Brincando às Linguagens com rigor: Engenharia Gramatical

Pedro manuel Rangel santos Henriques
professor associado do
Departamento de Informática
da Escola de Engenharia da Universidade do Minho

9 de Novembro de 2013

Prefácio

O tema que quero tratar nesta lição (neste documento de síntese) é *a "engenharia gramatical" como forma rigorosa de trabalhar com linguagens.*

Este documento apresenta a lição que considero sintetizar adequadamente a matéria de uma UC de pós-graduação em Engenharia Gramatical.
Pretendo que os tópicos abordados e interligados justifiquem claramente o desenvolvimento de uma panóplia de métodos, técnicas e ferramentas para *conceber*, *desenvolver* (escrever, analisar e depurar) e *processar* **gramáticas**, como *forma sistemática e rigorosa de manusear linguagens.*

No texto presente discorre-se (de forma original e pessoal) sobre um tema que me é grato e de uma forma que julgo sintetizar a minha aprendizagem e visão pessoal da área de Processamento de Linguagens Formais, reflectindo sobre a *qualidade de linguagens e de gramáticas*, e introduzindo *métricas para gramáticas.*

Conteúdo

Capítulo 1

Contextualização

Explicado, no prefácio, o conteúdo e razão de ser deste documento, fala-se, neste primeiro capítulo, nas maiores implicações de recorrer a linguagens escritas (textuais, clássicas) para estabelecer a comunicação entre dois agentes. Dentro desse contexto, justifica-se a necessidade de recorrer a um instrumento formal (a *gramática*) para definir linguagens e para possibilitar o reconhecimento das suas frases. Nesse âmbito, surge como necessidade premente a existência de métodos e técnicas de trabalho que possibilitem manusear (conceber e implementar) gramáticas da forma como em engenharia se procede normalmente.

1.1 Enquadramento e motivação

A Linguagem, sendo algo tão simples como *um conjunto de frases (sequências de símbolos de um alfabeto)*, é a peça chave para a vida em colectividade—entre muitos, como sucede na sociedade humana; ou entre dois, como sucede na interacção pessoa-computador. É através da linguagem que se comunica tudo quanto é necessário para essa co-existência e cooperação.

Essa comunicação pode ser síncrona, com os interlocutores presentes—por emissão de sons (linguagens sonoras como a linguagem humana falada, as linguagens dos animais, a linguagem de apitos, a música, etc.), por gesticulação (como as linguagens gestuais dos surdos-mudos) ou produção de outros efeitos visuais (como as linguagens por sinais de fumo, bandeiras, etc.)—ou pode ser assíncrona, em que o emissor deixa registados os símbolos nalgum suporte (pedra, madeira, papel, memória electrónica) para serem lidos depois pelo receptor (linguagem escrita).

É importante notar que, no contexto desta lição, restringe-se o discurso à comunicação por escrito, entre duas entidades (pessoa-pessoa ou pessoa-computador). Além disso, quando se fala em *linguagem*, está-se a referir a um exemplar do conjunto das **linguagens artificiais formais**[1], **textuais**[2]. Dentro destas, o foco recai sobre o subconjunto das **linguagens de programação**[3] (LPs).

As **linguagens visuais** (LVs) e em particular as **linguagens de programação visuais** (LPVs), com símbolos gráficos/icónicos compostos de formas diversas no plano ou no espaço tri-dimensional, estão fora desta discussão.

[1] Por oposição às Linguagens (ou Línguas) Naturais, designam-se, genericamente, por Linguagens Artificiais as que são concebidas e desenhadas para a comunicação técnica/profissional não-ambígua. Destas, o subconjunto das que são definidas rigorosamente por um formalismo matemático (por exemplo a Gramática) designam-se por Linguagens Formais.

[2] Linguagens em que os símbolos terminais são palavras formadas pela concatenação de caracteres ASCII e as frases são, por sua vez, sequências desses símbolos terminais.

[3] Linguagens para instruir o computador sobre a forma de resolver um determinado problema.

Para que a comunicação seja efectiva o emissor tem de saber escrever exactamente aquilo em que está a pensar e o receptor tem de compreender exactamente o significado daquilo que o emissor lhe transmitiu. Além disso, ambos tem de o fazer (escrever e interpretar) em tempo útil. Note-se ainda que faz parte do ser humano, na sua incessante procura do *óptimo* e da *perfeição*, gostar de *aprimorar o seu estilo de escrita*, lutando sempre por escrever de *forma elegante*...

O primeiro requisito—*o emissor saber escrever e o receptor compreender*—requer a existência de um suporte único, por ambos partilhado, que diga: ao primeiro, como se escrevem frases válidas e com o significado correcto; e ao segundo, como se interpretam essas frases válidas para delas extrair o significado correcto.

Neste contexto propõe-se o recurso à **gramática** como forma de, simultaneamente, definir a linguagem (guiando a escrita das frases) e suportar o seu reconhecimento (guiando a extracção do significado). Para que nos seja útil[4], a gramática deve assegurar que um símbolo tenha sempre o mesmo valor qualquer que seja o local onde é usado, isto é, deve ser uma Gramática Independente de Contexto [Cho62, Bac79, HMU06a] (GIC); para que seja completa e cubra todo o processo de tratamento, deve conter atributos que descrevam o significado dos símbolos, isto é, deve ser uma Gramática de Atributos [Knu68, DJL88, Hen92] (GA).

O segundo—*processar em tempo útil*—é uma questão técnica ligada ao referido suporte único, concretamente à gramática, que tem de garantir, por um lado, que o processo de derivação de uma frase termine e, por outro lado, que possa ser interpretada em tempo finito.

O terceiro requisito—*optimização do processo e aperfeiçoamento da escrita*—têm um caracter mais filosófico e está intimamente relacionado com várias características intrínsecas à linguagem (ao vocabulário e às construções previstas); é claramente um problema de *estética*, relacionado com o conceito do *belo* e dependente de *épocas* e *escolas* (ou seja, de *modas*). Nesta lição e por razões óbvias, o terceiro requisito não será considerado.

Tal como na vida real as diferentes comunidades humanas inventaram os seus vocabulários e as suas línguas (linguagens naturais), com regras sintáctico-semânticas próprias, também no campo das linguagens formais as várias comunidades científicas têm inventado muitas linguagens distintas [TB07], quer para resolver diferentes tipos de problemas, quer para seguir paradigmas de resolução distintos[5]. Isso faz com que a oferta seja variada e surja a necessidade de escolher entre elas a que se vai usar para determinado fim. Associada à escolha, anda sempre a questão da *qualidade*—todos somos pródigos a afirmar que queremos optar pela alternativa de *maior qualidade*!

Esta lição é precisamente sobre qualidade das linguagens e qualidade das gramáticas subjacentes. Assim, vai propor-se que a gramática, além de ser um instrumento de definição da linguagem e seu processamento, sirva também de suporte à avaliação da sua qualidade. Pretende-se, pois, apresentar um conjunto de métricas que permitam raciocinar sobre a qualidade da gramática e, por fim, procura-se relacionar essa avaliação com a qualidade da linguagem.

1.2 Organização da Lição

Apresentada a motivação para o uso de gramáticas e para a criação de uma disciplina rigorosa e quantificável que permita trabalhar com gramáticas como em engenharia se lida com a construção e uso de ferramentas, é altura de apresentar o esquema da lição que aqui se descreve.

Ao propósito *qualidade de linguagens*, dedica-se o capítulo 2 com três secções. Uma (secção 2.2) em que se defende a necessidade de abordar esse tema, mas na qual se mostra que discutir informalmente, sem critérios objectivos, a qualidade das linguagens é como *discutir o sexo dos anjos*, ou seja, é uma discussão sem fim que não trás nenhuma conclusão. Outra (secção 2.3) em que se definem diversos critérios precisos, à luz dos

[4] Do ponto de vista do seu processamento automático.

[5] A este propósito—diversidade, evolução histórica e cronologia das Linguagens de Programação—recomenda-se a consulta ao site `http://www.levenez.com/lang/`

quais se podem comparar linguagens e discutir a sua qualidade. E uma secção inicial (a secção 2.1) onde resumidamente se relembram as noções básicas relativas às LPs.

Como aqui se sugere que a gramática (já recomendada para especificar e reconhecer linguagens) seja também usada para estudar a qualidade da linguagem que gera e do processo de interpretação, serão reservados os capítulos 3 e 4 para o tema. No primeiro relembram-se as definições de Gramática Independente de Contexto e de Gramática de Atributos, bem como o processo de reconhecimento associado; enquanto que no segundo se introduzem critérios (secção 4.1) e métricas (secção 4.2) para avaliar a qualidade das gramáticas. A secção 4.3 completa esse capítulo, ilustrando todos os conceitos com dois casos de estudo, em que se transforma a gramática dada e se vai medido em cada metamorfose por forma a compreender a evolução, quer da gramática em si, quer da linguagem gerada.

A síntese da lição, com enfoque nos objectivos e resultados atingidos, bem como a referência a outros tópicos relacionados com a utilização e manipulação de gramáticas que, apesar do grande relevo e actualidade, não foram aprofundados por limites de tempo e para manter o discurso conciso, serão apresentados no capítulo 5.

Capítulo 2

Qualidade das Linguagens

Neste capítulo desenvolve-se o tema que motivou a lição, a qualidade das linguagens. Qualidade que deveria permitir, em termos absolutos, dizer o grau de satisfação com que uma linguagem serve o fim para que foi concebida e que deverá, no mínimo, permitir seleccionar uma entre várias alternativas.

Embora se considere que a leitura deste documento pressupõe um razoável conhecimento teórico e experimental sobre linguagens de programação, decidiu-se iniciar este capítulo relembrando os principais conceitos associados às linguagens de programação e deixando apontadores para a literatura fundamental sobre esse tópico.

Depois, na segunda secção, mostra-se o quão relevante é poder escolher uma boa linguagem para assegurar a eficácia da actividade de programação, mas alerta-se para os inúmeros factores humanos que tornam essa classificação quase impossível se conduzida de forma subjectiva.

Então e após apresentar a definição de *qualidade* que vai suportar todo o discurso, discutem-se, na terceira secção, os critérios para aferir a qualidade da linguagem à luz dessa definição.

2.1 Relembrando as Linguagens de Programação

Num texto pedagógico recente [Per10], M. João Varanda resume convenientemente a evolução e caracterização das Linguagens de Programação, bem como identifica os conceitos básicos da programação que devem ser contemplados no desenho e implementação das ditas linguagens (a propósito ver as obras de referência [Hoa73, Wat04, Seb09, Sco09], bem como a já citada [TB07]). Será com base nessas notas que se recapitulam aqui os referidos conceitos e classificações.

Mas antes de avançar e com o intuito de aumentar a motivação para um trabalho desta índole, citam-se abaixo as 6 razões apresentadas por Sebesta [Seb09] para se estudar uma Linguagens de Programação:

1. aumentar a capacidade de expressão (escrita) de ideias;
2. melhorar a sabedoria básica para se poder escolher a linguagem apropriada para cada situação;
3. aumentar a capacidade de aprender fácil e rapidamente novas linguagens;
4. aumentar a capacidade de conceber/desenhar novas linguagens;
5. melhorar a percepção da relevância que a implementação tem no uso da linguagem;
6. melhorar a compreensão sobre todo o domínio da programação (resolução de problemas por computador).

Conceitos básicos da programação Para começar é importante lembrar os conceitos básicos envolvidos na resolução de problemas por computador, conceitos esses que devem ser tidos em linha de conta quando se concebem Linguagens de Programação.

Os dois conceitos essenciais são o de **valor**—representado por uma **constante**, **variável** ou **expressão**—e o de **operação**. O primeiro para descrever os *dados* e o segundo para realizar as transformações que conduzem aos *resultados*. Para assegurar a correcção das transformações, os valores tem **tipos**—que podem ser **atómicos**, ou **estruturados** e, neste caso, podem ser *agregados* (como os registos) ou *recursivos* (como as listas). Associado ao conceito de tipo e à referida correcção, levanta-se de imediato uma questão fundamental na moderna programação: a **verificação de tipos**.

As várias entidades do programa (as operações, as variáveis, eventualmente as constantes, os tipos e até o próprio programa e seus subprogramas) tem de ser identificados por um **nome** único. Ao conceito de nome anda sempre ligado um outro requisito: a **associação** (ou, em inglês, **binding**) do nome à entidade que denota, ou a um valor. Essa associação pode ser *implícita* (ou *primitiva*), ou pode ser *explícita* (ou seja, *declarada* pelo programador). Além disso, a associação de um nome a uma entidade/valor pode ter carácter *global* a toda o programa, ou *local* a um determinado bloco; o **âmbito de validade de uma declaração**, usualmente designado por ***scope***, é outro conceito que tem de ser regulado e verificado.

A ordem de aplicação das operações aos operandos, ou seja, o **fluxo de execução**, e a capacidade de **controlar** esse fluxo dinamicamente através de condições, quer escolhendo caminhos alternativos, quer executando repetidamente, são outros dois conceitos a ter em consideração.

A capacidade de **abstracção** a nível das *operações* (criando *subprogramas* com carácter procedimental ou funcional, *módulos* ou *pacotes*) e a nível dos *dados* (criando *tipos de dados abstractos* ou *objectos*), o **encapsulamento** e a **herança**, são outros conceitos mais sofisticados, que surgiram recentemente com vista a lidar melhor com problemas de complexidade crescente, e que também devem ser considerados na concepção de Linguagens de Programação.

Classificação das Linguagens de Programação Como já foi muitas vezes referido até aqui, ao longo dos últimos 60/70 anos foram criadas mesmo muitas Linguagens de Programação com o intuito de considerar os conceitos enumerados no parágrafo anterior da forma que melhor ajude o programador a resolver bem e com facilidade os problemas. Assim sendo, o Ser Humano, na sua habitual tendência para agrupar as coisas com o intuito de perceber similaridades e diferenças, tem vindo a classificar as Linguagens de Programação associando-lhes a designação da classe a que pertencem. Porém essa classificação tem vindo a ser feita segundo diferentes critérios, o que faz com que muitas vezes surjam confusões porque as designações de classe atribuídas correspondem a critérios distintos não comparáveis (normalmente complementares).

No resto desta secção, apresentam-se as classificações arrumadas pelos critérios subjacentes:

- quanto à cronologia e grau de abstracção[1]:

  1ª Geração, código-máquina ou linguagem de baixo nível binária (p.ex. o código do Intel 80486).

  2ª Geração, Assembly ou linguagem de baixo nível por mnemónicas (p.ex. o Assembly do Intel 80486).

  3ª Geração, linguagem de alto nível para o programador (do FORTRAN ao C♯, do Perl ao Python, do Lisp ao Haskell, Prolog, etc.).

  4ª Geração, linguagem de alto nível para o utilizador, em domínios específicos (do SQL ao às Macros do Excel, etc.).

- quanto ao processamento:

[1] Relativamente à linguagem do Processador.

compilada, executada só após totalmente reconhecida e traduzida para código-máquina (p.ex. FORTRAN ou o C♯).

interpretada, executada durante o reconhecimento com tradução para um representação intermédia parcial (p.ex. o Basic).

- quanto ao estilo/paradigma de programação:

  imperativo ou procedimental (p.ex. o Basic, o FORTRAN ou o C♯).

  declarativo lógico (p.ex. Prolog) ou funcional (p.ex. Haskell).

- quanto à organização do código:

  monolítico, todo o texto de um programa está junto num único ficheiro porque a linguagem não suporta formas de o separar por várias fontes (tipicamente as lingaugens Assembly são desta classe) .

  modular, a linguagem inclui mecanismos para que o texto de um programa possa ser distribuído por vários ficheiros de texto, devendo principal localizar os módulos externos que invoca; estes mecanismos podem ser muito simples e primitivos (como acontece nas linguagens C, ou Pascal), elaborados contendendo a noção de interface e implementação (p.ex., Modula-2), ou muito sofisticadas incluindo as noções de encapsulamento, de classe e instância e de herança (p.ex., todas as linguagens para Programação Orientada-a-Objectos, como Oberon, Co, Java, ou C♯).

- quanto à forma de execução:

  sequencial, a linguagem não dispõe de mecanismos para distribuir a execução das instruções, pelo que todo o código é executado por um processador num único processo (p.ex., Basic, ou Pascal).

  concorrente, a linguagem fornece directivas para distribuir a execução do código, lançando processos concorrentes (p.ex., C + threads) ou enviando-o para processadores paralelos (p.ex. Occam).

- quanto ao sistema de tipos:

  - **fracamente** tipada (como o FORTRAN ou o Basic) *versus* **fortemente** tipada (como o Java, o Haskell ou o Perl).
  - **estaticamente** tipada (como o Java[2], ou o Haskell[3]) *versus* **dinamicamente** tipada (como o Perl).

- quanto à forma de expressão:

  textual, os símbolos terminais são palavras, formadas por caracteres ASCII (como exemplo surgem todas as linguagens referidas nos items anteriores do Assembly ao C♯).

  visual, os símbolos terminais são icónicos, formados por símbolos gráficos ou imagens (p.ex., ProGraph, LabView, ou VisualLISA).

- quanto ao propósito, ou âmbito:

  domínio geral, também designadas por GPL (*general purpose language*), são linguagens que visam servir para resolver problema em qualquer domínio ou universo de discurso (p.ex., FORTRAN, Prolog, Haskell, Pascal ou Java).

  domínio específico, também designadas por DSL (*domain specific language*), são linguagens que visam servir para resolver problema em determinado domínio concreto (p.ex., SQL, SVG ou FDL).

[2]Em que todos os tipos tem de ser explicitamente declarados.

[3]Que tem um potente mecanismo de inferência de tipos.

2.2 Em busca do impossível

Uma linguagem de programação (de especificação, modelação, coordenação, etc.) é o instrumento de trabalho de quem resolve problemas por computador (i.e., do *programador*).

De uma forma mais *imperativa*, ou mais *declarativa*, a linguagem permite ao programador dizer o que é o problema a resolver ou a forma como deve ser resolvido. A complexidade dos problemas a resolver requer, a maioria das vezes, descrições deveras complexas; essas descrições devem ser facilmente lidas por humanos e eficientemente processadas pela máquina e, sobretudo, devem ser produzidas (escritas) com facilidade.

Daí a preocupação com o facto de que *a linguagem forte e decisivamente influencia o trabalho do programador* e, portanto, implica *a qualidade da sua actividade e do respectivo resultado.*

Tal como para qualquer pintor, cujo pincel é o melhor (quer seja da mais fina cerda de porco, quer seja de barbas de milho), para o programador a sua *linguagem de trabalho* é a melhor...

Cada um adapta-se ao instrumento com que trabalha, aprende todos os truques necessários para dele (neste caso, dela, a linguagem) tirar o máximo rendimento e a partir de certo momento é indistinguível o que é apanágio da linguagem e o que é perícia do seu utilizador.

Sem critérios objectivos e inequívocos, esta é uma discussão infrutífera e uma batalha perdida... nunca se chegará à conclusão qual a melhor linguagem e qual a mais adequada para uma determinada pessoa ou um determinado fim.

Este problema constata-se mesmo ao nível individual. Cada um de nós tem, em geral, mais de uma linguagem de trabalho—seja por imposições externas, seja por escolha própria (relacionada com as especificidades das tarefas a executar e as particularidades de cada linguagem)—e a verdade é que nos é muito difícil dizer qual a melhor.

O problema levanta-se também na comparação entre Linguagens de Uso Genérico (GPL[4]) e Linguagens para Domínios Específicos (DSL[5], ver a propósito [vDK98, SMH05, KLBM08]). Quando se utiliza uma GPL para resolver um problema muito particular, sente-se que a linguagem oferece muita construção que não é útil e é omissa numa série de instruções que seriam muito cómodas para aquela família específica de problemas ... Quando se recorre a uma DSL para resolver certo tipo de problemas, frequentemente se pensa que a linguagem seria muito mais interessante se nos disponibilizasse mais este ou aquele operador, ou nos desse a liberdade de programar à vontade esta ou aquela componente.

Tentar avaliar qual a melhor linguagem, de acordo com as vantagens citadas por cada utilizador, é uma discussão interminável, que não leva a qualquer resultado prático.

Mas, por outro lado e como já foi dito, a escolha de uma *linguagem de trabalho* é tarefa complicada, porque a oferta é efectivamente grande, o que justifica que seja realmente relevante discutir a **qualidade das linguagens**, quer no sentido da legibilidade, quer no sentido da eficiência. No campo da **legibilidade** para o humano[6] distinguem-se 3 factores: *facilidade de aprendizagem*; *facilidade de escrita*, para desenvolvimento de uma descrição completa; *facilidade de compreensão*, para se perceber essa descrição completa (i.é, para ler uma frase da linguagem). No campo da **eficiência** do computador que vai processar, analisa-se apenas a questão do reconhecimento automático das frases, visto que a questão da execução é mais específica—só faz sentido falar de execução se for uma Linguagens de Programação—e depende de muitos factores externos à própria linguagem.

Definição 1 (Qualidade de uma Linguagem) *: A **qualidade de uma linguagem** afere-se em termos da facilidade com que se aprende, se usa e se compreende e em termos da eficiência com que as suas frases são processadas.*

[4]Do inglês, General Purpose Language.

[5]Do inglês, Domain Specific Language.

[6]Quer no papel de programador (que vai escrever frases da linguagem), quer no papel de analista (que vai ler essas frases).

Diz-se, então, que uma linguagem tem qualidade *se facilita a legibilidade (numa das três vertentes acima identificadas) sem degradar o processamento.*

É, pois, fundamental encontrar um conjunto de características que permitam um raciocínio claro e objectivo sobre as propriedades que contribuem para a qualidade das linguagens; e depois importa encontrar uma forma de as avaliar.

2.3 Caracterização de Linguagens

Vários autores (não muitos) têm-se preocupado ao longo dos tempos em definir critérios a ter em conta no desenho de uma linguagem de programação, ou a ponderar na escolha da linguagem mais adequada para resolver um problema específico (estabelecendo argumentos de comparação entre elas).
Um dos primeiros foi Hoare [Hoa73]. Recentemente, Oliveira-Silva na sua tese de doutoramento [Sil06] toma por base as ideias originais de Hoare e propõe critérios como: expressividade; abstracção; compreensibilidade; documentação; unicidade; consistência; extensibilidade; eficiência dos programas; segurança; realizabilidade; sinergia; e ortogonalidade.
Por seu lado Selic menciona, entre outros, os seguintes critérios para o sucesso de uma linguagem: expressividade; simplicidade de uso; eficiência de execução; familiaridade do programador com esse tipo de linguagens; e ambiente e ferramentas de suporte.
De acordo com Howatt (ver [How95]) os critérios a considerar estão organizados nos seguintes 4 grandes grupos:

- Desenho e Implementação: correcto desenho, sem ambiguidades; extensibilidade; dificuldade em implementar um processador; velocidade e eficiência desse processador.
- Factores Humanos (interacção com o programador): facilidade de uso e carácter amigável; tendência para fazer erros; facilidade de apreensão e compreensão.
- Engenharia de Software: portabilidade; segurança; capacidade de reutilização; facilidade de manutenção.
- Domínio de Aplicação.

Na opinião de Sebesta [Seb09] há 4 grandes critérios, a saber:

- legibilidade – simplicidade; ortogonalidade; instruções de controlo de fluxo; estruturas de dados; questões sintácticas (palavras-reservadas, identificadores, etc.).
- facilidade de escrita – simplicidade; ortogonalidade; suporte para abstracções; expressividade.
- fiabilidade – verificação de tipos; suporte a excepções.
- custo – de treino dos programadores; de escrita; de compilação; de execução; de manutenção.

Por seu lado Watt em [Wat04] enumera os seguintes critérios para selecção de uma linguagem: escalabilidade; modularidade; facilidade de reutilização; nível de abstracção; portabilidade; fiabilidade; eficiência; legibilidade; capacidade para modelação dos dados e dos processos; familiaridade do utilizador; existência de um processador e ferramentas de suporte ao desenvolvimento. Além desses factores a considerar na escolha, o mesmo autor David Watt, refere que o desenhador da linguagem deve ter em consideração aspectos como: os detalhes léxico-sintácticos; a regularidade (ortogonalidade); a simplicidade; e a eficiência no processamento.

Procurando organizar as várias propostas à luz da experiência e opinião pessoal, de forma a encontrar o maior subconjunto comum que deixe de fora os critérios muito específicos de linguagens de programação (e portanto restritivos), e, sobretudo, os que se prendam com o compilador e ambiente de execução (e que, portanto, não são factores intrínsecos à linguagem), chegou-se ao conjunto de características que se definem a seguir e que suportam o raciocínio desta lição.
Enumeram-se abaixo as **Características para avaliar a Qualidade de uma Linguagem** ou permitir compará-la com outras:

- (CL1) **expressividade** (inclui a *abstracção* e a *clareza léxico-sintáctica*[7])
- (CL2) **documentação**
- (CL3) **unicidade**
- (CL4) **consistência** e **ortogonalidade** (inclui a *fidelidade ao paradigma*)
- (CL5) **extensibilidade**
- (CL6) **escalabilidade**
- (CL7) **fiabilidade** ou confiança (inclui a *clareza semântica*, mas distingue-se da *segurança* cf. referido abaixo)
- (CL8) **modularidade** (interfere na *reutilização*)

De entre as características referidas pelos vários autores citados acima, no início da secção, foram excluídos da lista anterior os seguintes factores:

- *ambiente / ferramentas de suporte* — apesar de enorme importância para o sucesso de utilização de uma linguagem, é claramente um factor externo ao desenho da linguagem (ver a propósito o Apêndice B).
- *portabilidade* — tendo também enorme influência na escolha e aplicabilidade de uma linguagem[8], depende do compilador (tem essencialmente a ver com o código-máquina gerado).
- *segurança na execução* — depende do compilador, isto é, do código-máquina extra que é gerado para minimizar erros durante a execução[9]. É neste ponto que surge o tema, agora tão em voga, designado por Proof-Carrying Code [NL96, Nec97, Nec98, CLN00, BRS05, GMdSP06] (abreviadamente PCC), em que o compilador além do código-máquina gera um conjunto de condições de prova que poderão ser verificadas no ambiente destino (antes de correr o código) de modo a garantir que o *contrato*-[Mey92] inicial[10] é cumprido.
- *familiaridade do programador com essa família de linguagens* — de novo um factor que tem grande influência no custo da programação (agora a nível de treino e de escrita), mas que só depende dos hábitos de quem usa a linguagem, sendo independente do seu desenho.
- *simplicidade* — não é um factor objectivo, é antes o resultado que se espera de uma linguagem com qualidade.

[7] Que outros autores consideram separadamente.
[8] Reduzindo o custo da programação, quer ao nível da escrita quer da manutenção.
[9] Como, p.ex., divisão por 0, indexação de um array fora dos limites, referencia a uma posição de memória num endereço ilegal, etc.
[10] Formulado originalmente numa linguagem de especificação de alto-nível, tipo JML ou Spec♯, e testado a nível do código-fonte.

No resto da secção explicam-se as características enumeradas e mede-se a influência de cada um sobre os 4 factores críticos para a qualidade de uma linguagem mencionados na definição 1 acima:

- aprendizagem
- escrita
- compreensão
- eficiência no reconhecimento

No apêndice A mostra-se um exercício de aplicação a linguagens reais.

(CL1) Expressividade é a *facilidade* e *naturalidade* com que se exprime a mensagem a comunicar, ou seja, a satisfação com que a linguagem permite escrever as instruções a transmitir ou explicitar as operações a realizar.
Nas palavras de Sebesta [Seb09]

> Expressiveness requires very powerful operators that allow a great deal of computations to be accomplished with a very small program.

Exemplo 1 *: Supondo que se pretende* calcular a expressão $3 \times min + ang^2$ e guardar o resultado na variável *res, considere-se a sua descrição em 4 linguagens diferentes.*
Hipótese 1:

```
PSHA E(res)
PUSH 3
PSHA E(min)
LOAD
MUL
PSHA E(ang)
LOAD
PSHA E(ang)
LOAD
MUL
ADD
STORE
```

Hipótese 2:

```
LET N = 3*M
LET B = A*A
LET R = N+B
```

Hipótese 3:

```
res = 3*min + ang*ang;
```

Hipótese 4:

```
res <- triple(min) + sqr(ang)
```

É óbvio que a Hipótese 4 corresponde à linguagem mais expressiva, pois é a que permite apresentar uma descrição mais simples e natural, isto é, mais próxima da intenção.

Além da **disponibilidade dos operadores básicos** necessários para resolver problemas no universo de discurso ao qual a linguagem se aplica, dois outros factores influenciam esta característica:

- **Clareza** – o léxico usado[11] e a sintaxe das operações.
- **Abstracção** – o nível de complexidade dos operadores e operações, em relação ao código-máquina[12].

Exemplo 2 *: Retomando o exemplo anterior, é agora fácil verificar porque é que a 4ª hipótese é a mais expressiva em termos de ser a mais clara e a que apresenta maior nível de abstracção.*
A hipótese 2 oferece um nível de abstracção maior do que a hipótese 1 e o léxico é inegavelmente mais claro, apesar de só permitir identificadores com uma letra. Porém o facto de apenas possibilitar a escrita de uma operação de cada vez, é nitidamente insatisfatório na medida em que não torna ainda simples, nem natural, a escrita da expressão.
Tal como a 3ª, a 4ª hipótese permite escrever a expressão naturalmente, isto é, em notação in-fix *mantendo os nomes das variáveis (sem ter de mencionar os endereços de memória correspondentes, nem tão pouco reduzir a uma letra); permite ainda escrever a atribuição ao nível das demais operações, ainda em* in-fix*. Porém, em relação à 3ª hipótese, a 4ª permite uma maior abstracção ao facultar o uso de funções pré-definidas (de complexidade superior aos operadores aritméticos base) e oferece uma ortografia (neste caso em relação aos sinais) mais clara.*

Um outro exemplo que reforça este conceito, agora na área das bases de dados, é o que se apresenta a seguir.

Exemplo 3 *: Supondo que se quer contar os registos de uma tabela* `T1` *onde o campo* `cidade` *é igual a* `"braga"`*, basta escrever em* SQL *a seguinte instrução:*

```
SELECT COUNT FROM T1 WHERE cidade = "braga"
```

enquanto que, para descrever precisamente a mesma operação usando a GPL *C, já seria preciso escrever o programa (sequência de instruções) seguinte:*

```
count=0;
read(&reg,sizeof(reg),1,T1);
while (!eof(T1)) { if (strcmp(reg.cidade,"braga")==0) {count++;}
                   read(&reg,sizeof(reg),1,T1); }
```

Devido à ortografia dos operadores, à sintaxe da instrução e, sobretudo, ao elevado nível de abstração desses ditos operadores, a 1ª linguagem é manifestamente mais expressiva que a 2ª.

No caso específico das linguagens de programação, são ainda factores que concorrem para a expressividade:

[11] A ortografia dos nomes dos operadores e dos sinais.
[12] O qual é aqui tomado como referência, ou nível 0 de abstracção; é em relação a essa notação e complexidade efectiva/real que se pode falar em "mais abstracto", ou "mais alto nível".

- A existência de tipos de dados estruturados, nomeadamente o enumerado, o agregado[13], o conjunto e o ficheiro de acesso directo[14].
- A possibilidade de criar novos tipos.
- A possibilidade de manusear listas ligadas dinâmicas sem qualquer referência à alocação de memória, ou aos apontadores.
- A possibilidade de manusear funções-finitas (ou *mappings*), implementadas como tabelas-de-hashing ou de outra forma.

O *overload* de operadores, essencialmente concebido para elevar o nível de parametrização dos programas e assim facilitar a reutilização do código, também contribui regra geral para aumentar a expressividade. Obviamente, se mal utilizado pode tornar-se confuso e ser indesejável do ponto de vista da clareza.

Exemplo 4 *: Considere-se por exemplo o caso do sinal de adição, +. No contexto das operações aritméticas tradicionais, é sem dúvida muito cómodo e útil que se possa usar o mesmo sinal para operandos inteiros e reais. Pensando agora na concatenação de* strings *(operação equivalente à adição numérica no domínio das sequências de caracteres) ou na adição de um par chave-valor a uma função-finita, é também claro que se torna cómodo e expressivo poder manter o uso do sinal + para denotar essas duas operações.*

A expressividade influencia positivamente a produtividade porque acelera o tempo de

- aprendizagem,
- escrita,
- compreensão.

e não interfere significativamente na

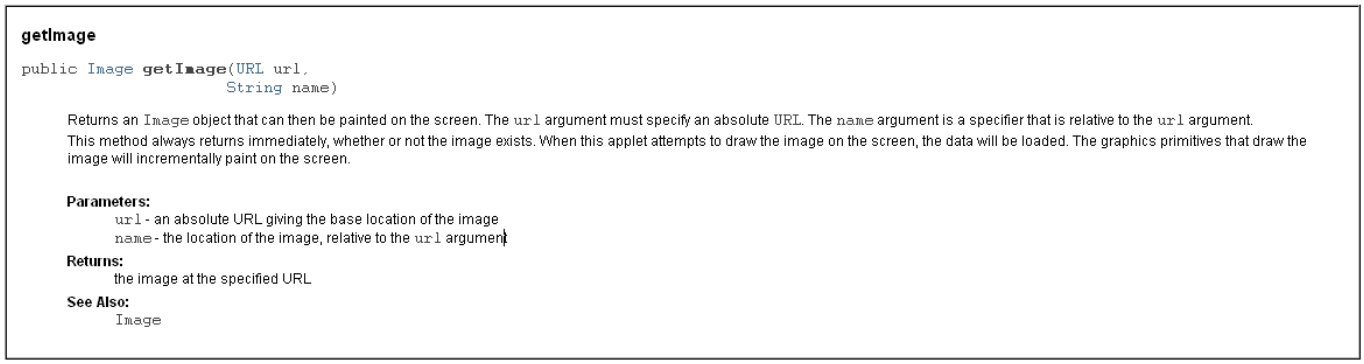
- eficiência do reconhecimento

pois os factores que melhoram a clareza requerem, normalmente, um esforço de análise ligeiramente superior, mas um nível de abstracção elevado não degrada de forma relevante o processo de reconhecimento.

(CL2) Documentação é a capacidade de incluir na mensagem[15] informação extra que explique a própria mensagem.
A forma típica que as linguagens oferecem para permitir documentar é através do uso de comentários. Um **comentário** é uma construção linguística válida que permite inserir um texto qualquer a partir de um determinado ponto até ao fim-de-linha (`eol`), ou num bloco bem delimitado (envolvendo, ou não, várias linhas). Esse texto livre é convenientemente assinalado para indicar, precisamente, que não faz parte da mensagem.

Exemplo 5 *: Praticamente todas as linguagens formais contemplam a possibilidade de inserir comentários de um tipo ou de outro.*

[13] `Tuple`, `Record` ou `Struct`.
[14] Tipo de ficheiro (disponível em FORTRAN, Cobol e Basic, mas não disponível em C nem em Pascal) em que se pode explicitar o número do registo onde se quer escrever/ler, como se faz a indexação de um array.
[15] A frase que se escreve na linguagem em causa para transmitir a especificação ou descrição básica pretendida.



getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute `URL`. The `name` argument is a specifier that is relative to the `url` argument.
This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
`url` - an absolute URL giving the base location of the image
`name` - the location of the image, relative to the `url` argument

Returns:
the image at the specified URL

See Also:
`Image`

Figura 2.1: Documentação de um método produzida pelo JavaDoc.

