Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Temple - Uma linguagem de Programação para o Ensino de Programação

José Miguel Mestre Pimenta

Orientador(es) | Miguel Barão

Vasco Fernando de Figueiredo Tavares Pedro

Évora 2019

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Temple - Uma linguagem de Programação para o Ensino de Programação

José Miguel Mestre Pimenta

Orientador(es) | Miguel Barão

Vasco Fernando de Figueiredo Tavares Pedro

Évora 2019

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

- Presidente | Paulo Quaresma (Universidade de Évora)
- Vogal | Artur Miguel Andrade Vieira Dias (Universidade Nova de Lisboa - Fundação da Faculdade de Ciências e Tecnologias)
- Vogal-orientador | Vasco Fernando de Figueiredo Tavares Pedro (Universidade de Évora)

Évora 2019

A quem ambiciona conhecimento.

Agradecimentos

A realização desta dissertação não teria sido possível sem a ajuda de todos os que contribuíram para o seu desenvolvimento.

Começo por agradecer aos meus orientadores, os professores Vasco Pedro e Miguel Barão, pelo tempo disponibilizado, pela paciência, por todo o conhecimento que me transmitiram ao longo das inúmeras reuniões, sugestões e comentários. Agradeço também ao professor Vasco Pedro pelas inúmeras revisões das várias versões que a dissertação teve durante a sua evolução. Sem a ajuda de ambos, o trabalho desenvolvido não seria possível.

Gostaria de agradecer aos colegas do meu curso que ajudaram com as suas opiniões e sugestões para o desenvolvimento da linguagem de programação da tese. Agradeço também aos meus colegas de mestrado, que embarcaram comigo neste desafio e com quem aprendi bastante.

Por fim, queria agradecer aos meus pais, à minha namorada Maria e à minha família e pessoas próximas pela paciência demonstrada ao longo deste desafio, pelo tempo que abdicaram da minha presença em prol do desenvolvimento desta tese, e pelos incentivos que me foram dando.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Glossário

bug (software bug) Erro ou falha num programa de computador ou sistema que faz com que ele produza um resultado incorreto ou inesperado

built-in Parte integrante de uma estrutura

IDE (Integrated Development Environment) Ambiente de Desenvolvimento Integrado, é um programa que facilita o desenvolvimento de software para o programador

markup Sistema de anotação de um texto de modo sintaticamente distinguível de texto

open source Modelo de desenvolvimento descentralizado de Software sem licença comercial, que promove modelo cooperativo intelectual

overflow Quando o resultado de uma operação aritmética é um valor fora do limite de valores representáveis

ranking Classificação ordenada de acordo com critérios determinados

scripting Linguagem de programação que automatiza a execução de tarefas; linguagem de programação interpretada

spaghetti code Código mal estruturado

statement Instrução de uma linguagem de programação

UML (Unified Modeling Language) Linguagem de modelação para elaboração da estrutura de projetos de software

Sumário

A disciplina introdutória de programação é muitas vezes o primeiro contacto que os alunos de engenharia informática têm com a programação. Muitos alunos demonstram dificuldades nessa primeira disciplina e a linguagem de programação utilizada pode influenciar essas dificuldades. Nesta dissertação pretende-se averiguar o estado atual das disciplinas introdutórias de programação para se poder criar uma linguagem de programação adequada aos problemas existentes. Foi realizado um estudo para verificar a situação atual a nível mundial e de Portugal, observando quais as linguagens de programação utilizadas na disciplina introdutória de programação, considerando os cuidados a ter no ensino da programação, as dificuldades manifestadas pelos alunos e professores e ainda qual o paradigma mais adequado para ser utilizado nesta primeira abordagem à programação. Com base na informação adquirida foi criada uma linguagem que pretende colmatar alguns dos problemas descobertos e ser uma alternativa às linguagens de programação existentes no ensino de programação.

Palavras chave: Linguagem de programação, Programação, Ensino de programação, Interpretadores, Compiladores

Abstract

Temple - A Programming Language for Teaching Programming

The introductory programming course is often the first contact that computer engineering students have with programming. Many students demonstrate difficulties in this first discipline and the programming language used can influence these difficulties. In this dissertation we intend to investigate the current state of the introductory programming courses in order to create a new programming language appropriate to the existing problems. A study was carried out to verify the current situation worldwide and in Portugal. We took a look at which programming languages are used in the introductory programming courses, the problems that we should pay attention to, the difficulties manifested by students and teachers and the most appropriate paradigm to be used in this first approach to programming. Based on the information acquired, a language was created that aims at solving some of the problems discovered and to be an alternative to the existing programming languages to teach programming.

Keywords: Programming language, Teaching programming, Programming, Interpreters, Compilers

1

Introdução

1.1 Motivação

Com a evolução tecnológica atual, muitos alunos candidatam-se para o ensino superior e entram em cursos relacionados com a área da tecnologia e um desses cursos é a engenharia informática. Nestes cursos, as disciplinas de programação são de grande importância, em particular no 1º ano, onde se dão os primeiros passos. No entanto, nem todos os cursos funcionam do mesmo modo em todas as universidades, pois nem todas as universidades utilizam a mesma linguagem de programação na primeira disciplina.

A primeira disciplina de programação pretende dotar os alunos com as bases fundamentais para que estes fiquem preparados para resolverem problemas de média dificuldade e que saibam adaptar os conhecimentos e conceitos a um leque de linguagens de programação que não a lecionada nesta disciplina. A linguagem escolhida e o seu paradigma poderão não ser os mais adequados nesta fase de aprendizagem para que no futuro o aluno tenha uma boa adaptação a outro paradigma ou facilidade na aprendizagem de outras linguagens e, assim, a escolha da linguagem é de grande importância. A linguagem escolhida deve ainda permitir construções básicas para algoritmos, que não dê demasiada liberdade aos alunos e que permita um bom diagnóstico de erros.

Deste modo, a grande motivação para esta dissertação é criar uma linguagem de programação orientada ao ensino de programação para dar uma alternativa às linguagens utilizadas atualmente no início dos estudos de programação, uma linguagem de programação que prepare melhor os alunos para o futuro académico e profissional.

A razão para se criar uma nova linguagem é a não existência de uma linguagem de programação perfeita para o ensino, apesar de existirem algumas linguagens criadas com este objetivo. Qualquer uma das linguagens que são utilizadas atualmente tem problemas, sejam eles de sintaxe ou mais técnicos, e muitas das linguagens são feitas para utilização na indústria por pessoas experientes.

1.2 Objetivos

O objetivo desta dissertação é a criação de uma linguagem de programação específica para o ensino de programação, que seja adaptada aos métodos de ensino atuais, que seja preparada para uma boa utilização pelos alunos e que toque nos pontos essenciais do que é programar, dando aos alunos os conhecimentos adequados e necessários para que cumpram o currículo do primeiro ciclo de ensino da melhor forma.

