visitor foca na seção `VARIABLES`. Ele itera por cada declaração de variável e utiliza a tabela de tipos construída anteriormente para verificar coerência:

    - **Tipo declarado existe:** para cada variável com tipo T, verifica se T está presente na tabela de tipos. Se não estiver, é um erro (*"Variável 'v' declara tipo desconhecido 'T'"*). Isso previne uso de tipos não definidos.

    - **Nome de variável duplicado:** verifica se o nome da variável já foi declarado antes (na mesma seção). Se sim, erro (*"Variável 'x' declarada mais de uma vez"*).

    - **Verificação de valor inicial:** se a variável possui uma expressão de inicialização, o visitor valida se essa expressão é compatível com o tipo da variável. Por exemplo, se `altitude: Number = 100;`, espera-se que `100` seja um NUMBER literal – ok (scalar <- scalar). Se `position: Point = (5,5,0);`, espera-se que o literal seja um vetor compatível (vetor de números) já que `Point` foi definido como vector. Se alguém escrevesse `distance: Number = (5,5);`, o visitor detectaria que está atribuindo um vetor ao tipo escalar Number e marcaria erro (*"Inicialização de 'distance' com tipo incompatível: esperado scalar, obtido vector"*). Essa verificação de tipo da expressão pode exigir determinar o *kind* resultante da expressão:

        - Literais numéricos produzem scalar.
        - Literais de vetor produzem vector.
        - Literais de array produzem array.
        - Operações aritméticas: aqui, o plugin pode definir regras simples, por exemplo: se ambos operandos são scalar, resultado é scalar; se ambos são vector, talvez resultado vector (se as dimensões batem, embora a DSL não especifica dimensões explicitamente); se um é scalar e outro vector, ou qualquer mistura inadequada, pode ser considerado erro. Provavelmente, para simplicidade, definiu-se que só se pode operar escalares com escalares – operações entre vetores talvez não sejam suportadas a nível de DSL genérico (a não ser que definam semântica de soma vetorial). De qualquer forma, o VariablesValidationVisitor pode checar subexpressões: se encontra uma expressão `expr` na inicialização, ele pode recorrer aos tipos: verificar cada operando se é coerente e inferir um tipo resultante para `expr` para então comparar com o tipo da variável. Essa é uma forma de *type checking* estático.
        - No caso de arrays, uma variável do tipo array requer que o literal de inicialização (se presente) seja um array literal cujos elementos sejam do tipo base correto. Ex: `route: Path = [ (0,0,0), (5,5,5) ];` – o visitor confirmaria que `Path` é *array of Point*, e então verificaria que cada elemento do literal array é um vector compatível com `Point`. Se encontrasse um elemento não compatível (um número solto ou um vetor de dimensão errada, caso dimensionassem), reportaria erro.

    - Esse visitor também registra cada variável em uma tabela de símbolos de variáveis, associando o nome ao seu tipo (ou referência ao TypeInfo correspondente). Assim, ao final, teremos um *Symbol Table* de variáveis disponíveis para uso nas instruções.

- Todos os problemas identificados nesta fase são adicionados à lista de erros. Importante: se um tipo estava indefinido e já gerou erro no passo anterior, usar esse tipo em variáveis pode disparar novos erros. O plugin, dependendo da implementação, ou repete o erro de tipo desconhecido para cada variável (menos desejável), ou ignora erros cascata (pode checar “se tipo não existe, já marcamos erro, pular outras verificações nessa var”).

- **InstructionsValidationVisitor:** Este é o visitor final que analisa a seção `INSTRUCTIONS`. Ele verifica cada instrução da sequência de acordo com as regras semânticas:

  - **Uso de variáveis:** sempre que a instrução refere-se a uma variável (por nome), verifica se essa variável foi declarada na seção `VARIABLES`. Por exemplo, uma instrução `MOVE destination;` requer que `destination` exista. Se não existe, erro (*"Variável 'destination' não declarada"*). Similarmente, em uma atribuição `x = expr;`, verifica-se se `x` existe e também verifica o tipo do lado direito `expr` contra o tipo da variável `x` (semelhante à checagem já feita em inicialização de variáveis). Assim, `x = y;` só é válido se `y` for declarado e o tipo de `y` for igual ou compatível com o de `x` (provavelmente devem ser exatamente o mesmo tipo na DSL, pois não há conversões automáticas).