Por exemplo em LaTeX só existem comentários de linha, desde `"%"` *até* `eol`*.*
Já em HTML *só existem blocos, intercalados entre* `"<!--"` *e* `"-->"`*.*
Por sua vez em C♯ *existem os dois tipos: de linha, desde* `"//"` *ate* `eol`*; ou blocos, entre* `"/*"` *e* `"*/"`*.*

Actualmente começa a ser frequente que dentro dos comentários, e além do texto livre, se possa usar uma **linguagem de documentação**. Essa sub-linguagem permite incluir meta-informação que, além de útil directamente, pode ser usada posteriormente para gerar uma documentação mais sofisticada.

Exemplo 6 *: O caso mais usado e conhecido é o da linguagem de documentação associada à linguagem de programação* Java*. Dentro dos blocos de comentários* Java *(que começam com dois asteriscos em vez de um só) pode-se escrever texto que será passado para a documentação a gerar e podem inserir-se alguns items (palavras-reservadas precedidas pelo caracter* `"@"`*) com meta-informação, como se ilustra abaixo*

```
/**
 * texto dentro do comentário que deve surgir no manual on-line a ser criado
 * mais texto, possivelmente com HTML embebido,.....
 *                       ....ou outros comandos especificos ({@link URL},etc.)
 *
 * @param  nome-param  descrição de um parametro do método
 * @param  nome-param  descrição de outro parametro do método
 * @return             descrição do resultado retornado pelo método
 * @see                identificação de fontes complementares a consultar ("seealso")
*/
```

Esses comentários com informação extra são processados à parte pelo programa JavaDoc *para gerar, em* HTML*, um manual online sobre as classes. A figura 2.1 mostra uma página* HTML *gerada pelo utilitário* JavaDoc *a partir da documentação de um método* `getImage`*.*

Quanto ao C♯*, permite que se embebam anotações* XML *nos comentários (iniciados com 3 barras) que serão usadas para efeitos de documentação, como se ilustra abaixo. O* parser *de* C♯ *expande as anotações* XML *com informação adicional e exporta tudo para um ficheiro que fica disponível para posterior processamento.*

```
/// <summary>
```

```
/// descrição sobre o método para produção de um manual
/// </summary>
```

Também o Visual Studio C++ *admite uma solução deste género (linguagem de documentação para passar informação dentro dos comentários), oferecendo a possibilidade de inserir texto anotado em* XML *como acontece em* C♯*. O* Visual Studio C++ *aceita a notação do* JavaDoc *(para bloco de comentários) e a do próprio* C♯ *(para comentários até* `eol`*).*

Embora as duas possibilidades anteriores—*inserção de comentários* e *adição de dados para documentação dentro dos comentários*—sejam cruciais para a adição, ao texto base, de explicações complementares, ambas são opcionais; tal significa que a qualidade da documentação final depende do uso, facultativo e não-guiado, que o programador delas faça.
Em contrapartida, várias linguagens actuais obrigam (através da respectivas regras sintácticas) a incluir meta-informação. Essa informação adicional (que neste caso não é opcional e cuja estrutura é agora bem definida) fica incluída em qualquer frase e portanto imediatamente disponível como elemento de documentação. Mas, além disso, pode também ser usada por processadores adequados para auxiliar o arquivo das mensagens (frases) e sua posterior consulta.

Exemplo 7 *: Um caso bem conhecido é o da linguagem de anotação de documentos* XML*; nessa linguagem as primeiras linhas de uma anotação tem de descrever o* DTD/XSD *em uso e o* namespace*, como se mostra a seguir.*

```
<?xml version="1.0" encoding="US-ASCII?>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:tns="http://www.library.com"
          targetNamespace="http://www.library.com" >
```

Estas (meta-)informações, além de serem necessárias para a validação e processamento do documento anotado, ajudam também à documentação da anotação.

A propósito da linguagem de anotação de documentos XML, deve referir-se que actualmente—no âmbito da aproximação ao desenvolvimento de software dita design by contract—chama-se *linguagem de anotação de programas* a uma linguagem declarativa (lógica ou algébrica) que pode ser combinada com uma linguagem de programação para incluir especificações formais nos procedimentos (pré/pós-condições) e nos ciclos (invariantes) com vista a ser possível provar matematicamente (tipicamente com base em *theorem-provers*) a correcção do código face a uma especificação.

Exemplo 8 *: Os dois exemplos actuais mais em voga são a* JML *e a* Spec♯*; a primeira (Java Modelling Language) para validação de programas escritos em* Java *e a segunda para validação de programas escritos em* C♯*.*

Para além do primeiro intuito—verificação rigorosa de software—esta possibilidade de acrescentar ao código anotações formais, acaba por se traduzir numa outra facilidade adicional de documentação de programas.

Para concluir este parágrafo dedicado à 2ª característica de uma linguagem relacionada com a qualidade, é muito importante referir o valor que tem para a documentação de uma mensagem escrita o uso de identificadores apropriados (que sejam próximos dos nomes reais). A possibilidade léxica oferecida por uma linguagem de denominar todas as entidades descritas na frase através de identificadores longos e formados por um conjunto vasto de caracteres (e não só por letras) é outro factor que influencia positivamente a

auto-documentação da frase. Essa facilidade da linguagem permite que o utilizador escolha identificadores coincidentes ou muito próximos dos nomes reais dos objectos em jogo, bem como lhe permite denotar as operações com identificadores que designam explicitamente seu significado.
A documentação complementa a expressividade no sentido de facilitar/acelerar a

- compreensão

e não influencia significativamente a

- aprendizagem

pois a notação para os comentários e meta-informação é normalmente muito simples e natural.
Apesar de ter um efeito negativo na

- escrita

que é forçosamente mais lenta—quer quando se pretende usar identificadores mais longos e explícitos ou incluir comentários, quer quando é necessário incluir meta-informação no texto base—acaba por ter um efeito positivo no desenvolvimento porque obriga a reflectir mais profundamente no que se está a fazer e facilita o retomar da escrita quando esta não é feita de uma só vez.
Quanto à

- eficiência no reconhecimento

apenas poderá haver uma muito ligeira degradação.

(CL3) Unicidade é a existência de uma forma única para escrever a mesma instrução ou operação.
A unicidade é, como se costuma dizer, *um pau de dois bicos*. Por um lado é, em todos os sentidos, mais simples de trabalhar quando não há escolhas, quando só há um caminho para cumprir determinado objectivo. Por outro lado é, sem dúvida, *anti-natura*; o ser humano (por muito que se queixe) gosta sempre de poder escolher e tomar as suas opções, de modo a personalizar a sua actividade e senti-la mais adaptada, mais sua...

Exemplo 9 *: Supondo que é necessário ler e adicionar 10 números para resolver determinado problema, numa linguagem como a C—que não apresenta de todo esta característica de unicidade—essa tarefa pode ser implementada pelo menos de 3 maneiras distintas:*

```
tot=0; soma=0;
while (tot < 10)
  { scanf("%d",&num);
    soma = soma + num;
    tot = tot +1;
  }
```

```
tot=soma=0;
do { scan("%d",&num);
     tot++;
     soma += num;
   }
while (tot != 10);
```

```
soma=0;
for( tot=0; tot<10; tot++ )
   { scan("%d",&num);
     soma += num;
   }
```

Note-se que neste exemplo além de se usarem construções iterativas alternativas, usam-se também diferentes formas de fazer a atribuição, a adição e o incremento.
É óbvio que é mais complicado conhecer/aprender as 3 alternativas, quer para as escrever, quer para as perceber, mas depois é agradável que cada programador possa escolher a forma que lhe sai mais natural, mais intuitivo, e portanto rentabiliza a sua actividade. As tarefas de análise, quer de quem tem de compreender o texto, quer do reconhecedor automático, ficam nitidamente mais complexas, devido à diversidade.

Positivamente, a unicidade

- acelera o tempo de aprendizagem
- acelera o tempo de reconhecimento (aumenta a sua eficiência)
- facilita a compreensão

na verdade a unicidade torna mais simples conhecer a linguagem e perceber frases escritas por diferentes programadores (ou até por um só), na medida em que não há grande hipótese de recorrer a formas diferentes/alternativas de escrever a mesma coisa.
Mas interfere negativamente na

- escrita

pois não permite que cada um procure a forma que mais lhe agrada e convém para exprimir suas ideias; além disso, vulgarmente, a unicidade é conseguida à custa de operações mais elementares (para servirem vários fins) e portanto acaba por se traduzir numa frase mais longa para dizer a mesma coisa.
É verdade que a unicidade anda normalmente associada a um nível de abstracção menor (e, portanto, a legibilidade decresce) mas de facto esse é um problema de expressividade (CL1) e não um problema intrínseco ao conceito de forma única.

(CL4) Consistência é a conservação da estratégia, da disciplina ou princípio de trabalho, para expressão dos conceitos análogos (da mesma família), ou afins. Dito por outras palavras, uma linguagem revela esta característica se a forma de escrever o mesmo tipo de ideias é coerente, é sempre a mesma.
A consistência tem muito que ver com o historial da linguagem, ou seja com a forma como a mesma evoluiu ao longo do tempo. De facto, uma das maiores fontes de inconsistência é o ser acrescentada, quando já está em uso, por alguém que não o seu criador. Tal situação acontece com frequência quando uma linguagem—criada para um determinado fim, com um propósito restrito, ou ainda para uma comunidade limitada—começa a ser muito usada (por muita gente ou em muitas situações); nessas circunstâncias é vulgar os seus utilizadores sentirem necessidade de maior poder expressivo e impelirem o criador da linguagem, ou outra equipa, a proceder a uma extensão.
Tal como no caso precedente da *unicidade*, neste caso da *consistência* é mais fácil ilustrar o conceito com contra-exemplos.

Exemplo 10 *: Registam-se abaixo algumas inconsistências nítidas da linguagem C na qual, por princípio, todas as palavras-reservadas e identificadores pré-definidos são o mais curto possível e são escritos em minúsculas:*

```
FILE     // este identificador de tipo pré-definido escreve-se em maiúsculas;
typedef  // esta palavra-reservada está longe de ser mínima;
#include // esta palavra-reservada está longe de ser mínima,
         // ... começa necessariamente na coluna 1 e com "#";
```

Também a existência de uma função sobre ficheiros como o `fseek()` *é manifestamente inconsistente com a filosofia dos ficheiros sequenciais que a linguagem diz disponibilizar; segundo essa filosofia o ficheiro deve ser lido sempre do princípio para o fim e sempre a partir do byte a seguir ao último acedido, portanto não faz sentido posicionar a cabeça de leitura num byte qualquer do ficheiro. Porém essa inconsistência foi necessária para possibilitar o acesso directo aos ficheiros, o que mais tarde se mostrou imprescindível para desenvolver aplicações reais.*

Exemplo 11 *: Num campo de trabalho diferente (agora o processamento de documentos), outra linguagem bem conhecida e muito usada, que apresenta notórias inconsistências, é o LaTeX.*
Em LaTeX para criar, dentro de um documento, um determinado ambiente (ou contexto de trabalho) usa-se a construção sintáctica

```
\begin[params-opcionais]{nome-ambiente}
   bloco de texto ao qual se quer aplicar o ambiente
\end{nome-ambiente}
```

que é extremamente lógica e clara, ou seja, faz todo o sentido e portanto apresenta grande legibilidade. Porém, se for para aplicar determinado estilo de formatação ao bloco de texto, já se segue uma sintaxe diferente:

```
\nome-estilo{bloco de texto ao qual se quer aplicar o formato}
\textbf{conceito} ou \emph{nota importante} ou \texttt{pedaço de código} etc.
```

Apesar de ainda claro, é já inconsistente com o anterior; mas o pior é que também se pode obter o mesmo resultado com a seguinte construção sintáctica

```
{\nome-abrev-estilo bloco de texto ao qual se quer aplicar o formato}
{\bf conceito} ou {\em nota importante} ou {\tt pedaço de código} etc.
```

o que, além de voltar a ser inconsistente com os dois princípios indicados, viola também a unicidade.
Mas na realidade surgem muitas outras contradições. Por exemplo, para definir uma nota-de-rodapé, que em princípio é um ambiente como outro qualquer, usa-se esta última forma:

```
\footnote{nota-de-rodapé.}
```

Para definir cada uma das componentes estruturais de um documento—capítulo, secção, subsecção, ou parágrafo—terá de se usar uma sintaxe diferente das duas anteriores, em que o respectivo bloco de texto não é delimitado; o texto entre chavetas é apenas o título dessa componente (o que deveria ser encarado como um parâmetro do dito ambiente)

```
\componente-estrut{título}
\chapter{Introdução} ou \section{Antecedentes} etc.
```

Como todos os utilizadores e ensinantes de LaTeX muito bem sabem, estas inconsistências dificultam muito a aprendizagem e a leitura.

Mas também se podem encontrar exemplos positivos, pela afirmativa, como se pode ler a seguir.

Exemplo 12 *: O identificador* `this` *existente nas linguagens orientadas aos objectos, como por exemplo em* Java *ou* C♯[16]*, permite manter a consistência e identificar sempre o objecto a quem um atributo pertence, ou identificar o objecto a quem se está a enviar uma mensagem, quando esse objecto é o próprio sujeito da acção.*

[16] Equivalentemente designado por `me` em Visual Basic, ou por `self` em OCL (Object Constraint Language é uma linguagem para definir restrições em classes de maneira a formalizar modelos UML).

A reforçar a ideia anterior, pode dizer-se que a consistência é tanto maior quanto mais maior for a sua **fidelidade a um paradigma** de trabalho. Esta é uma das características mais importantes—ser muito evidente na linguagem a existência de um princípio forte de trabalho subjacente, ou seja, a adesão clara a uma determinada conduta com suas estratégias e respectiva discipina[17] De certa forma, o facto de uma linguagem ser concebida e desenhada para traduzir um paradigma acaba por determinar todas as demais características. Ao longo do tempo assistiu-se com frequência à evolução de linguagens tradicionais, tipicamente imperativas (p.ex., com as linguagens C, Php e Perl) ou mesmo declarativas (p.ex., Common-Lisp), muito bem aceites e muito usadas, de modo a passarem a suportar também o paradigma, mais recente, da orientação-a-objectos (POO). Mantiveram o mercado, ou seja, a *carteira de utilizadores* que tinham; possivelmente até aumentaram essa bolsa ... mas introduziram muitas irregularidades, inconsistências, na nova linguagem!

Embora seja um conceito distinto, convém incluir neste ponto a característica designada por **ortogonalidade** que surgiu há anos atrás no âmbito do código-máquina mas que é facilmente estendido às demais linguagens. Uma linguagem diz-se **ortogonal** se o conjunto de operadores pode ser aplicado a qualquer um dos tipos de dados disponíveis; ao nível do Assembly isto significa que os diversos modos de endereçamento de memória (para acesso aos operandos) podem ser usados com qualquer um dos operadores definidos[18]. Isto significa que o programador se pode mexer livremente no eixo das instruções ou dos dados, que eles se podem sempre cruzar sem restrições. É óbvio que esta característica—que desempenha na qualidade da linguagem um papel em tudo semelhante ao da consistência—facilita consideravelmente a aprendizagem porque reduz muito os casos especiais. Citando Michael Scott [Sco09]:

> Orthogonality means that features can be used in any combination, that the combinations all make sense, and that the meaning of a given feature is consistent, regardless of the other features with which it is combined.

A consistência e a ortogonalidade influenciam positivamente a produtividade porque aceleram o tempo de

- aprendizagem
- escrita
- compreensão

e não interferem de todo na

- eficiência do reconhecimento

(CL5) Extensibilidade é a capacidade que uma linguagem oferece para ser modificada (acrescentada) pelo utilizador.

Curiosamente sendo um tema em voga nos dias de hoje, sobretudo devido ao desenvolvimento de aplicações para a Web, a primeira referência à problemática da extensibilidade das linguagens de programação surge em 1960 com o artigo [Mcl60] de Douglas Mcllroy, tendo tido um tempo áureo em fins de 60 inicio de 70, como o atestam as conferências [CS69, Sch71] e o *survey* [Sta75]. Após o desinteresse pelo tema (muito provavelmente devido à dificuldade de manutenção e à ineficiência da implementação), o tópico ressurge por volta de 2000 [Wil05].

Desde o início, a extensão de linguagens esteve sempre fortemente associada ao uso de *macros*. Uma **macro** é, pura e simplesmente, uma regra que define como um texto de entrada (geralmente uma palavra

[17] É precisamente a esta disciplina de trabalho que chamo *Paradigma*.

[18] Este conceito aparece para distinguir dos *instruction-sets* em que cada operador exigia um determinado modo de endereçamento, ou restringia o conjunto dos modos permitidos.

mnemónica) é expandido para outro texto de saída (geralmente mais longo e complexo que o de entrada). Essa expansão pode ser textual[19]—um texto fixo ou variável (à custa de parâmetros posicionais ou especiais) é sempre produzido à saída—ou pode ser procedimental—a macro contém pelo meio do texto instruções que são executadas previamente e o resultado da execução é incluído no texto fixo que é produzido à saída. Embora esta ideia tenha surgido no seio da programação de baixo-nível em Assembly, quando apareceram os chamados macro-assemblers com o intuito de aliviar um pouco esta árdua tarefa (reduzindo o tempo e diminuindo as hipóteses de erro) através de permitir o uso de instruções que não eram nativas mas sim sequências de instruções nativas, os exemplos actualmente mais conhecidos são a directiva `#define` da linguagem C, o comando `def` do LaTeX e as Macros do MS-Office (tão comuns em Excel ou em Word).

Existem várias formas de estender uma linguagem, umas mais simples e habituais que outras. Antes de mais, importa distinguir **extensões estáticas**—que alargam a linguagem, durante a compilação, permitindo que se usem novos elementos ao traduzir uma dada frase (um programa)—das **extensões dinâmicas**—que permitem modificar um programa, graças a ser possível retirar ou juntar código, o qual é compilado só em *runtime*.

Neste parágrafo desenvolve-se a primeira faceta, pois a segunda é essencialmente uma extensão ou alteração ao programa (ou ao sistema de software) embora requeira, claro, que a linguagem tenha primitivas para manusear essas facilidades—criar ou remover código, modificar objectos, associar tipos dinamicamente, e invocar o compilador na fase de execução. Tipicamente associada à programação na área da Inteligência Artificial, ao Scripting ou, actualmente, à Web—surge em linguagens como Prolog, ou Lisp(essencialmente os dialectos Common-Lisp e Scheme), Simula, Forth, Perl, Python, Ruby, Php, Javascript—a extensibilidade dinâmica normalmente é suportada por um construtor do tipo `eval`, `compile`, ou `call`. Presentemente anda muito próxima do conceito de *dynamic typing*, segundo o qual à partida uma variável (sem tipo declarado) pode conter valores de qualquer tipo ou classe (muita vezes pode conter dados e até código), sendo o tipo concreto determinado durante a execução. Por fim, convém mencionar que a extensão, ou programação, dinâmica surge muitas vezes relacionada com as *linguagens reflexivas*. A **reflexão** é a capacidade oferecida por certos linguagens de inspeccionar, eventualmente modificar, o comportamento de um programa durante a sua execução. Esta observação refere-se ao valor das estruturas de dados, mas também pode abranger o próprio código. De acordo com o definição de Jason Voegele, que aparece no documento *Programming Language Comparison*[20]

> Reflection is the ability for a program to determine various pieces of information about an object at run-time. This includes the ability to determine the type of the object, its inheritance structure, and the methods it contains, including the number and types of parameters and return types. It might also include the ability for determining the names and types of attributes of the object.

No caso da extensibilidade estática, importa distinguir três formas:

1. acrescentando tipos e operações (neste caso é como se se estendesse o alfabeto, ou seja, se acrescentasse símbolos terminais);
2. acrescentando novas construções frásicas (estendendo as formas sintácticas);
3. acrescentando ou refinando/especializando o significado das construções e operações (extensão semântica).

A 1ª é a mais corrente; a grande maioria das actuais linguagens de programação (e mesmo as de especificação ou modelação) permitem que se definam tipos, procedimentos e funções e dessa forma se enriqueça o léxico

[19] Normalmente realizada por um pré-processador que corre antes do compilador.

[20] Acessível na Web em `http://www.jvoegele.com/software/langcomp.html`, cf. consultado em 2010.Set.02.

disponível. A possibilidade de incluir ficheiros/bibliotecas (ou declarar o uso de pacotes) é apenas uma forma simples e prática de fazer a extensão por definição de tipos ou operações.

Exemplo 13 *: Considere-se a seguinte definição de tipo em C*

```
typedef struct E {
          int c1, c2;
          struct E *seg;
          } tCel, *tAptL;
```

Após a declaração precedente ter sido reconhecida e guardada pelo processador[21]*, torna-se possível usar os identificadores* `tCel` *e* `tAptL` *como se fossem símbolos terminais nativos da linguagem, como se vê no extracto de programa abaixo.*

```
tCel elemento;
tAptL lista;
    lista = (tAptL) malloc(sizeof(tAptL));
```

Em termos práticos, de quem programa, tudo se passa como se o alfabeto, ou vocabulário, da linguagem tivesse sido estendido com mais esses 2 vocábulos.

Nas linguagens lógicas e funcionais (tipicamente interpretadas) é vulgar poder-se também definir novos operadores (designação, precedência, associatividade, modo[22], etc.), como se exemplifica abaixo.

Exemplo 14 *: Em* Prolog *um novo operador (designado por* `NovoOp`*) define-se da seguinte forma:*

```
    op( Preced, Tipo, NovoOp)
sendo,
    Preced um inteiro no intervalo 0..1200
    Tipo um valor no conjunto {xf, yf, xfx,  xfy, yfx, yfy, fy, fx}

Exemplos de operadores pré-definidos em prolog:
   :-op( 1200, xfx, :- ).
   :-op( 1200, fy,  ?- ).
   :-op(  500, yfx, +, - ).
   :-op(  400, yfx, *, /, mod ).
```

e depois usa-se normalmente seguindo a sintaxe prefix, infix *ou* postfix *e a aridade (unária ou binária) explicitadas em* `Tipo` *(note-se que 'f' designa a posição do novo operador em relação aos seus operandos).*

Para estender a sintaxe, a linguagem terá de permitir definir novas regras gramaticais que correspondam a conceitos mais abstractos ou mais específicos, ou seja, introduzir novos símbolos não-terminais que de qualquer forma se possam exprimir à custa dos já existentes. Tal vai implicar que o reconhecedor consulte a gramática de cada vez que vai processar uma nova frase, uma vez que esta pode ser alterada pelo próprio programa; ou seja, esta facilidade implica que não se possam usar as habituais técnicas estáticas de geração das tabelas de parsing (cf. se discute à frente na secção 3.2) Apesar de ser interessante e um desafio ao processamento de linguagens, não tem grande expressividade prática.

A extensão semântica encontra-se vulgarmente disponível nas linguagens orientadas a objectos, sendo assegurada através dos conhecidos mecanismos de extensão associados à hierarquia de classes. Esses mecanismos

[21] Neste caso concreto, o Compilador de C.

[22] infix, prefix, ou postfix.

permitem estender os atributos e métodos (funcionando como uma extensão ao alfabeto, como no 1ºcaso), mas acima de tudo permitem também dar nova semântica às operações, instanciando interfaces, ou reescrevendo operações concretas; tal revela-se na maioria das vezes muito útil e importante para a eficaz resolução de problemas.

Exemplo 15 *: Considere-se a seguinte declaração de uma classe geral* `Pessoa` *em* C♯.

```
class Pessoa {
      private int idade;
      public  t1 metod1 (a, b) { ... }
      }
```

A classe `Aluno`*, abaixo declarada, estende a semântica de* `Pessoa` *acrescentando-lhe um novo atributo e um novo método.*

```
class Aluno extends Pessoa {
      private string numero;
      public  t2 metod2 (x) { ... }
      }
```

Já a classe `Docente`*, abaixo declarada, estende a semântica de* `Pessoa` *redefinindo o método* `metod1`*.*

```
class Aluno extends Pessoa {
      public  t1 metod1 (a, b) { bla, bla, bla }
      }
```

A extensibilidade acelera o tempo de

- escrita

porque aumenta a abstracção e, portanto, a expressividade; além disso, permite que o utilizador crie os seus próprios vocábulos e construtores sintácticos ou semânticos e os utilize com eficiência dai para a frente. Este último argumento é, sem dúvida, muito forte porque permite adaptar a ferramenta (a linguagem) às mãos de quem trabalha com ela (quem escreve especificações ou programas) e por isso é o motivo que a torna uma das características mais desejáveis numa linguagem.
Mas, claramente, tem efeito negativo na

- aprendizagem
- eficiência do reconhecimento

porque, sendo mais complexa, aumenta o tempo de ambas.
Quanto à

- compreensão

não é seguro afirmar se facilita ou complica. Por um lado é óbvio que dificulta a interpretação; estar a ler uma frase numa linguagem conhecida e de repente aparecerem construções ou operações novas, torna naturalmente o processo de compreensão mais complicado. Uma vez identificadas as extensões e percebidas, a continuação do processo de análise e entendimento da frase, depende muito da qualidade dessas extensões. Ou seja, o seu valor real, para quem analisa o texto, é maior ou menor conforme a habilidade do programador para escolher novas construções ou operações que de facto facilitem (além da sua tarefa de escrever) a tarefa dos outros para compreender.

(CL6) Escalabilidade é a capacidade que uma linguagem demonstra de servir para lidar uniformemente—isto é, mantendo as suas características e qualidade—com problemas sucessivamente mais complexos.

De uma forma geral, um sistema diz-se *escalável* quando está preparado para crescer ou mudar de escala de grandeza da carga, isto é, quando é capaz de lidar com uma quantidade crescente de trabalho mantendo-se estável e exibindo um comportamento constante, sem degradar a sua performance. No caso concreto de uma linguagem, diz-se que é *escalável* se permite escrever, compreender ou processar com a mesma facilidade frases longas (para trabalhar com problemas de grande escala) como o permite para frases curtas (desenvolvidas para lidar com problemas simples[23], de pequena escala).

Para clarificar este conceito, pense-se nas linguagens visuais que são um bom contra-exemplo de linguagem escalável. É sabido que um diagrama/esquema, ou uma expressão icónica, são fáceis de escrever e extremamente fáceis de compreender se são pequenos (com a dimensão de 1 página A4, ou de 1 linha no segundo caso), mas tornam-se impraticáveis de usar quando deixam de ter dimensão reduzida.

Em princípio, a escalabilidade não afecta a aprendizagem da linguagem. Um factor que muito influencia a escalabilidade é a verbosidade da linguagem. Uma linguagem excessivamente carregada de longas palavras-reservadas ou, no outro extremo, excessivamente desprovida de açúcar sintáctico é, em geral pouco escalável.

Mas além destas questões léxico-sintácticas, várias outras ao nível semântico—como por exemplo as políticas de tipos, ou os níveis de parametrização relacionados com a invocação de subprogramas—contribuem para o facto de uma linguagem ser ou não escalável.

Exemplo 16 *: A título de exemplo pense-se no caso da linguagem* Pascal *e no caso da linguagem* Lisp*. Devido ao excesso de longas palavras-reservadas, um programa* Pascal *para resolver um problema real, complexo, torna-se demorado de escrever, extenso para ler (embora não perca a clareza) e mais lento para processar. Pode, pois, dizer-se que a qualidade da linguagem, em termos de legibilidade e processamento, se degrada quando se passa de programas de pequena/média dimensão para programas realmente longos. Aliás a própria disciplina rígida de tipos*[24] *contribui também fortemente para essa fraqueza da linguagem.*
Por seu lado, um programa em Lisp *de média dimensão torna-se muito difícil de ler ou compreender devido à excessiva pobreza sintáctica que não ajuda a distinguir com clareza blocos do programa, nem operadores de operandos.*

Exemplo 17 *: Noutro âmbito diferente das linguagens de programação, a linguagem de anotação de documentos* XML *e seus derivados é igualmente pouco escalável devido à possibilidade de ter qualquer número de marcas com qualquer nível de aninhamento e à sintaxe que obriga a fechar todas as marcas. Facilmente um documento anotado atinge tamanho muito grande (ou mesmo o esquema* XSD *que define um dialecto, ou uma especificação de transformação* XSL*) que só se consegue escrever ou compreender com a ajuda de editores e ambientes apropriados.*

A escalabilidade não degrada

- eficiência do reconhecimento

e preserva a legibilidade, ou seja, não altera a facilidade de

- aprendizagem
- escrita
- compreensão

[23] Muitas vezes meramente académicos, com dimensão e complexidade muito aquém dos problemas reais.
[24] Dois operandos são são compatíveis se forem do mesmo tipo.

(CL7) Fiabilidade (Reliability) é uma característica que se avalia em função do conjunto de facilidades que uma linguagem oferece para aumentar a confiança no código que vai ser executado, isto é, dos mecanismos disponíveis para aumentar a segurança que o código que vai ser executado corre sem erros e realiza as operações realmente especificadas no programa-fonte.

A primeira faceta que contribui para a fiabilidade é a linguagem *ter uma semântica precisa e não ambígua, muito claramente definida.* Infelizmente e sobretudo porque na grande maioria dos casos a semântica é descrita em linguagem natural, através da apresentação de casos/exemplos, em vez de ser esta definida formalmente, esta faceta nem sempre está presente, dificultando muito a aprendizagem e retirando muito a confiança que a execução realmente realize o que se espera ou necessita.

Por exemplo, a definição clara que é garantida a verificação dos indices dos arrays (assegurando que respeitam o intervalo com que foram declarados), ou do valor dos apontadores (garantindo que não se acede a uma zona da memória interdita), ou da variante activa de uma união[25] (garantido que não se acede a uma componente da estrutura, se a respectivo selector não estiver activado) são facilidades da linguagem que muito contribuem para agilizar o desenvolvimento com confiança no código que se vai produzir. Esta problemática aqui descrita é a maior distinção entre as filosofias subjacentes às linguagens Pascal e C.

Uma das facilidades mais importantes que uma linguagem pode oferecer para ser fiável, é ter uma *politica de tipos rígida*—obrigar a que todos as variáveis, e mesmo todos os valores, estejam associadas a um tipo—e, durante o reconhecimento do texto-fonte, *fazer uma análise de tipos completa*[26]—verificar se os operandos de todos os operadores tem o tipo correcto e se o resultado é do tipo esperado e permitido no contexto. Esta análise de tipos implica que o reconhecedor também faça uma verificação exaustiva da declaração de todos os identificadores usados (não permitindo re-declarações no mesmo contexto de visibilidade) ou disponha de um *sistema de inferência de tipos* poderoso[27].

A grande maioria das Linguagens de Programação mais em voga actualmente fazem análise estática de tipos, mas um caso emblemático desta disciplina fortemente tipada com inferência de tipos é a linguagem funcional ML, ou a sua parceira Haskell.

No seio da comunidade da Programação Orientada-a-Objectos, vem-se assistindo desde há alguns anos a uma longa discussão sem saída entre os defensores das políticas de análise estática de tipos—que argumentam que é muito mais seguro e aumenta a eficiência durante a execução—e os defensores da análise de tipos dinâmica—que sustentam a ideia que o aumento de flexibilidade do programa é de tal forma relevante para o custo de desenvolvimento que ultrapassa a perda de eficiência durante a execução.

Outra facilidade muito relevante é a disponibilização de construções para o *controlo de erros durante a execução*[28] , permitindo programar como reagir em situações de excepção (em que os valores reais, manipulados durante a execução, não pertencem ao domínio dos respectivos operadores). É claro que para o programador é muito mais confortável, mesmo que isso implique demorar um pouco mais de tempo na escrita, poder estabelecer como quer contornar os regimes excepcionais do que deixar ser o sistema operativos a tomar o controlo da situação, ou ficar mesmo por controlar.

Este tipo de construtores aparece hoje em quase todas a linguagens orientadas a objectos, como é o caso do Java e do C♯.

Naturalmente, a fiabilidade

- aumenta o tempo de processamento (diminuindo a sua eficiência)

[25] Uma estrutura variável, como é o caso do `record-variant` em Pascal ou da `union` em C.

[26] Em inglês, *static type checking.*

[27] Para conseguir fazer a análise de tipos severa, sem obrigar a declarar explicitamente o tipo de todos os identificadores e valores manuseados.

[28] Em inglês, *Exception handling mechanisms.*

- aumenta o tempo de escrita

mas facilita a

- compreensão

quanto ao

- tempo de aprendizagem

é dúbio. Se por um lado, ter uma semântica clara e precisa facilita a aprendizagem, ter um sistema de tipos forte e rígido e mecanismos para controlo e excepções implica ter mais coisas par estudar e aprender e portanto aumenta o tempo desta fase.

(CL8) Modularidade é a facilidade oferecida pela linguagem de organizar partes da mensagem (instruções/tarefas afins) em unidades, ou componentes, individuais (distintas e independentes) que podem depois ser reutilizadas.

A necessidade de agrupar o código de um programa surgiu desde logo no início da programação (na geração do Assembly) assim que a complexidade dos problemas aumentou e começou a exigir a escrita de frases (mais longas). E essa necessidade fez-se sentir desde sempre, também, por duas fortes razões: facilitar a escrita, evitando repetir subfrases iguais ou semelhantes (assim nasce o conceito de reutilização); isolar subfrases que descreviam uma determinada ideia importante, facilitando a compreensão e manutenção.

Partindo do conceito primitivo de subrotina, em Assembly, evolui depois para a forma de procedimento ou função, p. ex., em Pascal ou C, mas tomou a verdadeira forma de módulo em linguagens como o Modula-2 ou o ML. Neste novo estágio o módulo surge associado aos conceitos de *interface* e *implementação*, permitindo separar a *forma como se reutiliza* da *forma como se definem as estruturas de dados* ou *se descreve o corpo dos procedimentos/funções*. Nessa geração, a noção de módulo proposta satisfaz bem ambos os requisitos, acima mencionados, de *usar várias vezes o que só se escreveu uma vez* e de *isolar em blocos de código distintos as ideias cruciais* agilizando escrita e entendimento.

Posteriormente—por exemplo com as linguagens SmallTalk, Eiffel, Co, Java, C♯ ou OCAML—este conceito de módulo foi apurado no conceito de classe (mantendo a separação interface/implementação), mas providenciando agora uma metodologia de desenvolvimento que disciplina a forma de organização dos dados e respectivas operações.

Positivamente, a modularidade

- acelera o tempo de escrita
- facilita a compreensão

pois permite estruturar melhor o código ("arrumar" melhor as várias componentes da frase) concentrando em determinadas partes determinadas ideias ou tarefas e permite reutilizar.

E interfere muito ligeiramente de forma negativa na

- aprendizagem (há mais alguns construtores a aprender)
- no tempo de reconhecimento (diminui a eficiência porque há mais componentes a processar e mais informação manter em memória e a gerir)

Mas neste caso o balanço é totalmente favorável em termos do que se ganha em redução no custo do desenvolvimento (escrita) e da manutenção (compreensão).