Pretende-se que a linguagem criada colmate problemas existentes nas linguagens utilizadas na disciplina introdutória de programação, fazendo com que o aluno aprenda mais facilmente e possivelmente os resultados se mostrem em melhores classificações na disciplina.

O objetivo final é que a linguagem possa ser utilizada no primeiro ano de ensino de programação.

1.3 Contribuições

Com esta dissertação obteve-se uma visão global de como se encontra o ensino de programação no ensino superior, com foco na primeira disciplina de programação e na linguagem de programação utilizada. Observaram-se alguns dos problemas que se encontram nessa disciplina segundo alguns autores e o modo como se podem resolver.

A dissertação contribuiu ainda com uma linguagem de programação, a linguagem Temple, que pode ser uma alternativa às linguagens utilizadas atualmente e que tenta resolver alguns dos erros mais graves encontrados.

1.4 Plano da tese

O plano geral da tese inclui um estado da arte no Capítulo 2 onde é observado um pouco do passado histórico da programação no ensino, são verificadas quais as linguagens de programação utilizadas atualmente na primeira disciplina de programação, quais os paradigmas mais adequados, comparações entre as linguagens utilizadas e ainda considerações a ter na escolha ou criação de uma linguagem de programação, com base em problemas encontrados em algumas linguagens e dificuldades dos alunos.

No Capítulo 3 encontra-se o desenvolvimento do trabalho, onde primeiramente são discutidas as escolhas e decisões tomadas durante o desenvolvimento da linguagem, seguido por uma secção onde a linguagem é definida formalmente. No Capítulo 3 são ainda incluídos alguns detalhes acerca da implementação que levantaram mais questões ou foram de maior dificuldade, uma secção que compara a eficiência da linguagem criada com outras linguagens e ainda alguns exemplos de código da linguagem criada.

O Capítulo 4 apresenta as conclusões da dissertação e o trabalho que ficou para o futuro.

Existem ainda alguns anexos, entre eles as listas das universidades utilizadas no estudo no Capítulo 2 e a gramática da linguagem criada, Temple.

2

Estado da arte

Neste capítulo, é feita uma panorâmica do estado atual do ensino da programação e são debatidos vários temas repartidos por várias secções. A primeira secção ilustra um pouco da história da programação no ensino e como se chegou aos dias de hoje. Na segunda secção, através de um estudo, são observadas quais as linguagens de programação utilizadas atualmente e quais os possíveis motivos para a sua utilização. Um dos motivos para a utilização das linguagens é a utilização da linguagem na indústria, que é observado na terceira secção. Na quarta e quinta secções, são avaliadas boas características e coisas a evitar em linguagens de programação, assim como dificuldades que se observam nos alunos e bons métodos para o ensino da programação. As últimas secções são mais técnicas e tratam a escolha da linguagem com base no paradigma de programação, a comparação da sintaxe das linguagens mais utilizadas e problemas encontrados em algumas dessas linguagens.

2.1 História da programação no ensino

Ao longo do tempo, a escolha da linguagem de programação para ensinar os alunos foi-se alterando à medida que novas técnicas e metodologias foram implementadas e desenvolvidas. Rocky Ross [Ros00] refere como

aprendeu Fortran IV (lançado em 1962), e como foi a primeira linguagem que ensinou como professor assistente. Ross explica o caso do `if-statement` em Fortran IV e como o resultado da sua utilização em programas era chamado de *spaghetti code*, devido à necessidade da utilização da instrução `GO TO`. Posteriormente, com o desenvolvimento do Algol 60 e Algol 68, passa a ser possível escrever instruções `if-then-else` de uma forma mais estruturada, gerando influências visíveis em linguagens utilizadas atualmente. O desenvolvimento do Algol e os passos dados com a criação do mesmo geraram o crescimento do interesse pelo ensino de linguagens de programação nas universidades. Ross explica que o interesse era providenciar os alunos com as bases de programação e não ensinar-lhes conceitos mais técnicos de linguagens utilizadas comercialmente. Foi nesta altura que foram criados compiladores com base em linguagens utilizadas, mas com inovações sobre o que era uma boa sintaxe, e com uma maior deteção de erros sintáticos de modo a ajudar os alunos. Exemplos dados por Ross, que tiveram uma grande utilização, foram as linguagens Watfor e Watfiv (criadas em 1965 e 1968 respetivamente), criadas na Universidade de Waterloo, baseadas em Fortran IV, que introduziam estruturas parecidas com as do Algol.

A IBM, criadora de Fortran, criou na altura a linguagem PL/1 (ano de 1964) com influências do Algol, e a Universidade de Cornell criou um compilador PL/C (no ano de 1973) como um compilador para ser utilizado na universidade e tornar a linguagem acessível aos alunos. Foi nesta altura que linguagens como o Pascal (lançado em 1970) apareceram, uma linguagem conhecida como uma linguagem com muitas características boas para o ensino de programação. Andrew Black, Kim B. Bruce e James Noble [BBN10] referem que a escolha generalizada de Pascal beneficiou bastante os alunos e professores, que facilmente podiam transferir os conhecimentos adquiridos na linguagem entre instituições e utilizarem livros de diferentes autores. A escolha de Pascal beneficiou também os empregadores que sabiam que os alunos partilhavam conhecimentos na mesma linguagem de programação. Com a passagem do tempo, o Pascal começou a mostrar a sua idade e tornou-se popular a ideia de que uma linguagem de programação no ensino tem de ser uma linguagem popular e que os alunos utilizem no mercado de trabalho (tema debatido mais à frente). Sem uma linguagem que tenha aparecido para tomar o lugar de Pascal, chegámos aos tempos mais recentes, em que várias linguagens são utilizadas, dependendo dos estabelecimentos de ensino.

A influência de linguagens de programação orientadas a objetos e o aparecimento de linguagens como Smalltalk (1972), C++ (1985) e Java (1995), fragmentaram a comunidade educacional. Python (1990), pela sua simplicidade, foi também uma linguagem adotada, mas a falta de declaração de tipos e falta de disciplina de tipos é considerada um dos motivos para a preocupação de professores na utilização desta linguagem como introdução. A utilização de programação orientada a objetos tem tido uma influência bastante grande na escolha da primeira linguagem (tema debatido mais à frente), e George McMaster e Michael Zastre [MZ11] observaram que muitas universidades no Canadá optaram por introduzir o Pascal como linguagem de programação procedimental na primeira disciplina de programação, e só então Java como linguagem de programação orientada a objetos na segunda disciplina de programação, algo que se verifica atualmente na Universidade de Évora, com uma divisão semelhante, apesar de Pascal ser substituído por C.