  - **Parâmetros de comandos:** para comandos específicos do domínio, o visitor conhece que tipo de argumento esperam, e valida. Por exemplo, supondo que `MOVE <dest>` espera um ponto (vector) como destino: se o código for `MOVE altitude;` onde `altitude` é Number (scalar), isso é semanticamente incorreto. O visitor detecta e gera erro (*"Comando MOVE esperando variável do tipo Point, mas recebeu Number"*). Assim, cada instrução definida na DSL tem regras:

    - comandos sem parâmetro (e.g. `TAKEOFF;`, `LAND;`) são triviais – apenas verificar que realmente são tokens válidos (o parser já garantiu) e possivelmente qualquer pré-condição (ex: se exigisse que altitude inicial fosse definida antes de decolar, mas isso já seria lógica de negócio fora do escopo da linguagem).
    - comandos com parâmetros variáveis (e.g. `MOVE x;`, `SET_SPEED n;` etc): checar existência e tipo do parâmetro.
    - atribuições e operações: checar tipos compatíveis como mencionado.
    - eventuais estruturas de controle (por ex, se existissem `IF cond THEN ...` na linguagem): verificar que `cond` seja booleano, etc. Não está citado nada sobre condicional/loop na pergunta, então provavelmente não há, simplificando a validação.

  - **Outros:** verificar coisas como instruções proibidas em certas condições, ou uso de constantes mágicas, etc., caso a DSL defina (não aparenta ter nada assim no enunciado).

  - Esse visitor usa tanto a tabela de variáveis quanto de tipos construídas pelos anteriores. Assim, tem informação do tipo (kind) de cada variável para conferir compatibilidades. Por exemplo, sabe que `destination` é do tipo `Point` (vector) e `altitude` é `Number` (scalar) e portanto pode apontar incompatibilidade se usados trocados.

  - Erros encontrados aqui também são acumulados. Exemplos de erros semânticos nesta fase: variável não declarada, tipos incompatíveis em atribuição ou em argumentos, tentar usar um array onde esperava-se um scalar, etc.

Em resumo, cada visitor implementa o método `visitX` apropriado para os nós relevantes da parse tree (onde X é a regra correspondente) e realiza verificações contextuais que vão além do que a gramática pôde fazer. Essa organização em múltiplos passes torna o código mais claro e cada passo foca em um *subset* de regras:

1. **Versão/estrutura geral** (ProgramLanguageVersionVisitor)
2. **Definições de tipos** (TypesValidationVisitor)
3. **Definições de variáveis** (VariablesValidationVisitor)
4. **Sequência de instruções** (InstructionsValidationVisitor)

Este fluxo é análogo ao de compiladores tradicionais (coleta de símbolos, depois verificação de uso) e garante que as mensagens de erro podem ser mais precisas e agrupadas. Inclusive, como cada fase acumula erros em vez de lançar exceções imediatas, o usuário final pode receber um relatório contendo *todas* as discrepâncias encontradas de uma vez, ao invés de precisar corrigir iterativamente erro por erro. Essa é uma característica importante em design de DSLs, pois melhora a usabilidade do feedback.

Acumulação e Reporte de Erros

Um aspecto fundamental do **DroneGenericPlugin** é como ele lida com erros de validação. Diferente do parser (que pode parar no primeiro erro de sintaxe, embora ANTLR tente recuperar e continuar), os visitors não interrompem o processo no primeiro erro semântico. Em vez disso, cada visitor adiciona os problemas encontrados a uma estrutura de acumulação (por exemplo, uma lista `List<Error>` mantida pelo plugin). Cada erro provavelmente registra informação como: tipo do erro, mensagem descritiva, e posição (linha/coluna) no código fonte onde ocorreu, obtida do contexto dos nós da árvore. Somente após executar todos os visitantes, o plugin terá a lista completa de erros semânticos. A integração final do plugin pode então:

- Se a lista estiver vazia: declarar o programa válido.
- Se houver erros: reportá-los todos ao usuário. Isso pode ser feito lançando uma exceção contendo todas as mensagens, ou retornando um objeto de resultado com as informações. Em ambientes de IDE, poderia destacar cada erro no editor, etc.