(CL) Síntese A tabela 2.1 resume precisamente tudo o que foi dito ao longo desta secção sobre a influência das características escolhidas sobre os 4 factores, identificados na definição 1, para avaliar a qualidade de uma linguagem.

Caract x Factores	Aprendizagem	Escrita	Compreensão	Reconhecimento
CL1:Expressividade	+	+	+	x
CL2:Documentação	x	−	+	x
CL3:Unicidade	+	−	+	+
CL4:Consistência	+	+	+	X
CL5:Extensibilidade	−	+	+/−	−
CL6:Escalabilidade	=	=	=	=
CL7:Fiabilidade	x	+	X	−
CL8:Modularidade	x	+	+	x

Tabela 2.1: Influência das Características de uma Linguagem sobre o Critério de Qualidade

(Legenda) A Tabela 2.1 foi preenchida de acordo com o seguinte critério:

\+ **(influência positiva)** – contribui para facilitar o factor em causa

− **(influência negativa)** – contribui para dificultar o factor em causa

+/− **(influência variável)** – o seu contributo não é sempre o mesmo, depende de vários outros factores

X **(não tem influência)** – não interfere com o factor em causa

x **(influência mínima)** – contribui de forma pouco significativa para dificultar o factor em causa

= **(influência conservativa)** – contribui para manter a facilidade do factor em causa

No apêndice A mostra-se um exercício de aplicação em que se avaliaram 11 linguagens reais diferentes de acordo com as características aqui apresentadas, com o intuito de averiguar se todas elas são facilmente identificadas e medidas e de apreciar se no fim se obtém uma tabela que permita distinguir as linguagens entre si.

Analisando a Tabela de síntese acima, que evidência a influência das características no critério de qualidade, pode-se postular que:

Definição 2 (Aferição da Qualidade de uma Linguagem) *: A qualidade de uma linguagem, entendida como expresso na definição 1, pode ser decidida atestando a presença das características CL1 a CL8, sendo a linguagem tanto melhor quanto mais características verificar.*

A questão está agora em saber como verificar se a linguagem em avaliação tem essa característica ou não.

Manualmente é fácil constatar a presença dessas características. Tal como sucede na vida quotidiana com os bens de consumo, em que a presença de determinada característica num produto final, quando não é uma grandeza mensurável directamente, pode ser apurada através de ensaios cujos resultados são recolhidos em inquéritos e depois tabelados, também as características CL1 a CL8 (que não são mensuráveis) podem ser recolhidas sistematicamente por um humano dessa mesma forma.

Então o problema interessante que aqui se levanta é mesmo saber como conseguir fazê-lo de forma mais objectiva e automaticamente.

Assim sendo, o que se pretende discutir na continuação desta lição é até que ponto a qualidade de uma gramática pode ser medida automaticamente e pode ajudar a avaliar a qualidade da linguagem que define.

Capítulo 3

Gramaticando as Linguagens

Gramaticar uma linguagem[1] significa dar-lhe **forma** e **sentido** através de uma notação rigorosa para descrever as regras sintácticas e semânticas.

Para dar estrutura à linguagem, isto é, definir o seu vocabulário e as formas como os símbolos se podem agrupar para exprimir cada uma das ideias subjacentes à linguagem, defende-se o uso de uma GIC, ou seja uma Gramática Independente de Contexto, cuja definição é recordada abaixo. Para dar significado às frases e restringir algumas construções sintácticas que não fazem sentido, defende-se o uso de uma GA, ou seja uma Gramática de Atributos, que também se define abaixo.

As linguagens são, no âmbito da compilação e do processamento automático, formalmente especificadas por gramáticas; é pois expectável que a qualidade da primeira dependa da qualidade da segunda, como se dizia no fim do capítulo anterior, mesmo sendo certo que a qualidade de uma especificação ou a qualidade de uma ferramenta nunca podem implicar totalmente a qualidade do produto final.

3.1 Definições Gramaticais

Mas antes de avançar para essa temática, o foco da presente lição, relembram-se as definições básicas sobre as quais assentará o resto do discurso.

Ao iniciar esta secção será mais que justo prestar um singelo mas justo tributo ao linguista e filósofo (ainda vivo) que tornou possível o processamento automático de linguagens artificiais, concretamente ao americano Avram Noam Chomsky, nascido em Filadélfia a 7 de Dezembro de 1928 e ainda hoje professor de Linguística no Instituto de Tecnologia de Massachusetts (MIT). Noam Chomsky criou a noção de gramática generativa[2] transformacional e organizou as gramáticas em 4 grandes classes criando aquilo a que ainda hoje se chama *Hierarquia de Chomsky*. Devido à sua brilhante personalidade como pensador e ao seu enorme contributo para as ciências da computação e toda a Humanidade que actualmente depende do computador ficam aqui referidas algumas das suas principais obras escritas nesta área [Cho55b, Cho53, Cho55a, Cho56, Cho59b, Cho59a, Cho62, Cho57, Cho64, Cho65].

Definição 3 (GIC) *: Uma* Gramática Independente de Contexto *define-se como sendo um tuplo*

$$GIC =< T, N, S, P >$$

onde
T é o conjunto dos ***símbolos terminais*** *da linguagem (o* alfabeto *ou* vocabulário*).*

[1] Liberdade linguística da minha autoria.
[2] Em inglês, *gerative* ou *generative*.

N é o conjunto dos ***símbolos não-terminais*** *da gramática.*
$S \in N$ é o ***símbolo inicial*** *ou* axioma *da gramática.*
P é o ***conjunto de produções*** *ou* regras de derivação *da gramática.*
Cada produção $p \in P$ é uma regra da forma

$$p : X_0 \rightarrow X_1 \dots X_i \dots X_n$$

em que p é o identificador da regra, $\rightarrow$ é o operador derivação*, $X_0 \in N$ e $X_i \in (N \cup T),\ 1 \leq i \leq n$*
Numa produção com etiqueta p o lado esquerdo do operador de derivação, sempre um não-terminal, denota-se por LHS*(p)*[3] *e o lado direito do operador de derivação, uma sequência de símbolos terminais ou não-terminais, denota-se por* RHS*(p)*[4].
O conjunto T dos símbolos terminais *divide-se em 3 subconjuntos disjuntos — $T = PR \cup Sin \cup TV$ — das* ***Palavras-Reservadas****, dos* ***Sinais*** *e dos* ***Terminais-Variáveis***[5].

Dada uma GIC, o conceito de produção unitária é muito relevante para a discussão sobre qualidade e métricas gramaticais que será tida mais à frente e por isso será já introduzido aqui.

Definição 4 (Produção Unitária) *: Designa-se por* ***produção unitária****, uma produção $pu \in P$ com um único símbolo à direita, ou seja, da forma:*

$$X_0 \rightarrow X_1$$

e que é a única alternativa para X_0 (note-se que será sempre $X_0 \neq S$).
O conjunto dessas produções será denotado por PU, sendo $PU \subset P$.

Nestas circunstâncias alguns autores (p.ex., Hopcroft & Ullman em [HMU06b]) chamam a X_0 **símbolo inútil**[6] uma vez que acaba por ser o mesmo que X_1, ou seja não introduz qualquer riqueza estrutural adicional; é apenas um *alias*.

Apresenta-se abaixo, como exemplo de GIC, a gramática que define a linguagem funcional Lisp. Apesar de muito pequena é sempre um caso de estudo muito interessante por ser a gramática de uma linguagem de programação real e por ser altamente recursiva.

Exemplo 18 *: A gramática listada a seguir diz que uma frase (um programa) em* Lisp *é uma* symbolic expression *e que uma* symbolic expression *(`SExp`) é um valor atómico—número (`num`) ou nome (`pal`)—ou é uma lista de* symbolic expressions *(`SExplist`) entre parêntesis curvos.*

```
T = { num, pal, "(", ")" }
N = { Lisp, SExp, SExplist }
S = Lisp
P = {
      p0: Lisp     --> SExp
      p1: SExp     --> num
      p2: SExp     --> pal
      p3: SExp     --> "(" SExplist ")"
      p4: SExplist --> SExp SExplist
      p5: SExplist --> &
    }
```

[3] Do inglês, Left Hand Side.
[4] Do inglês, Right Hand Side.
[5] Também designados por muitos autores como Classes-Terminais.
[6] *Useless Symbol* em inglês.

Segundo esta definição, uma lista pode conter 0 ou mais elementos de qualquer tipo e por qualquer ordem; assim uma lista pode conter listas aninhadas até qualquer nível.
Duas frases válidas da linguagem gerada por esta gramática são:

```
(defun quad x (mul x x))           // quad(x) = x*x
(let res (add (mul 1 2)(div 8 4))) // res = 1*2 + 4/8
```

Esta gramática não tem produções unitárias; embora `p0` *assim o pareça, não cai na definição dada pois o símbolo do lado esquerdo é justamente o axioma,* `S`*, da gramática.*

Visto que uma gramática é uma especificação (e guia um processo), é importante que exista forma de certificar que está bem escrita já que é impossível provar que semanticamente a especificação traduz a ideia do seu criador. Para isso introduz-se a noção de gramática *bem-formada*.

Definição 5 (GIC bem-formada) *: Uma* Gramática Independente de Contexto *diz-se* ***bem-formada*** *se:*

- *para todo o $X \in N$ existir pelo menos 1 produção com X à esquerda;*
- *para todo o $X \in N$, X for* ***alcançável****, i.e., existir pelo menos 1 derivação a partir do axioma* que use *X;*
- *para todo o $X \in N$, X for* ***terminável*** *de acordo com a definição abaixo.*

Para terminar a definição anterior, diz-se que

Definição 6 (Símbolo Terminável) *Um símbolo gramatical diz-se* ***terminável*** *se:*

- *for um símbolo terminal.*
- *for um símbolo não-terminal e existir pelo menos 1 produção com esse símbolo à esquerda cujo lado direito seja uma* sequência terminável*.*

sendo que uma ***sequência de símbolos*** *$N \cup T$* ***é terminável*** *se:*

- *for a sequência vazia.*
- *cada símbolo dessa sequência for terminável.*

O exemplo seguinte clarifica estes conceitos.

Exemplo 19 *Apesar de altamente recursiva e pouco comum, a gramática da linguagem* Lisp*, acima apresentada no exemplo 18, é* bem-formada *porque:*

- *existe pelo menos 1 produção para cada um dos 3 símbolos não-terminais:* `p0` *para* `Lisp`*;* `p1, p2, p3` *para* `SExp`*;* `p4, p5` *para* `SExplist`*.*
- *os dois não-terminais além do axioma são alcançáveis a partir de* `Lisp`*:* `SExp` *é usada directamente em* `p0`*; e* `SExplist` *é usada indirectamente (via* `SExp`*) em* `p3`*.*
- *todos os não-terminais são termináveis:* `SExp` *é terminável graças a* `p1` *porque o seu* RHS *é um terminal (ou* `p2`*, pela mesma razão);* `SExplist` *é terminável graças a* `p5` *cujo* RHS *é a sequência vazia; por fim,* `Lisp` *é terminável por ambos os símbolos no* RHS*(*`p0`*) são termináveis.*

Para ajudar, mais à frente, na caracterização das gramáticas, introduz-se ainda o conceito de Grafo de Dependência entre os Simbolos.

Definição 7 (GDS) *: Chama-se* ***Grafo de Dependências entre Símbolos****,* GDS*, a um grafo em que os vértices são os símbolos $N \cup T$ da gramática e os ramos, ou arcos, vão do vértice Y para o vértice X sempre que existir em P uma produção p da forma $p: \; X \; \rightarrow \; \ldots \; Y \ldots$, dizendo-se então que X* `depende-de` *Y (porque Y entra na definição de X) ou que Y* `deriva-de` *X.*

O exemplo seguinte esclarece esta definição.

Exemplo 20 *No caso da gramática de* Lisp *do exemplo anterior, o* GDS *é formado pelos seguintes ramos:*

```
(SExp,Lisp)
(num,SExp)
(pal,SExp)
("(",SExp)
(SExplist,SExp)
(")",SExp)
(SExp,SExplist)
(SExplist,SExplist)
```

Um outro conceito fundamental, associado às gramáticas, quer do ponto de vista teórico da formalização, quer na perspectiva mais prática da implementação dos processadores, é a noção de árvore de derivação que se define a seguir.

Definição 8 (Árvore de Derivação duma produção) *: Associada a cada produção $p \in P$ da gramática existe uma* ***árvore de derivação*** *cuja raiz é X_0 e os descendentes, ou filhos, são os n símbolos X_i do lado direito.*

As árvores de derivação das produções podem juntar-se, colando na folha X_i de uma árvore, outra árvore que tenha X_i por raiz. Dessa forma se constrói uma **árvore de derivação** completa (AD).

Definição 9 (Árvore de Derivação) *: Começando por uma árvore cuja raíz é o símbolo inicial da gramática, isto é, uma árvore correspondente a uma produção como o axioma à esquerda, substitui-se cada um dos seus descendentes não-terminais por uma árvore que tenha esse símbolo por raíz e repete-se o processo sucessiva e recursivamente até que todas as folhas sejam símbolos terminais. A árvore completa assim formada a partir do axioma é uma* ***árvore de derivação*** *da gramática.*

Daqui surge a noção de frase válida.

Definição 10 (Frase Válida) *: A fronteira de uma árvore de derivação, isto é, a sequência de símbolos terminais obtida por concatenação das folhas, recolhidas da esquerda para a direita, é uma* ***frase válida da linguagem****. Além disso, uma frase (sequência de terminais) só é válida, só pertence à linguagem, se for fronteira de uma árvore de derivação dessa gramática.*

Assim sendo, gerar (ou derivar) uma frase corresponde a construir uma árvore de derivação; e validar uma frase (aceitá-la como pertencente a uma linguagem) corresponde a descobrir uma árvore de derivação para essa frase.

Definição 11 (Linguagem Não-Ambigua) *: Uma linguagem diz-se* ***não-ambigua*** *se para toda a frase válida existir uma e só uma árvore de derivação possível.*

Assim sendo, se a mesma frase poder ser derivada da gramática por mais do que uma árvore de derivação diferente, a linguagem diz-se ambigua e está fora do alcance das abordagens e técnicas de processamento formal e automático aqui discutido.
Embora não se use neste contexto como instrumento de reflexão ou trabalho, convém apresentar também a definição de Gramática Tradutora, uma vez que vai sendo mencionada aqui ou ali.

Definição 12 (GT) *: Uma* Gramática Tradutora *é uma extensão à* Gramática Independente de Contexto *para definir (semi-formalmente) como processar as frases da linguagem gerada pela* GIC*. Assim acrescenta-se à* GIC *um conjunto de* ***símbolos semânticos*** *AS (identificadores das Acções Semânticas) e modificam-se as produções de P, passando-se a ter o seguinte tuplo*

$$GT =< T, N, S, AS, Px >$$

onde
Cada produção $p \in Px$ passa agora a ser uma regra da forma

$$p : X_0 \rightarrow X_1 \dots X_i \dots X_n \ as$$

em que $as \in AS$ indica a Acção Semântica a executar[7] *após reconhecer todos os símbolos no lado direito da produção.*

Como foi dito atrás, para especificar a semântica da linguagem usa-se uma Gramática de Atributos, cuja defnição se segue.

Definição 13 (GA) *: Uma* Gramática de Atributos *define-se como sendo um tuplo*

$$GA =< GIC, A, RC, CC, RT >$$

onde
$A = \bigcup A(X), \forall X \in (N \cup T)$ *é o conjunto dos* ***atributos*** *de todos os símbolos da gramática.*
$RC = \bigcup RC(p), \forall p \in P$ *é o conjunto das* ***regras de cálculo dos atributos*** *em todas as produções da gramática.*
$CC = \bigcup CC(p), \forall p \in P$ *é o conjunto das* ***condições de contexto*** *em todas as produções da gramática.*
$RT = \bigcup RT(p), \forall p \in P$ *é o conjunto das* ***regras de tradução*** *em todas as produções da gramática.*
Os atributos $A(X)$ de cada símbolo dividem-se em dois subconjuntos disjuntos

$$A(X) = AI(X) \cup AS(X), \quad AI(X) \cap AS(X) = \emptyset$$

em que
$AI(X)$ *é o conjunto dos* ***atributos herdados*** *do símbolo X, ou seja, dos atributos que transportam a informação do contexto esquerdo ou direito pela subárvore abaixo*[8]*.*
$AS(X)$ *é o conjunto dos* ***atributos sintetizados*** *do símbolo X, ou seja, dos atributos que sintetizam a informação a partir das folhas e a transportam pela árvore acima*[9]*.*

[7] Pelo facto de apenas indicar o nome da acção e não descrever de qualquer forma essa acção, é que se afirma que é uma especificação semi-formal.

[8] Como é óbvio, o axioma e os terminais não têm atributos herdados.

[9] Nas abordagens convencionais, os atributos associados aos símbolos terminais, como são calculados ao nível da análise léxica e são usados para injectar os valores iniciais a partir dos quais se sintetizam os demais, são normalmente agrupados à parte e designados por atributos *intrínsecos*. Porém, neste trabalho serão considerados indistintamente como atributos *sintetizados*.

O conjunto $A(X)$ caracteriza completamente, do ponto de vista semântico, cada símbolo $X \in N \cup T$. Numa frase concreta cada instância do símbolo X na árvore de derivação ficará associada a um conjunto de **propriedades**—*atributos com os seus valores concretos* que descrevem completamente o seu significado. Uma árvore de derivação assim preenchida com as propriedades de cada símbolo, ou seja carregando o significado de cada nodo, chama-se uma **árvore de derivação decorada** (ou abreviadamente, uma árvore decorada). As regras de cálculo indicam precisamente como valorar os atributos, face ao contexto concreto de X, para obter o dito significado. A partir do significado de cada símbolo constrói-se o significado da frase, que se considera como sendo dado pelas propriedades do axioma da gramática (precisamente a raíz da árvore decorada). Tendo o significado da frase, e de cada símbolo, é possível transformar, ou traduzir, a frase para obter o resultado pretendido—para definir rigorosamente essa transformação usam-se as regras de tradução. Porém uma frase só pode ser processada se for sintacticamente válida e se estiver correcta semanticamente. A correcção sintática é assegurada pelas produções da GIC; a validade semântica é estabelecida pelo conjunto de condições de contexto da GA, as quais explicitam, ao nível de cada produção, as restrições que os valores concretos dos atributos terão de respeitar para a frase *fazer sentido*.

Dada a quantidade de informação que tem de ser fornecida para desenvolver a GA de uma linguagem real, propõe-se que esse *desenvolvimento seja feito incrementalmente*, orientado por objectivos a atingir[10]. No exemplo seguinte e no resto da lição esclarece-se pragmaticamente esta abordagem.

Abaixo retoma-se o exemplo anterior—a gramática da linguagem Lisp—para ilustrar cada um dos elementos de uma GA.

Exemplo 21 *: Neste exemplo toma-se por base a* GIC *acima que define* symbolic expressions *e constrói-se gradualmente uma* GA *para:*

1. *calcular a quantidade de números e palavras da lista*
2. *construir uma lista plana (todos os elementos ao mesmo nível) com os elementos originais associados ao nível a que aparecem*
3. *verificar que todos os elementos da lista sejam do mesmo tipo do 1º elemento*
4. *gerar código post-fix como se a SExp fosse calculada numa máquina de stack.*

1º passo – cálculo dos totais de elementos de cada tipo:
Antes de escrever a GA *retomam-se as duas frases válidas do exemplo acima para mostrar os valores a calcular*

```
(defun quad x (mul x x))           ==> Lisp.contaN=0; Lisp.contaP=6
(let res (add (mul 1 2)(div 8 4))) ==> Lisp.contaN=4; Lisp.contaP=5
```

Para especificar este cálculo definem-se os seguintes atributos:

```
AS(Lisp) = { contaN:int, contaP:int }
AI(SExp) = AI(SExplist) = { contaN_in:int,  contaP_in:int }
AS(SExp) = AS(SExplist) = { contaN_out:int, contaP_out:int }
```

e as seguintes regras de cálculo:

[10]Isto é, em função das informações a retirar da frase (valores a calcular), condições contextuais a validar, ou resultados a produzir com a tradução.

```
RC(p0) = { SExp_1.contaN_in=0; SExp_1.contaP_in=0;
           Lisp_0.contaN=SExp_1.contaN_out; Lisp_0.contaP=SExp_1.contaP_out }
RC(p1) = { SExp_0.contaN_out=SExp_0.contaN_in+1; SExp_0.contaP_out=SExp_0.contaP_in }
RC(p2) = { SExp_0.contaN_out=SExp_0.contaN_in;   SExp_0.contaP_out=SExp_0.contaP_in+1 }
RC(p3) = { SExp_0.contaN_out=SExplist_2.contaN_out; SExp_0.contaP_out=SExplist_2.contaP_out;
           SExplist_2.contaN_in=SExp_0.contaN_in; SExplist_2.contaP_in=SExp_0.contaP_in }
RC(p4) = { SExplist_0.contaN_out=SExplist_2.contaN_out; SExplist_0.contaP_out=SExplist_2.contaP_out;
           SExp_1.contaN_in=SExplist_0.contaN_in; SExp_1.contaP_in=SExplist_0.contaP_in;
           SExplist_2.contaN_in=SExp_1.contaN_out; SExplist_2.contaP_in=SExp_1.contaP_out }
RC(p5) = { SExplist_0.contaN_out=SExplist_0.contaN_in; SExplist_0.contaP_out=SExplist_0.contaP_in }
```

2º passo – linearização da lista com níveis:
De novo antes de escrever a GA *retomam-se as duas frases válidas do exemplo acima e mostram-se as listas a calcular.*

```
(defun quad x (mul x x))           ==> Lisp.lista=<(defun,1),(quad,1),(x,1),
                                                   (mul,2),(x,2),(x,2)>
(let res (add (mul 1 2)(div 8 4))) ==> Lisp.lista=<(let,1),(res,1),(add,2),
                                         (mul,3),(1,3),(2,3),(dic,3),(8,3),(4,3)>
```

Para realizar esta operação é agora necessário definir os seguintes atributos:

```
AS(Lisp) = { lista:seq(pares(atomo,int)) }
AI(SExp) = AI(SExplist) = { nivel_in:int, lista_in:seq(par(atomo,int)) }
AS(SExp) = AS(SExplist) = { lista_out:seq(par(atomo,int)) }
AS(num)  = AS(pal)      = { val:atomo }
```

e as seguintes regras de cálculo:

```
RC(p0) = { SExp_1.nivel_in=0; SExp_1.lista_in=NULL;
           List_0.lista=SExp_1.lista_out }

RC(p1) = { SExp_0.lista_out=insere1(num_1.val,SExp_0.nivel_in, SExp_0.lista_in) }
RC(p2) = { SExp_0.lista_out=insere1(pal_1.val,SExp_0.nivel_in, SExp_0.lista_in) }

RC(p3) = { SExplist_2.nivel_in=SExp_0.nivel_in+1; SExplist_2.lista_in=SExp_0.lista_in;
           SExp_0.lista_out=SExplist_2.lista_out }
RC(p4) = { SExp_1.nivel_in=SExplist_0.nivel_in; SExplist_2.nivel_in=SExplist_0.nivel_in;
           SExp_1.lista_in=SExplist_0.lista_in; SExplist_2.lista_in=SExp_1.lista_out;
           SExplist_0.lista_out=SExplist_2.lista_out }
RC(p5) = { SExplist_0.lista_out=SExplist_0.lista_in }
```

3º passo – verificar a restrição ao tipo dos elementos da lista:
Considerando de novo as duas frases válidas do exemplo acima, mostra-se a seguir o resultado da verificação:

```
(defun quad x (mul x x))           ==>  //não é emitida qualquer mensagem de erro
                                        // ...(são todos "nomes")
(let res (add (mul 1 2)(div 8 4))) ==> "Erro 1: 'numero' quando devia ser 'nome'"
                                       //esta msg aparece 4 vezes
```

Para realizar esta validação é ainda necessário definir os seguintes atributos:

```
AI(SExp) = AI(SExplist) = { tipo_in: enum(nil,num,pal) }
AS(SExp) = AS(SExplist) = { tipo_out:enum(nil,num,pal) }
```

e as seguintes regras de cálculo

```
RC(p0) = { SExp_1.tipo_in=nil }
RC(p1) = { se (SExp_0.tipo_in == nil)
              entao {SExp_0.tipo_out=num} senao {SExp_0.tipo_out=SExp_0.tipo_in} }
RC(p2) = { se (SExp_0.tipo_in == nil)
              entao {SExp_0.tipo_out=pal} senao {SExp_0.tipo_out=SExp_0.tipo_in} }
RC(p3) = { SExp_0.tipo_out=SExplist_2.tipo_out; SExplist_2.tipo_in=SExp_0.tipo_in }
RC(p4) = { SExplist_0.tipo_out=SExplist_2.tipo_out;
           SExp_1.tipo_in=SExplist_0.tipo_in; SExplist_2.tipo_in=SExp_1.tipo_out }
RC(p5) = { SExplist_0.tipo_out=SExplist_0.tipo_in }
```

sendo então possível definir as seguintes condições contextuais (ou restrições semânticas):

```
CC(p1) = { se (SExp_0.tipo_in == pal) entao {erro(semantica,1)} }
CC(p2) = { se (SExp_0.tipo_in == num) entao {erro(semantica,2)} }
```

4º passo – geração de código post-fix:
Gerando código post-fix para uma linguagem simples com 3 instruções:

```
LAB pal  // LABel significa que o argumento 'pal' é uma constante do tipo alfanumérico
                                                          (identificador ou etiqueta)
CONS num // CONS significa que o argumento 'num' é uma constante do tipo numérico
OPER op  // OPER significa que o argumento 'op' é um operador
```

o resultado de processar as duas frases válidas do exemplo acima seria:

```
(defun quad x (mul x x))    ==> LAB quad, LAB x, LAB x, LAB x, OPER mul, OPER defun
(let res (add (mul 1 2)(div 8 4))) ==> LAB res, CONS 1, CONS 2, OPER mul, CONS 8,
                                                CONS 4, OPER div, OPER add, OPER let
```

Para realizar esta operação é agora necessário definir os seguintes atributos:

```
AS(Lisp) = { codigo:seq(inst-maq) }
AI(SExp) = AI(SExplist) = { cod_in:seq(inst-maq)) }
AS(SExp) = AS(SExplist) = { cod_out:seq(inst-maq)) }
```

e as seguintes regras de cálculo:

```
RC(p0) = { SExp_1.cod_in=NULL;
           List_0.codigo=SExp_1.cod_out }

RC(p1) = { SExp_0.cod_out=insere2("CONS"++num_1.val, SExp_0.cod_in) }
RC(p2) = { SExp_0.cod_out=insere2("LAB" ++pal_1.val, SExp_0.cod_in) }
RC(p3) = { SExplist_2.cod_in=SExp_0.cod_in;
           SExp_0.cod_out=append(tail(SExplist_2.cod_out),conv_oper(head(SExplist_2.cod_out)) }

RC(p4) = { SExp_1.cod_in=SExplist_0.cod_in; SExplist_2.cod_in=SExp_1.cod_out;
           SExplist_0.cod_out=SExplist_2.cod_out }
RC(p5) = { SExplist_0.cod_out=SExplist_0.cod_in }
```

É então possível definir a regra de tradução que grava em ficheiro o código sintetizado.

```
RT(p0) = { grava(Lisp_0.codigo); }
```

A Gramática de Atributos *que especifica completamente a semântica da linguagem* Lisp *definida no início deste exemplo, obtém-se fundido as várias partes construídas nos 4 passos apresentados; o resultado final é mostrado no apêndice F.*
Nota: As funções/procedimentos auxiliares usadas ao longo do exemplo,

```
insere1(atomo,int, seq(par(atomo,int))) -> seq(par(atomo,int))
insere2(inst-maq, seq(inst-maq)) -> seq(inst-maq)
append(seq(inst-maq),inst-maq) -> seq(inst-maq)
tail(seq(inst-maq)) -> seq(inst-maq)
head(seq(inst-maq)) -> inst-maq
conv_oper(inst-maq) -> inst-maq

grava(inst-maq) -> void
erro(tipo_erro,codigo_erro) -> void
```

têm o significado óbvio e a sua implementação não será aqui detalhada.

No apêndice G testa-se a GA construida no exemplo acima, usando a ferramenta AnTLR.
As regras de cálculo, as condições de contexto e as regras de tradução afectas a uma produção $p \in P$ induzem localmente dependências entre os atributos associados aos símbolos $N \cup T$ dessa produção, dependências essas que é fundamental conhecer para determinar a ordem de cálculo correcta[11]. Para estudar essas dependências e determinar a ordem de cálculo, usa-se um grafo, dito Grafo de Dependências Local à produção p (GDL(p)) no qual os vértices são todas as ocorrências de atributos de todos os símbolos que intervém na produção e os ramos vão de uma ocorrência para outra sempre que a primeira é usada no lado direito de uma regra de cálculo com a segunda à esquerda, ou a primeira é uma parâmetro de uma condição de contexto ou de uma regra de tradução (a qual será nesse caso o vértice destino do ramo). Este conceito de Grafo de Dependências Local a uma produção é exemplificado no apêndice F onde se apresentam os GDL(.) associados a três das produções, visto que os ditos grafos serão depois usados para cálculo de uma das métricas definida adiante.
Tal como se fez atrás para as GIC, define-se abaixo o conceito de GA *bem-formada*, que permite ter confiança na forma como foi escrita a especificação sintáctico-semântica da linguagem.

Definição 14 (GA bem-formada) *: Uma* Gramática de Atributos *diz-se* ***bem-formada*** *se:*

- *a* GIC *subjacente for* bem-formada*;*
- *para cada produçao $p \in P$ for fornecida uma e uma só regra da forma*

$$X_i.a = f(...X_j.b...);$$

para cálculo dos atributos a de X_i podendo ser usados os atributos b de X_j, de acordo com as restrições seguintes.

  - *Os atributos a a calcular terão se ser obrigatoriamente os sintetizados de X_0 e os herdados de $X_i,\ 1 \leq i \leq n$;*
  - *Na fórmula de cálculo podem ser usados atributos b herdados de X_0 ou sintetizados de $X_j,\ 1 \leq j \leq n$.*

[11] Antes de calcular um atributo de um símbolo é preciso garantir que o valor de todos os atributos dos quais ele depende já foram calculados; de igual forma, o teste de uma condição contextual, ou a realização de uma acção de tradução, requer que todos os argumentos já estejam valorados.

- *se o Grafo de Dependências induzido sobre cada árvore de derivação por essas regras de cálculo for* aciclico.

Note-se que não há imposições aos atributos a usar como argumentos dos predicados das condições de contexto nem dos procedimentos das regras de tradução, conquanto se respeite sempre o princípio da localidade *que diz que numa produção $p \in P$ só podem ser referidos os atributos (sintetizados ou herdados) afectos aos símbolos $N \cup T$ dessa produção.*

3.2 Processando uma Linguagem

Seja `txt-fnt` um *texto-fonte*, ou seja, uma sequência de caracteres tipicamente armazenado num ficheiro sequencial. Nas definições que se seguem supõe-se que `txt-fnt` é lido por um analisador léxico que, depois de desprezar todos os caracteres de formatação comentários e meta-informação, junta os demais caracteres em palavras e reconhece nelas os vocábulos, ou símbolos terminais, da gramática (os elemento de T). O analisador léxico devolve uma sequência de símbolos que se designa a seguir por *frase*; devolverá erro caso não consiga reconhecer apenas palavras válidas, i. é, que correspondem a símbolos.

Na prática, quando é necessário reconhecer uma frase é conveniente considerar que o texto-fonte termina com um símbolo especial, o fim-de-ficheiro (`eof`) normalmente representado pelo símbolo "$". Para formalizar tal questão estende-se a GIC com um novo axioma Z e uma nova produção da forma

$$Z \ \rightarrow \ S \ \$$$

Conforme se verá mais à frente, o símbolo que denota o `eof`, embora não pertencendo ao alfabeto (T), conta como terminal nas estruturas de dados que suportam os algoritmos de reconhecimento.

Definem-se a seguir, de uma forma genérica, os programas que manipulam frases.

Definição 15 (Processador) *: Dada uma* GA *G e uma frase f, um* ***processador da linguagem*** *$\mathcal{L}(\mathcal{G})$ é uma função*

$$processador :: G \ \times f \ \rightarrow \ res$$

que reconhece f como frase válida da linguagem, $f \in \mathcal{L}(\mathcal{G})$, e a transforma num resultado qualquer.

Para realizar a sua tarefa, um processador de linguagens procede em duas fases: primeiro reconhece e depois transforma. Pelo meio, ou seja entre estas duas fases principais, existe ainda uma etapa de verificação. Assim define-se:

Definição 16 (Reconhecedor) *: Dada uma* GA *G e uma frase f, um* ***reconhecedor da linguagem*** *$\mathcal{L}(\mathcal{G})$ é uma função*

$$reconhecedor :: G \ \times f \ \rightarrow \ ASAD$$

*que reconhece a estrutura, ou sintaxe, e a semântica de f e devolve o seu significado na forma de uma árvore de sintaxe abstracta decorada (*ASAD*).*

e ainda:

Definição 17 (Tradutor) *: Dada uma* GA *G e uma árvore de sintaxe abstracta decorada,* ASAD*, um* ***tradutor da linguagem*** *$\mathcal{L}(\mathcal{G})$ é uma função*

$$tradutor :: G \ \times ASAD \ \rightarrow \ res$$

que recebe o significado de uma frase semanticamente válida $f \in \mathcal{L}(\mathcal{G})$, na forma de uma ASAD*, e lhe aplica uma série de regras de transformação de maneira a produzir o resultado desejado.*

Definição 18 (Verificador) *: Dada uma* GA *G e uma árvore de sintaxe abstracta decorada,* ASAD*, um* ***verificador, ou validador da linguagem*** *$\mathcal{L}(\mathcal{G})$ é uma função*

$$verificador :: G \times ASAD \rightarrow ASAD$$

que recebe uma árvore de sintaxe (abstracta) com os valores dos atributos já calculados e associados aos nodos e verifica se é uma frase semanticamente válida, $f \in \mathcal{L}(\mathcal{G})$, isto é, se as restrições contextuais são verificadas pelos valores concretos dos atributos. Caso a árvore dada não corresponda a uma frase com sentido (não seja válida), o verificador devolve uma árvore vazia.

Sendo assim, é possível dizer-se que um processador resulta de compor estas três funções:

$$processador(G, f) \equiv tradutor(verificador(reconhecedor(G, f)))$$

Na pratica há duas modos distintos de implementar este processo. Um **compilador** processa uma linguagem em modo não-interactivo. Após a edição da frase completa (o programa, ou a descrição/especificação), o compilador procede à sua leitura, reconhecimento e tradução integral sem intervenção nem diálogo com o utilizador; no fim devolve o resultado esperado ou reporta todos os erros detectados.
Em contrapartida, um **interpretador** comporta-se de forma oposta, sendo um processo interactivo. O interpretador vai processando a frase durante a edição, reconhecendo e traduzindo as sub-frases com significado e reportando os erros à medida que os encontra; dá ao utilizador a oportunidade de rever e corrigir a frase à medida que a mesma vai sendo processada e validada.
Enquanto que o comportamento do interpretador previlegia a fase de desenvolvimento, pela interacção permitida, o funcionamento do compilador é preferível numa fase de geração do resultado final, pois permite optimizar o resultado produzido.