2.2 Linguagens de programação utilizadas na disciplina introdutória

De modo a ter-se uma noção de como se encontra a disciplina introdutória de programação foi necessário realizar um estudo sobre quais as linguagens de programação mais proeminentes. Como tal, foi feita uma lista de universidades mundiais de renome e com base em *rankings*, tanto a nível geral, como da área de engenharia informática. A pesquisa feita resultou na listagem de dados no Anexo A. Foi estudado também o caso de Portugal, listando as universidades portuguesas públicas e quatro institutos politécnicos de maior renome no Anexo B. Este estudo pretendeu averiguar quais as linguagens utilizadas, a variedade de linguagens e qual a importância dada a determinado tipo de paradigmas. O estudo foi realizado no final do ano de 2017.

2.2.1 A primeira linguagem de programação a nível mundial

Para se verificar o estado atual da primeira disciplina de programação no primeiro ciclo de engenharia informática pelo mundo, foi feito um estudo com universidades do continente Europeu, América do Norte e Sul e da Oceânia, num total de dezasseis países. Os países foram escolhidos com base na importância da universidade e *rankings* da especialidade, pelo que apenas se escolheram países de cultura ocidental. Os países escolhidos foram os seguintes:

- Américas - Brasil, Estados Unidos da América, Canadá;
- Europa - Reino Unido, Alemanha, Espanha, Itália, França, Suíça, Bélgica, Holanda, Suécia, Dinamarca, Finlândia, Áustria;
- Oceânia - Austrália.

Após terem sido selecionados os países, o estudo teve como objetivo apurar quais as linguagens dadas na disciplina introdutória de programação em cursos da área da engenharia informática. Verificou-se que havia bastante variedade nos cursos. Determinados cursos têm um currículo em que o aluno pode escolher as disciplinas que pretende frequentar, e assim, o leque de disciplinas introdutórias era amplo, podendo o aluno escolher de entre várias linguagens para a sua primeira disciplina de programação. Em outros casos, a universidade dispõe de vários cursos na área, pelo que o foco geral é o mesmo, mas contém uma especialização no final do curso diferente. Existe depois o caso de cursos em que o aluno tem uma base comum e tem de escolher o ramo da informática que pretende estudar nos anos seguintes. Finalmente, há o caso de cursos nos quais a disciplina introdutória ensina ao aluno várias linguagens de programação ou dispõe de uma linguagem de programação própria de modo a ensinar as metodologias de programação sem ensinar necessariamente uma linguagem de programação específica. Com base na informação encontrada e análise dos dados, tal como se observa na Figura 2.1, as linguagens de programação mais utilizadas nas disciplinas introdutórias de programação são, por ordem descendente de utilização, Java, Python, C, C++ e Haskell, perfazendo um total de aproximadamente 91% (não incluindo universidades portuguesas nesta parte do estudo, que serão analisadas mais à frente).

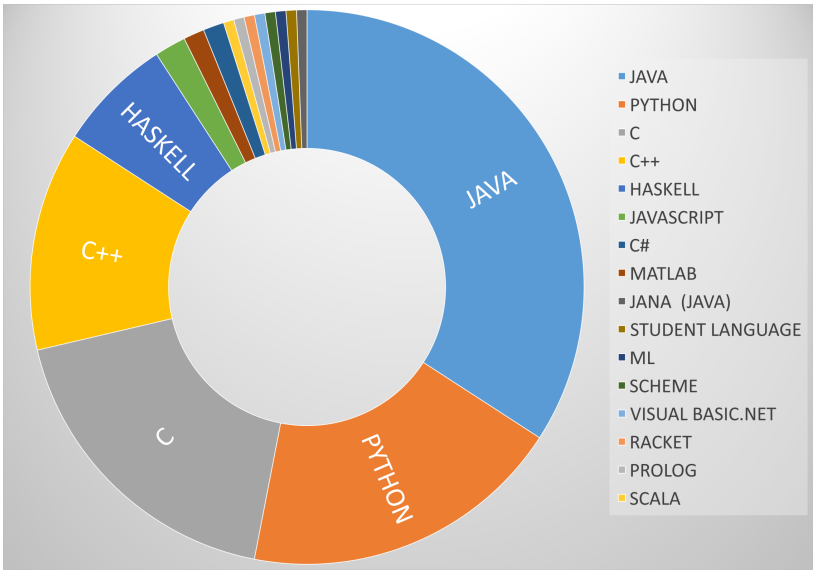

Figura 2.1: Utilização de linguagens de programação no mundo

Após a pesquisa realizada para tentar descobrir quais as linguagens de programação utilizadas nas primeiras disciplinas de programação em cursos de engenharia informática, foram encontradas duas linguagens que foram criadas com o mesmo propósito que este trabalho. Essas linguagens são JANA, uma linguagem feita com base em Java, e Student Language, uma linguagem funcional. Estes dados permitiram ainda verificar que a percentagem acumulada das linguagens demonstra que as linguagens mais utilizadas compõem, como referido, aproximadamente 91% da amostra. A amostra demonstra assim que o leque de linguagens utilizadas não é muito grande. Esta análise pode ser observada na Figura 2.2.

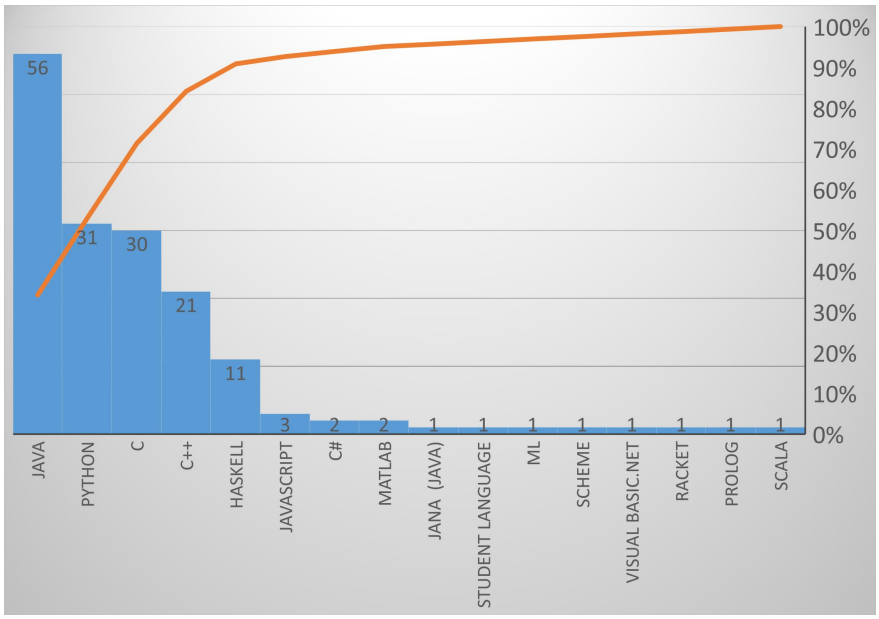

Figura 2.2: Percentagem acumulada e quantidade das linguagens de programação no mundo

2.2.2 A primeira linguagem de programação em Portugal

Em Portugal, a situação referente à utilização da linguagem de programação na disciplina introdutória de programação não foge muito à situação mundial. Para analisar a situação em Portugal, foram utilizados dados das universidades públicas portuguesas e ainda de quatro politécnicos de boa referência, encontrando-se no Anexo B mais informação acerca das universidades e cursos considerados. Em termos de análise, verificou-se que as linguagens utilizadas em Portugal são sobretudo Java, Python e C, tal como se verifica na Figura 2.3. As três linguagens mais utilizadas nos estabelecimentos de ensino selecionados perfazem aproximadamente 90% das linguagens utilizadas. De referir ainda que uma das linguagens utilizadas, para além das três referidas, é o Haskell, com uma percentagem considerável nos dados obtidos a nível mundial.