Essa abordagem segue a recomendação de projeto de compiladores, onde primeiro se coleta todos erros possíveis para apresentá-los juntos. Por exemplo, uma variável não declarada `y` usada 5 vezes gera 5 ocorrências de erro "variável não declarada" – o plugin poderia reportar todas, ou talvez consolidar em uma única mensagem dizendo "variável y não declarada (usada 5 vezes)". O importante é que o plugin **não para** na primeira detecção, permitindo ao desenvolvedor corrigir múltiplos erros de uma vez. A forma exata de armazenamento de erros no plugin não foi detalhada, mas é comum implementar uma classe ou interface de reporter de erros. Por exemplo, poderia haver uma interface `ErrorReporter` com um método `reportError(String mensagem, int linha, int coluna)`. Os visitors chamam esse método para cada problema. No final, o plugin pode imprimir ou lançar as mensagens acumuladas. Esse padrão é mencionado pela comunidade ANTLR como boa prática: *"colete as informações sobre erros encontrados e forneça-as na interface de usuário de forma adequada para seu aplicativo"*.

Registro de **Tipos** e **Variáveis** por *Kind*

Conforme citado, o plugin constrói tabelas internas para rastrear os símbolos definidos no código:

- **Tabela de Tipos:** armazena cada tipo definido na seção TYPES, associado a informações como seu *kind*. Provavelmente o plugin possui uma estrutura de dados (uma classe `TypeSymbol` ou similar) que possui campos como `name` (nome do tipo) e `kind` (enum com valores possíveis `SCALAR`, `VECTOR`, `ARRAY`). Se o kind for ARRAY, ele pode ter também um campo `elementType` referenciando outro `TypeSymbol` (a tipo base). Essa tabela de tipos permite que, dado um nome de tipo usado em variável ou em definição de array, o plugin rapidamente obtenha se é válido e qual sua categoria. Tipos *built-in* da DSL (se existirem implicitamente) também poderiam estar aí – por exemplo, pode ser que `scalar` e `vector` em si não sejam usáveis como nomes no código do usuário, apenas como palavras-chave para definir tipos. Mas o plugin pode internamente representar um tipo implícito "scalar" base para todos escalares, etc. De qualquer forma, os tipos nomeados pelo usuário (como `Number`, `Point`, etc.) entram nessa tabela.

- **Tabela de Variáveis:** similarmente, para cada variável declarada (caso passe nas validações iniciais), registra o nome da variável e uma referência para o tipo associado (provavelmente apontando para o `TypeSymbol` correspondente). Poderia também armazenar o *kind* diretamente para acesso rápido. Assim, ao validar instruções, quando vê um nome de variável, o plugin consulta essa tabela para verificar existência e recuperar qual o tipo e kind daquela variável.

- **Classificação por kind:** essa informação de *kind* é crucial para as regras de compatibilidade. Exemplos:

    - Ao tentar somar duas coisas, se ambos são `SCALAR`, ok (resultado scalar). Se ambos são `VECTOR` e talvez de mesmo tamanho, pode ser ok (dependendo se a DSL quer suportar operações vetoriais – não explicitado, mas poderia). Se misturados, o plugin decide se permite (ex: somar scalar com scalar = scalar; scalar com vector talvez proíbe, ou interpretaria como soma de uma constante em todos componentes do vetor – mas isso seria sofisticado, provavelmente não implementado sem indicação).
    - Ao atribuir valor a uma variável, compara o kind do valor com o kind do tipo da variável. Precisa bater exatamente (ou em casos de array, valor do elemento bater com tipo base).
    - Ao passar variável para um comando, checa o kind esperado. Ex: `MOVE` espera `VECTOR`, então a variável passada deve ter kind `VECTOR`.
    - Essa classificação também ajuda na geração de código ou execução, caso existisse (por ex, saber como armazenar ou manipular aquele valor).

- **Uso de Kind no código:** Nota-se que *kind* (scalar, vector, array) não é algo que o usuário escreve junto ao nome da variável explicitamente, mas está associado via o tipo. Ou seja, o usuário define que `Point = vector` (aqui ele informou o kind do tipo) e depois declara `destination: Point`. O plugin deduz que `destination` é kind vector porque seu tipo `Point` tem esse kind. Similarmente, se `Path = array of Point`, então uma variável `route: Path` será kind array (de elementos vector). O plugin possivelmente classifica as variáveis também em categorias semelhantes para facilitar.