A complexidade do processo de reconhecimento requer vulgarmente que se decomponha esta tarefa em duas partes que se definem a seguir.

Definição 19 (*Parser*) *: Dada uma* GA *G e uma frase f, um* ***analisador sintáctico, ou*** **Parser*****, da linguagem*** *$\mathcal{L}(\mathcal{G})$ é uma função*

$$analSint :: G \times f \rightarrow ASA$$

*que verifica se f é uma frase sintacticamente válida (ou correcta) e devolve a sua estrutura gramatical (isto é, as produções usadas na sua derivação) na forma de uma árvore de sintaxe abstracta (*ASA*). O analisador sintáctico devolve uma árvore vazia se a frase f não pertence à linguagem, isto é, se não existe nenhuma árvore de derivação com raiz em S (axioma de G) cuja fronteira seja f.*

Definição 20 (Calculador-de-Atributos) *: Dada uma* GA *G e uma árvore de sintaxe abstracta* ASA*, um* ***analisador semântico, ou calculador-de-atributos, da linguagem*** *$\mathcal{L}(\mathcal{G})$ é uma função*

$$analSem :: G \times ASA \rightarrow ASAD$$

*que calcula os atributos dos símbolos nos nodos da árvore de sintaxe abstracta e devolve a árvore decorada (*ASAD*).*

É então possível definir o reconhecimento com base nas duas componentes:

$$reconhecedor(G, f) \equiv analSem(analSint(G, f))$$

Um **Tradutor Dirigido pela Semântica** segue à risca esta definição teórica e implementa cada fase de análise separadamente, bem como separadando o reconhecimento da verificação e tradução, enquanto que um **Tradutor Dirigido pela Sintaxe** simplifica o processo e implementa a análise semântica, verificação e tradução em simultaneo e sob controlo da análise sintática.

O reconhecimento, ou análise sintáctica é talvez a operação mais complexa e delicada, pelo que foi alvo de inúmeros estudos. No apêndice C recordam-se os principais algoritmos de parsing, de modo a clarificar a referência que lhes é feita nas secções seguintes. A seguir definem-se as duas estratégias, ou abordagens possíveis ao problema.

Definição 21 (*Parser* TD) *: Um* Parser *diz-se* Top-Down *(*TD*), ou descendente, se reconstrói a árvore de derivação da raiz (de cima) para as folhas (para baixo).*
Nesta abordagem preditiva*, o analisador tem sempre um símbolo($N \cup T$) para reconhecer; se for um terminal, pois o próximo símbolo de f terá de ser precisamente esse símbolo—se for, avança-se na frase para o próximo, se não for, termina-se em erro; se for um não-terminal, escolhe-se uma produção*[12] *em que esse símbolo seja o* LHS *e substitui-se o objectivo de reconhecer o símbolo pelo objectivo de reconhecer todos os símbolos do* RHS *dessa produção. O processo começa colocando o axioma como objectivo a reconhecer e termina quando se chegar a um erro, ou quando não houver mais símbolos para reconhecer.*

Há duas instâncias muito conhecidas desta abordagem TD, ambas recordadas no apêndice C: o *Parser* Recursivo-Descendente e o *Parser* LL.

Definição 22 (*Parser* BU) *: Um* Parser *diz-se* Bottom-Up *(*BU*), ou ascendente, se reconstrói a árvore de derivação das folhas (de baixo) para a raiz (para cima).*
Nesta abordagem expectativa*, o analisador vai acumulando símbolos da frase f numa stack até reconhecer um sufixo que seja o* RHS *de uma produção da gramática; nessa altura, substitui esse sufixo pelo símbolo do* LHS *respectivo e junta-o á stack. O processo começa com a stack vazia e termina quando na stack estiver apenas o axioma da gramática.*

A instanciação mais conhecida desta estratégia BU é o *Parser* LR, também referido no apêndice C.

[12] Caso não seja possível escolher nenhuma, atinge-se uma situação de erro.

Capítulo 4

Qualidade das Gramáticas

Discutir a qualidade de uma gramática, sendo esta um instrumento de trabalho, consiste em verificar se cumpre bem as funções para que foi concebida. Relativamente à discussão análoga, tida no capítulo 2, é agora possível estabelecer um conjunto de métricas que vão permitir aferir quantitativamente os critérios de qualidade, por se tratar de um objecto rigorosamente definido.
Tal como foi explicado no capítulo 1, uma gramática desempenha um duplo papel:

- define (ou gera) uma linguagem e, portanto, diz como se escrevem frases válidas[1];
- guia o reconhecimento das frases dessa linguagem que gera.

Esta segunda função de suporte ao reconhecimento é usada, quer pelo humano que tem de compreender as frases que alguém derivou da gramática para delas retirar o respectivo significado, quer pelos programas encarregues de fazerem o processamento (reconhecimento e transformação) dessas frases. No contexto da Engenharia Gramatical, é esta última vertente que tem particular interesse na medida em que é possível derivar tais programas sistemática e automaticamente a partir da gramática[2].
Sendo assim, o estudo que se segue será organizado segundo estas duas ópticas: gramática como geradora de linguagens; gramática como geradora de programas[3]. Na realidade não parece possível falar-se de qualidade de uma gramática em termos absolutos, de forma geral, visto que estes dois eixos tem exigências muitas vezes opostas.

Note-se que nesse estudo está-se interessado em avaliar uma gramática concreta e sua influência na linguagem e processadores específicos que dela derivam; não se discutem abordagens nem notações (meta-linguagens), específicas de sistemas/ambientes concretos para desenvolvimento de compiladores, para criar ou escrever gramáticas.

4.1 Caracterização de Gramáticas

Enumeram-se abaixo as Características para avaliar a qualidade de uma Gramática ou permitir compará-la com outras segundo as duas perspectivas acima identificadas.

[1] Com a forma correcta e com sentido.

[2] Este processo de produzir o código dos Processadores de Linguagens automaticamente, a partir da Gramática da Linguagem, é implementado pelos chamados Geradores de Compiladores, GCs (em inglês *Compiler Generators*) ou Compiladores de Compiladores, CCs (em inglês *Compiler Compilers*).

[3] Os processadores para a linguagem gerada pela gramática.

- (**CG1**, como *geradora de linguagens*) **usabilidade** da gramática enquanto instrumento para derivar frases de uma linguagem:
  - (CG1.1) facilidade de compreensão
  - (CG1.2) facilidade de derivação
  - (CG1.3) facilidade de manutenção
- (**CG2**, como *geradora de programas*) **eficiência** da gramática enquanto instrumento para derivar processadores para uma linguagem:
  - (CG2.1) eficiência no reconhecimento (processamento) das frases da linguagem gerada.
  - (CG2.2) eficiência na geração automática do processador.

Definição 23 (Qualidade de uma Gramática) *: A **qualidade de uma gramática, enquanto especificação que gera uma linguagem** afere-se em termos da facilidade com que se aprende (lê e compreende o que ela descreve), se usa para derivar frases e se mantém (correctiva ou evolutivamente). Neste sentido, diz-se, então, que uma* gramática tem qualidade *se facilita a usabilidade.*
*A **qualidade de uma gramática, enquanto especificação que gera um processador** afere-se em termos da eficiência do programa que dela deriva e da eficiência do próprio processo de geração. Neste sentido, diz-se, então, que uma* gramática tem qualidade *se permite gerar processadores de linguagens eficientes sem degradar a facilidade de geração automática.*

Discutem-se a seguir as características, acima enumeradas, que definem a qualidade de uma gramática. Para manter a discussão mais concisa e clara, faz-se uma primeira caracterização da Gramática Independente de Contexto, a qual só define a sintaxe da linguagem, e depois na subsecção seguinte caracteriza-se a Gramática de Atributos, que especifica ainda a semântica da linguagem.

4.1.1 Caracterização de GICs

Nesta subsecção estuda-se a usabilidade e eficiência de uma gramática, enquanto geradora de uma linguagem e enquanto geradora de um programa, tendo em atenção que o objecto em apreço é uma Gramática Independente de Contexto, conforme definida em 3.1.

(CG1) Usabilidade da gramática é a clareza e facilidade com que o utilizador final lê a gramática e a usa, enquanto instrumento para derivar frases da linguagem que essa gramática define, e o engenheiro gramatical a compreende e mantém.
Faz-se aqui referência separada aos dois grupos de pessoas que manuseiam uma gramática porque as perspectivas do utilizador final e do engenheiro gramatical são parcialmente distintas. Ambos têm de a ler com facilidade para a compreender, mas o primeiro olha-a mais num perspectiva linguistica, de quem tem que escrever frases para comunicar suas ideias, enquanto que o segundo a olha mais numa perspectiva técnica, de quem tem de cumprir os requisitos especificando ou mantendo a linguagem que lhe encomendam. O primeiro está mais preocupado com a semântica e a estética da frase que vai derivar; o segundo mais preocupado com *desenhar* uma linguagem que satisfaça o pedido mas aumentando a eficiência do seu processamento. De qualquer forma, a característica que aqui se discute é a clareza—que ajuda a uma rápida e fácil compreensão daquilo que a gramática descreve—e, nesse sentido, discute-se o que para ambos os grupos é comun; a perspectiva de engenharia de quem cria/mantem a gramática será discutida no contexto da próxima secção.
Como foi dito atrás, esta característica geral de **usabilidade** desmonta-se em três componentes: compreensão, derivação e manutenção; serão essas vertentes que se analisam agora.
A **compreensão** (CG1.1) está intrinsecamente relacionada com os seguintes elementos objectivos:

- a escolha dos identificadores para os símbolos terminais e não-terminais
- o uso de produções unitárias
- o comprimento dos lados direitos das produções (número de símbolos nos RHS)
- a notação empregue (BNF puro ou extendido)
- o esquema de recursividade (direita ou esquerda, directa ou indirecta) adoptado

O exemplo abaixo clarifica os 3 primeiros conceitos.

Exemplo 22 *: Veja-se a diferença entre `G1` e `G2`*

```
G1                                    G2
Colaborad --> Dados Tarefa             Col    --> Dds Trf
Dados     --> Identif PeriodVal        Dds    --> Id PV
PeriodVal --> Intervalo                PV     --> Interv
Intervalo --> DatInic DatFim           Interv --> DI DF
DatInic   --> data                     DI     --> data
DatFim    --> data                     DF     --> data
.....                                  .....
```

em que na primeira se recorre a longos identificadores, esclarecedores dos conceitos em jogo, e na segundo se reduzem esses identificadores a um mínimo de letras de significado enigmático. Parece óbvio que a primeira facilita a vida ao utilizador final no sentido de o ajudar a perceber o que representam as várias componentes das frases da linguagem.
Para o engenheiro gramatical que tem de trabalhar a gramática escrita por outro para afinar ou modificar a linguagem, é evidente que `G1` também lhe facilita a vida; mas para aquele que cria a gramática com vista a automatizar o desenvolvimento dum processador para a linguagem, `G2` é sem dúvida mais simples, porém é decisivamente menos legível.
Comparando agora `G1` com `G3`, abaixo, nota-se que o facto de `G1` ter 3 produções unitárias a torna mais clara e fácil de compreender do que a alternativa `G3`, em que essas 3 produções foram eliminadas.

```
G3
Colaborad --> Dados Tarefa
Dados     --> Identif Intervalo
Intervalo --> data data
.....
```

De forma semelhante, comparando agora `G1` com `G4`, pode observar-se que gramáticas com lados direitos mais longos, isto é, mais símbolos à direita, tornam-se mais confusas sendo portanto menos legíveis.

```
G4
Colaborad --> Identif DatInic DatFim Tarefa
DatInic   --> data
DatFim    --> data
.....
```

Quanto aos dois outros elementos, que estão relacionados entre si, têm claro impacto na compreensão mas a sua influência não é sempre no mesmo sentido, como se verá no exemplo seguinte.

Exemplo 23 *: Considere-se de novo a gramática* `G1` *do exemplo 22, supondo que se pretende agora dizer que uma frase pode descrever 1 ou mais colaboradores (*`Colaborad`*).*
A primeira decisão a tomar é se se usa a notação BNF *pura ou extendida.*
Em pure-BNF *a repetição de símbolos (as listas) é descrita recorrendo explicitamente à recursividade; nesse caso, poderia ser usado qualquer um dos esquemas* `G1a`*,* `G1b` *(ambos com recursividade à direita) ou* `G1c` *(com recursividade à esquerda) que se ilustra a seguir.*

```
G1a                                  G1b
Colaboras --> Colaborad              Colaboras --> Colaborad Outros
Colaboras --> Colaborad Colaboras    Outros    --> &
Colaborad --> Dados Tarefa           Outros    --> Colaborad Outros
.....                                .....

G1c
Colaboras --> Colaborad
Colaboras --> Colaboras Colaborad
.....
```

Também a variante `G1d`*, a seguir—que corresponde a um caso de recursividade à direita, mas agora indirecta—poderia ser usada como alternativa a* `G1b`*.*

```
G1d
Colaboras --> Colaborad Outros
Outros    --> &
Outros    --> Colaboras
.....
```

Em ex-BNF *a repetição de símbolos é expressa através do uso dos operadores iterativos + (para dizer 1 ou mais vezes),* AST *(para dizer 0 ou mais vezes) ou ? (para dizer 0 ou 1 vez), como se ilustra na variante G5 abaixo.*

```
G5
Colaboras --> Colaborad+
Colaborad --> Dados Tarefa
.....
```

A preservação da mesma notação e do mesmo esquema recursivo ao longo de toda a gramática é, sem dúvida, um factor importante para uma mais fácil e rápida compreensão.
Quanto ao tipo de notação usada ou ao esquema seguido (recursividade à direita, ou à esquerda), embora tenham nítida influência no processo de compreensão e de utilização da gramática (isto é, na sua legibilidade), não é fácil dizer se interferem positiva ou negativamente, pois essa influencia depende muito da formação de base de quem manuseia a gramática. Tipicamente para um engenheiro gramatical a notação ex-BNF *é preferível por ser mais concisa e sucinta, enquanto que qualquer esquema de recursividade lhe é familiar e igualmente legível; para um utilizador final, sem formação prévia em linguagens (nem em notações do tipo das expressões regulares) ou com pouca formação em matemática, tipicamente a notação* pure-BNF *é mais amigável, bem como a recursividade à direita lhe parece mais natural e fácil de aprender.*
Em geral o uso de recursividade indirecta, tal como o esquema misto, priora a leitura e compreensão.

Porém os elementos que ajudam a tornar uma gramática facilmente compreendida, não são sempre os mesmos que a tornam fácil de a usar para derivar frases ou de a manter.

No caso da **derivação** (CG1.2), a redução do número de produções (p.ex., eliminando produções unitárias) e do número de não-terminais (p.ex., aumentado o comprimento dos lados direitos) facilita o processo.
Por isso mesmo, também se constata que a notação ex-BNF é mais cómoda, ou a recursividade à direita mais natural e portanto facilitadora da geração de frases.
Em qualquer circunstância é óbvio que a conservação da mesma notação/esquema e o uso de identificadores esclarecedores para os símbolos são factores que intervêm positivamente para a realização desta tarefa.

Quanto à **manutenção** de uma gramática (CG1.3)—tarefa realizada pelo engenheiro gramatical para corrigir deficiências na linguagem gerada ou introduzir modificações nessa linguagem—além dos elementos que contribuem positivamente para a leitura e compreensão (identificadores esclarecedores, recurso a produções unitárias, lados direitos curtos), surgem mais dois elementos importantes:

- modularidade
- complexidade

A modularidade, no sentido de construir uma gramática importando outras subgramáticas já existente que depois se reutilizam e completam, ajuda a criação e a manutenção. Talvez para compreender, ou mesmo derivar, não seja tão fácil, mas para manter fica mais organizado e permite um trabalho por partes, mais seguro. Mecanismos de herança e extensão, típicos das abordagens orientadas a objectos, podem ser úteis e ajudar nesta tarefa.
Entendendo que a complexidade de uma gramática diz respeito à forma como os símbolos dependem uns dos outros—isto é, de quantos outros símbolos um dados símbolo precisa para derivar, ou em quantos símbolos esse símbolo intervém—é óbvio que a manutenção é mais simples quanto menor for a complexidade. Este factor não parece ter uma influência muito directa na compreensão e na derivação.
Neste caso da manutenção, nota-se que o uso de ex-BNF é preferível, enquanto que o esquema recursivo à direita ou à esquerda parece ser indiferente; a recursividade indirecta ou mista, tal como acima, também aqui é indesejável.

(CG2) Eficiência na Geração de Programas Como foi dito acima, uma dos importantes papéis da gramática é a sua capacidade de servir como instrumento para derivar programas, os processadores para a linguagem especificada pela mesma gramática. A maneira como a gramática é escrita influencia, quer a eficiência do processador que dela deriva, quer o próprio processo de geração, como se discute nos dois próximos parágrafos.

(CG2.1) Eficiência no Reconhecimento das frases da linguagem gerada é uma característica que se mede em termos de:

- tempo de *Parsing*
- tamanho/complexidade das Tabelas de Parsing, ou dos mecanismos de decisão.

O que importa considerar é o impacto que os vários elementos da GIC[4] têm sobre os factores indicados, pois é evidente que o tempo real de parsing e a complexidade das tabelas depende dos métodos e técnicas escolhidos (por quem cria o GC) para implementar o Reconhecedor[5] e da forma como o sistema os implementa.
Assim sendo e neste ponto específico, o que se pode dizer de objectivo e genérico é que um aumento no tamanho da gramática (no número de símbolos ou produções) implicará um aumento no tamanho das Tabelas

[4] Recordar o que foi dito na secção 3.1.
[5] Sobre este tema recorde-se o que foi dito na secção 3.2 e mais especificamente no apêndice C.

de Parsing, ou estruturas de decisão, e nas estruturas de dados complementares.
Em princípio esse aumento no tamanho das tabelas, se directamente implementadas em *arrays* ou em estruturas de controlo de selecção, não deve afectar o tempo de reconhecimento. Mas devido a possíveis optimizações no espaço requerido para armazenar as tabelas, vamos admitir que na prática o aumento do tamanho da gramática degrada ligeiramente o tempo de reconhecimento.
Curiosamente o comprimento dos lados direitos vai provocar um aumento na *stack de parsing* (e, portanto, requer mais memória na fase de reconhecimento), mas diminui as *operações de parsing* e consequentemente o tempo.
O tamanho dos identificadores, a notação usada, o esquema recursivo e o próprio recurso à modularidade ou a complexidade não afectam a eficiência do reconhecimento uma vez que são questões de escrita e clareza da gramática que são resolvidas pelo GC.

(CG2.2) Eficiência na Geração automática do processador é uma característica que se mede em termos de:

- tempo geração
- tamanho das estruturas de dados usadas internamente para armazenar e transformar a gramática.

Note-se que, novamente, nesta perspectiva o que importa considerar é o impacto que os vários elementos da gramática têm sobre os dois factores indicados, pois é evidente que o tempo real de geração e a complexidade das EDs depende dos métodos e técnicas escolhidos por quem cria cada GC e da forma como o sistema implementa esses algoritmos e estruturas de dados.
Assim sendo e neste ponto específico, o que se pode dizer de objectivo e genérico é que um aumento no tamanho da gramática (no número de símbolos ou de produções) implicará um aumento no tempo de geração e no espaço de armazenamento requerido durante esse processo. Neste caso, o tamanho dos identificadores dos símbolos também pode degradar tanto o tempo de análise da gramática quanto o tamanho das estruturas.

Exemplo 24 *: Para clarificar a afirmação e a título de exemplo, note-se que uma gramática com 10 produções requer maior tempo de análise que uma gramática com 5 produções e a* ED *para armazenar essas produções exige mais espaço em memória. De igual forma, uma gramática com 100 símbolos ocupa mais memória do que uma gramática com 50 símbolos; o tempo de análise será também mais lento embora o número de símbolos não tenha o mesmo impacto que o número de produções. Reconhecer identificadores curtos como os da variante* `G2` *no exemplo 22 é mais eficiente, a nível da análise léxica da gramática, do que reconhecer identificadores extensos como os usados na variante* `G1` *do mesmo exemplo.*

O comprimento dos lados direitos das produções não parece ter uma influência significativa, pois se, por um lado, requer mais memória para armazenar sequências mais longas, por outro lado, também significa que terá menos símbolos e produções e, portanto, os efeitos antagónicos compensam-se.
A complexidade, mais relacionada com o esforço de manutenção, e o esquema recursivo não afectam a eficiência nesta fase de geração. Mas em contrapartida o recurso à modularidade degrada o tempo de processamento da gramática pois mais módulos tem de ser abertos e consultados, mas não afecta o tamanho das estruturas. De igual forma a notação usada deveria ter uma influência semelhante no tempo de análise (com ex-BNF a requer maior esforço) e no tamanho das estruturas (não afectando); na prática, não é credível que assim seja pois cada GC estará optimizado para suportar o tipo de notação que aceita, não devendo este afectar a eficiência do reconhecimento.

Em suma, pode-se dizer que, quer o reconhecedor, quer o gerador, são afectados negativamente quando o tamanho da gramática aumenta. São considerações deste genéro que levam a que muitas vezes se trabalhe com duas gramáticas, uma (*maior*) para guiar a geração da linguagem e outra (*menor*) para servir de base à geração do processador[6].
Esta síntese corrobora a ideia de separar em duas partes a definição 23 de modo a considerar separadamente os conceitos de qualidade conforme a perspectiva gramatical em vista.

(CG - GIC) Síntese A tabela 4.1 resume precisamente tudo o que foi dito ao longo desta subsecção sobre a influência dos vários elementos de uma Gramática Independente de Contexto (GIC) sobre as características que afectam a qualidade de uma gramática à luz da definição 23.

Elems x Caracts	(CG1)Usabilidade			(CG2)Eficiência	
	Compreensão	Derivação	Manutenção	Reconh-L	Geraç-Rec
Ids Símbolos claros	+	+	+	*X*	+Te,Ta
Prods Unitárias	+	−	+	+Te,Ta	+Te,Ta
Comprimento RHS	−	+	−	*x*Te,Ta	*X*
Notação	+/−	−p, +ex	−p, +/−ex	*X*	*X*
Esquema Recursivo	+/−	+d, −e	+/−	−Te,Ta	*X*
Modularidade	−	−	+	*X*	+Te
Complexidade Sint	*X*	*X*	−	*X*	*X*

Tabela 4.1: Influência dos Elementos de uma GIC nas Características do Critério de Qualidade

(Legenda) A Tabela 4.1 foi preenchida de acordo com o seguinte critério:

\+ **(influência positiva)** – contribui para facilitar o factor em causa

− **(influência negativa)** – contribui para dificultar o factor em causa

+/− **(influência ambivalente)** – tanto pode contribuir para facilitar como para dificultar o factor em causa

X **(não tem influência)** – não interfere com o factor em causa

x **(influência mínima)** – contribui de forma pouco significativa para dificultar o factor em causa

Te – Tempo de processamento/geração

Ta – Tamanho das EDs internas de suporte ao processamento/geração

p – pure-BNF

ex – ex-BNF

d – recursividade à direita

e – recursividade à esquerda

[6]Note-se que neste caso a manutenção deverá ser feita sobre a primeira versão, para se manter mais simples.

4.1.2 Caracterização de GAs

Retoma-se a análise da usabilidade e eficiência de uma gramática, enquanto geradora de uma linguagem e enquanto geradora de um programa, mas agora tendo em atenção que o objecto em apreço é uma Gramática de Atributos, conforme definida em 3.1.
De acordo com a definição 13, uma gramática de atributos tem sempre subjacente uma gramática independente de contexto, por isso ao fazer-se o estudo que se segue mantém-se válida a caracterização da GIC feita atrás. Por outras palavras, dada uma Gramática de Atributos e pretendendo-se estudar a sua qualidade (em qualquer uma das duas vertentes referidas) deve-se caracterizar e avaliar primeiro a Gramática Independente de Contexto subjacente e depois completar com a caracterização e estudo da componente atributiva.

(CG1) Usabilidade da gramática é a clareza e facilidade com que o utilizador final lê a GA e a usa, enquanto instrumento para derivar frases da linguagem que essa gramática define, e o engenheiro gramatical a compreende e mantém.
Como foi dito atrás, esta característica geral de **usabilidade** desmonta-se em três componentes: compreensão, derivação e manutenção; essas vertentes serão agora analisadas.
A **compreensão** (CG1.1) de uma GA está fortemente relacionada com todos os elementos objectivos identificados na secção anterior a propósito da facilidade com que se lê/compreende uma GIC (os quais se continuam a revelar francamente relevantes) e ainda com os seguintes elementos específicos:

- a escolha dos identificadores para os atributos
- a complexidade dos atributos
- o número de atributos
- o número de regras de cálculo, de condições de contexto e de regras de tradução
- a notação empregue e a simplicidade com que as operações atributivas[7] são escritas

Tal como já foi discutido para os símbolos da gramática, também a escolha de identificadores esclarecedores para os atributos (identificadores que descrevam bem o conceito que cada atributo representa) é um dos factores que mais ajuda a compreender a gramática.
A complexidade do tipo dos atributos aumenta a dificuldade de entendimento. Atributos que armazenam valores estruturados, complexos, são mais dificeis de perceber do que atributos de tipos atómicos.
A notação, ou linguagem de especificação/programação, escolhida para escrever as operações atributivas (regras de cálculo, condições de contexto e regras de tradução) e o estilo usado para as escrever é outro factor determinante para influenciar a facilidade com que se compreende a GA. Nesse sentido são normalmente preferiveis linguagens declarativas que tenham capacidade de manusear tipos de dados complexos; tipicamente linguagens imperativas só manuseiam tipos de dados muito básicos e portanto é preciso um código muito mais extenso e mais rebuscado para descrever as mesmas operações. Também em princípio é preferivel o recurso a linguagens standard do que a linguagens específicas de cada GC, porque nesses casos exige-se do utilizador da GA a aprendizagem de uma nova linguagem.
O número de atributos e de operações atributivas em princípio ao crescer deveria provocar um aumento na dificuldade de compreensão; porém esta relação não é assim directa, pois depende da complexidade de ambos. Se o número de atributos crescer mas o seu tipo for mais simples, bem como as operações que sobre eles trabalham forem igualmente menos elaboradas, o resultado final é que é mais rápido e fácil perceber a GA.

[7] Designação genérica que engloba as regras de cálculo, as condições de contexto e as regras de tradução.

O tipo de recursividade, à direita ou à esquerda, não aparece alterar muito a compreensão, mas a recursividade indirecta ou mista continuam a ser indesejáveis.

Quanto à notação e apesar do ex-BNF ser mais conciso e simples para ler uma GIC, essa alternativa não é a mais conveniente e legível; o pure-BNF permite a escrita das regras de cálculo dos atributos associados aos símbolos das produções de forma mais sistemática, simples e clara.

O recurso à modularidade pode à primeira vista manter a GA mais compacta e arrumada e nesse sentido facilitar uma primeira leitura, mas para compreender a fundo acaba por dificultar a legibilidade.

A complexidade, em termos de dependência entre símbolos ou atributos, não influência significativamente a facilidade de compreensão.

No caso da **derivação** (CG1.2), a componente atributiva da GA pouco influencia o processo de construção de frases e portanto a implicação da gramática nesta tarefa é, essencialmente, a que já se descreveu a propósito da GIC—essa sim, determina a forma das frases válidas e guia o modo de as obter.

Concretamente, os atributos, regras de cálculo e regras de tradução essencialmente especificam o que o processador da linguagem gerada pela GA tem de fazer e pouca informação útil contêm para o vulgar programador que usa a gramática para escrever as frases que precisa.

Por seu turno, as condições contextuais já dão informação importante para o utilizador da linguagem, pois esclarecem o que é válido ou inválido em cada contexto de derivação específico. Assim e em termos de qualidade da GA enquanto instrumento para derivar frases, dir-se-á que a localização das condições de contexto na produção mais adequada e a forma clara e concisa como se encontram escritas interferem na facilidade com que o utilizador deriva frases semanticamente válidas; por isso, o número de condições aumenta a dificuldade do processo. No sentido explicado, é provável que o utilizador final precise de perceber o que são os atributos envolvidos nas restrições semânticas da linguagem (condições de contexto) e para isso poderá ter de compreender a informação que cada atributo transporta, bem como as respectivas regras de cálculo, mas essa questão da compreensão (em que o próprio tamanho da GA interfere) já foi discutida no item anterior.

Para facilitar a **manutenção** de uma gramática (CG1.3) intervêm todos os elementos já identificados para as GIC—identificadores esclarecedores, recurso a produções unitárias, lados direitos curtos, modularidade, complexidade sintáctica— e ainda os elementos referidos acima e que contribuem positivamente para a leitura e compreensão da GA. Aqui neste caso é bem evidente que o tamanho da GA dificulta a manutenção. Além desses, surge mais um factor importante:

- complexidade semântica

Entendendo agora por complexidade semântica de uma gramática a forma como os atributos dependem uns dos outros—isto é, de que outros atributos um dado atributo precisa para ser calculado, ou em quantas outras definições ou cálculos esse atributo intervém—é óbvio que a manutenção é mais simples quanto menor for a complexidade. Este factor não tem uma influência directa na compreensão nem na derivação.

Relativamente à manutenção, o uso de ex-BNF parece ser menos natural e portanto mais complexo, enquanto que o esquema recursivo à direita ou à esquerda parece ser indiferente; a recursividade indirecta ou mista, tal como nas situações anteriores, também aqui é inconveniente.

(CG2) Eficiência na Geração de Programas Se no caso da GIC é verdade que a maneira como a gramática é escrita influencia, quer a eficiência do reconhecedor que dela deriva, quer o próprio processo de geração, mais forte e notória ainda é essa influência no caso da GA, como se discute nos dois próximos parágrafos, pois agora a gramática é muito maior e mais complexa e dela já se deriva não apenas um reconhecedor mas também um verificador e um tradutor[8].

[8] Relembrar estes conceitos em 3.2.

(CG2.1) Eficiência no Processamento das frases da linguagem gerada é uma característica que se mede em termos de:

- tempo de análise e tradução
- tamanho/complexidade das estruturas que guiam a análise e a tradução.

Como se sabe, os algoritmos de análise léxica e sintáctica derivam apenas da GIC subjacente à GA. Sendo assim, a influência da gramática no tempo de análise sintáctica, e correspondente análise léxica, e no tamanho das estruturas de decisão associadas já foi discutida na subsecção anterior.
O que importa considerar agora é o impacto que os vários elementos da GA[9] têm sobre o tempo de verificação/tradução e o tamanho das estruturas de suporte. Recordando o que foi dito ao longo do capítulo 3, a verificação e tradução resolvem-se da mesma forma testando predicados, ou invocando procedimentos, sobre os valores dos atributos, à medida que se atravessa a árvore de sintaxe abstracta decorada (ASAD). Tal processo requer o prévio cálculo desses valores dos atributos, invocando funções que manuseiam outros atributos. Essas funções, predicados ou procedimentos—que genericamente se designarão por *operações atributivas*—são definidas por quem escreve a GA, isto é, não são geradas automaticamente a partir da gramática. Pode, então, afirmar-se que a eficiência dessas operações é dependente do engenheiro gramatical que cria a GA, sendo, portanto, influenciado pela linguagem de programação e pelos algoritmos e EDs que ele usa. Note-se, porém, que o tempo final/real de processamento e a complexidade efectiva das estruturas de dados depende dos métodos e técnicas escolhidos por quem cria o GC para desenvolver o Processador[10] e da forma concreta como o sistema os implementa[11].

Assim a influência mais directa que a GA pode ter nesse processo é através do número de operações a realizar. Afirma-se, então, que em termos de tamanho de uma GA, o número total de operações atributivas, embora não afecte o tamanho das estruturas internas, determina o esforço de verificação e tradução, aumentando o tempo quando esse número cresce. Note-se, porém, que estas duas grandezas não variam numa proporção directa, porque uma GA pode ter mais operações atributivas que outra GA, mas contudo essas operações podem ser de menor complexidade e portanto mais rapidamente realizadas.

O número de símbolos, de produções, ou mesmo o número de atributos, embora esteja de alguma forma relacionado, não determina directamente o número de operações de cálculo, verificação ou tradução e, portanto, não condiciona directamente o tempo de processamento. Mas condiciona indirectamente, porque uma maior gramática (com mais símbolos, produções ou atributos) vai dar origem a uma maior ASAD[12] (a estrutura interna que suporta a verificação/tradução) e, assim sendo, além de aumentar o consumo de memória, vai implicar um maior tempo de travessia.

Tal como visto para a GIC, o tamanho dos identificadores, a notação usada, o esquema recursivo e o próprio recurso à modularidade ou a complexidade sintáctica não afectam a eficiência do processamento[13].

(CG2.2) Eficiência na Geração automática do processador é uma característica que se mede em termos de:

- tempo geração
- tamanho das estruturas de dados usadas internamente para armazenar e transformar a gramática.

[9] Recordar o que foi dito na secção 3.1.
[10] Sobre este tema recorde-se o que foi dito na secção 3.2.
[11] Por isso mesmo, porque não depende da gramática, esse tempo total efectivo está fora do âmbito desta discussão.
[12] Maior em termos do número de nodos, se tem mais símbolos ou produções, ou maior em termos do tamanho requerido por cada nodo para armazenar todos os correspondentes atributos.
[13] São questões que ficam filtradas aquando da geração do processador.

Note-se que, novamente, neste ponto o que importa considerar é o impacto que os vários elementos da GA têm sobre os dois factores indicados, pois é evidente que o tempo real de geração e a complexidade das EDs depende dos métodos e técnicas escolhidos por quem cria cada GC e da forma como o sistema implementa esses algoritmos e estruturas de dados.

O que se pode dizer de objectivo e genérico neste caso é muito semelhante ao que se discutiu em relação à GIC e à geração de reconhecedores léxico-sintácticos: um aumento no tamanho da GA (no número de símbolos, número de produções, número de atributos, ou número de operações atributivas) implicará também um aumento no tempo de geração do verificador/tradutor e no espaço de armazenamento requerido durante esse processo (sobretudo para guardar os grafos de dependências entre os atributos).

Se é verdade que a complexidade, tal como definida atrás em termos sintácticos (nível de dependência entre os símbolos), continuará a não afectar o processo de geração, agora a complexidade semântica, definida em termos de dependências entre os atributos, pesa muito significativamente no processo de geração. Isto acontece porque a grande tarefa do gerador de processadores em relação à componente semântica da linguagem (verificação e tradução) é a determinação de uma ordem total para realização das operações atributivas durante as travessias a cada árvore de sintaxe abstracta decorada. Assim sendo, é de esperar que a eficiência do gerador decresça quando essa complexidade semântica aumenta.

Também neste caso, o tamanho dos identificadores dos símbolos e dos atributos pode degradar tanto o tempo de análise da gramática quanto o tamanho das estruturas.