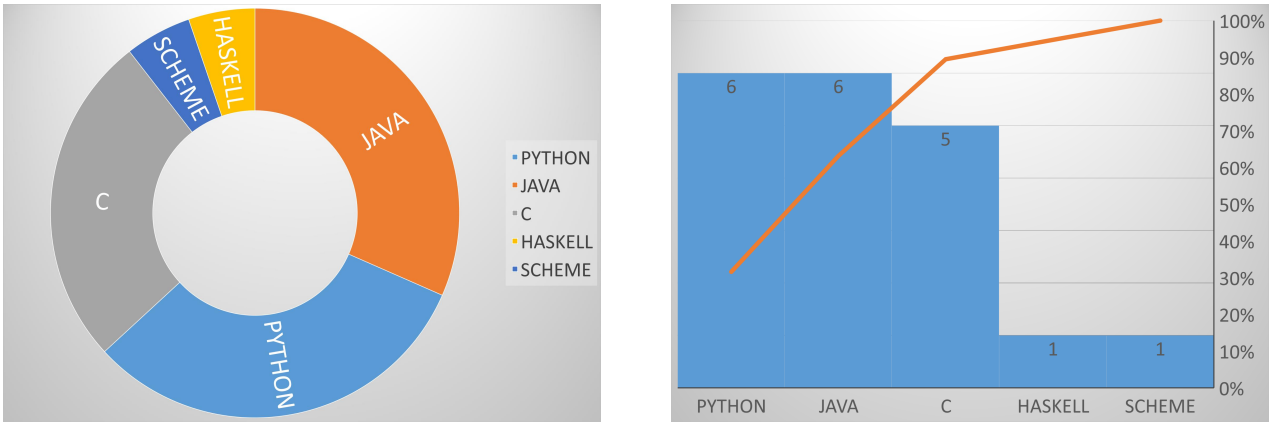

Figura 2.3: Utilização de linguagens de programação em Portugal

2.2.3 Paradigmas das linguagens de programação utilizadas

Foi criado um gráfico (Figura 2.4) com base nas linguagens mais utilizadas a nível mundial e de Portugal, que ilustra a percentagem das cinco linguagens mais utilizadas.

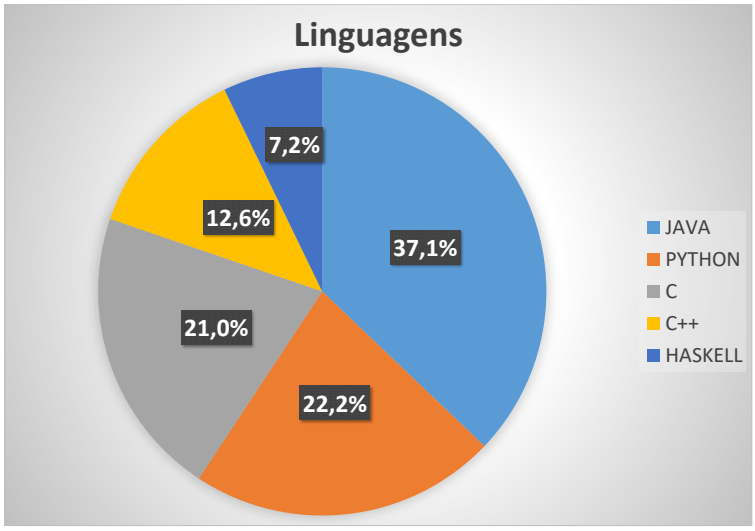

Figura 2.4: Linguagens de programação mais utilizadas

Estes dados serviram depois para determinar a proporção de cada tipo de paradigma de programação e se a linguagem contém ou não objetos. Os gráficos obtidos encontram-se na Figura 2.5.

Com os resultados identificados na Figura 2.5, verificamos que em termos de paradigma as linguagens mais utilizadas tendem a ser as linguagens imperativas, fazendo 92,8% das linguagens utilizadas, enquanto que os restantes 7,2% pertencem ao Haskell, a única linguagem funcional identificada com um grau de utilização mundial muito alto. De referir ainda que muitas das linguagens são flexíveis e permitem a utilização do paradigma funcional, não sendo inteiramente funcionais e enquadram-se mais no aspeto imperativo. Em termos de objetos, 71,9% das linguagens analisadas utilizam objetos, entre elas Java, Python e C++, sendo Java a única linguagem que obriga à utilização de objetos. Dos restantes 28,2%, as linguagens que não utilizam objetos são

C e Haskell. Pode-se assim concluir que nos cursos considerados existe um grande predomínio na utilização de uma linguagem imperativa e que esta contenha objetos.

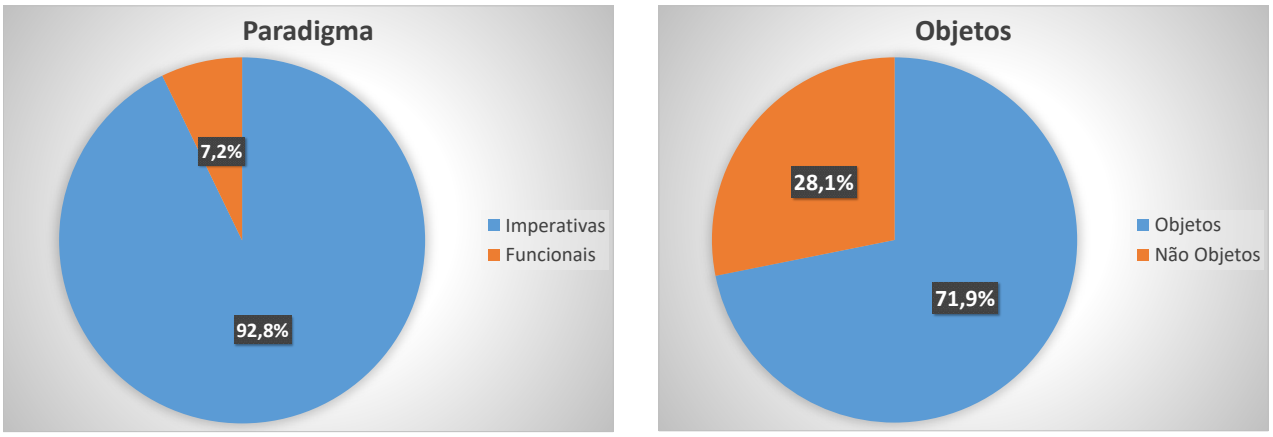

Figura 2.5: Panorama das linguagens mais utilizadas em Portugal e no mundo

2.2.4 O caso da Austrália

Foi realizado um estudo, na Austrália, por Michael de Raadt, Richard Watson e Mark Toleman [dRWT03] em 2001, idêntico em parte ao estudo anterior, com base em 57 cursos de 37 universidades Australianas.