Em suma, **registrar tipos e variáveis com sua categoria** permite ao sistema de validação tomar decisões de tipo corretamente. Essa fase de construção de *symbol table* é realizada pelos dois primeiros visitors e depois consultada pelos demais. Essa separação de concerns garante que *todos os usos de um símbolo* (tipo ou variável) possam ser validados com conhecimento das declarações anteriores. Se tentarmos usar um símbolo antes de declará-lo, o plugin pega na hora (pois ao chegar na instrução correspondente verá que não está nas tabelas). Essa técnica de *symbol table + visitors* é comum em implementações de DSLs sobre ANTLR.

Campos Obrigatórios e **RequiredFields** (TYPES, INSTRUCTIONS)

Na especificação da linguagem **droneGeneric**, duas seções são consideradas obrigatórias para qualquer programa válido: a seção de **TYPES** e a seção de **INSTRUCTIONS**. Ou seja, mesmo que um drone mission script não precise definir variáveis ou um cabeçalho específico, ele **deve** definir pelo menos um tipo e pelo menos uma instrução para fazer sentido. Essas exigências estão codificadas no plugin (já que a gramática em si pode ter permitido essas seções vazias ou ausentes para simplificar a sintaxe).

O plugin contém algo referido como *RequiredFields* que provavelmente é uma enumeração ou lista das seções que **devem** aparecer. Com base nisso:

- Se a seção `TYPES` estiver ausente no arquivo (ou vazia, sem nenhum tipo definido), o plugin adiciona um erro do tipo *"Seção TYPES obrigatória ausente ou vazia"*. Sem tipos, não há como declarar variáveis adequadamente, então é uma condição inválida.
- Se a seção `INSTRUCTIONS` estiver ausente ou não contiver nenhuma instrução, também é reportado erro (*"Seção INSTRUCTIONS obrigatória ausente – é necessário pelo menos um comando/instrução"*). Afinal, um script de drone sem instruções não faz nada.
- A seção `VARIABLES` não é listada como obrigatória em RequiredFields, portanto entende-se que pode ser opcional – um programa pode não ter variáveis declaradas (por exemplo, só usar constantes nas instruções) e ainda assim ser válido. Nesse caso, o parser aceitaria a ausência de `VARIABLES` e o plugin não reclamaria.
- O cabeçalho (`PROGRAM ... LANGUAGE VERSION ...`) aparentemente também não é considerado obrigatório pela enum RequiredFields (já que só menciona TYPES e INSTRUCTIONS). Isso sugere que o header é opcional: se ausente, possivelmente o plugin assume a versão padrão suportada (por exemplo 1.0). No entanto, se o header estiver presente, aí sim a versão é validada. Portanto, a presença do cabeçalho não é estritamente exigida para execução, mas se fornecida deve ser correta.

A checagem dessas seções obrigatórias é feita provavelmente logo após a fase de parsing, antes ou dentro do **ProgramLanguageVersionVisitor** conforme discutido. Garantir essas seções também simplifica a lógica subsequente – por exemplo, outros visitors podem assumir que a parse tree terá um nó de `types` e `instructions`. Caso estivessem faltando, ou o parse falharia (se gramática exigisse) ou então sem esses checks posteriores poderia causar NullPointer. Por isso o plugin toma cuidado de verificar e abortar com mensagens claras se estiverem faltando. Essa abordagem também permite uma mensagem de erro mais clara do que um erro de sintaxe genérico. Em vez de "*mismatched input EOF expecting 'INSTRUCTIONS'*" que o ANTLR daria se a gramática tornasse obrigatório, o plugin pode dar uma mensagem mais amigável em português explicando a ausência.

Resumindo, **RequiredFields = {TYPES, INSTRUCTIONS}** significa que **todo programa** deve ter pelo menos uma seção de tipos e uma seção de instruções. Essa é uma decisão de design da DSL: pelo menos um tipo (mesmo que trivial) deve ser definido – possivelmente para incentivar explicitar tipos, ou porque todas variáveis devem ter um tipo nomeado (não usam tipos anônimos) – e pelo menos uma instrução deve existir (senão o drone não faz nada). Desenvolvedores que estenderem a DSL devem ter em mente ao adicionar novas seções se elas serão obrigatórias ou opcionais e atualizar essa lógica de RequiredFields conforme necessário.