De igual forma a influência das restantes características é semelhante ao que já se disse acima: o esquema recursivo não afecta a eficiência nesta fase de geração; a modularidade degrada o tempo de processamento da GA sem afectar o tamanho das estruturas; a notação usada, embora pudesse afectar (de novo com o ex-BNF a requer maior esforço de processamento) também não será muito relevante para o tempo de análise, visto que cada GC estará adaptado ao tipo de entrada que é suposto aceitar.

Em suma, pode-se dizer mais uma vez que, quer o reconhecedor, quer o gerador, são afectados negativamente quando o tamanho da gramática aumenta.

(CG - GA) Síntese A tabela 4.2 resume precisamente tudo o que foi dito ao longo desta subsecção sobre a influência dos vários elementos de uma Gramática de Atributos (GA) sobre as características que afectam a qualidade de uma gramática à luz da definição 23.

Elems x Caracts	(CG1)Usabilidade			(CG2)Eficiência	
	Compreensão	Derivação	Manutenção	Reconh-L	Geraç-Rec
Ids Atributos claros	+	X	+	X	+Te,Ta
Nuºatributos	$\surd$	X	−	xTe,Ta	+Te,Ta
NuºOpers Atribs	$\surd$	X	−	+Te	+Te,Ta
NuºSimbolos+Prods	$\surd$	−	−	xTe,Ta	+Te,Ta
Complexidade Atribs	−	x	−	xTe,Ta	X
Complexidade OpersAts	−	x	−	xTe,Ta	X
Colocação/clarez CCs	+	+	+	X	X
Notação	+p, −ex	x	+p, −ex	X	X
Esquema Recursivo	+/−	+/−	+/−	−Te,Ta	X
Modularidade	−	−	+	X	+Te
Complexidade Semant	X	X	−	X	+Te,Ta

Tabela 4.2: Influência dos Elementos de uma GA nas Características do Critério de Qualidade

(Legenda) A Tabela 4.2 foi preenchida de acordo com o seguinte critério:

$+$ **(influência positiva)** – contribui para facilitar o factor em causa

$-$ **(influência negativa)** – contribui para dificultar o factor em causa

$+/-$ **(influência ambivalente)** – tanto pode contribuir para facilitar como para dificultar o factor em causa

$\surd$ **(influência dependente)** – tem influência, mas o sinal depende de outros factores

X **(não tem influência)** – não interfere com o factor em causa

x **(influência mínima)** – contribui de forma pouco significativa para dificultar o factor em causa

Te – Tempo de processamento/geração

Ta – Tamanho das EDs internas de suporte ao processamento/geração

p – pure-BNF

ex – ex-BNF

d – recursividade à direita

e – recursividade à esquerda

4.2 Métricas para Gramáticas

Por se tratar de um objecto rigorosamente definido, é possível neste caso estabelecer um conjunto de métricas que vão permitir aferir quantitativamente o critério de qualidade das gramáticas. Nesta secção procura-se, após apresentar as métricas, relacioná-las com as características da gramática, identificadas na secção anterior, que determinam a sua qualidade, bem como relacionar com as características das linguagens, descritas no capítulo 2. Neste estudo, vai-se manter uma abordagem a dois níveis: primeiro introduzem-se as métricas para as GICs, e depois far-se-á uma apresentação semelhante para as GAs.

Apesar de haver muito trabalho, e até antigo, na área das métricas aplicadas a programas[14]—sendo de salientar o contributo de Fenton [Fen91, FN00], de Finkbine [Fin96], de Coleman [CALO] ou de Pan [PKEJW06], entre muitos outros preocupados em perceber a complexidade para avaliar o esforço de análise e de manutenção do software—muito poucos foram os autores que se debruçaram a adaptar este trabalho à análise e avaliação de gramáticas. Essencialmente surge o trabalho pioneiro de Power & Malloy [PM00, PM04], seguido por Alves & Visser [AV05, AV07, AV09] e agora, muito recentemente explorado por Cervelle e Matej [CCF$^+$09, CKM$^+$10].

4.2.1 Métricas para as GICs

Alguns dos autores acima referidos, nomeadamente Power & Malloy [PM00] e Alves & Visser [AV05], consideram dois Tipos de Métricas:

- **de Complexidade ou de Tamanho**
- **Estruturais**

Nesta lição e porque se discorda parcialmente, não se segue à risca essa classificação, mas adopta-se a nomenclatura apresentada de seguida, onde nomeadamente se designam as *estruturais* por *métricas de forma*. Também por decisão própria, não se usam as métricas de tamanho referidas por Power & Malloy e conhecidas por *McCabe Cyclomatic Complexity* [McC76, WM96] e *Halstead Effort*[15] [Cur81]; em sua substituição serão apresentadas outras que se julga darem a mesma informação final e serem mais simples (claras) de calcular. Inclui-se ainda um terceiro grupo, *métricas lexicográficas* ou *de feitio*, que se considera poderem conter mais informação relativa à qualidade da linguagem gerada.

Seja G uma Gramática Independente de Contexto *bem-formada* e GDS o respectivo Grafo de Dependência entre os Simbolos[16].

Definem-se a seguir (divididas em 3 grandes grupos) várias medidas para aferição da qualidade de G:

- Métricas de Tamanho:
  - (MT1) **tamanho da Gramática**, mede-se em termos de:

[14] Em inglês *Software Metrics*.

[15] Ver mais em `http://www.verifysoft.com/en\_halstead\_metrics.html` ou em `http://www.virtualmachinery.com/sidebar2.htm`

[16] Como G está bem-formada, mesmo que existam ciclos no Grafo de Dependência entre os símbolos $N \cup T$, é sempre possível escolher para cada símbolo um caminho que termina.

Parâmetro	Descrição
#T	número de símbolos terminais
#N	números de símbolos não-terminais
#P	número de produções
#PU	número de produções unitárias
#R	número de símbolos directa ou indirectamente recursivos
§RHS	número médio de símbolos nos lados direitos
§RHS-Max	número máximo de símbolos num lado direito, ou seja, $max(length(RHS))$
§Alt[1]	número médio de produções alternativas para os mesmos lados esquerdos
§Alt-Max	número máximo de produções alternativas para um mesmo lado esquerdo
#Mod	número de módulos gramaticais importados

– (MT2) **complexidade sintáctica da Gramática**, mede-se em termos de:

Parâmetro	Descrição
FanIn[2]	número médio de ramos de entrada nos nodos (não-terminais) do GDS.
FanOut[3]	número médio de ramos de saída dos nodos do GDS.

– (MT3) **tamanho do *Parser***, mede-se em termos de:

Parâmetro	Descrição
#RD	número de funções do *Parser* Recursivo-Descendente Puro ($\#(N \cup T)$)
§TabLL	dimensão da Tabela de Parsing LL(1) ($\#N \times (\#T + 1)$)
§$\mathcal{AD}$-LR	dimensão do Autómato Determinista LR(0) ($\#Q$)
§TabsLR	dimensão das Tabelas de Parsing LR(0) ($\#Q \times (\#T + 1) + \#Q \times \#N$)

Notas:

1. Para o cálculo de §Alt, considera-se que cada símbolo não-terminal tem sempre 1 alternativa; assim sendo, este valor é $\#P/\#N$.
2. **factor de derivação**, mede quantos símbolos derivam de um símbolo (ou quantos símbolos precisa um símbolo para sua definição).
3. **factor de dependência**, mede as vezes que um símbolo é usado na definição de outros símbolos.

- Métricas de Forma:

  – (MF1) **forma de Recursividade**, podendo tomar um dos seguintes valores:

Valor	Descrição
RecDirecta	todos os casos de recursividade seguem o padrão $X \rightarrow \cdots X \cdots$
RecIndirecta	todos os casos de recursividade seguem o padrão $X \rightarrow \cdots Y \cdots ; Y \rightarrow \cdots X \cdots$
RecFMista	ambas as formas de recursividade são usadas

  – (MF2) **tipo de Recursividade**, podendo tomar um dos seguintes valores:

Valor	Descrição
RecD	os casos de RecDirecta seguem o padrão recursivo à direita $X \rightarrow \varepsilon \mid e\ X$
RecD-LL	os casos de RecDirecta seguem o padrão recursivo LL(1) $X \rightarrow e\ C; C \rightarrow \varepsilon \mid e\ C$
RecE	os casos de RecDirecta seguem o padrão recursivo à esquerda $X \rightarrow \varepsilon \mid X\ e$
RecTMisto	vários padrões de recursividade directa são usados

– (MF3) **notação**, podendo tomar um dos seguintes valores:

Valor	Descrição
BNF	todas as produções estão escritas em Backus-Naur Form puro
ex-BNF	todas as produções estão escritas em Extended Backus-Naur Form
NMista	ambos as notações são usadas

- Métricas Lexicográficas:
  – (ML1) **identificadores esclarecedores** para os símbolos não-terminais e terminais-variáveis, calcula-se através da fórmula

$$\#IdCompl/(\#IdCompl + \#IdAbrev)$$

considerando a definição 24, o nome do conceito denotado por cada símbolo N ou TV da gramática e sendo:

Valor	Descrição
#IdCompl	num. símbs em que o identificador *deriva* do nome do conceito
#IdAbrev	num. símbs em que o identificador *não deriva* do nome do conceito

  – (ML2) **palavras-reservadas e sinais esclarecedores** da linguagem definida por G calcula-se através da fórmula

$$\#PRCompl/(\#PRCompl + \#PRAbrev)$$

considerando a definição 24, o nome de cada conceito descrito pela linguagem e sendo:

Valor	Descrição
#PRCompl	num. casos em que palavra-reservada *deriva* do nome do conceito
#PRAbrev	num. casos em que palavra-reservada *não deriva* do nome do conceito

  – (ML3) **flexibilidade dos terminais-variáveis**, calcula-se através da fórmula

$$\#TFlex/(\#TFlex + \#TRigid)$$

sendo:

Valor	Descrição
#TFlex	num. terms com flexibilidade na construção do valor ou identificador (T-variável)
#TRigid	num. terms sem flexibilidade na construção do valor ou identificador (T-variável)

  – (ML4) **tipo de comentários**, calcula-se através da fórmula

$$inline + bloco + metaI$$

sendo:

Valor	Descrição
inline	1 se aceita comentários dum ponto ao `eol`; senão 0
bloco	1 se aceita comentários formados por blocos de uma ou mais linhas; senão 0
metaI	1 se aceite meta-informação dentro dos blocos de comentários; senão 0

Para completar a formulação das métricas lexicográficas é necessário dizer com rigor o que se intende por uma palavra *derivar* de um termo; para isso introduz-se a definição seguinte.

Definição 24 (Derivação de um Identificador) *: Diz-se que um Identificador (uma palavra)* ***deriva*** *do Nome de um Conceito (um termo multi-palavra) se:*

1. *o identificador for igual ao nome (p.ex.,* `Lisp` *deriva de* Lisp*);*
2. *o identificador for um prefixo do nome, com 3 ou mais letras (p.ex.,* `num` *deriva de* numero*);*
3. *o identificador tiver um prefixo que é prefixo do nome e as restantes letras poderem ser obtidas do nome por remoção de algumas letras do nome (p.ex.,* `SExp` *deriva de* Symbolic Expression*).*

De forma análoga, diz-se que um identificador ***denota-bem*** *uma operação, se esse identificador for formado por 1 ou mais sinais que na vida quotidiana se usam para designar, ou representar, a operação em causa (p.ex., o identificador* `+` *denota-bem a* adição *e o identificador* `==` *denota-bem a* igualdade*).*

As métricas acima introduzidas vão ter um impacto diferente nas várias características, apresentados na secção anterior, para avaliação de gramáticas e, consequentemente, nas várias características para avaliar linguagens (cf. capítulo 2.3). A tabela 4.3 mostra precisamente as circunstâncias em que se manifesta essa influência.

Metrica x Caract	CG1.1	CG1.2	CG1.3	CG2.1	CG2.2	QL
MT1	√	√	√	√	√	√
MT2	√		√			
MT3				√	√	√
MF1	√	√	√			√
MF2	√	√		√		√
MF3	√	√	√			√
ML1	√	√	√		√	√
ML2				√		√
ML3				√		√
ML4				√		√

Tabela 4.3: Relação das Métricas com as Características para aferição duma GIC

Uma entrada na tabela preenchida com √ significa que a métrica da respectiva linha interfere, positiva ou negativamente, na característica da respectiva coluna.

A tabela 4.4 detalha a influência dos vários parâmetros associados às métricas de tamanho nas várias características usadas para estabelecer os critérios de qualidade de uma gramática G; regista-se também a interferência que tenham na qualidade da linguagem gerada por G.

MT x Caract	CG1.1	CG1.2	CG1.3	CG2.1	CG2.2	QL
#T	+	−	+	xTe,+Ta	+Te,+Ta	+L,−R
#N	+	−	+	xTe,+Ta	+Te,+Ta	+L,−R
#P	+	−	+	xTe,+Ta	+Te,+Ta	+L,−R
#PU	+	−	+	xTe,+Ta	+Te,+Ta	+L,−R
#R	−	X	−	X	X	−L,XR
§RHS	−	+	−	−Te,+Ta	X	−L,+R
§Alt	−	−	−	X	X	−L,XR
#Mod	−	−	+	X	+Te,XTa	−L,XR
FanIn/$\#(N \cup T)$	−	X	−	X	X	X
FanOut/$\#(N \cup T)$	X	X	−	X	X	X
#RD	X	X	X	+Te,XTa	+Te,+Ta	XL,−R
§TabLL	X	X	X	xTe,+Ta	+Te,+Ta	XL,−R
§TabsLR	X	X	X	xTe,+Ta	+Te,+Ta	XL,−R

Tabela 4.4: Relação das Métricas de Tamanho com as Características para aferição da GIC

(Legenda) A Tabela 4.4 foi preenchida de acordo com o seguinte critério:

\+ **(influência positiva)** – contribui para facilitar/aumentar o factor em causa

− (influência negativa) – contribui para dificultar/diminuir o factor em causa

X **(não tem influência)** – não interfere com o factor em causa

x **(influência mínima)** – contribui de forma pouco significativa para dificultar o factor em causa

Te – Tempo de processamento/geração

Ta – Tamanho das EDs internas de suporte ao processamento/geração

L – Legibilidade da linguagem

R – eficiência no Reconhecimento da linguagem

Todos estes parâmetros (excepto os do 2ºgrupo) afectam a qualidade da linguagem gerada por G.
Quanto ao 3º grupo, MT3, essa influência é óbvia: se os parâmetros influenciam o reconhecimento da frases (CG2.1), então vão afectar a eficiência no processamento da linguagem; quanto maior for o tempo de reconhecimento, menor será a eficiência.
Quanto ao 1ºgrupo, MT1, ai a relação é mais curiosa pois nenhum dos parâmetros considerados permite directamente tirar conclusões relativas às 6 características, CL1 a CL6, identificadas no capítulo 2; mas o que sucede na realidade é que, sempre que esses parâmetros influenciam a facilidade de leitura e compreensão da gramática (CG1.1), estão a influenciar, no mesmo sentido, a aprendizagem da linguagem e, consequentemente, a sua legibilidade.

A tabela 4.5 detalha a influência dos vários valores que caracterizam as métricas de forma nas várias características usadas para estabelecer os critérios de qualidade de uma gramática G; regista-se também a interferência que tenham na qualidade da linguagem gerada por G.

MF x Caract	CG1.1	CG1.2	CG1.3	CG2.1	CG2.2	QL
RecDirecta	+	+	+	X	X	+L,XR
RecIndirecta	−	−	−	X	X	−L,XR
RecFMista	−	−	−	X	X	−L,XR
RecD	+/−	+	X	+Te,+Ta	X	+L,XR
RecD-LL	+/−	+	X	+Te,+Ta	X	+L,XR
RecE	+/−	−	X	−Te,−Ta	X	−L,XR
RecTMisto	−	−	−	+/−Te,+/−Ta	X	−L,XR
BNF	+/−	−	−	X	X	+/−L,XR
ex-BNF	+/−	+	+	X	X	+/−L,XR
NMista	−	−	−	X	X	−L,XR

Tabela 4.5: Relação das Métricas de Forma com as Características para aferição da GIC

(Legenda) A Tabela 4.5 foi preenchida de acordo com o seguinte critério:

\+ **(influência positiva)** – contribui para facilitar/aumentar o factor em causa

− **(influência negativa)** – contribui para dificultar/diminuir o factor em causa

+/− **(influência ambivalente)** – tanto pode contribuir para facilitar como para dificultar o factor em causa

X **(não tem influência)** – não interfere com o factor em causa

Te – Tempo de processamento/geração

Ta – Tamanho das EDs internas de suporte ao processamento/geração

L – Legibilidade da linguagem

R – eficiência no Reconhecimento da linguagem

Relativamente à influência deste grupo de métricas de forma, MF1 a MF3, de G na qualidade da linguagem gerada por G, a discussão é análoga à que foi tida para o grupo anterior.
Neste caso nenhum desses parâmetros afecta o tempo de reconhecimento e por isso sua influência na eficiência do processamento da linguagem é nula; todavia todos afectam a leitura/compreensão da gramática e, consequentemente, todos interferem no mesmo sentido na legibilidade da linguagem, visto facilitarem/dificultarem a sua aprendizagem.

A tabela 4.6 detalha a influência dos vários parâmetros que caracterizam as métricas lexicográficas nas várias características usadas para estabelecer os critérios de qualidade de uma gramática G; regista-se também a interferência que tenham na qualidade da linguagem gerada por G.

ML x Caract	CG1.1	CG1.2	CG1.3	CG2.1	CG2.2	QL
IdSimbolos	+	+	+	X	+Te,+Ta	+L,XR
PalReservadas	X	X	X	+Te,+Ta	X	+L,xR
SVariaveis	X	X	X	+Te,+Ta	X	+L,xR
Comentarios	X	X	X	+Te,XTa	X	+L,xR

Tabela 4.6: Relação das Métricas Lexicográficas com as Características para aferição da GIC

(Legenda) A Tabela 4.6 foi preenchida de acordo com o seguinte critério:

\+ **(influência positiva)** – contribui para facilitar/aumentar o factor em causa

X **(não tem influência)** – não interfere com o factor em causa

x **(influência mínima)** – contribui de forma pouco significativa para dificultar o factor em causa

Te – Tempo de processamento/geração

Ta – Tamanho das EDs internas de suporte ao processamento/geração

L – Legibilidade da linguagem

R – eficiência no Reconhecimento da linguagem

Este grupo de métricas lexicográficas, ML1 a ML4, tem agora uma influência mais forte e objectiva na qualidade da linguagem gerada por G.
Em termos de eficiência no processamento da linguagem, a influência é pequena fazendo-se sentir, no sentido inverso, apenas nos parâmetros que de alguma forma aumentam (ligeiramente) o tempo de reconhecimento das frases. A interferência de ML1 na legibilidade da linguagem continua a ser explicada por contribuir para uma maior facilidade de leitura/compreensão de G; porém a influência de ML2, ML3 e ML4 é devida à implicação que estes parâmetros têm em algumas das características de aferição de linguagem, conforme se detalha a seguir na tabela 4.7 (ver legenda anterior).

ML x Caract	CL1	CL2	CL3	CL4	CL5	CL6	QL
IdSimbolos							+L,XR
PalReservadas	+					x	+L,xR
SVariaveis	+					x	+L,xR
Comentarios		+					+L,xR

Tabela 4.7: Relação da Métricas Lexicográficas com as Características de aferição de L

4.2.2 Métricas para as GAs

Seja GA uma Gramática de Atributos *bem-formada*. Sobre GA definem-se os seguintes parâmetros atributivos para medida, ou aferição, da qualidade (mantendo válidas todas as métricas introduzidas na subsecção anterior para avaliação da Gramática Independente de Contexto subjacente):

- Métricas de Tamanho:
  - (MTA1) tamanho de GA:

Parâmetro	Descrição
#A	número de atributos
#AI	número de atributos herdados
#AS	número de atributos sintetizados
#RC	número de regras de cálculo
#CC	número de condições de contexto
#RT	número de regras de tradução

  - (MTA2) complexidade semântica da gramática:

Parâmetro	Descrição
$\mathsf{FanIn}^{(1,2)}$	número médio de ramos de entrada nos atributos dos GDLs das produções.
$\mathsf{FanOut}^{(1,3)}$	número médio de ramos de saída dos atributos dos GDLs das produções.

    Notas:

    1. calcula-se analisando o Grafo de Dependências Local, GDL, associado a cada produção e tomando para cada atributo o máximo.
    2. mede quantos atributos precisa um atributo para sua definição.
    3. mede as vezes que um atributo é usado na definição de outros atributos.

- Métricas de Forma:
  - (MFA1) **complexidade dos atributos**, calcula-se através da fórmula

$$\#AComplex/(\#AComplex + \#AAtom)$$

    sendo:

Parâmetro	Descrição
#AAtom	número de atributos de tipo atómico
#AComplex	número de atributos de tipo estruturado (com/sem apontadores, ou hashing)

  - (MFA2) **complexidade das operações atributivas**, calcula-se através da fórmula

$$\#OComplex/(\#OComplex + \#OSimples)$$

    sendo:

Parâmetro	Descrição
#OSimples	num. de RC, CC, ou RT que se restringem apenas a 1 instrução
#OComplex	num. de RC, CC, ou RT que são 1 bloco de instruções

– (MFA3) **esquema de cálculo** para escrita das RCs, sendo medido pela adição de dois valores, um que avalia a forma de agregação de acordo com a tabela seguinte:

Valor	Descrição
RCAgreg	o cálculo dos atributos segue o padrão de agregação de valores
RCNAgreg	o cálculo dos atributos segue o padrão de não-agregação de valores

e outro que classifica a forma de acumulação de valores recursivos de acordo com a tabela abaixo (sugere-se a consulta ao apêndice D onde se padronizam os esquemas das operações atributivas):

Valor	Descrição
RCpureS	o cálculo dos atributos de acumulação segue o padrão puramente sintetizado
RCpureI	o cálculo dos atributos de acumulação segue o padrão puramente herdado
RCmixIS	o cálculo dos atributos de acumulação segue o padrão misto h/s
RCvar	o cálculo dos atributos não segue um padrão típico

– (MFA4) **esquema de restrição semântica** para escrita das CCs, podendo tomar um dos seguintes valores (sugere-se a consulta ao apêndice D onde se padronizam os esquemas das operações atributivas):

Valor	Descrição
CCTop	a colocação das CCs segue essencialmente o padrão sintetizado
CCCentrad	a colocação das CCs segue essencialmente o padrão do ponto certo
CCBottom	a colocação das CCs segue essencialmente o padrão herdado
CCvar	a colocação das CCs não segue um padrão típico

– (MFA5) **esquema de tradução** para escrita das RTs, podendo tomar um dos seguintes valores (sugere-se a consulta ao apêndice D onde se padronizam os esquemas das operações atributivas):

Valor	Descrição
RTTop	a colocação das RTs segue essencialmente o padrão sintetizado
RTInterm	a colocação das RTs segue essencialmente o padrão do ponto certo
RTBottom	a colocação das RTs segue essencialmente o padrão herdado
RTvar	a colocação das RTs não segue um padrão típico

– (MFA6) **estilo da linguagem** para escrita das operações atributivas, sendo neste caso relevante registar se é uma *linguagem declarativa* (funcional ou lógica), ou não (se for imperativa).

– (MFA7) **especificidade da linguagem** para escrita das operações atributivas, sendo neste caso relevante registar se é uma linguagem de programação standard, comum, ou não (se for uma linguagem talhada especificamente para o efeito pelo GC em causa).

• Métricas Lexicográficas:

– (MLA1) **identificadores esclarecedores** para os atributos, calcula-se através da fórmula

$$\#IdACompl/(\#IdACompl + \#IdAAbrev)$$

considerando a definição 24, o nome do conceito denotado por cada atributo da gramática e sendo:

Valor	Descrição
#IdACompl	num atribs cujo identificador *deriva* do nome do conceito
#IdAAbrev	num atribs cujo identificador *não deriva* do nome do conceito

– (MLA2) **identificadores esclarecedores** para operadores atributivos (nomes das funções, predicados e procedimentos invocados nas regras de cálculo dos atributos, nas condições de contexto e nas regras de tradução), calcula-se através da fórmula

$$\#IdOCompl/(\#IdOCompl + \#IdOAbrev)$$

considerando a definição 24, o nome de cada operação que efectivamente se pretende realizar em cada regra e sendo:

Valor	Descrição
#IdOCompl	num operações cujo identificador *denota-bem* ou *deriva* do nome da operação
#IdOAbrev	num operações cujo identificador *não denota* nem *deriva* do nome da operação

Relativamente às métricas de forma, é importante notar que não se voltaram a incluir nesta parte atributiva os parâmetros relacionados como o *esquema recursivo (tipo e forma)* nem com o *recurso à modularidade*, pois a sua influência nas característica da gramática é precisamente a mesma que já foi analisada, na subsecção anterior, para a parte da Gramática Independente de Contexto, como aliás já tinha sido discutido ao fazer a caracterização da Gramática de Atributos na subsecção 4.1.2. Apenas a notação (pure-BNF ou ex-BNF) tem agora uma influencia em sentido oposto ao que foi visto no caso da GIC, uma vez que a escrita das operações atributivas em ex-BNF é mais complexa, menos natural, do que em pure-BNF apesar da primeira forma ser mais sucinta e clara em termos sintácticos; porém como só afecta a compreensão e a manutenção, decidiu-se não incluir de novo nesta parte.

A tabela 4.8 mostra a influência das várias métricas envolvendo atributos sobre as características atrás definidas para avaliar a qualidade de uma gramática, indicando-se também a sua relação com a qualidade da linguagem gerada pela *GA* (cf. capítulo 2.3).

Métrica x Caract	CG1.1	CG1.2	CG1.3	CG2.1	CG2.2	QL
MTA1	√	√	√	√	√	√
MTA2	√		√	√	√	√
MFA1	√	√	√	√		√
MFA2	√	√	√			√
MFA3	√	√	√	√	√	√
MFA4	√	√	√	√	√	√
MFA5	√		√	√	√	√
MFA6	√	√	√	√		√
MFA7	√	√	√	√	√	√
MLA1	√	√	√		√	√
MLA2	√	√	√		√	√

Tabela 4.8: Relação da Métricas Atributivas com as Características para aferição duma GA

Tal como na subsecção anterior, uma entrada na tabela preenchida com √ significa que a métrica da respectiva linha interfere, positiva ou negativamente, na característica da respectiva coluna.

Para se fazer um estudo mais detalhado da influência de cada métrica, definida para a parte atributiva da gramática, nas suas várias características de classificação e avaliação, expandem-se na tabela 4.9 as métricas nos respectivos parâmetros.

Note-se que nesta tabela se consideram os parâmetros estabelecidos acima, não em valores absolutos, mas relativizados ao tamanho da gramática, por se julgar que esse ratio dá uma ideia mais precisa/rigorosa da

influência no contexto de cada GA concreta.

Métrica x Caract	CG1.1	CG1.2	CG1.3	CG2.1	CG2.2	QL
#A/#($N \cup TV$)	+/−	x	+/−	+Te,+Ta	+Te,+Ta	+/−L,−P
#RC/#P	+/−	x	+/−	+/−Te,XTa	+Te,+Ta	+/−L,+/−P
#CC/#P	+/−	−	+/−	+/−Te,XTa	+Te,+Ta	+/−L,+/−P
#RT/#P	+/−	X	+/−	+/−Te,XTa	+Te,+Ta	+/−L,+/−P
FanIn/#A	−	X	−	+/−Te,XTa	+Te,+Ta	−L,+/−P
FanOut/#A	X	X	−	−Te,−Ta	+Te,+Ta	XL,+P
#AComplex/#A	−	−	−	+Te,+Ta	X	−L,−P
#OComplex/#O	−	−	−	X	X	−L,XP
RCAgreg	−	x	−	+Te,XTa	−Te,−Ta	−L,−P
RCNAgreg	+	x	+	−Te,XTa	+Te,+Ta	+L,+P
RCpureS	−	x	−	−Te,−Ta	−Te,−Ta	−L,+P
RCpureI	+	x	+	+Te,+Ta	+Te,+Ta	+L,−P
RCmixIS	+	x	+	+Te,+Ta	+Te,+Ta	+L,−P
CCTop	−	−	−	+Te,+Ta	x	−L,−P
CCCentrad	+	+	+	−Te,−Ta	x	+L,+P
CCBottom	−	−	−	+Te,+Ta	x	−L,−P
RTTop	−	X	−	+Te,+Ta	x	−L,−P
RTInterm	+	X	+	−Te,−Ta	x	+L,+P
RTBottom	−	X	−	+Te,+Ta	x	−L,−P
LDeclarativa	+	x	+	+/−Te,+/−Ta	X	+L,+/−P
LStandard	+	x	+	+/−Te,+/−Ta	−Te,−Ta	+L,+/−P
#IdACompl/#A	+	x	+	X	+Te,+Ta	+L,XP
#IdOCompl/#O	+	x	+	X	xTe,XTa	xL,XP

Tabela 4.9: Relação detalhada das Métricas Atributivas com as Características para aferição da GA

(Legenda) A Tabela 4.9 foi preenchida de acordo com o seguinte critério:

\+ **(influência positiva)** – contribui para facilitar/aumentar o factor em causa

− **(influência negativa)** – contribui para dificultar/diminuir o factor em causa

+/− **(influência ambivalente)** – tanto pode contribuir para facilitar como para dificultar o factor em causa

X **(não tem influência)** – não interfere com o factor em causa

x **(influência mínima)** – contribui de forma pouco significativa para dificultar o factor em causa

Te – Tempo de processamento/geração

Ta – Tamanho das EDs internas de suporte ao processamento/geração

L – Legibilidade da linguagem

P – eficiência no Processamento da linguagem

Curiosamente todas as métricas atributivas—de tamanho, forma ou lexicográficas—consideradas (de MTA1 a MLA1) afectam a qualidade da linguagem gerada pela gramática, embora nenhum dos parâmetros considerados permita directamente tirar conclusões relativas às 6 características, CL1 a CL6, identificadas no capítulo 2.
O que sucede na realidade é que todos esses parâmetros influenciam a facilidade de leitura e compreensão da gramática (CG1.1), estando assim a influenciar, no mesmo sentido, a aprendizagem da linguagem e, consequentemente, a sua legibilidade. Além disso todos eles interferem também no tempo de execução e na memória consumida pelo processador gerado automaticamente a partir da gramática (CG2.1) e, assim, vão afectar a eficiência no processamento da linguagem; quanto maior for o tempo de processamento, menor será a eficiência.

4.3 Casos de estudo

Nesta secção trabalham-se dois casos, um que envolve uma linguagem de programação (Lisp), e outro que envolve uma linguagem de domínio específico.

4.3.1 Caso 1: a linguagem Lisp

Considere-se como primeiro caso de estudo a gramática da linguagem Lisp introduzida na secção 3.1 (mais precisamente no exemplo 18). Na tabela 4.10 mostram-se as medidas que vão permitir avaliar essa gramática.

Metrica	Medida	Obs.
MT1		
#T	4	=0(PR)+2(Sin)+2(TV)
#N	3	
#P	6	
#PU	0	
#R	2	
§RHS	1,3	=(1+1+1+3+2+0)/6
§RHS-Max	3	
§Alt	2	=(1+3+2)/3
§Alt-Max	3	
#Mod	0	
MT2		
FanIn/$\#(N \cup T)$	2,6/7	=(1+5+2)/3
FanOut/$\#(N \cup T)$	1,2/7	=(0+2+1+1+1+2+1)/7
MT3		
§RD	7	=(4+3)
§TabLL	15	=(3*(4+1))
§$\mathcal{AD}$-LR	10	
§TabsLR	50;30	=(10*(4+1);10*3)
MF1	RecFMista	Nota 1
MF2	RecD	Nota 2
MF3	BNF	Nota 3
ML1	5/5	Nota 4
ML2	2/2	Nota 5
ML3	?	Nota 6
ML4	?	Nota 6

Tabela 4.10: Medidas da Gramática Independente de Contexto da linguagem Lisp

Notas relativas à tabela 4.10):

1. há recursividade directa na produção $p4$ e indirecta na $p3$.
2. a recursividade directa é à direita.
3. todas as produções estão escritas em BNF-puro.

4. os identificadores dos símbolos não-terminais, `Lisp, SExp, SExpList` são considerados completos pois todos *derivam* (à luz da definição 24) dos nomes dos conceitos em causa Lisp, Symbolic Expression, Symbolic Expression List; de igual forma os 2 terminais-variáveis, `num, pal`, também se consideram completos por *derivarem* dos conceitos que representam, número e palavra.

5. o alfabeto não tem palavras-reservadas e os 2 sinais são uma possível denotação para o conceito de `Lista` em causa.

6. não se pode inferir da gramática dada (as ERs que descrevem os terminais/comentários não são apresentadas).

Olhando para os valores que se registaram na tabela 4.10 e atendendo ao que se disse na subsecção 4.2.1 e sobretudo à discussão ao longo da subsecção 4.1.1 (sintetizada na tabela 4.1), é possível proceder-se à análise da Gramática Independente de Contexto da linguagem Lisp:

- Relativamente à informação que se retira das métricas de tamanho (e tendo em conta a tabela 4.4), pode dizer-se que é uma gramática difícil de compreender e manter, mas uma vez compreendida não é difícil de usar para derivar as frases; em contrapartida os reconhecedores que se podem gerar da gramática são simples e eficientes. Estas afirmações justificam-se com base nas seguintes evidências:
  - a GIC é muito pequena, em termos de símbolos e produções, o que é agravado pela inexistência de produções unitárias e pelo facto de 2 dos 3 símbolos não-terminais serem recursivos—tudo isto dificulta a compreensão;
  - apesar dos lados direitos não serem muito longos, em média cada símbolo tem 2 alternativas, o que é bastante para uma gramática pequena;
  - embora pelo valor baixo do FanOut não se preveja uma extrema complexidade na manutenção (cada símbolo só é usado na definição de 1 outro), o valor do FanIn (quase 3 símbolos derivam de cada não-terminal, em média) confirma a complexidade intrínseca avaliada nos 2 items anteriores;
  - as dimensões do *Parser*-RD ou das tabelas de parsing são pequenas (sobretudo as do *Parser*-LL(1)) o que torna o reconhecimento pouco consumidor de memória e rápido.

- A dificuldade em ler/perceber esta gramática é ainda aumentada pelo uso de recursividade directa e indirecta (forma mista) e notação pure-BNF, ambas consideradas—de acordo com a tabela 4.5—pouco favoráveis, sobretudo para os peritos em engenharia gramatical;

- ainda em termos de métricas de forma, o uso de recursividade à direita não é dos piores tipos, sobretudo para os leigos na área;

- para contrabalançar a dificuldade em compreender esta gramática inerente à sua pequenez e recursividade, verifica-se, olhando para as métricas lexicográficas e para a tabela 4.6, que os identificadores dos símbolos não-terminais e terminais-variáveis são elucidativos do conceito que denotam, o que facilita a leitura da gramática.

Relativamente à linguagem gerada pela GIC acabada de analisar, pode-se apenas dizer que:

- a sua aprendizagem não será muito fácil, pelo menos pelo lado da compreensão da gramática geradora;

- o seu reconhecimento pode ser eficiente, se o respectivo *Parser* for bem escolhido e implementado;

- a sua expressividade, no que se pode deduzir das métricas lexicográficas, não será muito facilitada pela ausência de palavras-reservadas (ver tabela 4.7); mas, ao contrário, a escalabilidade será assegurada, pelo menos em termos de escrita e de processamento de longos programas[17].