Foi verificado que as linguagens mais utilizadas (contando a percentagem de alunos por linguagem) foram Java com 43,9% dos alunos, Visual Basic com 18,9%, C++ com 15,2%, Haskell com 8,8%, C com 5,5% e Eiffel com 3,3%, como se verifica na Figura 2.6.

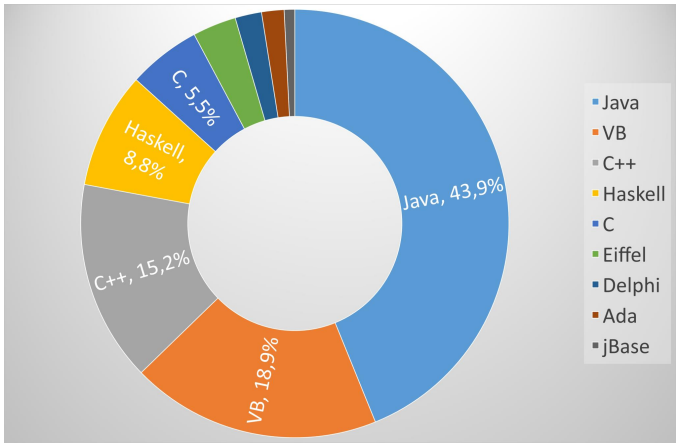

Figura 2.6: Linguagens de programação ensinadas na Austrália por percentagem de alunos

Foi perguntado aos professores como abordavam o paradigma de programação no ensino da linguagem. Os valores a que se chegaram foram os da Figura 2.7, com 51% a seguirem o paradigma procedimental, 40% a introduzir os objetos de início e apenas 9% a utilizarem o paradigma funcional. O interessante nestes valores é

que aproximadamente metade dos alunos foram ensinados com o paradigma procedimental, apesar de 81% das linguagens serem orientadas a objetos. Os dados das Figuras 2.8 e 2.9 ilustram como Java foi ensinado em maioria por um paradigma orientado a objetos, enquanto Visual Basic e C++ foram introduzidos através do paradigma procedimental.

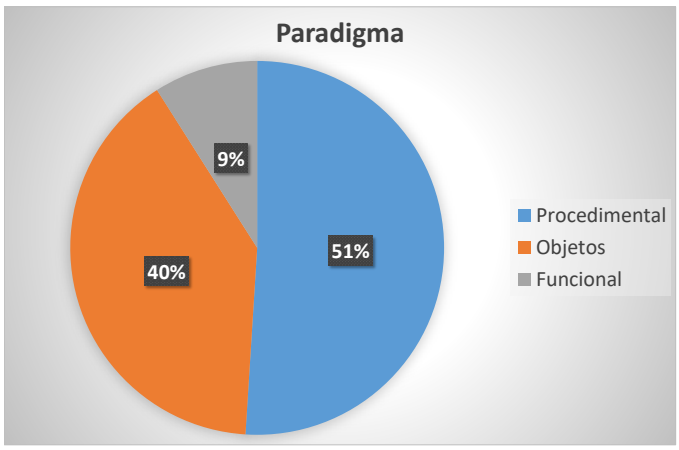

Figura 2.7: Tipo de paradigma utilizado

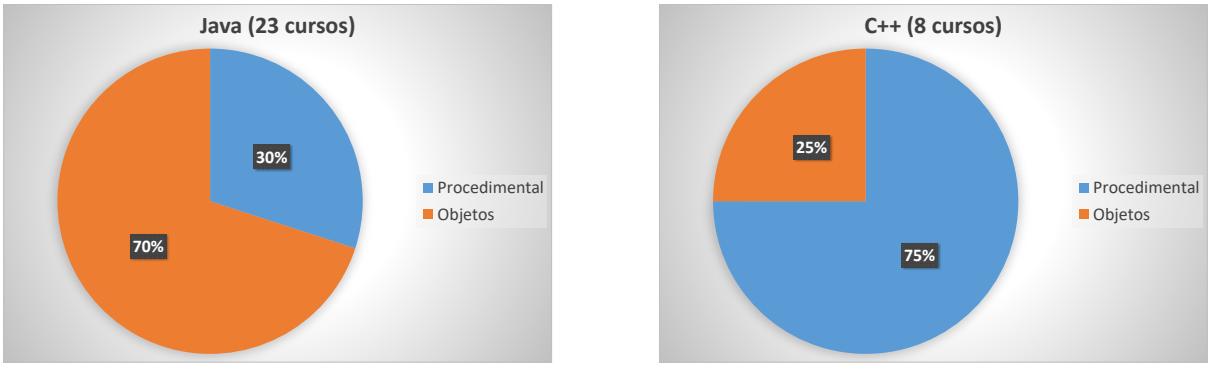

Figura 2.8: Tipo de paradigma utilizado em Java e C++

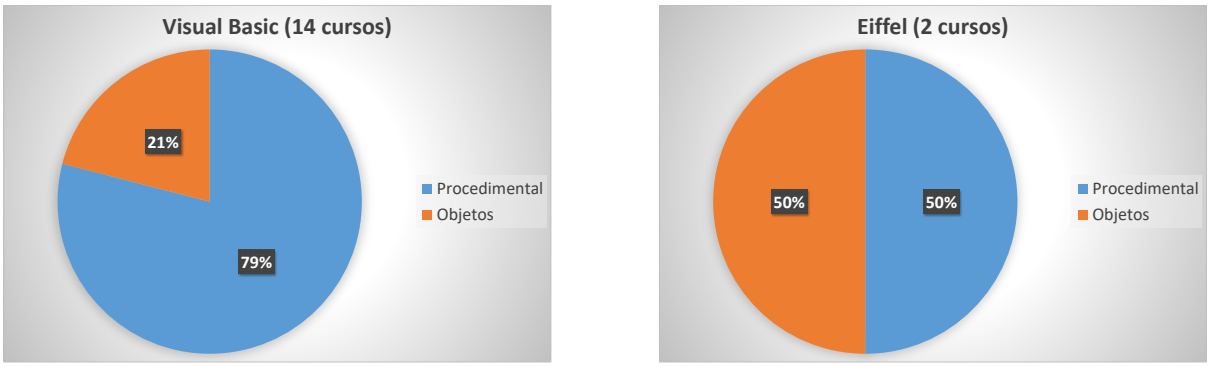

Figura 2.9: Tipo de paradigma utilizado em Visual Basic e Eiffel

2.3 A influência da indústria na escolha da linguagem de programação

A escolha da linguagem de programação utilizada no ensino da programação, neste caso na primeira disciplina de programação, deveria recair sobre uma linguagem que seja uma ferramenta para os alunos aprenderem as noções básicas sobre o que é programar, de modo a que consigam posteriormente aplicar os conhecimentos e conceitos adquiridos em qualquer linguagem de programação que utilizem no contexto do paradigma selecionado.