Exemplos de Uso da DSL (Válidos e Inválidos)

A seguir, apresentamos alguns exemplos ilustrativos de códigos na linguagem **droneGeneric**, incluindo um programa válido e casos de erros sintáticos/semânticos comuns. Isso ajuda a demonstrar as regras da gramática e as validações do plugin em ação.

Exemplo de Programa Válido:

```
PROGRAM DemoMission LANGUAGE VERSION 1.0

TYPES:
    Number = scalar;
    Point  = vector;
    Path   = array of Point;

VARIABLES:
    altitude: Number = 100;
    destination: Point = (10, 20, 5);
    route: Path = [ (0,0,0), (10,20,5), (0,0,0) ];

INSTRUCTIONS:
    TAKEOFF;
    MOVE destination;
    altitude = altitude + 20;
    LAND;
```

Análise: Este programa define no cabeçalho a versão da linguagem como 1.0. Em `TYPES`, cria três tipos: `Number` (escalar), `Point` (vetor) e `Path` (array de `Point`). Em `VARIABLES`, declara: `altitude` do tipo Number com valor inicial 100; `destination` do tipo Point com um vetor 3D como coordenada; e `route` do tipo Path inicializado com uma lista de pontos (note que usamos dois pontos iguais para talvez indicar ida e volta no caminho). A seção `INSTRUCTIONS` contém quatro comandos: decolagem, mover até o ponto `destination`, ajustar a altitude somando 20, e pouso. Todos os identificadores referenciados estão devidamente declarados, e os tipos de cada operação são consistentes (somar altitude+20: altitude é Number, 20 é Number literal, resultado Number ok; atribuir a altitude de volta a Number ok; passar `destination` (Point) para `MOVE` que espera um Point ok). Portanto, o plugin não reportaria erros. A gramática também aceitou a estrutura (todas seções presentes e na ordem correta). Esse seria um exemplo de script válido para uma missão simples.

Exemplos de Programas Inválidos (Erros):

Abaixo, listamos trechos de código ou programas completos com erros, explicando o motivo de cada erro conforme as regras de sintaxe (parser) ou semântica (plugin):

1. **Falta de seção obrigatória** – *Seção INSTRUCTIONS ausente*:

```
PROGRAM MissingInstructions LANGUAGE VERSION 1.0

TYPES:
    Number = scalar;
VARIABLES:
    x: Number = 5;

// INSTRUCTIONS section is missing here
```

Erro: O plugin exigirá a presença da seção `INSTRUCTIONS`. A ausência dessa seção fará o **ProgramLanguageVersionVisitor** (ou lógica similar) reportar erro: *"Seção INSTRUCTIONS obrigatória não encontrada."* O programa não possui comandos para o drone executar, portanto é considerado incompleto/inválido. (Obs: Caso a gramática não permita simplesmente terminar o arquivo sem `INSTRUCTIONS`, isso poderia até gerar erro de sintaxe imediato. Mas assumindo que a gramática permitiu opcional, então aqui passa na sintaxe e falha na validação semântica.)

2. **Uso de Tipo não definido**:

```
TYPES:
    Number = scalar;
VARIABLES:
    y: Distance = 200;   // 'Distance' type not defined in TYPES
INSTRUCTIONS:
    TAKEOFF;
```

Erro: Na linha da variável `y`, tenta-se usar o tipo `Distance` que nunca foi declarado na seção TYPES. O **VariablesValidationVisitor** detecta isso consultando a tabela de tipos (que só tem `Number` registrado) e não encontrando `Distance`. Será gerado um erro como *"Tipo 'Distance' não foi definido (usado na variável y)"*. Corrigir requer adicionar `Distance` em TYPES ou trocar para um tipo existente.

3. **Uso de Variável não declarada**:

```
TYPES:
    Number = scalar;
VARIABLES:
    x: Number = 1;
INSTRUCTIONS:
    x = 5;
    y = 10;    // 'y' not declared in VARIABLES
```

Erro: A instrução `y = 10;` tenta atribuir valor à variável `y`, que não foi declarada anteriormente. O **InstructionsValidationVisitor** consultará a tabela de variáveis após percorrer `VARIABLES:` e não encontrará `y`. Isso resulta em erro *"Variável 'y' não declarada"* apontando para essa linha. A instrução `x = 5;` por si está ok (x existe e recebe um Number 5). Note que esse é um erro semântico (a gramática permite `y = 10` sintaticamente porque `y` satisfaz a regra de ID e `10` é uma expressão válida; somente na validação é descoberto que `y` é desconhecida).