De modo semelhante a tabela 4.11 mostra as medidas da Gramática de Atributos relativa à semântica definida para a linguagem Lisp no exemplo 21.

Metrica	Medida	Obs.
MTA1		
#A/#($N \cup TV$)	28/5	
#AI	12	=(0+6+6+0+0)
#AS	16	=(4+5+5+1+1)
#RC/#P	53/6	=(10+5+5+11+17+5)
#CC/#P	2/6	=(0+1+1+0+0+0)
#RT/#P	1/6	=(1+0+0+0+0+0)
MTA2		
FanIn/#A	0,8/28	Ver Apêndice F
FanOut/#A	1,2/28	Ver Apêndice F
MFA1		
#AComplex/#A	11/28	=(11/(11+17))
MFA2		
#OComplex/#O	0/56	=(0/(0+56))
MFA3	RCNAgreg+RCpureI	Nota 1
MFA4	CCCentrad	Nota 2
MFA5	RTTop	Nota 3
MFA6	LImperativa	Ling. à la Pascal
MFA7	LStandard	Ling. Algorítmica genérica (Nota 4)
MLA1		
#IdACompl/#A	28/28	=(28/(28+0)) (Nota 5)
MLA2		
#IdOCompl/#O	7/10	=(7/(7+3)) (Nota 6)

Tabela 4.11: Medidas da Gramática de Atributos da linguagem Lisp

Notas relativas à tabela 4.11:

1. as regras de cálculo, por um lado seguem o padrão da não agregação de valores simples em atributos compostos (os valores parciais são mantidos em atributos simples distintos) e por outro lado seguem o padrão puramente herdado, em que a informação de contexto desce na árvore até às produções mais elementares (com folhas como descendentes), através de atributos herdados, para depois acumular, em atributos de saída dessas produções, a nova informação que será sintetizada pela árvore acima; note-se, para corroborar esta classificação, que o número de atributos herdados é quase igual ao de número de sintetizados—tirando os 4 do axioma e os 2 intrínsecos aos terminais-variáveis, os outros atributos praticamente surgem aos pares, como é típico do esquema `RCpureI`.

2. as duas condições de contexto estão associadas às produções onde efectivamente se pretende impor a restrição.

[17]Seguramente, a compreensão de um programa muito longo sem palavras-reservadas vai tornar-se difícil.

3. a única regra de tradução está associada à produção $p0$ o que obrigada sintetizar todo o código num atributo associado ao axioma da GA.

4. todas as operações atributivas estão especificadas numa linguagem algorítmica típica usada com propósito genérico; não se trata de uma linguagem específica de determinado GC.

5. os identificadores de todos os atributos *derivam* do nome do conceito que representam (segundo a definição 24); por exemplo os atributos que guardam o Contador de Números, o Contador de Palavras, a Lista, o Código-Maquina, o Tipo de Elemento, ou o Valor do símbolo são identificados por `contaN`, `contaP`, `lista`, `codigo` (ou `cod`), `tipo` e `val`.

6. a maioria (7) dos identificadores das operações atributivas—`=, +, ==, append, tail, head, grava`—podem considerar-se como *denotando-bem*, ou *derivando*, dos nomes das operações que implementam—atribuição (cópia de valores), adição, igualdade, junção de listas, cauda da lista, cabeça da lista, gravação do código—enquanto que os outros 3 operadores—`insere1, insere2, conv_oper`—não são identificadores esclarecedores.

Considerando os valores registados na tabela 4.11 e atendendo ao que se disse na subsecção 4.2.2, em particular às relações exaradas na tabela 4.9, e à discussão ao longo da subsecção 4.1.2 (sintetizada na tabela 4.2), é possível proceder-se à análise da Gramática de Atributos da linguagem Lisp. Mantendo a avaliação já feita acima à componente estrutural, ou sintáctica, pode agora acrescentar-se que:

- se trata de uma GA 'pesada' na medida em que a componente atributiva é grande: cerca de 6 atributos por símbolo, e cerca de 10 regras de cálculo por produção; *pesada* não quer dizer *complexa*, como sucede neste caso, apenas significa que tem muitos atributos e operações atributivas para ler e compreender.

- apesar de 'pesada' a GA em causa não é 'complexa', pois:

  - os atributos são na maioria de tipo atómico (só 11 em 28 são de tipos estruturados[18]), e as operações atributivas são todas simples (descrevem-se em 1 instrução);
  - os identificadores de ambos, atributos e operações atributivas, são na sua quase totalidade esclarecedores e, portanto, facilitadores de uma rápida compreensão;
  - o grau de dependência entre atributos é consideravelmente baixo, como o comprovam o FanIn e FanOut que rondam o valor 1 (um atributo depende de 1 outro para sua definição e participa na definição de 1 outro);
  - os padrões, ou esquemas usados para a escrita das regras de cálculo e colocação das condições de contexto (`RCNAgreg`, `RCpureI` e `CCCentrada`, respectivamente), são os que mais facilitam a compreensão;
  - o esquema de escrita das regras de tradução, `RTTop`, embora não seja o mais fácil de analisar, é um dos mais usuais e não pesa muito visto só existir 1 regra;
  - a linguagem usada na escrita das operações atributivas, apesar de não ser declarativa, é perfeitamente adequada ao tipo simples de operações envolvidas (essencialmente atribuições, que correspondem a regras de cópia) e apresenta a vantagem de não ser uma linguagem específica para manuseamento de GAs (é uma linguagem algorítmica usual).

- a gramática, uma vez bem percebida, não levanta especiais dificuldades nem a quem tem de a usar para escrever frases da linguagem (além da simplicidade sintáctica, só há 2 restrições de tipo a preservar), nem a quem tem de a manter (pelo menos a complexidade semântica é baixa).

[18] Que neste caso não vão além de sequências de pares ou de strings, não se encontrando outros tipos mais pesados, como tabelas de hashing ou grafos.

- o processador que dela se gera, ou deriva, apesar da ineficiência inerente à sua dimensão (que requer mais memória para armazenar todos os atributos e, consequentemente, um tempo de cálculo grande), é beneficiado pela simplicidade das operações (facilmente implementáveis com eficiência numa linguagem procedimental clássica).

- o gerador do processador, embora também seja um pouco afectado pela dimensão da GA, não enfrenta dificuldades especiais na análise de dependências e de determinação da ordem total de cálculo, sobretudo devido a um baixo grau de complexidade (FanIn+ FanOut) e à escolha do esquema puramente herdado juntamente com operações simples.

Relativamente à linguagem gerada pela GA acabada de analisar, pouco há a acrescentar ao que já foi dito acima, aquando da avaliação da GIC subjacente. Em termos de balanço final, pode-se então e apenas dizer que:

- a sua aprendizagem não será muito fácil, pelo menos pelo lado da compreensão da gramática geradora, quer devido à complexidade sintáctica, quer ao peso da parte atributiva;

- o seu processamento pode ser satisfatório, se o respectivo compilador/interpretador for bem escolhido e implementado, graças à simplicidade do reconhecedor sintáctico que se pode gerar e ao calculador de atributos e tradutor razoavelmente eficientes que também se podem produzir a partir da GA;

- a sua expressividade, no que se pode deduzir das métricas lexicográficas, não será muito facilitada pela ausência de palavras-reservadas (ver tabela 4.7); mas, ao contrário, a escalabilidade será assegurada, pelo menos em termos de escrita e de processamento de longos programas[19].

[19] Seguramente, a compreensão de um programa muito longo sem palavras-reservadas vai tornar-se difícil.

4.3.2 Caso 2: a linguagem GCI

Considere-se, como segundo caso de estudo, uma DSL para *gestão de um centro de investigação* de uma universidade (GCI), que é introduzida (contextualizada e descrita) a seguir através da respectiva Gramática Independente de Contexto.
Tal como no caso anterior, estudar-se-á primeiro a linguagem do ponto de vista sintáctico, medindo e avaliando a sua GIC, e depois procede-se de forma análoga para a componente semântica, medindo e aferindo a respectiva GA. Porém neste caso, a Gramática Independente de Contexto original sofrerá diferentes metamorfoses (será alvo de três transformações) para se fazer um estudo comparativo de cada versão.

Enunciado do problema – fase 1 Nos tempos que correm, até a produção científica de uma instituição académica tem de ser contabilizada. Para isso é preciso colectar determinados resultados obtidos pelos membros do centro de investigação em causa e fazer contagens por anos em avaliação, ou outros parâmetros. Para organizar toda essa informação, descrevendo resultados e associando-os a quem os produziu, pretende-se criar uma linguagem específica[20] (que se denominará GCI – *Gestão de Centros de Investigação*) que facilite a recolha e permita automatizar os cálculos, além de servir para consolidar certos dados.
A gramática independente de contexto G, abaixo apresentada, define uma versão reduzida dessa linguagem GCI para contabilização de todas as orientações de pós-graduação[21] (PG) de um dado grupo de investigação, permitindo descrever cada projecto concluído ou em andamento dentro do grupo.

```
T = { num, id, str, PHD, MSC, CO-ORIENT, INI, FIM, ".", ";", "(", ")" }
N = { Gci, PGs, Pg, IdOrient, Tipo, CoOrient, Aluno, Titulo, Nome, Inic, Fim, Ano }
S = Gci
P = {
     p0:  Gci      -->  PGs '.'
     p1:  PGs      -->  Pg
     p2:             |   PGs ';' Pg
     p3:  Pg       -->  IdOrient Tipo CoOrient Aluno '(' Titulo ')' Inic Fim
     p4:  IdOrient -->  id
     p5:  Tipo     -->  PHD
     p6:             |   MSC
     p7:  CoOrient -->  &
     p8:             |   CO-ORIENT IdOrient
     p9:  Aluno    -->  Nome
     p10: Titulo   -->  str
     p11: Nome     -->  str
     p12: Inic     -->  INI Ano
     p13: Fim      -->  &
     p14:            |   FIM Ano
     p15: Ano      -->  num
    }
```

sabendo-se ainda que os símbolos terminais-variáveis e os comentários válidos são definidos pelas seguintes expressões regulares (escritas segundo a notação convencional em Lex):

```
num : [0-9]+
id  : [a-zA-Z][a-zA-Z0-9/_]*
```

[20] Uma DSL.
[21] Isto é, as supervisões de Doutoramentos (phd) e Mestrados (msc).

```
str : \"[^"]*\"
comentario1 : "//".*
comentario2 : \{[^}]*\}
```

Para estudar a qualidade da GIC G dada, pretende-se fazer a sua avaliação de acordo com as métricas para Gramáticas Independentes de Contexto introduzidas na secção 4.2.1, seguindo a análise feita para a gramática Lisp na subsecção anterior. Depois pretende-se efectuar sobre G cada uma das 3 transformações abaixo descritas[22] e fazer a medição de cada versão, segundo os mesmos parâmetros, de modo a ser possível compará-las.
As transformações pedidas são:

T1) Reduza a gramática G eliminando as produções unitárias.

T2) Altere a gramática G de modo a simplificar o lado direito da produção `p3`, permitindo escrevê-lo na forma:

```
p3:  Pg       -->  IdOrient DescOrientac Periodo
```

T3) Altere a gramática G para agrupar numa única descrição todas as orientações do mesmo Orientador.

Note-se que as duas primeiras transformações afectam a cosmética da gramática sem afectar a linguagem por ela gerada, enquanto que a terceira altera também a linguagem final devendo torná-la mais fácil de usar.

Resolução – fase 1 Por uma questão de sistematização e facilidade de apresentação começam-se por fazer as três transformações pedidas e depois mostram-se, na mesma tabela, as medidas das quatro versões (a original e as 3 transformadas) a comparar.

Transformação (T1) Eliminando as produções unitárias de G, obtém-se o seguinte novo conjunto de produções:

```
P' = {
     p0:  Gci      -->  PGs '.'
     p1:  PGs      -->  Pg
     p2:             |   PGs ';' Pg
     p3:  Pg       -->  id Tipo CoOrient str '(' str ')' Inic Fim
     p5:  Tipo     -->  PHD
     p6:             |   MSC
     p7:  CoOrient -->  &
     p8:             |   CO-ORIENT id
     p12: Inic     --> INI num
     p13: Fim      -->  &
     p14:            |  FIM num
     }
```

Como se vê acima, foram retiradas cinco produções unitárias, desaparecendo cinco símbolos não-terminais (N foi assim reduzido). O alfabeto T mantém-se inalterado, bem como o axioma S. Os símbolos no RHS das produções alteram-se mas o seu comprimento mantém-se.

[22] Note que as alterações são não-cumulativas, isto é, são independentes e realizadas sempre sobre a gramática original.

Transformação (T2) Para realizar a segunda transformação, será necessário acrescentar dois não-terminais a N e consequentemente duas produções a P, como se mostra abaixo:

```
P''= {
     p0:  Gci      -->  PGs '.'
     p1:  PGs      -->  Pg
     p2:             |   PGs ';' Pg
     p3:  Pg       -->  IdOrient DescOrientac Periodo
     p4:  IdOrient -->  id
     P5:  Tipo     -->  PHD
     p6:             |   MSC
     p7:  CoOrient -->  &
     p8:             |   CO-ORIENT IdOrient
     p9:  Aluno    -->  Nome
     p10: Titulo   -->  str
     p11: Nome     -->  str
     p12: Inic     -->  INI Ano
     p13: Fim      -->  &
     p14:            |   FIM Ano
     p15: Ano      -->  num
     p16: DescOrientac --> Tipo CoOrient Aluno '(' Titulo ')'
     p17: Periodo  -->  Inic Fim
    }
```

Uma vez mais, T e S mantém-se inalterados, bem como a linguagem gerada.

Transformação (T3) A terceira e última transformação, que visa tornar mais simples a forma de organizar e fornecer a informação, embora não altere o axioma S, vai mudar a linguagem à custa da introdução de dois novos símbolos não-terminais e três novas produções, conforme se ilustra a seguir.

```
P'''={
     p0:  Gci      -->  PGs '.'
     p1:  PGs      -->  Pg
     p2:             |   PGs  Pg
     p3:  Pg       -->  IdOrient '[' Orientacs ']'
     p4:  IdOrient -->  id
     p5:  Tipo     -->  PHD
     p6:             |   MSC
     p7:  CoOrient -->  &
     p8:             |   CO-ORIENT IdOrient
     p9:  Aluno    -->  Nome
     p10: Titulo   -->  str
     p11: Nome     -->  str
     p12: Inic     -->  INI Ano
     p13: Fim      -->  &
     p14:            |   FIM Ano
     p15: Ano      -->  num
     p16: Orientacs-->  Orientac
     p17:            |   Orientacs ';' Orientac
```

```
 p18: Orientac -->  Tipo CoOrient Aluno '(' Titulo ')' Inic Fim
}
```

Apesar do enunciado não o explicitar, julgou-se conveniente acrescentar a T dois novos símbolos terminais (os parêntesis rectos) para identificar a lista de orientações de um mesmo orientador; note-se que em consequência disso $p2$ foi alterado pois deslocou-se o separador " ;"para dentro dessa nova lista.

Avaliação – fase 1 Na tabela 4.12 mostram-se as medidas que vão permitir avaliar a gramática original G e as três metamorfoses. Note-se que apenas se mostram (na coluna das Observações) os cálculos conducentes ao valor tabelado para o caso da gramática original (`GOrig`); os outros omitem-se por serem obviamente semelhantes.

Metrica	GOrig	Obs.	T1	T2	T3
MT1					
#T	12	=5(PR)+4(Sin)+3(TV)	12	12	14
#N	12		7	14	14
#P	16		11	18	19
#PU	5		0	5	5
#R	1		1	1	2
§RHS	1,7	Nota 1	2	1,7	1,8
§RHS-Max	9		9	6	8
§Alt	1,3	Nota 2	1	1,3	1,4
§Alt-Max	2		2	2	2
#Mod	0	Nota 3	0	0	0
MT2					
FanIn/$\#(N \cup T)$	2,3/24	Nota 4	2,9/19	2,1/26	2,3/28
FanOut/$\#(N \cup T)$	1,2/24	Nota 5	1,1/19	1,1/26	1,2/28
MT3					
§RD	24	=(12+12)	19	26	28
§TabLL	156	=(12*(12+1))	91	182	210
§$\mathcal{AD}$-LR	29		24	31	34
§TabsLR	377;348	=(29*(12+1));(29*12)	312;168	403;434	510;476
MF1	RecDirecta	Nota 6	RecDirecta	RecDirecta	RecDirecta
MF2	RecE	Nota 7	RecE	RecE	RecE
MF3	BNF	Nota 8	BNF	BNF	BNF
ML1	15/15	Nota 9	10/10	17/17	17/17
ML2	9/9	Nota 10	9/9	9/9	11/11
ML3	3/3	Nota 11	3/3	3/3	3/3
ML4	2	Nota 12	2	2	2

Tabela 4.12: Medidas da Gramática Independente de Contexto da linguagem GCI

Notas relativas à tabela 4.12:

1. =(2+1+3+9+1+1+1+0+2+1+1+1+2+0+1+1)/16.
2. =(1+2+1+1+2+2+1+1+1+1+2+1)/12

3. não há inclusão de qualquer módulo gramatical.
4. =(2+3+9+1+2+2+1+1+1+2+1+2)/12.
5. =(0+2+1+1+1+1+2+1+1+1+1+1+2+1+1+1+1+1+1+1+2+1+1+1)/24.
6. só há recursividade directa (na produção $p2$).
7. a recursividade presente é à esquerda.
8. todas as produções estão escritas em BNF-puro.
9. á luz da definição 24 os identificadores dos símbolos não-terminais e terminais-variáveis são esclarecedores do conceito que denotam.
10. o alfabeto é formado por palavras-reservadas *esclarecedoras* pois `PHD, MSC, CO-ORIENT, INI, FIM` derivam (de acordo com a definição 24) dos nomes dos respectivos conceitos (Ph.D., M.Sc., Co-orientador,Inicio, Fim) e por sinais `".", ";", "(", ")"` também considerados *esclarecedores*por serem separadores convencionais e claros.
11. os Terminais-Variáveis da gramática (`num, id, str`) são considerados os 3 flexíveis do ponto de vista léxico por serem suficientes, no contexto em causa, para representar as grandezas respectivas (anos, identificadores dos orientadores e nomes dos alunos).
12. aceita comentários da marca "//" até fim-de-linha (inline) e blocos de comentários entre "{" e "}".

Tendo em consideração a caracterização das Gramáticas Independentes de Contexto feita ao longo da subsecção 4.1.1 (sintetizada na tabela 4.1)[23], atendendo às métricas definidas na subsecção 4.2.1 e às tabelas associadas que descrevem o sentido em que cada qual interfere na qualidade, pode então usar-se as medidas registadas na tabela 4.12 e proceder à análise da GIC da linguagem GCI. Essa análise será feita em relação à gramática original (cujos valores medidos se encontram na primeira coluna); posteriormente usam-se as medidas das três variantes para estudar o impacto de cada transformação realizada. Assim sendo, pode dizer-se que:

- Relativamente à informação que se retira das métricas de tamanho (e tendo em conta a tabela 4.4), a GIC em avaliação é uma gramática de médio porte *fácil de compreender*, *manter* e, também, *de usar para derivar as frases* (possivelmente um pouco maçadora); os reconhecedores que se podem gerar da gramática são igualmente de médio porte e por conseguinte *satisfatoriamente eficientes*. Estas afirmações justificam-se com base nas seguintes evidências:
  - a existência de uma percentagem grande de produções unitárias (5 em 16) e apenas 1 símbolo não-terminal recursivo são os principais factores que contribuem para facilitar a compreensão e manutenção desta GIC;
  - os lados direitos curtos (em média inferior a 2 símbolos) e um número de produções alternativas baixo (inferior a 1,5) são mais duas características que simplificam a compreensão e manutenção, embora esse excesso de lados direitos curtos e de produções unitárias tornem o uso para derivação um pouco mais lento e fastidioso;
  - os valores claramente baixos do FanOut (cada símbolo só é usado na definição de 1 outro) e do FanIn (em média 2.3 símbolos derivam de cada não-terminal) confirmam que não se trata realmente de uma gramática complexa e que não se espera grande dificuldade na sua manipulação, como o comprovam as três transformações[24] que foram facilmente executadas;

[23] Onde se discutiu a influência dessas características na qualidade da gramática.
[24] Que na vida real podiam perfeitamente ser tarefas de manutenção.

  - as dimensões do *Parser*-RD ou das tabelas de parsing são médias (sobretudo as do *Parser*-LL(1)) o que torna o reconhecimento razoável em termos de consumo de memória e de tempo;
  - o não-recurso a outros módulos, com partes da gramática, aumenta também a velocidade de leitura e derivação, sem afectar significativamente a manutenção.

- As métricas de forma, de acordo com a tabela 4.5, abonam em termos da boa usabilidade da gramática em causa e da eficiência dos reconhecedores que dela se derivam, visto que:
  - a facilidade em ler/perceber e manter esta gramática é ajudada pelo uso exclusivo de recursividade directa;
  - o uso da notação pure-BNF, embora não seja o preferido pelos peritos em engenharia gramatical, não é complicado nem para esses nem para os utilizadores leigos;
  - o uso de recursividade à esquerda poderá ser, neste caso concreto, a maior dificuldade para os leigos na área, embora não seja complexa para os peritos; em contrapartida, este tipo revela-se como o mais favorável na implementação dos reconhecedores.

- Olhando para as métricas lexicográficas e para a tabela 4.6, constata-se que os identificadores dos símbolos não-terminais e terminais-variáveis são elucidativos do conceito que denotam, o que ainda vem facilitar mais a usabilidade da gramática sem interferir na eficiência do reconhecimento.

Relativamente à linguagem gerada pela GIC acabada de analisar, pode-se apenas dizer que:

- a sua aprendizagem é muito fácil, pelo menos pelo lado da compreensão da gramática geradora;
- o seu reconhecimento pode ser satisfatório, se o respectivo *Parser* for bem escolhido e implementado;
- a sua expressividade, no que se pode deduzir das métricas lexicográficas, será boa pelo menos no que se refere ao uso de palavras-reservadas e sinais expressivos e terminais-variáveis suficientemente flexíveis para escrever os valores em causa (ver tabela 4.7), não pondo, por esse lado, em risco a escalabilidade.
- a documentação permitida, para auxiliar a legibilidade das frases da linguagem gerada pela GIC em apreço, é boa, conforme o confirmam ainda as métricas lexicográficas (2 pontos em 3).

Uma vez estudada a qualidade da gramática original da linguagem GCI, é importante usar as métricas e os valores tabelados em 4.12 para perceber a forma como cada alteração pode influenciar a qualidade. A este respeito verifica-se que:

- A transformação (T1), ao *eliminar as produções unitárias*, sem alterar a linguagem reduziu significativamente o tamanho da gramática (sem afectar as outras métricas de forma ou lexicográficas), o que se traduz num claro aumento da eficiência do reconhecimento sacrificando a usabilidade (menos compreensível e um pouco mais complexa para manter, mas menos aborrecida para derivar frases).
- A transformação (T2) reduziu o tamanho máximo do RHS duma produção (de 9 para 6) preservando a linguagem com o intuito de facilitar a leitura/compreensão. Em termos de métricas (de tamanho, porque as outras ficam inalteradas), esta transformação não parece ter um impacto francamente positivo, embora a complexidade sintáctica diminuia e o aumento ligeiro de #N e #P possam ajudar um pouco a legibilidade à custa de um substancial aumento das tabelas de parsing e consequente degradação da eficiência do reconhecimento.

- A transformação (T3) tinha por objectivo alterar a linguagem gerada no sentido de aumentar a sua expressividade tornando as frases mais curtas (para o mesmo conteúdo) e mais agradáveis de escrever. Admitindo que a alteração introduzida nesta 3ª metamorfose atingiu esse objectivo, permitindo reunir em lista todas as orientações do mesmo orientador, é interessante analisar que o impacto na gramática foi em sentido contrário tornando o reconhecimento mais pesado e degradando muito ligeiramente a sua usabilidade (com um tamanho maior e mais produções recursivas) — provavelmente uma transformação do tipo de (T1) para eliminar as produções unitárias após (T3) seria a transformação ideal.

Analisada a componente sintáctica da linguagem, a GIC, repete-se o estudo acima para a componente semântica tomando apenas em consideração a gramática original (as transformações não serão aqui estudadas).

Enunciado do Problema – fase 2 Do ponto de vista semântico, uma frase só será considerada válida se se observarem as seguintes restrições (semântica estática da linguagem):

- um orientador não pode ter mais de duas teses de cada tipo;
- a data de fim de uma tese concluída nunca pode ser menor ou igual à sua data de início.

Ao processar automaticamente as frases da linguagem GCI, pretende-se produzir os resultados seguintes (semântica dinâmica da linguagem):

- gerar uma listagem com as siglas de todos Orientadores (sem repetições);
- imprimir, por ordem crescente de ano de início, os alunos de Doutoramento (nome, título da tese e orientador) com teses em andamento (iniciadas mas não acabadas);
- contar e imprimir o número total de teses concluídas por ano de conclusão e por tipo (phd, ou msc).

Resolução – fase 2 Para resolver este problema, isto é, escrever uma Gramática de Atributos que especifique formalmente as restrições—definindo as frases com significado (semanticamente válidas)—e o resultado pretendido—a obter após o processamento das frases—aplica-se uma vez mais a abordagem incremental, sendo neste caso propostos 5 passos ou fases, 2 para as condições contextuais (*fases 1 e 2*) e 3 para cada um dos pedidos (*fases 3 a 5*).
Como resultado desta *abordagem estratificada por objectivos* para rápido desenvolvimento de GAs, associam-se aos símbolos os atributos que se organizam na tabela 4.13, indicando-se a fase que motiva a sua inclusão.

Os atributos identificados na tabela 4.13 têm os seguintes tipos de dados:

```
tabTeses  = hashMap( str,int );           //complexo
tipoT     = enum( d, m );                 //atómico
listaO    = seq( str );                   //complexo
listaDA   = seq( tup(int,str,str,str) );  //complexo
tabTConc  = array( int,tipoT ) de int;    //complexo
str       = array( int ) de char;         //complexo
int;                                      //atómico
```

Quanto às regras de cálculo e tradução e às condições contextuais, lista-se a seguir o resultado da fusão dos 5 passos, mencionando também a fase a que dizem respeito.
Regras de Cálculo:

Símbolo	AHerdados	Fase	ASintetizados	Fase
Gci				
PGs	contD_in: tabTeses	1	contD_out: tabTeses	1
	contM_in: tabTeses	1	contM_out: tabTeses	1
	lstOrs_in: listaO	3	lstOrs_out: listaO	3
	lstDsA_in: listaDA	4	lstDsA_out: listaDA	4
	contTC_in:tabTConc	5	contTC_out:tabTConc	5
Pg			idO: str	1,3,4
			tipT: tipoT	1,4
			al: str	4
			tit: str	4
			ini: int	4
			fim: int	4
	contTC_in:tabTConc	5	contTC_out:tabTConc	5
IdOrient			idO: str	1,3,4
Tipo			tipT: tipoT	1,4
CoOrient				
Aluno			nome: str	4
Titulo			nome: str	4
Nome			nome: str	4
Inic			aa: int	2,4
Fim			aa: int	2,4
Ano			aa: int	2,4
num			val: int	-
id			val: str	-
str			val: str	-

Tabela 4.13: Atributos da GA da linguagem GCI

```
RC(p0) = { PGs_1.contD_in= inicTabTeses();                          //fase 1
           PGs_1.contM_in= inicTabTeses();                          //fase 1
           PGs_1.lstOrs_in= NULL;                                   //fase 3
           PGs_1.lstDsA_in= NULL;                                   //fase 4
           PGs_1.contTC_in= inicTConc()                             //fase 5
         }
RC(p1) = {
           PGs_0.contD_out= (Pg_1.tipT==d)?
                            incr(PGs_0.contD_in,Pg_1.idO) :
                            PGs_0.contD_in;                 //fase 1
           PGs_0.contM_out= (Pg_1.tipT==d)?
                            PGs_0.contM_in :
                            incr(PGs_0.contM_in,Pg_1.idO);   //fase 1
           PGs_0.lstOrs_out= insSRep(Pg_1.idO,PGs_0.lstOrs_in);     //fase 3
           PGs_0.lstDsA_out= ((Pg_1.tipT==d)e(Pg_1.fim==0))?
                            insOrd(PGs_0.lstDsA_in,Pg_1.ini,Pg_1.al,Pg_1.tit,Pg_1.idO) :
                            PGs_0.lstDsA_in;                 //fase 4
           Pg_1.contTC_in= PGs_0.contTC_in;                         //fase 5
           PGs_0.contTC_out= Pg_1.contTC_out                        //fase 5
         }
RC(p2) = { PGs_1.contD_in= PGs_0.contD_in;                          //fase 1
```

```
            PGs_1.contM_in= PGs_0.contM_in;                          //fase 1
            PGs_0.contD_out= (Pg_3.tipT==d)?
                             incr(PGs_1.contD_out,Pg_3.idO) :
                             PGs_1.contD_out;                        //fase 1
            PGs_0.contM_out= (Pg_3.tipT==d)?
                             PGs_1.contM_out :
                             incr(PGs_1.contM_out,Pg_3.idO);         //fase 1
            PGs_1.lstOrs_in= PGs_0.lstOrs_in;                        //fase 3
            PGs_0.lstOrs_out= insSRep(Pg_3.idO,PGs_1.lstOrs_out);    //fase 3
            PGs_0.lstDsA_out= ((Pg_3.tipT==d)e(Pg_3.fim==0))?
                             insOrd(PGs_1.lstDsA_out,Pg_3.ini,Pg_3.al,Pg_3.tit,Pg_3.idO) :
                             PGs_1.lstDsA_out;                       //fase 4
            PGs_1.lstDsA_in= PGs_0.lstDsA_in;                        //fase 4
            PGs_1.contTC_in= PGs_0.contTC_in;                        //fase 5
            Pg_3.contTC_in=  PGs_1.contTC_out;                       //fase 5
            PGs_0.contTC_out= Pg_3.contTC_out                        //fase 5
          }
RC(p3) = { Pg_0.idO  = IdOrient_1.idO;                               // fase 1,3,4
           Pg_0.tipT = Tipo_1.tipT;                                  // fase 1,4
           Pg_0.fim = Fim_9.aa;                                      // fase 4
           Pg_0.ini = Inic_8.aa;                                     // fase 4
           Pg_0.al = Aluno_4.nome;                                   // fase 4
           Pg_0.tit = Titulo_6.nome;                                 // fase 4
           Pg_0.contTC_out= (Fim_9.aa=/=0)?
                             incrTC(Pg_0.contTC_in,Fim_9.aa,Tipo_1.tipT) :
                             Pg_0.contTC_in                          //fase 5
          }
RC(p4) = { IdOrient_0.idO = id_1.val                                 //fase 1
          }
RC(p5) = { Tipo_0.tipT = d                                           //fase 1
          }
RC(p6) = { Tipo_0.tipT = m                                           //fase 1
          }
RC(p9) = { Aluno_0.nome = Nome_1.nome                                //fase 4
          }
RC(p10)= { Titulo_0.nome = str_1.val                                 //fase 4
          }
RC(p11)= { Nome_0.nome = str_1.val                                   //fase 4
          }
RC(p12)= { Inic_0.aa = Ano_2.aa                                      //fase 2
          }
RC(p13)= { Fim_0.aa = 0                                              //fase 2
          }
RC(p14)= { Fim_0.aa = Ano_2.aa                                       //fase 2
          }
RC(p15)= { Ano_0.aa = num_1.val                                      //fase 2
          }
```

Condições de contexto:

```
CC(p0) = { para_todo (id in dom(PGs_1.contD_out)) { (PGs_1.contD_out[id] <= 2) } //fase 1
           para_todo (id in dom(PGs_1.contM_out)) { (PGs_1.contM_out[id] <= 2) } //fase 1
         }
CC(p3) = { se (Fim_9.aa > 0) então { (Fim_9.aa > Inic_8.aa) }                    //fase 2
         }
```

Regras de tradução:

```
RT(p0) = { imprimeO(  PGs_1.lstOrs_out ); //fase 3
           imprimeDA( PGs_1.lstDsA_out ); //fase 4
           imprimeTC( PGs_1.contTC_out )  //fase 5
         }
```

Na Gramática de Atributos acima usam-se as seguintes funções/procedimentos

```
inicTabTeses() -> tabTeses
incr( tabTeses,str )  -> tabTeses
insSRep( str,listaO ) -> listaO
insOrd( listaDA,int,str,str,str ) -> listaDA
inicTConc() -> tabTConc
incrTC( tabTConc,int,tipoT ) -> tabTConc
imprimeO( listaO )   -> void
imprimeDA( listaDA ) -> void
imprimeTC( tabTConc )-> void
```

cuja definição não se apresenta por ser óbvia.

Avaliação – fase 2 De modo semelhante ao que se fez na fase anterior para avaliação da componente sintáctica, mostra-se, agora, na tabela 4.14 as medidas que vão permitir avaliar a gramática da linguagem GCI na sua componente atributiva.

Notas relativas à tabela 4.14:

1. as regras de cálculo, por um lado seguem o padrão da não agregação de valores parciais em atributos compostos (os valores parciais são mantidos em atributos distintos) e por outro lado seguem maioritariamente o padrão intermédio herdado/sintetizado, em que a informação de contexto desce na árvore até às produções recursivas colocando-se a operação de acumulação ao nível dessas produções (como se confirma analisando as `RCs` associadas às produções $p1$ e $p2$—note-se que a nível das produções mais elementares apenas há sintese de atributos, como se confirma olhando para `RC(3)`). Para corroborar esta classificação, note-se que os atributos do símbolo `PGs` surgem aos pares herdado/sintetizado, enquanto que o elemento desta lista, o símbolo `Pg`, não tem praticamente atributos herdados. Nesta GA ainda surge um caso de cálculo que foge deste princípio e segue o esquema descrito pelo padrão puramente herdado, em que a informação de contexto desce na árvore até às produções mais elementares (com folhas como descendentes), através de atributos herdados, para depois acumular, em atributos de saída dessas produções, a nova informação que será sintetizada pela árvore acima; é o que se passa com o cálculo do atributo `contTC_out` do símbolo `Pg`, o qual é sintetizado ao nível da produção elementar $p3$.

2. as duas condições de contexto estão associadas às produções onde efectivamente se pretende impor a restrição.

3. as três condições de contexto estão associadas às produções onde a respectiva restrição semântica faz sentido ser verificada.

4. as três regras de tradução estão associadas à produção raiz, $p0$, o que obrigada a sintetizar a informação de saída até ao topo.