Apesar disso, a influência da indústria e das linguagens utilizadas no mercado de trabalho sobrepõe-se a opções que talvez sejam mais benéficas para os alunos, tal como se verificou num estudo realizado com universidades da Austrália por Michael de Raadt, Richard Watson e Mark Toleman [dRWT03], em que 56,1% dos professores inquiridos referiram a indústria como uma das razões para a escolha da linguagem, e apenas 33,3% referiu os benefícios pedagógicos como uma das razões. Neste estudo é verificado ainda que das linguagens mais utilizadas, Java, Visual Basic, C++ e C, correspondiam, à data do artigo, às linguagens que apareciam em maior quantidade em anúncios de propostas de emprego. A escolha da linguagem pela influência da indústria ficou registada como 70% no caso do Java, 43% em Visual Basic, 88% em C++ e 25% no caso de C. Em geral, 83% dos alunos do estudo estavam a ser ensinados por uma linguagem de programação influenciada pela indústria. Observou-se também que a quantidade de linguagens que o aluno sabe permite-o candidatar-se a mais propostas.

Outra prova da influência da indústria na escolha da linguagem é o estudo com universidades da Austrália de Linda Mannila e Michael de Raadt [MdR06], em que o principal motivo para a escolha de uma linguagem de programação na introdução à programação foi uma mais fácil inserção no mercado de trabalho, o que ajuda a atrair potenciais alunos. Só depois é que aparecem, em segundo lugar, os motivos pedagógicos, com um número substancialmente menor que o da indústria. Os autores defendem ainda que apesar da escolha da linguagem não ser uma escolha tão importante como o conteúdo programático da disciplina em si, é importante escolher uma linguagem que seja adequada ao programa da disciplina.

Em termos de paradigmas, a influência da indústria verifica-se novamente, pois o paradigma que apareceu mais nos anúncios foi de programação orientada a objetos, em 81% dos anúncios, seguido de 39% com linguagens de *scripting/markup* e, só depois, de linguagens procedimentais com 22% (cada anúncio podia conter mais do que um tipo de linguagem e paradigma). Isto reflete-se sobretudo na utilização de Java, Visual Basic e C++, que pertencem ao paradigma de programação orientada a objetos e que 78% dos alunos das universidades que participaram no estudo utilizaram na primeira linguagem.

Um dos motivos dado por Mark Lewis [LBBO16] para não nos submetermos à indústria na escolha da linguagem de programação é a complexidade que determinadas linguagens têm, com utilidade para grandes projetos industriais mas que não é adequada a alunos principiantes, tornando as linguagens menos puras, mais complexas e mais difíceis de ensinar.

2.4 Avaliação de características e escolhas das linguagens de programação

2.4.1 Características de uma boa primeira linguagem de programação

O desenvolvimento de uma boa linguagem de programação pressupõe a criação de bons fundamentos, tal como numa casa, pelo que há mais para além da sintaxe e expressividade de uma linguagem. Há vários aspetos a ter em conta quando se cria uma linguagem de programação, ainda para mais neste caso específico em que a linguagem é criada com o objetivo de ser apelativa e educar alunos que entram no mundo do ensino superior. Os alicerces criados nesta primeira fase irão determinar o modo como os alunos encaram determinadas estruturas de programação e, quando num futuro próximo aprenderem uma nova linguagem de programação, como utilizam as bases adquiridas nesta primeira linguagem. Uma boa abordagem nesta primeira linguagem irá preparar muito melhor os alunos para a futura aprendizagem de novas linguagens. Algumas características a considerar de modo a tornar essa linguagem inicial apelativa são, segundo Diwaker Gupta [Gup04], Linda McIver e Damian Conway [MC96], Andrew Black, Kim B. Bruce e James Noble [BBN10], Linda Mannila e Michael de Raadt [MdR06], Michael Kölling, Bett Koch e John Rosenberg [KKR95]:

- Ensino: A linguagem deve ter o intuito do ensino em mente, e não se deixar influenciar por outras linguagens apenas por popularidade. Deve-se ter em conta, no desenvolvimento da linguagem, que os alunos não estão preparados para certas noções de gramática ou pensamento estruturado para resolver um determinado tipo de problemas;
- Simplicidade: A simplicidade do código torna a aprendizagem do aluno muito mais fácil. É importante a linguagem ter uma sintaxe simples e uma semântica natural. Coisas mais complexas podem ser adicionadas posteriormente através de bibliotecas;
- Integração no currículo: A linguagem deve poder facilitar a aprendizagem de fundamentos e princípios de programação como base para a aprendizagem e adaptação do aluno a outras linguagens mais tarde;
- Recursos disponíveis: Com uma nova linguagem, é necessária a criação de novos materiais de estudo para os alunos, o que pode tornar a vida difícil aos professores e alunos. Por outro lado, devido à pouca bibliografia, a deteção de plágio é muito mais fácil;
- Ortogonalidade: Tentar minimizar os diferentes modos de criar uma determinada coisa, fazendo com que o aluno saiba o que fazer e não confundindo ainda mais o aluno com várias hipóteses de realizar a mesma ação;
- Cobertura: Deve-se ter em atenção as construções básicas de sintaxe e semântica, como condicionais (`if..then..else`, `switch...case`), ciclos (`while, for`), estruturas de dados (*structs*, classes, objetos) e algum tipo de exceções;
- Regularidade: Manter a consistência da linguagem, e evitar problemas inesperados;
- Compilação e deteção de erros: Ter uma linguagem que compile rapidamente e que envie mensagens adequadas e ajustadas aos erros cometidos inicialmente pelos alunos;

- Sistema de tipos: A linguagem deve ser fortemente tipificada e permitir verificação estática e dinâmica de tipos. A quantidade de tipos predefinidos não deve ser muito grande e os tipos não devem depender da máquina nem da implementação;
- Rapidez: O aluno deve poder implementar fácil e rapidamente pequenos blocos de código;
- Eficiência: A eficiência não deve ser uma preocupação da linguagem escolhida.