4. **Incompatibilidade de tipos em atribuição**:

```
TYPES:
    Number = scalar;
    Point  = vector;
VARIABLES:
    p: Point = (10,20,30);
    n: Number = 0;
INSTRUCTIONS:
    p = 5;          // Error: assigning scalar to a vector variable
    n = (1,2,3);    // Error: assigning vector to a scalar variable
```

Erro: Existem duas atribuições inválidas aqui. Na primeira, `p` é do tipo `Point` (vector), mas o valor atribuído é `5` (um NUMBER literal, scalar). **InstructionsValidationVisitor** (ou **VariablesValidationVisitor** se olhasse inits, mas aqui está em instructions) verifica o tipo de `p` (vector) vs tipo do valor (`5` → scalar) e emite erro do tipo *"Tipo incompatível: variável 'p' é do tipo Point (vetor), não pode receber um valor escalar"*. Na segunda, `n` é Number (scalar) e está recebendo `(1,2,3)` que é um literal vetor, resultando em erro análogo *"Variável 'n' é scalar, não pode receber vetor"*. Esses erros demonstram o checagem de tipos do plugin. (A gramática por si aceita `p = 5;` pois sintaticamente é ID '=' expr ';' – tudo tokens válidos – mas semanticamente é inválido.)

5. **Outros possíveis erros** (não exibidos com código aqui, mas para compreensão):

   - *Duplicação:* Se repetirmos `Number = scalar;` duas vezes em TYPES, o plugin reclamaria de tipo duplicado. Idem para duas variáveis com mesmo nome em VARIABLES.
   - *Versão de linguagem inválida:* `PROGRAM X LANGUAGE VERSION 9.9` se só 1.0 é suportado – erro do ProgramLanguageVersionVisitor.
   - *Sintaxe incorreta:* Esquecendo ponto e vírgula no final de instrução ou declaração, ou uma vírgula entre valores do vetor. Esses seriam erros sintáticos pegos pelo parser ANTLR imediatamente (ex: *"syntax error: missing ';'..."*). Por exemplo, `altitude: Number = 100` (sem `;`) não parseia corretamente. Ou `route: Path = [ (0,0,0) (10,10,10) ];` faltando a vírgula entre elementos resultaria em erro próximo ao `(10,10,10)` indicando token inesperado.

Em conclusão, a DSL **droneGeneric** possui uma gramática bem definida que impõe a estrutura básica dos programas de drones, e o **DroneGenericPlugin** implementa a lógica de validação semântica necessária para garantir que esses programas façam sentido antes de serem executados. Com a compreensão da gramática (seções, expressões, tokens) e do fluxo de validação (visitors para versão, tipos, variáveis, instruções, utilizando tabelas de símbolos e acumulando erros), um desenvolvedor pode estender a linguagem de forma segura – por exemplo, adicionando novos comandos na

seção INSTRUCTIONS exigirá ajustar a gramática e talvez adicionar casos no InstructionsValidationVisitor para validar novos parâmetros. Da mesma forma, incluir um novo *kind* (digamos, *matrix* para matriz 2D) implicaria atualizar a gramática (permitir `matrix` em TYPES) e ensinar o plugin a lidar com compatibilidade de `matrix` nas atribuições e operações. Felizmente, graças à arquitetura modular, essas mudanças ficariam concentradas em partes claras do código.

Este documento serve como referência para desenvolvedores manterem e evoluírem a DSL, garantindo que entendam as regras atuais e onde no código essas regras são verificadas e reforçadas. Com esse conhecimento, é possível ampliar a linguagem droneGeneric preservando sua coerência e robustez.

Fonte de Referência:

- Conceitos de estrutura de gramática ANTLR e organização por seções
- Definições típicas de tokens e resolução de ambiguidades entre keywords e IDs
- Exemplo de regra de lista (array) em gramática semelhante (JSON)
- Exemplo de definição de expressão aritmética com precedência em ANTLR
- Definição de tokens numéricos com parte decimal opcional
- Separação entre sintaxe e semântica – parser versus lógica no código