Metrica	Medida	Obs.
MTA1		
#A/#($N \cup TV$)	29/15	
#AI	6	=(0+5+1+0+0+0+0+0+0+0+0+0+0+0+0)
#AS	23	=(0+5+7+1+1+0+1+1+1+1+1+1+1+1+1)
#RC/#P	39/16	=(5+6+11+7+1+1+1+0+0+1+1+1+1+1+1+1)
#CC/#P	3/16	=(2+0+0+1+0+0+...)
#RT/#P	3/16	=(3+0+0+0+0+0+...)
MTA2		
FanIn/#A	1,3/29	=(38/29)
FanOut/#A	1,2/29	=(35/29)
MFA1		
#AComplex/#A	21/29	=(21/(21+8))
MFA2		
#OComplex/#O	0/45	=(0/(0+45))
MFA3	RCNAgreg + RCmixIS/RCpureI	Nota 1
MFA4	CCCentrad	Nota 3
MFA5	RTTop	Nota 4
MFA6	LImperativa	Ling. à la Pascal
MFA7	LStandard	Ling. Algorítmica genérica (Nota 4 da tabela 4.11)
MLA1		
#IdACompl/#A	26/29	=(26/(26+3)) (Nota 5)
MLA2		
#IdOCompl/#O	9/10	=(9/(9+1)) (Nota 6)

Tabela 4.14: Medidas da Gramática de Atributos da linguagem GCI

5. os identificadores dos atributos *derivam*, na sua grande maioria, do nome do conceito que representam (segundo a definição 24); por exemplo os atributos que guardam o Contador de Doutorandos, o Contador de Mestrandos, a Lista de Orientadores, a Lista de Doutoramentos em Andamento, a Lista de Teses Concluídas, ou o Valor do símbolo são identificados por `contD`, `contM`, `lstOrs`, `lstDsA`, `lstTC` e `val`. São excepção a este princípio três atributos, todos designados por `aa`, cujo identificador não se pode considerar o esclarecedor do conceito que denota, Ano.

6. a maioria (9) dos identificadores das operações atributivas—`=, inicTabTeses, inicTConc, insSRep, insOrd, incrTC, imprimeO,imprimeDA,imprimeTC`—podem considerar-se como *denotando-bem*, ou *derivando*, dos nomes das operações que implementam—atribuição (cópia de valores), inicializa tabela de teses, inicializa tabela teses concluídas, insere sem repetições, insere ordenado, incrementa a tabela de teses concluídas, imprime orientadores, imprime doutoramentos em andamento, imprime teses concluídas—enquanto que o operador restante—`ins`—não é um identificador esclarecedor.

Considerando os valores registados na tabela 4.14 à luz das relações patentes na tabela 4.9 (ver subsecção 4.2.2) e da discussão sustentada na subsecção 4.1.2 e sintetizada na tabela 4.2, procede-se abaixo à análise da Gramática de Atributos da linguagem `GCI`. Mantendo a avaliação já feita acima à componente estrutural, ou sintáctica, pode agora acrescentar-se que:

- se trata de uma GA nem muito 'pesada' nem muito 'complexa' na medida em que a componente atributiva não é demasiado grande: cerca de 2 atributos por símbolo, e cerca de 2 regras de cálculo por produção.

- esta ideia de que a GA em causa não é 'complexa', é corroborada pelos seguintes factos:
  - as operações atributivas são todas simples (descrevem-se em 1 instrução);
  - os identificadores de ambos, atributos e operações atributivas, são na sua quase totalidade esclarecedores e, portanto, facilitadores de uma rápida compreensão;
  - o grau de dependência entre atributos é consideravelmente baixo, como o comprovam o FanIn e FanOut que rondam o valor 1 (um atributo depende de 1 outro para sua definição e participa na definição de 1 outro);
  - a linguagem usada na escrita das operações atributivas, apesar de não ser declarativa, é perfeitamente adequada ao tipo simples de operações envolvidas (essencialmente atribuições, que correspondem a regras de cópia) e apresenta a vantagem de não ser uma linguagem específica para manuseamento de GAs (é uma linguagem algorítmica típica).
  - os padrões, ou esquemas usados para a escrita das regras de cálculo e colocação das condições de contexto (`RCNAgreg`, `RCpureI`/`RCmixIH` e `CCCentrada`, respectivamente), são os que mais facilitam a compreensão;
  - o esquema de escrita das regras de tradução, `RTTop`, embora não seja o mais fácil de analisar, é um dos mais usuais e não pesa muito visto as 3 regras existentes serem muito simples e semelhantes;
- a gramática, uma vez bem percebida, não levanta especiais dificuldades nem a quem tem de a usar para escrever frases da linguagem (além da simplicidade sintáctica, só há 3 restrições contextuais a preservar), nem a quem tem de a manter (além do tamanho não ser grande, a complexidade semântica é baixa).
- o principal foco de dificuldade a perceber esta GA e sobretudo de ineficiência no processamento resulta da maioria dos atributos (21 no total de 29) serem de tipo complexo (e ainda para mais os tipos estruturados usados são sequências, arrays e tabelas de hashing).
- o processador que dela se gera, ou deriva, não pode ser muito eficiente devido a: dimensão absoluta (que requer mais memória para armazenar todos os atributos e, consequentemente, um maior tempo de travessia à árvore e tempo de cálculo); complexidade dos atributos que se traduz na complexidade das operações que os manipulam.
- o gerador do processador, que também é um pouco afectado pela dimensão da GA, não enfrenta dificuldades especiais na análise de dependências e de determinação da ordem total de cálculo, sobretudo devido a um baixo grau de complexidade (FanIn+ FanOut) entre atributos e à escolha do esquema misto herdado/sintetizado para escrita das regras de cálculo.

Relativamente à linguagem gerada por esta GA, pouco há a acrescentar ao que já foi dito atrás, aquando da avaliação da GIC subjacente. Em termos de balanço final, pode-se então e apenas dizer que:

- a sua aprendizagem será fácil, pelo menos pelo lado da compreensão da gramática geradora, devido à simplicidade sintáctica e ao peso da parte atributiva;
- o seu processamento, apesar de não muito afectado pelo tamanho geral da gramática (quer da componente sintáctica, quer da componente semântica), é prejudicado pela complexidade dos valores dos atributos e respectivas operações;
- a sua expressividade, no que se pode deduzir das métricas lexicográficas, será boa pelo menos no que se refere ao uso de palavras-reservadas e sinais expressivos e terminais-variáveis suficientemente flexíveis para escrever os valores em causa (ver tabela 4.7), não pondo, por esse lado, em risco a escalabilidade.

- a documentação permitida, para auxiliar a legibilidade das frases da linguagem gerada pela GIC em apreço, é boa, conforme o confirmam ainda as métricas lexicográficas (2 pontos em 3).

Capítulo 5

Reflexões finais

Assumindo que as linguagens formais—para programar a resolução de problemas ou o controlo de sistemas, para especificar problemas, ou para anotar documentos ou programas—são o instrumento de trabalho de uma vasta classe de profissionais do ramo da informática e considerando que a oferta é bastante grande, partiu-se para esta lição com o intuito de discutir o que é a qualidade de uma linguagem.

5.1 Sintetizando...

Tendo-se constatado que é possível caracterizar a linguagem de modo a estabelecer um critério de qualidade, percebeu-se que as características envolvidas não são facilmente mensuráveis, o que torna difícil sistematizar a sua aferição, ou seja, a avaliação da qualidade.
Sendo certo que as linguagens formais são definidas rigorosamente por gramáticas (nesta contexto, ao dizer-se *gramática* está-se a considerar a Gramática de Atributos, GA, com a subjacente Gramática Independente de Contexto, GIC) considerou-se um desafio estudar a qualidade das gramáticas com base em características mensuráveis e investigar o impacto que essa avaliação possa ter na avaliação da linguagem.

Assim foi objectivo da lição apresentada, explorar (descobrir e justificar) a importância da Engenharia Gramatical no manuseamento rigoroso das linguagens (formais) clássicas, as linguagens textuais.
Por ser um assunto vasto que daria uma outra lição, as Linguagens Visuais (ver conceito na secção 1.1) não foram abordadas—reconhece-se porém a sua importância actual e assume-se a necessidade de estender o presente estudo de modo a abarcar o que essa classe mais lata traga de específico, pois tudo o que foi dito também lhes é aplicado.

A primeira preocupação foi justificar a necessidade de falar em qualidade de linguagens e na sua aferição; dada a dificuldade da questão em termos absolutos, considerou-se pelo menos indispensável dispor de critérios para comparar linguagens. Nesse sentido apresentaram-se os factores (subjectivos) que podem ser usados para estabelecer o referido critério: *legibilidade* da linguagem (em termos de facilidade de *aprendizagem*, de *escrita* de frases e de *compreensão*) e *eficiência* no processamento. Depois, procedeu-se à caracterização das linguagens e à identificação do contributo que cada característica identificada—*expressividade, documentação, unicidade, consistência, extensibilidade, escalabilidade, fiabilidade, modularidade*—trás aos factores que vão ser usados na avaliação da qualidade.
Assumido o desafio acima enunciado, e sempre com o intuito de perceber o que daí se poderia tirar para avaliar a linguagem, focou-se o discurso na caracterização de gramáticas e na apresentação de um conjunto de *métricas*—de *tamanho*, *forma* e *feitio (lexicográficas)*—que possa ser usado para avaliar a sua qualidade

em termos de *usabilidade* (*compreensão*, *derivação* de frases e *manutenção*), enquanto instrumento para gerar uma linguagem, e *eficiência*, enquanto instrumento para gerar um processador para essa linguagem. Para ligar as duas partes (gramática e linguagem), foi necessário relembrar definições basilares que suportam essa ligação: Gramática Independente de Contexto e Gramática de Atributos; árvore de derivação e frase válida da linguagem gerada pela gramática; ambiguidade; processador de linguagem e seus componentes. Recordou-se ainda os principais conceitos ligados ao processamento de uma linguagem e alguns algoritmos, para que se possa discutir a sua eficiência.

A maior lição tirada deste estudo pode enunciar-se assim:

> Uma gramática é um objecto matemático—instrumento de trabalho para o engenheiro de linguagens—facilmente mensurável, quer na componente estrutural (sintáctica), quer na componente atributiva. Das medidas retiradas pode-se aferir a qualidade da gramática; podem-se também usar as medidas para comparar gramáticas equivalentes[1] ou para avaliar o impacto na qualidade quando se altera a gramática para mudar a linguagem. Porém a avaliação da gramática não é suficiente para avaliar a linguagem, visto que a maioria dos parâmetros mensuráveis da primeira não afecta directamente as oito características que determinam a qualidade da linguagem. Essencialmente, verifica-se que a gramática tem um efeito directo na facilidade de aprendizagem da linguagem e na eficiência com que as suas frases serão processadas—daí que o engenheiro gramatical deva usar a medição da gramática para procurar escolher entre as equivalentes aquela (ou aquelas) que melhor servem a qualidade da linguagem a desenvolver.

Ao fazer o balanço do que foi dito, convém relembrar que a gramática é um meio e não um fim, é o instrumento de especificação e não o produto final. Nestas circunstâncias, a gramática determina, influencia[2], a qualidade da linguagem, mas contudo não são uma só e a mesma coisa e as características mensuráveis de uma podem não ser suficientes para a avaliação total da qualidade da outra. É o mesmo que sucede em qualquer sistema de produção; as fichas técnicas estabelecem as características que irão determinar a qualidade do produto final, mas a sua análise não é suficiente para a avaliar.

5.2 Divagando, fechando...

Como se disse acima, o estudo feito restringiu-se às Linguagens de Programação escritas textuais deixando de fora um importante sub-conjunto das Linguagens de Programação Visuais. Uma evolução imprescindível deste estudo será fazer essa extensão. Uma vez que são linguagens formais, também definidas por gramáticas, não parece haver grande dificuldade nem surpresas na referida tarefa; os critérios de avaliação terão de ser revistos e adaptados, mas o caminho de trabalho e os resultados deverão ser idênticos.

Considerando a **gramática** como um especificação formal de determinada linguagem e o **gerador de processadores de linguagens** como um reificador dessa especificação, e vendo a **linguagem** como uma família (ou classe) de objectos concretos (as frases) que seguem o mesmo padrão sintáctico-semântico, é possível extrapolar e falar de Programação Gramatical como forma de *gramaticar a programação* . O conceito de Programação Gramatical, que não pôde ser introduzido extensivamente por falta de espaço mas que fica mencionado nesta reflexão final, tem duas vertentes. Uma delas vai no sentido de se falar em *abordagem gramatical à programação*; a outra dirige-se para o uso de Grammarware na análise dos programas a manter. Quanto à primeira vertente, defende-se, claramente, uma aproximação à programação que tire partido do conceito de dualidade inerente à gramática e que consiste na separação distinta entre a *forma* (estrutura)

[1] Que geram a mesma linguagem.
[2] Embora essa influência não seja total nem directa.

e o *conteúdo*[3]. A abordagem defendida propõe que se mantenha essa separação, quer na fase de análise e especificação dos problemas, quer na fase de arquitectar o sistema de Software. Na fase de análise e especificação, o que se propõe é tão simplesmente analisar e descrever primeiro a estrutura dos objectos no domínio do problema e depois, numa segunda fase e sempre sobre essa estrutura, descrever a parte das operações, validações e transformações. Na fase de concepção da arquitectura do sistema, propõe-se que os vários blocos sejam acamados separando as tarefas de aquisição e armazenamento dos dados das tarefas de transformação dessa informação.

No que se refere à segunda vertente e assumindo a definição proposta pelos criadores da ideia [KLV05], —**GrammarWare** *é a designação colectiva de um conjunto de utilitários formado por ferramentas para trabalhar com gramáticas e ferramentas baseadas (suportadas) em gramáticas*—sugere-se um duplo uso desses instrumentos para automatizar o desenvolvimento e manutenção de programas.

A capacidade de *gerar automaticamente software* de qualidade a partir de especificações gramaticais—permitindo assim elegante e sistematicamente reutilizar soluções algorítmicas e de implementação desenvolvidas por peritos numa dada área—é explorada sob o tópico *programação generativa* tendo precisamente por objectivo usar os geradores incluídos no Grammarware para implementar soluções computacionais para problemas em outras áreas que não apenas no foro do processamento de linguagens.

Por outro lado, existe também a possibilidade de usar a mesma panóplia de ferramentas contidas no Grammarware (e os métodos e técnicas subjacentes) para analisar software existente e permitir a sua evolução por *transformação*; com essa *abordagem gramatical à manutenção* pretende-se oferecer um suporte rigoroso e eficaz a actividades como a compreensão de programas, a engenharia reversa ou a re-engenharia.

Aderindo a esta ideia de gramaticar a programação, um tópico claramente desafiante, que é aqui deixado como assunto de investigação futura, consiste em *determinar a influência das métricas gramaticais na programação generativa e na evolução do software por transformação.*

[3] A primeira componente definida pela Gramática Independente de Contexto e a segunda pela Gramática de Atributos.

Apêndice A

Um Exercício de Caracterização de Linguagens reais

Para aferir a possibilidade de aplicar efectivamente as características (CL) apresentadas na secção 2.3, fez-se um exercício experimental em que se escolheram 11 linguagens reais diferentes (9 GPLs e 2 de Anotação) e se procurou averiguar se era fácil avaliar, para cada linguagem, essas 8 características identificadas como influindo na qualidade de uma linguagem. Uma vez preenchida a grelha de avaliação, tentou-se perceber se no fim essa tabela permitia realmente distinguir as linguagens entre si e fazer qualquer reflexão à cerca da qualidade relativa.

Caract x Ling.s	Pascal	Basic	C	Perl	Prolog	C++	C#	Java	Haskell
Paradigma	Imp	Imp	Imp	Script/Imp	Decl/L	OO/Imp	OO/Imp	OO/Imp	Decl/F
CL1:Expressividade	mB	S	mB	MB	S	mB	MB	MB	MB
CL2:Documentação	bás	bás-	evol	evol	bás-	evol-	evol	evol	bás
CL3:Unicidade	N	N	N	N	N	N	N	N	N
CL4:Consistência	B	S	S	S	B	S	B	B	B
CL5:Extensibilidade	bás	bás-	bás+	evol	evol	evol	evol	evol	evol
CL6:Escalabilidade	S	S	B	B	R	B	B	R	R
CL7:Fiabilidade	B	R	S	S	R	B	MB	MB	+
CL8:Modularidade	min	min	S	S	min	mB	MB	MB	S

Tabela A.1: Averiguação das Características de Linguagens de Programação usuais

(Legenda) A Tabela A.1 foi preenchida de acordo com o seguinte critério:

min – satisfaz minimamente (insatisfatório)

S – satisfaz normalmente

B – satisfaz bem

MB – satisfaz muito bem

N – não verifica

bas – solução básica

evol – solução evoluída

Analisando a Tabela A.1 pode ver-se que há uma clara supremacia das linguagens imperativas mais recentes, orientadas-a-objectos, pois no computo geral são as que melhor satisfazem todas as características. De entre estas, nota-se ainda a vantagem do C♯ no que respeita à escalabilidade, pois como é sabido grandes programas em Java são executados muito lentamente, podendo até não se conseguir executar a partir de certa dimensão—sendo a mais recente, C♯ foi criada para suprir os problemas das suas antecessoras. É também evidente a limitação, ou fraqueza do Basic, em relação às restantes; a semelhança e balanço conhecido entre as grandes concorrentes da década de 80, Pascal e C, também transparece nessa tabela.

O exercício foi repetido para 2 conhecidas linguagens de anotação de documentos, como se regista na Tabela A.2(preenchida de acordo com a legenda da tabela anterior).

Caract x Ling.s	LaTeX	XML
Paradigma	Decl	Decl
CL1:Expressividade	MB	mB
CL2:Documentação	bás	B
CL3:Unicidade	N	Sim
CL4:Consistência	min	B
CL5:Extensibilidade	B	N
CL6:Escalabilidade	S	R
CL7:Fiabilidade	NA	NA
CL8:Modularidade	min	min

Tabela A.2: Averiguação das Características de Linguagens de Anotação usuais

A análise da Tabela A.2 mostra que estas linguagens cumprem bem o fim para que foram concebidas sendo bastante expressivas; nota-se que no caso da linguagem XML, muito mais recente que LaTeX, se procurou já resolver problemas críticos da segunda no que se refere à consistência, unicidade e documentação.

Apêndice B

Ferramentas e Ambientes de Suporte à Programação

Como se disse em 2.3, para o sucesso de uma Linguagens de Programação não contribui só a sua qualidade (resultante das características intrínsecas com que foi concebida). Existem outros factores pragmáticos muito relevantes para uma ampla aceitação e utilização duma Linguagens de Programação, tais como: (a) a familiaridade que o programador já tem com o paradigma e linguagens do mesmo estilo; (b) existência de um bom conjunto de ferramentas para a processar ou auxiliar o seu uso, soltas ou integradas num ambiente de desenvolvimento, IDE[1].

Neste apêndice referem-se de passagem as principais ferramentas que, indispensáveis ou acessórias, realmente determinam o sucesso de uma linguagem:

1. **Editor**, que permite escrever, ver e alterar o texto-fonte. Distinguem-se:
   - *Editor assistido pela sintaxe* que conhece a Gramática Independente de Contexto da linguagem e a usa para oferecer facilidades do tipo:
     - *syntax highlighting*, uso de cores para distinguir palavras-reservadas (todas em geral, ou por categorias) ou, mesmo, blocos (sub-estruturas) da linguagem; esta facilidade acelera a escrita e facilita a detecção de erros sintácticos.
     - *code completion*, predição de palavras ou sub-estruturas da linguagem que o programador está e começar a escrever; se bem adaptada, a facilidade de *autocomplete* acelera consideravelmente o tempo de escrita.
     - *code folding*, compressão/expansão de blocos ou partes da frase, com base nas sub-estruturas da linguagem, para facilitar a leitura de código muito extenso .
   - *Editor estrutural, ou guiado pela sintaxe*[2], que usa a Gramática Independente de Contexto da linguagem para impor a forma válida e ir preenchendo automaticamente blocos, palavras-reservadas e sinais, só deixando mesmo escrever literais nos locais correctos.
   - *Editor assistido pela semântica* que conhece a Gramática de Atributos da linguagem e tem acesso à informação semântica recolhida pelo compilador (tal como a Tabela de Identificadores) para oferecer facilidades do tipo:
     - *validações contextuais*, de declarações, ou tipos.

[1] Do inglês, Integrated Development Environment.

[2] Em inglês, Syntax-oriented (ou directed) Editors, como é o caso dos editores gerados pelo Synthesizer Generator (SGen).

– *IntelliSense*, uma variante do *code completion*, mas que não usa as palavras-reservadas e sim os identificadores disponíveis em cada contexto; facilidade muito importante em POO porque permite, entre outras coisas, aceder aos atributos e métodos de uma classe sem ter de os saber de cor ou de recorrer a um manual.

Outra importante capacidade dos actuais editores é o uso, e gestão[3], de *snippets*—pequenos fragmentos ou blocos de código que são reutilizados com frequência na escrita de frases de uma determinada linguagem; os snippets podem ser estáticos (texto fixo) ou dinâmicos (texto com variáveis, ou *placeholders*, que são instanciados de diversas formas no momento da reutilização.

2. **Processador**, que reconhece e executa as ordens contidas em cada frase da linguagem, podendo ser um **Compilador** ou **Interpretador**, conforme discutido e definido na secção 3.2. Distinguem-se:

   - Compilador Estático, clássico, que reconhece todo o código-fonte, desde o início até ao fim e de uma só vez, e gera o código-máquina também de uma só vez e após proceder a diversos níveis de optimização.
   - Compilador Incremental, que recompila o programa após cada edição, só recalculando o que foi alterado.
   - Compilador Dinâmico, que gera código-máquina em *runtime* e à medida das necessidades de modo a permitir alterações/adaptações ao código-fonte durante a execução e associações também dinâmicas de tipos a variáveis. A técnica mais vulgar actualmente é designada por Just In-Time Compiler (JIT), a qual se baseia na geração, em tempo de execução e por-necessidade, de código-máquina a partir do pseudo-código; essencialmente permite aumentar consideravelmente a eficiência dos programas portáveis, compilados para pseudo-código (código intermédio abstracto ou Assembly de uma máquina-virtual), possibilitando ainda obter comportamentos especiais do programa (não definidos estaticamente) que são possíveis pela compilação se ir fazendo em *runtime* com *feedback* da execução.

3. **Debugger**, que permite seguir a execução do código-máquina gerado com o propósito de descobrir erros. Distingue-se:

   - Debugger de baixo-nível (só mostra o Assembly) versus alto-nível (relaciona o Assembly com o código-fonte).
   - Debugger textual (embora normalmente exiba uma interface gráfica, descreve em texto o conteúdo da memória) versus visual (usa representações icónicas para as estruturas de dados contidas na memória e para o controlo do fluxo).

4. **Leitor de Documentação**, para mostrar todo o manual da linguagem e das ferramentas, ou simplesmente para apresentar dicas e resumos.

5. **Analisador de Código**, que permite inspeccionar o código-fonte extraindo dele informações diversas—tais como todos os identificadores usados, fluxo de controlo ou de dados, componentes do sistema e suas interdependências, etc.—que ajudam a compreender e a avaliar (a dimensão ou a eficiência) o programa (ou a especificação); normalmente, essa informação extraída é apresentada graficamente (em termos de tabelas, árvores e grafos) por Visualizadores associados ao Analisador; em certos casos, esses visualizadores aparecem ainda associados a Animadores que mostram como as grandezas em observação evoluem ao longo do tempo durante a execução.

[3] O editor deve permitir criar e manter um repositório de snippets.

6. **Controlador de Versões**, que permite manter o texto original e todas as diferenças a partir dele para cada nova versão do programa.

7. **Importador/integrador de Bibliotecas**, que permite gerir os ficheiros onde residem os módulos importados

8. **Gestor de Projectos**, que permite criar e controlar os vários ficheiros que constituem uma aplicação, ou sistema de software

Apêndice C

Recordando o *Parsing*

Neste apêndice são recordados os três principais algoritmos de *Parsing*, a que se faz referência explicita na secção das métricas (sec. 4.2) relacionadas como os critérios CG2.1 e CG2.2 (ver sec. 4.1). A talhe de foice, relembra-se também o conceito de Autómato Determinista LR($\mathcal{AD}$-LR), igualmente referido nessa métrica de tamanho (MT2).
Este apêndice completa, assim, as definições básicas sobre reconhecimento sintáctico, apresentadas na secção 3.1.

Parser Recursivo-Descendente (RD) O *Parser* RD é o mais intuitivo, natural, e implementa a estratégia TD (ver definição em 3.1).
O *Parser* RD é formado por:

- $\#T$ funções distintas para reconhecer cada um dos símbolos terminais; essas funções seguem todas o esquema baixo descrito no algoritmo 1.
- $\#N$ funções distintas para reconhecer cada um dos símbolos não-terminais. essas funções seguem todas o esquema baixo descrito no algoritmo 2.

Algoritmo 1 *Seja* `s` *o próximo símbolo da frase* f *a analisar e seja* `T` *o símbolo terminal que se quer reconhecer. A função, cujo algoritmo se esquematiza abaixo, retorna normalmente se o símbolo* `s` *for aceite; senão termina a execução intempestivamente.*

```
void rec_T(s)
   { se ( s==T) entao { s=daSimbolo() }
                senao { erro }
   }
```

Algoritmo 2 *Seja* `s` *o próximo símbolo da frase* f *a analisar e seja* `N` *o símbolo não-terminal que se quer reconhecer. Sejam* $p_1, \ldots, p_k$ *as* k *produções com* `N` *do lado esquerdo.*
Cada uma dessas k *produções é da forma* $p_i : N \rightarrow X_{i1} \ldots X_{im}$ *sendo* $la(p_i)$ *o respectivo* lookahead*, ou seja, o conjunto dos símbolos que podem aparecer no iníco de qualquer frase derivada de* p_i*.*
A função, cujo algoritmo se esquematiza abaixo, retorna normalmente se o símbolo `s` *poder ser ser aceite; senão termina a execução intempestivamente.*

```
void rec_N(s)
   { se ( s in la(p1) ) entao { rec_X11(s);...rec_X1m(s) }
```

```
      senao { se ( s in la(p2) ) entao { rec_X21(s);...rec_X2m(s) }
      senao { .....
      senao { se ( s in la(pk) ) entao { rec_Xk1(s);...rec_Xkm(s) }
      senao { erro }
    }
```

Parser LL(1) O *Parser* LL(1), tal como o anterior, implementa também a estratégia TD. Mas, ao contrário do antecessor, é iterativo e baseia-se num algoritmo standard (independente da gramática) guiado por tabela[1], o qual se esquematiza a seguir.

Algoritmo 3 *Seja* `s` *o próximo símbolo da frase* f *a analisar e seja* `SP` *uma* stack *(a* Stack *de* Parsing*) que guarda os símbolos* $N \cup T$ *da gramática que faltam reconhecer em cada momento (os objectivos); note-se que dado o caracter* Top-Down *da estratégia seguida,* `SP` *será inicializada com o axioma* `S` *seguido do* `eof, $`.

```
void parserLL()
   { s = daSimbolo();
     empilha(SP, "S$");
     repetir {
         X = topo(SP); pop(SP);
         se (X in T) entao { se (X==s) entao {accao=skip} senao {accao=err} }
         senao { accao = TabLL[X,s].acc }
         caso ( accao ) seja
         {
          ac   : |
          err  : sinaliza_erro() |
          skip : s = daSimbolo() |
          prod : p = TabLL[X,s].pr;
                 empilha(SP, RHS[p])
         }
       } ate  ( ac V err )
   }
```

Uma vez que este algoritmo é fixo, toda a informação específica da gramática da linguagem em análise está guardada na Tabela de Parsing, neste caso denotada por TabLL. A tabela TabLL é um *array bi-dimensional* assim definido:

```
TabLL = array [ N x T ] de accao
accao = par{
            acc : enum{ ac, err, skip, prod };
            pr  : inteiro // numero da producao
           }
```

Parser LR(.) O *Parser* LR(.) implementa, agora, a estratégia BU (ver definição em 3.1). Tal como o antecessor, é iterativo e baseia-se num algoritmo fixo, independente da gramática, guiado por tabela, o qual se apresenta abaixo.

Neste caso, as Tabelas de Parsing que guiam o reconhecedor são construídas a partir de um autómato determinista, designado por $\mathcal{AD}$-LR, em que cada estado representa todo o percurso já feito e indica a

[1] É um algoritmo *table-driven*.

acção a realizar em função do próximo símbolo terminal de f ou do símbolo não-terminal reconhecido. No contexto desta lição não se inclui mais detalhes sobre o conteúdo de cada estado[2], nem sobre a forma de construir o dito $\mathcal{AD}$-LR; o leitor interessado é remetido para [ASU86, WG84, Cre98], embora no apêndice E se incluia o autómato LR(0) correspondente á GIC da linguagem Lisp que se usou para as métricas do 1º caso de estudo (ver subsecção 4.3.1). Apenas interessa conhecer a estrutura das duas Tabelas de Parsing, ditas TabACTION e TabGOTO, que dele se extraem e que vão determinar o movimento do reconhecedor. As TabsLR são ambas *arrays bi-dimensionais* assim definidas:

```
TabACTION = array [ Q x T ] de accA
accA  = tup{
            acc : enum{ ac, err, shift, red };
            stat: inteiro // numero do estado para o qual transitar
            pr  : inteiro // numero da producao pela qual reduzir
           }
TabGOTO = array [ Q x N ] de accG
accG  = par{
            acc : enum{ err, shift };
            stat: inteiro // numero do estado para o qual transitar
           }
```

Algoritmo 4 *Seja* `s` *o próximo símbolo da frase* f *a analisar e seja* `SP` *uma* stack *(a* Stack *de* Parsing*) que neste caso guarda os estados do autómato já percorridos mas ainda não reduzidos; a* stack `SP` *será inicializada com o estado inicial do autómato que corresponde a nada se ter ainda feito e nada se saber.*

```
void parserLR()
   { s = daSimbolo();
     push(SP, QS);
     repetir {
         Q = topo(SP);
         accao = TabACTION[Q,s].acc }
         caso ( accao ) seja
         {
          ac   : |
          err  : sinaliza_erro() |
          shift: Q = TabACTION[Q,s].stat;
                 push(SP Q);
                 s = daSimbolo() |
          red  : p = TabACTION[Q,s].pr;
                 desempilha(SP, length(RHS[p]));
                 Q = TabGOTO[ topo(SP), LHS(p) ];
                 push(SP Q)
         }
       } ate  ( ac V err )
   }
```

[2]Os chamados *items LR*, agrupados no *kernel* e no *fecho dos items*.

Apêndice D

Padronização do Esquema para escrita das Operações Atributivas

Tal como na mesma linguagem de programação se podem escrever vários programas correctos para resolver o mesmo problema, também usando a mesma notação se pode especificar a mesma linguagem escrevendo diferentes Gramáticas de Atributos todas elas válidas. Porém o estilo usado para criar a Gramática de Atributos—ou seja a estratégia seguida para associar atributos aos símbolos e o esquema adoptado para escrever as operações atributivas envolvendo esses atributos—vai ter influência na legibilidade da gramática final e até na eficiência do processador dela derivado, como se procurou argumentar na subsecção 4.1.2.
Devido á importância que se reconheceu a esse factor—estilo de escrita das operações atributivas—na avaliação da qualidade de uma Gramática de Atributos, decidiu-se introduzir, na subsecção 4.2.2, um conjunto de métricas de forma (MFA3 a MFA5) que avaliam precisamente o esquema atributivo adoptado.
Para que seja possível medir essa característica, pretende-se neste apêndice definir um conjunto de padrões que descrevam os esquemas mais típicos das diferentes abordagens à escrita das regras de cálculo dos atributos, das regras de tradução, ou das condições de contexto.
Admite-se, porém, que muitos autores de Gramáticas de Atributos possam não adoptar nenhum destes padrões, ou não seguir uma abordagem sistemática ao longo da mesma gramática (na avaliação desses casos, serão classificados como `RC/RT/CC-var`).

Padrões para as Regras de Cálculo A ideia geral de uma Gramática de Atributos é especificar a forma de retirar de cada subfrase e respectivos terminais que a constituem a informação concreta que agregada, ou adicionada, com a informação das outras subfrases vai dar origem ao significado, ou conteúdo preciso, da frase, o qual será processado para produzir o resultado esperado.
Sobre a forma de especificar esse processamento discorre-se no parágrafo seguinte; no último parágrafo deste apêndice fala-se no estilo de escrita das restrições que terão de ser verificadas para que o dito conteúdo seja válido e possa ser processado. Neste parágrafo pretende-se apresentar os esquemas mais importantes para agregar, ou acumular, os valores parciais de modo a formar um todo (o valor semântico), embora esta tarefa seja ingrata visto que há muitas formas válidas de especificar a construção do significado da frase, com a agravante de serem dependentes do tipo escolhido para os atributos.
Em relação à agregação de valores associada a uma produção da forma:

```
P= { .....
     p1:  X0   -->  X1 ...Xi... Xn
     ..........
   }
```

e supondo que cada uma das partes contribui com o valor de um atributo sintetizado `a` para o valor semântico do todo (X0), há essencialmente duas alternativas que são tipificadas a seguir através do padrão 1 e do padrão 2.

Padrão Atributivo 1 (RCAgreg) *Neste padrão associa-se ao símbolo do lado esquerdo um único atributo estruturado* `s` *que vai aglutinar o valor de cada uma das partes, resultando na escrita de uma única regra de cálculo, como se exemplifica a seguir:*

```
p1:  X0   -->  X1 ...Xi... Xn
     RC(p1) = { X0.s = makeTuple( X1.a, ...., Xn.a ) }
```

Embora esta pareça uma forma muito natural de agregar os valores parciais num único valor composto, não é necessariamente a forma mais conveniente em termos de eficiência de cálculo. Por isso, surge a alternativa muito usada que se mostra a seguir.

Padrão Atributivo 2 (RCNAgreg) *Neste padrão associa-se ao símbolo do lado esquerdo um atributo* `si` *por cada valor que se quer guardar, sem nunca chegar a fazer-se a aglutinação dos vários valores de cada uma das partes. Como resultado desta estratégia serão agora necessárias* `n` *regras de cálculo do tipo de* regras de cópia*, como se exemplifica a seguir:*

```
p1:  X0   -->  X1 ...Xi... Xn
     RC(p1) = { X0.s1 = X1.a; ....;
                X0.sn = Xn.a  }
```

Discutem-se agora os 3 padrões mais importantes a usar com produções recursivas da forma

```
P= {
     .....
     p1:  Lista   -->  Lista Elem
     p2:           |   Elem
     p3:  Elem    -->  t
     ..........
    }
```

para construir um atributo complexo `s` (tipo sequência ou função-finita) através da acumulação, ou adição, do valor de cada elemento, dado pelo atributo `e`, ambos sintetizados. Supõe-se, nos esquemas que se seguem, que `add(x,y)` é uma função que junta o valor `x` à estrutura `y`, retornando uma nova estrutura do tipo de `y`. O esquema mais pobre, mas mais simples para os noviços na área, é o tipificado no padrão 3. É um esquema que induz um processo de cálculo simples e uma leitura fácil quando essa extracção e aglutinação não precisa de recorrer a informação contextual (das subfrases à esquerda ou à direita)[1].