Existem ainda algumas características da linguagem que são referidas nos artigos, mas sobre as quais não se observa um consenso, tais como:

- ser orientada a objetos ou não;
- ter uma interface multimédia;
- suportar gráficos e multimédia;
- ser uma nova abordagem para ensinar software;
- ter uma comunidade;
- ser *open source*;
- ser suportada em vários sistemas operativos;
- ter bastante material de apoio;
- ser suportada por um IDE;
- suportar paralelismo;
- poder adaptar-se ao currículo e paradigma pretendido;
- ser composta por níveis, com vários subconjuntos;
- ter mensagens de erro adaptadas;
- utilizar partes da linguagem conforme o conhecimento dos alunos em dada parte da aprendizagem.

2.4.2 Coisas a evitar em linguagens de programação

No desenvolvimento da linguagem, há coisas com que ter cuidado e de evitar, tal como referido por Linda McIver e Damian Conway [MC96] e Diwaker Gupta [Gup04], que são as seguintes:

- Menos demais: A utilização de expressões demasiado curtas, apesar de ser algo que à primeira vista pode parecer ser algo útil para um novo aluno, torna a linguagem mais difícil de utilizar. O caso por exemplo do Lisp, que utiliza apenas listas e que usa a notação prefixa de cálculos, é muito diferente da notação infixa normalmente utilizada. Neste caso, a simplicidade não significa facilidade de aprendizagem e utilização.
- Mais demais: O contrário também é aplicável. Certas linguagens utilizam uma sintaxe demasiado longa para executar determinada ação. Um caso específico muito falado é a criação do método `main()` na linguagem Java e a maneira como o *input/output* funciona, necessitando duma aprendizagem mais específica de como funcionam esses métodos e classes *built in* específicas da linguagem.

- Tipos de dados: A existência em demasia de várias maneiras de representar um determinado tipo de dados é, por vezes, supérflua. A existência de `float` e de `double`, por exemplo, em determinadas linguagens, é desnecessária quando a maioria das vezes se utiliza `double`, mesmo que não seja necessário a precisão dupla que este tipo de dados nos garante. Outro caso é a variação de dimensão de um determinado tipo de dados com base na máquina na qual se corre o programa. Um tipo inteiro, `long` em C, pode ter 32 ou 64 bits, conforme a máquina em que o programa corre, o que para alunos inexperientes e até em utilizadores avançados que não tenham esse conhecimento, é uma grande desvantagem.
- Equilíbrio: O poder da linguagem ou as restrições impostas podem ser algo bom ou mau. Tudo o que é de extremos costuma ser mau e neste caso não é exceção. Há que haver equilíbrio da liberdade que uma linguagem nos dá, de modo a fazermos o que pretendemos, mas sem corrermos riscos desnecessários ou termos oportunidade de fazer algo que não corra como pretendemos. Um dos exemplos para este caso é a conversão de tipos, tanto implícita como explícita, que apesar de dar mais poder ao utilizador, pode gerar erros inesperados.

2.5 As dificuldades dos alunos e o ensino da programação

2.5.1 Dificuldades normalmente encontradas pelos alunos

O grande objetivo da criação de uma linguagem de programação para o ensino é tentar prevenir dificuldades existentes por parte dos alunos. Como tal, é necessário verificar quais são as dificuldades que os alunos mais manifestam e o modo como tratar essas dificuldades. Para isso, Essi Lahtinen, Kirsti Ala-Mutka e Hannu-Matti Järvinen [LAMJ05] realizaram um estudo sobre as dificuldades de alunos na primeira disciplina de programação e também o modo como os professores veem as dificuldades dos alunos. O questionário utilizado continha duas partes. A primeira parte foca o conteúdo da disciplina, perguntando aos alunos quais os problemas onde têm mais dificuldades e os conceitos mais difíceis de aprender. A segunda parte foca o modo como se aprende e ensina programação, averiguando o local e método mais propícios para aprendizagem e os materiais que ajudam mais na aprendizagem.

No estudo, foi concluído que os alunos consideravam como mais difícil:

- Perceber como desenhar um programa para resolver um determinado problema;
- Dividir funcionalidades em procedimentos;
- Descobrir *bugs* nos próprios programas.

Estes problemas demonstram que os alunos têm dificuldades quando colocados perante problemas de maior dimensão e não em apenas determinados detalhes. Os professores opinaram do mesmo modo que os alunos, apenas acrescentando o caso de compreensão de estruturas de programação à lista anterior.

Em termos de conceitos de programação, os alunos consideraram mais difícil:

- Recursão;
- Apontadores e referências;
- Tipos de dados abstratos;
- Tratamento de erros;

- Utilização de bibliotecas.

Os problemas verificados aqui, sobretudo recursão, apontadores e referências e tipos de dados abstratos necessitam dum pensamento mais abstrato e difícil de conseguir inicialmente pelos alunos. O tratamento de erros necessita de conhecimentos mais profundos do aluno na linguagem para saber onde tratar os erros no seu programa. A utilização de bibliotecas não é por vezes muito abordada nas aulas e depende da pesquisa independente por parte do aluno.

Em termos de aprendizagem, os alunos consideraram que estudarem sozinhos e trabalharem no trabalho final é mais útil do que assistir às aulas teóricas e às aulas práticas de exercícios. Já os professores consideraram as sessões práticas em salas de computadores, sessões de exercícios em pequenos grupos e trabalhar sozinho no trabalho final como mais útil.

O tipo de material considerado mais adequado à aprendizagem, tanto por alunos como professores, foram os programas de exemplo, enquanto que os professores valorizaram também a visualização interativa.

2.5.2 Bons métodos de ensino de programação

Uma referência de importância neste tema são os artigos de Marcus Crestani e Michael Sperber [CS10, SC12], onde chegaram a várias conclusões sobre quão efetivos eram os métodos de ensino abordados, que foram:

- A avaliação contínua do sucesso do ensino é importante;
- Supervisão pessoal é importante;
- Os métodos de avaliação tradicionais são quase inúteis;
- A forma interessa mais do que programas que funcionam;
- O que parece fácil ou natural para os professores não o é necessariamente para os alunos;
- O plágio é um problema significativo;
- A pressão das notas funciona;
- Os requerimentos para uma linguagem de ensino são diferentes dos requerimentos de uma linguagem de produção;
- Os alunos não são obrigados a gostar da linguagem ou dos professores para terem sucesso;
- Dizer aos alunos que o que estão a aprender é útil não interessa;
- Erros e dificuldades de um aluno são repetidos por outros alunos;
- Não se pode esperar diretamente dos alunos *feedback* para melhorar a disciplina e o *software* da disciplina;
- A falta de deteção de erros ou uma deteção não adequada aos alunos iniciantes dificulta um avanço independente.