Padrão Atributivo 3 (RCpureS) *Neste padrão só se usam atributos sintetizados (eventualmente poderá recorrer-se a variáveis globais) e as várias regras de cálculo, associadas a todas as produções envolvidas, vão sempre fazendo subir o valor da adição, como se exemplifica a seguir:*

```
p1:  Lista   -->  Lista Elem
     RC(p1) = { Lista0.s = add( Elem2.e, Lista1.s ) }
p2:           |   Elem
```

[1] Quando necessita de informação contextual, será necessário recorrer a variáveis globais tornando a abordagem manifestamente desaconselhável.

```
    RC(p2) = { Lista0.s = add( Elem1.e, null ) }
p3:  Elem    -->  t
    RC(p3) = { Elem0.e = fun( t1.a ) }
```

O esquema oposto a este é o que se tipifica no padrão 4.

Padrão Atributivo 4 (RCpureI) *Neste padrão, o mais comum em* Gramática de Atributos*, usam-se atributos herdados e sintetizados aos pares. A informação contextual desce (via os atributos herdados) até aos elementos e a adição do valor é realizada ao nível dessa produção elementar, sendo depois o resultado simplesmente sintetizado, como se exemplifica a seguir:*

```
p1:  Lista   -->  Lista Elem
    RC(p1) = { Lista1.in  = Lista0.in;
               Elem2.e_in = Lista1.s;
               Lista0.s   = Elem2.e}
p2:           |   Elem
    RC(p2) = { Elem1.e_in = Lista0.in;
               Lista0.s   = Elem1.e }
p3:  Elem    -->  t1
    RC(p3) = { Elem0.e = add( fun( t1.a ), Elem0.e_in ) }
```

Embora seja a forma mais tradicional de especificar o cálculo de atributos, por ser clara e sistemática, este último padrão, ao contrário do anterior, é muito mais consumidor de memória e de tempo de cálculo (exige muito mais operações nas visitas aos nodos aquando da travessia da árvore de sintaxe). Sugere-se, através do padrão 5, um esquema intermédio que muitos autores de GAs usam por se manter claro e sistemático e ser mais económico que o anterior.

Padrão Atributivo 5 (RCmixIS) *Neste padrão usa-se apenas um par de atributos herdado e sintetizado para a lista e a informação contextual desce (via os atributos herdados) só até às produções intermédias, procedendo-se à adição do valor a esse nível, como se exemplifica a seguir:*

```
p1:  Lista   -->  Lista Elem
    RC(p1) = { Lista1.in  = Lista0.in;
               Lista0.s   = add( Elem2.e, Lista1.s ) }
p2:           |   Elem
    RC(p2) = { Lista0.s   = add( Elem1.e, Lista0.in ) }
p3:  Elem    -->  t1
    RC(p3) = { Elem0.e = fun( t1.a ) }
```

Desta forma usam-se menos atributos e reduz-se ao número de cálculos a efectuar em cada visita aos nodos, mas mantém-se o recurso a atributos herdados para fazer descer na árvore a informação contextual, evitando o uso de variáveis globais conforme sugerido quando se adopta a abordagem descrita no padrão 3.

Padrões para as Regras de Tradução Quanto à escrita das regras de tradução, isto é, das operações atributivas que invocam procedimentos, levando atributos como parâmetros, para produzir o resultado final que se espera do processamento das frases da linguagem, o esquema mais usual consiste em associá-las à produção inicial (com o Axioma no LHS).
Esta visão, seguida até por muitos Geradores de Compiladores, corresponde à definição estrita de Gramática de Atributos segundo a qual o significado de uma frase é dado pelos atributos sintetizados do Axioma.
O padrão 6 tipifica este esquema.

Padrão Atributivo 6 (RTTop) *Neste padrão as várias regras de tradução estão associadas à produção inicial e manipulam os atributos do axioma, como se exemplifica a seguir (assumindo que há* ***n*** *regras designadas por* ***rt1*** *a* ***rtn****):*

```
p0:  S -->  X1 ...Xi... Xk
    RT(p0) = { rt1( S0.a ); ....; rtn( S0.b ) }
```

este esquema obriga a associar ao Axioma um atributo sintetizado por cada valor que se pretenda apresentar como resultado, sendo esses atributos calculados no contexto das várias produções ao longo da gramática.

Apesar de retardar a geração da saída (a apresentação dos resultados finais), e de requerer mais memória (para todos esses atributos sintetizados) bem como mais tempo de cálculo, este esquema é talvez o mais frequente pois além de arrumar bem todas estas regras num só ponto da gramática, permite manipular o resultado final como um todo, depois de completamente calculado. Esta abordagem é claramente vantajoso para tarefas como, por exemplo, a optimização de código realizada pelos compiladores. No caso tradicional dos compiladores é vulgar acumular os vários fragmentos de código, correspondentes às várias instruções que vão sendo reconhecidas, num atributo `codigo` que é sintetizado (por concatenação das parcelas) até ao Axioma. O exemplo 21, introduzido acima (ver capítulo 3) e sintetizado no apêndice F, ilustra bem esta ideia.
Uma abordagem intermédia, que pode ser utilizada, em alternativa ao esquema anterior, para diminuir o consumo de memória e o atraso na apresentação de resultados, e que de facto é seguida por muitos autores, é a descrita pelo padrão 7.

Padrão Atributivo 7 (RTInterm) *Neste padrão as regras de tradução vão sendo associadas às produções onde já se sintetiza um resultado parcial que pode ser apresentado. Supondo, a título de exemplo, que o atributo* `a` *é sintetizado para* ***X1*** *na produção* ***p1*** *e que o atributo* `b` *é sintetizado para* ***Xi*** *na produção* ***pi****, então as regras de tradução seriam colocadas da seguinte forma:*

```
p0:  S  -->  X1 ...Xi... Xk
p1:  X1 -->  .....
    RT(p1) = { rt1( X1.a ) }
pi:  Xi -->  .....
    RT(pi) = { rtn( Xi.b ) }
```

este esquema evita que se tenha de associar ao Axioma um atributo sintetizado por cada valor que se pretenda apresentar como resultado, mas por outro lado perde a visão global sobre todo o resultado final.

Um terceiro padrão, `RTBottom`, que também podia ser adoptado correspondia a levar ao extremo o esquema anterior (padrão 7) e distribuir as regras de tradução pelas produções terminais. Por não favorecer a clareza nem a optimização e consumir muito mais recurso que o Intermédio, este esquema não traz vantagens e é pouco usado na prática. Por isso não será aqui apresentado o respectivo padrão.

Padrões para as Condições de Contexto Suponha-se que numa determinada gramática existem várias alternativas para o símbolo `Operacao`

```
P= {
     .....
     p1:  Operacao -->  Oper1
     p2:            |   Oper2
     p3:            |   .....
```

```
p4:  Oper1    -->  OperandA OP OperandB
p5:  OperandA -->  t1
p6:  OperandB -->  t2
.....
}
```

e numa delas (`Oper1`, por exemplo) se pretende restringir o valor do atributo `a` de um `OperandA` ao valor do atributo `a` de outro `OperandB`. Admita-se que essa restrição é verificada pelo predicado binário `cc1`.
O esquema mais natural para escrever essa validação contextual é descrito pelo padrão 8.

Padrão Atributivo 8 (CCCentrad) *Neste padrão a condição de contexto é colocada na produção que ela pretende restringir:*

```
p4:  Oper1    -->  OperandA OP OperandB
    CC(p4) = { cc1( OperandA.a, OperandB.a ) }
```

onde tipicamente o atributo `a` *é sintetizado quando se reconhece o respectivo operando (neste caso concreto, seria nas produções* `p5` *e* `p6`*).*

Porém a mesma restrição poderia ser estabelecida, por outro autor, mais acima ao nível das produções que definem as alternativas para `Operacao`, como se tipifica no padrão 9.

Padrão Atributivo 9 (CCTop) *Neste padrão a condição de contexto é colocada numa produção mais para cima, mais perto do axioma da gramática:*

```
p1:  Operacao -->  Oper1
    CC(p1) = { cc1( Oper1.Esq, Oper1.Dir ) }
p4:  Oper1    -->  OperandA OP OperandB
    RC(p4) = { Oper1.Esq = OperandA.a;
               Oper1.Dir = OperandB.a
             }
```

o que obriga a acrescentar dois novos atributos sintetizados que serão calculados no contexto da produção `p4`*.*

Outra alternativa, adoptada por alguns autores, consiste em fazer as validações mais abaixo ao nível do símbolo que se pretende restringir, tal como ilustrado no padrão 10.

Padrão Atributivo 10 (CCBottom) *Neste padrão a condição de contexto é colocada numa produção mais abaixo, mais próxima das folhas:*

```
p4:  Oper1    -->  OperandA OP OperandB
    RC(p4) = { OperandA.aEsperado = OperandB.a
             }
p5:  OperandA -->  t1
    CC(p5) = { cc1( OperandA.a, OperandA.aEsperado ) }
```

o que obriga a acrescentar um novo atributo, agora herdado, que será calculado no contexto da produção `p4` *(continua-se a admitir que o atributo* `a` *de* `OperandA` *é sintetizado e calculado nesta produção* `p5`*).*

Apêndice E

Autómato LR(0) para a linguagem Lisp

Com o intuito de enriquecer a definição de autómato de reconhecimento LR, apresentada de passagem no apêndice C, e com isso ajudar a entender a discussão sobre as métricas de tamanho (do parser) introduzidas em 4.2.1, mostra-se neste apêndice um exemplo concreto de um $\mathcal{AD}$-LR previamente construído (o processo de construção, sistemático mas complexo, é irrelevante no presente contexto).
No exemplo segue-se a seguinte convenção:

```
Numero-do-Estado:
  KERNEL (formado por 1 ou mais Items-LR)
  ------------------
  FECHO  (formado por 0 ou mais Items-LR)
```

sendo Item-LR:

```
  [ LHS -> alfa "." beta ]
com
  LHS = X0
  alfa, beta: (N U T)*
```

O autómato determinista, $\mathcal{AD}$-LR, que guia um reconhecedor Bottom-Up LR(0) para a linguagem Lisp é formado por 10 estados constituídos pelos items LR, conforme se mostra a seguir:

```
Q0:
  [ Z -> . Lisp $ ]
  ------------------
  [ Lisp -> . SExp ]
  [ SExp -> . num ]
  [ SExp -> . pal ]
  [ SExp -> . ( SExplist ) ]
Q1:
  [ Z -> Lisp . $ ]    ac
Q2
  [ Lisp -> SExp . ]   #p0
Q3:
  [ SExp -> num .  ]   #p1
Q4:
  [ SExp -> pal .  ]   #p2
```

```
Q5:
  [ SExp -> ( . SExplist ) ]
  ------------------------------
  [ SExplist -> . SExp SExplist]
  [ SExplist -> . ]    #p5
  [ SExp -> . num ]
  [ SExp -> . pal ]
  [ SExp -> . ( SExplist ) ]
Q6:
  [ SExp -> ( SExplist . ) ]
Q7:
  [ SExp -> ( SExplist ) .  ]   #p3
Q8:
  [ SExplist -> SExp . SExplist]
  ------------------------------
  [ SExplist -> . ]    #p5
  [ SExp -> . num ]
  [ SExp -> . pal ]
  [ SExp -> . ( SExplist ) ]
Q9:
  [ SExplist -> SExp SExplist . ) ]   #p4
```

Aos estados apresentados acima, associaram-se as acções a fazer quando forem atingidos: aceitar a frase (`ac`); reduzir por uma dada produção p (#p).

A função de transição entre estados é descrita abaixo por uma lista dos tuplos no seguinte formato: `(estOrigem, simboloNouT, estFinal)`.

```
(Q0, Lisp, Q1)
(Q0, SExp, Q2)
(Q0, num,  Q3)
(Q0, pal,  Q4)
(Q0, "(",  Q5)

(Q5, SExplist, Q6)
(Q5, num, Q3)
(Q5, pal, Q4)
(Q5, "(", Q5)
(Q5, SExp, Q8)

(Q6, ")", Q7)

(Q8, SExplist, Q9)
(Q8, SExp, Q8)
(Q8, num, Q3)
(Q8, pal, Q4)
(Q8, "(", Q5)
```

Apêndice F

Gramática de Atributos para a linguagem Lisp

Para concluir o exemplo 21 (introduzido na sec. 3.1), mostra-se neste apêndice a Gramática de Atributos completa da linguagem Lisp, a qual resulta de fundir os fragmentos construidos nos 4 passos apresentados nessa secção.
O conjunto A dos atributos da linguagem é a união dos conjuntos de atributos herdados e sintetizados de cada símbolo $N \cup T$, a seguir discriminados.

```
AS(Lisp) = { contaN:int, contaP:int,
             lista:seq(pares(atomo,int)),
             codigo:seq(inst-maq) }
AI(SExp) = { contaN_in:int,  contaP_in:int,
             nivel_in:int, lista_in:seq(par(atomo,int)),
             tipo_in: enum(nil,num,pal),
             cod_in:seq(inst-maq)) }
AS(SExp) = { contaN_out:int, contaP_out:int ,
             lista_out:seq(par(atomo,int),
             tipo_out: enum(nil,num,pal)),
             cod_out:seq(inst-maq)) }
AI(SExplist) = { contaN_in:int,  contaP_in:int,
                 nivel_in:int, lista_in:seq(par(atomo,int)),
                 tipo_in: enum(nil,num,pal),
                 cod_in:seq(inst-maq)) }
AS(SExplist) = { contaN_out:int, contaP_out:int ,
                 lista_out:seq(par(atomo,int),
                 tipo_out: enum(nil,num,pal)),
                 cod_out:seq(inst-maq)) }
AS(num) = { val:atomo }
AS(pal) = { val:atomo }
```

As seguintes regras de cálculo completas obtém-se também juntando as RCs parciais por produção. O conjunto RC será a união de todos os subconjuntos indicados.

```
RC(p0) = { SExp_1.contaN_in=0;
           SExp_1.contaP_in=0;
           Lisp_0.contaN=SExp_1.contaN_out;
           Lisp_0.contaP=SExp_1.contaP_out;
           SExp_1.nivel_in=0;
           SExp_1.lista_in=NULL;
```

```
            Lisp_0.lista=SExp_1.lista_out;
            SExp_1.tipo_in=nil;
            SExp_1.cod_in=NULL;
            Lisp_0.codigo=SExp_1.cod_out }

RC(p1) = { SExp_0.contaN_out=SExp_0.contaN_in+1;
            SExp_0.contaP_out=SExp_0.contaP_in;
            SExp_0.lista_out=insere1(num_1.val,SExp_0.nivel_in, SExp_0.lista_in);
            SExp_0.tipo_out=(SExp_0.tipo_in == nil)?
                                        num :
                                        SExp_0.tipo_in;
            SExp_0.cod_out=insere2("CONS"++num_1.val, SExp_0.cod_in) }

RC(p2) = { SExp_0.contaN_out=SExp_0.contaN_in;
            SExp_0.contaP_out=SExp_0.contaP_in+1;
            SExp_0.lista_out=insere1(pal_1.val,SExp_0.nivel_in, SExp_0.lista_in);
            SExp_0.tipo_out=(SExp_0.tipo_in == nil)?
                                        pal :
                                        SExp_0.tipo_in;
            SExp_0.cod_out=insere2("LAB" ++pal_1.val, SExp_0.cod_in) }

RC(p3) = { SExp_0.contaN_out=SExplist_2.contaN_out;
            SExp_0.contaP_out=SExplist_2.contaP_out;
            SExplist_2.contaN_in=SExp_0.contaN_in;
            SExplist_2.contaP_in=SExp_0.contaP_in;
            SExplist_2.nivel_in=SExp_0.nivel_in+1;
            SExplist_2.lista_in=SExp_0.lista_in;
            SExp_0.lista_out=SExplist_2.lista_out;
            SExp_0.tipo_out=SExplist_2.tipo_out;
            SExplist_2.tipo_in=SExp_0.tipo_in;
            SExplist_2.cod_in=SExp_0.cod_in;
            SExp_0.cod_out=append(tail(SExplist_2.cod_out),conv_oper(head(SExplist_2.cod_out)) }

RC(p4) = { SExplist_0.contaN_out=SExplist_2.contaN_out;
            SExplist_0.contaP_out=SExplist_2.contaP_out;
            SExp_1.contaN_in=SExplist_0.contaN_in;
            SExp_1.contaP_in=SExplist_0.contaP_in;
            SExplist_2.contaN_in=SExp_1.contaN_out;
            SExplist_2.contaP_in=SExp_1.contaP_out;
            SExp_1.nivel_in=SExplist_0.nivel_in;
            SExplist_2.nivel_in=SExplist_0.nivel_in;
            SExp_1.lista_in=SExplist_0.lista_in;
            SExplist_2.lista_in=SExp_1.lista_out;
            SExplist_0.lista_out=SExplist_2.lista_out;
            SExplist_0.tipo_out=SExplist_2.tipo_out;
            SExp_1.tipo_in=SExplist_0.tipo_in;
            SExplist_2.tipo_in=SExp_1.tipo_out;
            SExp_1.cod_in=SExplist_0.cod_in;
            SExplist_2.cod_in=SExp_1.cod_out;
            SExplist_0.cod_out=SExplist_2.cod_out }

RC(p5) = { SExplist_0.contaN_out=SExplist_0.contaN_in; SExplist_0.contaP_out=SExplist_0.contaP_in;
            SExplist_0.lista_out=SExplist_0.lista_in;
            SExplist_0.tipo_out=SExplist_0.tipo_in;
```

```
        SExplist_0.cod_out=SExplist_0.cod_in }
```

De igual forma, o conjunto das condições de contexto, CC, obtem-se fazendo a união dos subconjuntos relativos a cada produção, que se listam a seguir:

```
CC(p1) = { se (SExp_0.tipo_in == pal) entao {erro(semantica,1)} }
CC(p2) = { se (SExp_0.tipo_in == num) entao {erro(semantica,2)} }
```

Também o conjunto RT de todas as regras de cálculo se obtém por união da regras de tradução de cada produção:

```
RT(p0) = { grava(Lisp_0.codigo); }
```

Apresentam-se a seguir os grafos de dependências locais (GDL) às 6 produções da gramática[1]. Para cada grafo mostram-se os pares de vértices `(origem, destino)` que pertencem à relação depende-de, devendo ler-se (destino depende-de origem).

```
GDL(p0)
          (SExp_1.contaN_out, Lisp_0.contaN);
          (SExp_1.contaP_out, Lisp_0.contaP);
          (SExp_1.lista_out,  Lisp_0.lista);
          (SExp_1.cod_out,    Lisp_0.codigo);
          (Lisp_0.codigo,     rt01(grava)).

GDL(p1)
          (SExp_0.contaN_in, SExp_0.contaN_out);
          (SExp_0.contaP_in, SExp_0.contaP_out);
          (num_1.val,        SExp_0.lista_out);
          (SExp_0.nivel_in,  SExp_0.lista_out);
          (SExp_0.lista_in,  SExp_0.lista_out);
          (SExp_0.tipo_in,   SExp_0.tipo_out);
          (num_1.val,        SExp_0.cod_out);
          (SExp_0.cod_in,    SExp_0.cod_out);
          (SExp_0.tipo_in,   cc11).

GDL(p2)
          (SExp_0.contaN_in, SExp_0.contaN_out);
          (SExp_0.contaP_in, SExp_0.contaP_out);
          (pal_1.val,        SExp_0.lista_out);
          (SExp_0.nivel_in,  SExp_0.lista_out);
          (SExp_0.lista_in,  SExp_0.lista_out);
          (SExp_0.tipo_in,   SExp_0.tipo_out);
          (pal_1.val,        SExp_0.cod_out);
          (SExp_0.cod_in,    SExp_0.cod_out).

GDL(p3)
          (SExplist_2.contaN_out, SExp_0.contaN_out);
          (SExplist_2.contaP_out, SExp_0.contaP_out);
          (SExplist_2.lista_out,  SExp_0.lista_out);
          (SExplist_2.tipo_out,   SExp_0.tipo_out);
          (SExplist_2.cod_out,    SExp_0.cod_out);
          (SExp_0.contaN_in,      SExplist_2.contaN_in);
          (SExp_0.contaP_in,      SExplist_2.contaP_in);
          (SExp_0.nivel_in,       SExplist_2.nivel_in);
          (SExp_0.lista_in,       SExplist_2.lista_in);
          (SExp_0.tipo_in,        Explist_2.tipo_in);
          (SExp_0.cod_in,         SExplist_2.cod_in).

RC(p4)
          (SExplist_2.contaN_out, SExplist_0.contaN_out);
          (SExplist_2.contaP_out, SExplist_0.contaP_out);
          (SExplist_0.contaN_in,  SExp_1.contaN_in);
          (SExplist_0.contaP_in,  SExp_1.contaP_in);
          (SExp_1.contaN_out,     SExplist_2.contaN_in);
          (SExp_1.contaP_out,     SExplist_2.contaP_in);
```

[1]Recorde-se que o Grafo de Dependência entre os Simbolos correspondente a esta gramática já foi mostrado atrás no âmbito do exemplo 20.

```
            (SExplist_0.nivel_in,   SExp_1.nivel_in);
            (SExplist_0.nivel_in,   SExplist_2.nivel_in);
            (SExplist_0.lista_in,   SExp_1.lista_in);
            (SExp_1.lista_out,      SExplist_2.lista_in);
            (SExplist_2.lista_out,  SExplist_0.lista_out);
            (SExplist_2.tipo_out,   SExplist_0.tipo_out);
            (SExplist_0.tipo_in,    SExp_1.tipo_in);
            (SExp_1.tipo_out,       SExplist_2.tipo_in);
            (SExplist_0.cod_in,     SExp_1.cod_in);
            (SExp_1.cod_out,        SExplist_2.cod_in);
            (SExplist_2.cod_out,    SExplist_0.cod_out).

GDL(p5)
            (SExplist_0.contaN_in,  SExplist_0.contaN_out);
            (SExplist_0.contaP_in,  SExplist_0.contaP_out);
            (SExplist_0.lista_in,   SExplist_0.lista_out);
            (SExplist_0.tipo_in,    SExplist_0.tipo_out);
            (SExplist_0.cod_in,     SExplist_0.cod_out).
```

Para medir, de acordo com as métricas apresentadas na secção 4.2.2, a complexidade semântica desta GA, a qual é determinada pelo grau de dependência entre os atributos que se estabelece nos seis GDLs acima, devem calcular-se os factores FanIn e FanOut. A tabela seguinte sintetiza esse cálculo:

Atributo	FanOut	FanIn
AS(Lisp)		
contaN	0	1
contaP	0	1
lista	0	1
codigo	1	1
AI(SExp)		
contaN_in	1	1
contaP_in	1	1
nivel_in	1	1
lista_in	1	1
tipo_in	2	1
cod_in	1	1
AS(SExp)		
contaN_out	1	1
contaP_out	1	1
lista_out	1	3
tipo_out	1	1
cod_out	1	2
AI(SExplist)		
contaN_in	1	1
contaP_in	1	1
nivel_in	1	1
lista_in	1	1
tipo_in	1	1
cod_in	1	1
AS(SExplist)		
contaN_out	1	1
contaP_out	1	1
lista_out	1	1
tipo_out	1	1
cod_out	1	1
AS(num)		
val	2	0
AS(pal)		
val	2	0

Apêndice G

Implementação da GA-Lisp em AnTLR

Neste apêndice mostra-se a GA introduzida no exemplo 21 (ver sec. 3.1) escrita na sintaxe do Gerador de Compiladores AnTLR.

A gramática abaixo corresponde ao 1º passo, cálculo da quantidade de números e palavras da lista.

```
grammar Lisp;

lisp returns [int contaN_out, int contaP_out]
@init{ int contaN_in = 0, contaP_in = 0; }
    :   sExp[contaN_in, contaP_in]      { $contaN_out = $sExp.contaN_out;
                                $contaP_out = $sExp.contaP_out;
                                System.out.println("Total de numeros: " + $contaN_out + "\n" +
                                           "Total de palavras: " + $contaP_out + "\n"); }
    ;

sExp[ int contaN_in, int contaP_in] returns [int contaN_out, int contaP_out]
    :   NUM         { $contaN_out = $contaN_in + 1;
                      $contaP_out = $contaP_in; }
    |   PAL         { $contaN_out = $contaN_in;
                      $contaP_out = $contaP_in + 1;
                      }
    |   '(' sExpList[contaN_in, contaP_in] ')'
                    { $contaN_out = $sExpList.contaN_out;
                      $contaP_out = $sExpList.contaP_out; }
    ;

sExpList[ int contaN_in, int contaP_in] returns [int contaN_out, int contaP_out]
@init{ int aux1, aux2; }
    :   a1=sExp[contaN_in, contaP_in] { aux1 = $a1.contaN_out; aux2 = $a1.contaP_out;    }
        a2=sExpList[aux1, aux2]
                    { $contaN_out = $a2.contaN_out;
                      $contaP_out = $a2.contaP_out; }
    |
                    { $contaN_out = $contaN_in;
                      $contaP_out = $contaP_in; }
    ;
```

```
NUM :   ('0'..'9')+
    ;

PAL :   ('a'..'z')+
    ;

WS  :   (' ' | '\t' | '\n' | '\r') { channel=HIDDEN; };
```

Uma vez testado o processador gerado pelo AnTLR com ambas as frases-exemplo, obtiveram-se os valores previstos.
A gramática abaixo corresponde ao 2º passo, planificação da lista associando a cada elemento o seu nível.

```
grammar Lisp2;

@header{
    import java.util.ArrayList;
}

@members{
    public class Pair<A, B> {
        public final A fst;
        public final B snd;

        public Pair(A fst, B snd) {
            this.fst = fst;
            this.snd = snd;
            }


        public <A,B> Pair<A,B> of(A a, B b) {
            return new Pair<A,B>(a,b);
            }

            public String toString() {
                return "(" + fst + "," + snd + ")";
        }

    }

    public ArrayList<Pair> insere(ArrayList<Pair> lista_in, String valor, int nivel)
    {
        Pair p = new Pair(valor,nivel);
        lista_in.add(p);
        return lista_in;
    }
}

lisp returns [ArrayList<Pair> lista_out]
```

```
@init{ ArrayList<Pair>  lista_in = new ArrayList<Pair>();
       int nivel_in = 0, nivel_out;
     }
    :   sExp[nivel_in, lista_in]        { $lista_out = $sExp.lista_out;
                                          System.out.println($lista_out);
                                        }
    ;

sExp[ int nivel_in, ArrayList<Pair> lista_in] returns  [ArrayList<Pair> lista_out]
    :   NUM         { $lista_out = insere($lista_in, $NUM.text, $nivel_in); }
    |   PAL         { $lista_out = insere($lista_in, $PAL.text, $nivel_in); }
    |   '(' sExpList[nivel_in+1, lista_in] ')'
                    { $lista_out = $sExpList.lista_out; }
    ;

sExpList[ int nivel_in, ArrayList<Pair> lista_in] returns  [ArrayList<Pair> lista_out]
@init{ ArrayList<Pair> aux; }
    :   a1=sExp[nivel_in, lista_in]     { aux = $a1.lista_out; }
        a2=sExpList[nivel_in, aux]
                    { $lista_out = $a2.lista_out; }
    |               { $lista_out = $lista_in; }
    ;

NUM :   ('0'..'9')+
    ;

PAL :   ('a'..'z')+
    ;

WS  :   (' ' | '\t' | '\n' | '\r') { channel=HIDDEN; };
```

Uma vez testado o processador gerado pelo AnTLR com ambas as frases-exemplo, obtiveram-se os valores previstos.

Bibliografia

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AV05] Tiago L. Alves and Joost Visser. Metrication of sdf grammars. Technical Report DI-Research.PURe-05.05.01, Departamento de Informática, Universidade do Minho, May 2005.

[AV07] Tiago L. Alves and Joost Visser. Sdfmetz: Extraction of metrics and graphs from syntax definitions. Tool Demonstration presented in Seventh Workshop on Language Descriptions, Tools and Applications (LDTA 2007), 2007.

[AV09] Tiago L. Alves and Joost Visser. A case study in grammar engineering. In Dragan Gasevic, Ralf Lämmel, and Eric van Wyk, editors, *Software Language Engineering, First International Conference, SLE 2008, Toulouse–France, September 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2009.

[Bac79] Roland Backhouse. *Syntax of Programming Languages: Theory and Practice*. Prentice Hall, 1979.

[BRS05] Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In *Formal Aspects in Security and Trust*, pages 112–126, 2005.

[CALO] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.

[CCF+09] Julien Cervelle, Matej Crepinsek, Rémi Forax, Tomaz Kosar, Marjan Mernik, and Gilles Roussel. On defining quality based grammar metrics. In *Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT 2009, – 2nd Workshop on Advances in Programming Languages (WAPL'2009)*, pages 651–658, Mragowo, Poland, October 2009. IEEE Computer Society Press.

[Cho53] Noami Chomsky. Systems of syntactic analysis. *The Journal of Symbolic Logic*, 18(3):242–256, Sep. 1953.

[Cho55a] Noami Chomsky. Logical syntax and semantics, their linguistic relevance. *Language*, 31(1):36–45, Jan-Mar 1955.

[Cho55b] Noami Chomsky. *Transformational Analysis*. PhD thesis, University of Pennsylvania, 1955.

[Cho56] Noami Chomsky. Three models for the description of languages. *IRE Transactions on Information Theory*, IT-2(3):113–124, Sep. 1956.

[Cho57] Noami Chomsky. *Syntactic Structures*. Mouton, London, 1957.

[Cho59a] Noami Chomsky. On certain formal properties of grammars. *Information and Control*, 2:136–167, Jun. 1959. Reprinted in Readings in Mathematical Psychology 2, edited by Luce, Bush, Galanter, 125-55. New York, Wiley and Sons, 1965.

[Cho59b] Noami Chomsky. The transformational basis of syntax. Paper presented at IVth University of Texas Symposium on Syntax, Jun. 1959.

[Cho62] Noami Chomsky. Context-free grammars and pushdown storage. RLE Quarterly Progress Report 65, MIT, Apr. 1962.

[Cho64] Noami Chomsky. *Current Issues in Linguistic Theory.* Mouton, The Hague, 1964.

[Cho65] Noami Chomsky. *Aspects of the Theory of Syntax.* MIT-Press, Cambridge, 1965.

[CKM+10] Matej Crepinsek, Tomaz Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, and Gilles Roussel. On automata and language based grammar metrics. *ComSIS – Computer Science an Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications*, 7(Issue 2):309 – 329, April 2010.

[CLN00] Christopher Colby, Peter Lee, and George C. Necula. A proof-carrying code architecture for java. In *Computer Aided Verification*, pages 557–560, 2000.

[Cre98] Rui Gustavo Crespo. *Processadores de Linguagens: da concepção à implementação.* IST Press, 1998.

[CS69] C. Christensen and C.J. Shaw, editors. *Proceedings of the Extensible Languages Symposium.* ACM, Aug. 1969. SIGPLAN Notices 4 no. 8.

[Cur81] B. Curtis. Chapter 12 – the measurement of software quality and complexity. In Alan J. Perlis, Frederick Sayward, and Mary Shaw, editors, *Software Metrics: An Analysis and Evaluation.* MIT-Press, May 1981.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Main results, existing systems and bibliography. In *LNCS 341.* Springer-Verlag, 1988.

[Fen91] Norman E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman & Hall, Ltd., London, UK, UK, 1991.

[Fin96] Ronald B. Finkbine. Metrics and models in software quality engineering. *SIGSOFT Softw. Eng. Notes*, 21(1):89, 1996.

[FN00] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*, pages 357–370, New York, NY, USA, 2000. ACM.

[GMdSP06] João Gomes, Daniel Martins, Simão Melo de Sousa, and Jorge Sousa Pinto. LISSOM, a source level proof carrying code platform. In *International Workshop on Proof Carrying Code Platform, (PCC2006) — Seattle, USA*, 2006.

[Hen92] Pedro Rangel Henriques. *Atributos e Modularidade na Especificação de Linguagens Formais.* PhD thesis, Universidade do Minho, Dec. 1992.

[HMU06a] John E. Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 5 – Context-Free Grammars and Languages. Addison-Wesley, 3rd ed. edition, 2006.

[HMU06b] John E. Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd ed. edition, 2006.

[Hoa73] C. A. R. Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, Stanford, CA, USA, 1973.

[How95] James Howatt. A project-based approach to Programming Language Evaluation. *ACM SIGPLAN Notices*, 30(7), 1995.

[KLBM08] Tomaz Kosar, Pablo Martínez Lopez, Pablo Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific languages. *IST – Information and Software Technology*, 50(5):390–405, 2008.

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec. 1976.

[Mcl60] M. Douglas Mcllroy. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3(4):214–220, Apr. 1960.

[Mey92] Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[Nec98] George C. Necula. *Compiling with Proofs.* PhD thesis, School of Computer Science, Carnegie Mellon University, Sep. 1998.

[NL96] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, CMU, Nov. 1996. (62 pages).

[Per10] Maria João Varanda Pereira. Paradigmas da programação. Texto pedagógico (mestrado em sistemas de informação), IPB–Instituto Politécnico de Bragança, Escola de Tecnologia e Gestão, Campus de Santa Apolónia, Bragança, Feb. 2010.

[PKEJW06] Kai Pan, Sunghun Kim, and Jr. E. James Whitehead. Bug classification using program slicing metrics. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42, Washington, DC, USA, 2006. IEEE Computer Society.

[PM00] James F. Power and Brian A. Malloy. Metric-based analysis of context-free grammars. In *IWPC'00, Proceedings of the 8th International Workshop on Program Comprehension*, page 171, Washington, DC, USA, 2000. IEEE Computer Society.

[PM04] James F. Power and Brian A. Malloy. A metrics suite for grammar-based software: Research articles. *J. Softw. Maint. Evol.*, 16(6):405–426, 2004.

[Sch71] S.A. Schuman, editor. *Proceedings of the International Symposium on Extensible Languages*. ACM, Dec. 1971. SIGPLAN Notices 6 no. 12.

[Sco09] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, third ed. edition, March 2009.

[Seb09] Robert Sebesta. *Concepts of Programming Languages*. Pearson, Addison-Wesley, nineth ed. edition, 2009.

[Sil06] Miguel Oliveira Silva. *Metodologias e mecanismos para Linguagens de Programação Concorrente Orientadas por Objectos*. PhD thesis, Universidade de Aveiro, 2006.

[SMH05] T. Sloane, M. Mernik, and J. Heering. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, Dec. 2005.

[Sta75] Thomas A. Standish. Extensibility in programming language design. *SIGPLAN Notices*, 10(7):18–21, Jul 1975.

[TB07] Allen Tucker and Robert Boonan. *Programming Language: Principles and Paradigms*. McGraw-Hill, nineth ed. edition, 2007.

[vDK98] A. van Deursen and Paul Klint. Little languages: Little maintenace? *Journal of Software Maintenance*, 10:75–92, 1998.

[Wat04] David Watt. *Programming Language Design Concepts*. John Wiley, 2004.

[WG84] William Waite and Gerhard Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer-Verlag, 1984.

[Wil05] Gregory V. Wilson. Extensible programming for the 21st century. *ACM Queue 2*, 9, Jan 2005.

[WM96] Arthur H. Watson and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical Report NIST Special Publication 500-235, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-0001, Aug. 1996.