Crestani e Sperber defendem que criar uma linguagem nova para o ensino é muito tentador (como se pode ver pelo tema desta dissertação), mas a diferença entre o conhecimento do professor e do aluno pode ser um

obstáculo ao desenvolvimento da linguagem perfeita para o ensino. Assim, a chave principal é não desenvolver uma linguagem, mas sim ir desenvolvendo ao longo do tempo, fazendo melhorias com base nas observações dos alunos. Os autores defendem ainda a criação de vários níveis da linguagem, que impõem várias restrições de modo a ajudar os alunos iniciantes a evitar erros comuns.

2.6 A escolha do paradigma de programação

2.6.1 Alteração de paradigma na primeira disciplina

Com base no estudo realizado na Secção 2.2.3, aferimos que, considerando as cinco linguagens mais utilizadas a nível mundial (Figura 2.3), Java tem uma percentagem de 37,1%, sendo que 71,9% das linguagens utilizadas contêm objetos (Figura 2.4). Como se sabe, Java impõe a utilização de objetos, enquanto que as outras linguagens que utilizam objetos, Python e C++, não obrigam à utilização de objetos. Já no estudo realizado na Austrália (Figura 2.7) observa-se que apenas 40% dos alunos são ensinados com um paradigma orientado a objetos, apesar de 81% usarem linguagens orientadas a objetos.

Desse modo, pode-se observar que grande parte dos cursos utilizam linguagens que possuem objetos. Mas será que esse tipo de linguagens é adequada numa primeira abordagem à programação? Foi isso que Ambikesh Jayal, Stasha Lauria, Allan Tucker e Stephen Swift pretenderam avaliar no estudo que realizaram [JLTS11]. Nesse estudo, durante um ano (ano letivo 2008-09), tiveram como objetivo lecionar os princípios de programação orientada a objetos utilizando Java desde o início da disciplina, enquanto que no ano seguinte (ano letivo 2009-10) começaram por ensinar os conceitos básicos de programação utilizando Python e só depois abordaram programação orientada a objetos com Java. Os alunos foram avaliados com base na matéria lecionada nas primeiras dez semanas, dispondo de uma hora para completar a prova. Os alunos do primeiro ano utilizaram Java e os do ano seguinte utilizaram Python. Os resultados obtidos demonstram que no ano em que Python foi utilizado, os alunos utilizavam muito mais `if`, `for`, `while`, importavam mais o pacote `random` e tinham muito menos *bugs* que não permitiam o programa correr sem algum *debugging* primeiro. Deste modo, 88% dos alunos da disciplina em que se lecionara Python tiveram nota positiva, enquanto que na disciplina em que foi lecionado apenas Java, apenas 50% dos alunos tiveram aprovação. Os resultados mostram que introduzir os conceitos básicos através da linguagem Python e só depois ensinar programação orientada a objetos melhora a compreensão dos conceitos básicos na disciplina introdutória de programação. Isto mostra que começar por uma linguagem que não obriga à utilização de objetos imediatamente é uma mais valia, focando o ensino nos aspetos mais básicos da programação primeiro. Uma hipótese será a possibilidade de ter uma linguagem que permita a utilização de objetos, mas que ao mesmo tempo não imponha essa utilização, tendo também a possibilidade de utilizar conceitos básicos facilmente.

Noutro estudo, realizado por Mirjana Ivanović, Zoran Budimac, Miloš Radovanović e Miloš Savić [IBRS15], os autores avaliaram a mudança de Modula-2 (uma linguagem do mesmo criador de Pascal, feita também para o ensino de programação) para Java (usando uma abordagem imperativa e só mais tarde com objetos no final da disciplina) e observou-se que os resultados dos alunos se mantiveram muito idênticos. A popularidade de Java na indústria e entre alunos foi vista como uma mais valia para a motivação dos alunos do que em comparação com as gerações que aprenderam Modula-2. Tendo em conta vários paradigmas, os autores referem que o paradigma imperativo é parte essencial na educação em universidades como um bom modo de ensinar programação e conceitos chave. O paradigma de programação orientada a objetos não é um bom paradigma para começar o ensino de programação, pois não é uma abordagem simples. O paradigma funcional é considerado algo mais exótico e raramente utilizado em primeiras linguagens de programação. A proposta adotada pelos autores no ensino de programação foi:

- Ensinar os alunos a pensar de forma algorítmica independentemente da linguagem;

- Utilizar uma linguagem educacional que não force uma ferramenta ou ambiente;
- Apenas depois de saber as bases de programação é que o aluno está preparado para uma disciplina de estruturas de dados e algoritmos e para o paradigma orientado a objetos.

2.6.2 Utilizar objetos de início ou não

Este é um tema em que vários autores divergem na sua opinião. Por um lado há quem sugira que os alunos aprendam primeiro os conceitos base de programação procedimental e só depois objetos, podendo separar por disciplinas ou dar ambos na mesma disciplina, como nos estudos da Secção 2.6.1. Ainda há o caso do Canadá, onde várias universidades introduziram Pascal como linguagem de programação procedimental na primeira disciplina, e Java como linguagem de programação orientada a objetos, na segunda disciplina de programação, como também é o caso da Universidade de Évora, como ilustrado na Secção 2.1.

Os autores Roger Duke, Eric Salzman, Jav Burmeister, Josiah Poon e Leesa Murray [DSB+00] estão de acordo que não é apropriado pegar numa linguagem orientada a objetos e usá-la para ensinar o paradigma procedimental, mas compreendem por que razão este modo é adotado, devido à quantidade de anos em que o paradigma procedimental foi utilizado nas disciplinas de introdução à programação. Apesar disto, os autores acham que a construção de objetos não deve ser dada logo na primeira semana.

Noutro artigo de Jason Hong [Hon98], é ilustrado como numa primeira disciplina (*Introduction to Computing*) ensinam aspetos fundamentais de programação através de pseudo-código sem compilação, e só numa segunda disciplina (*Introduction to Programming*) é que programas são construídos com uma linguagem, neste caso Java. Nesta disciplina, certas partes correspondentes aos objetos foram ocultadas através de APIs de modo a que os alunos aprendam a sintaxe de Java, escrevendo apenas o código de uma classe, ocultando aspetos como *input* e *output* e ainda o caso do `main` do Java, em que foi explicado que é necessário, e só mais tarde explicado de que modo funciona.

A utilização de objetos de início é algo defendido por alguns autores, tais como Michael Kölling, Bett Koch e John Rosenberg [KKR95] e Desmond Wesley Govender e Irene Govender [GG12], que referem como as disciplinas estão ainda muito focadas no paradigma procedimental na introdução à programação e só mais tarde introduzem os conceitos de programação orientada a objetos, quando os professores e alunos deviam adotar os objetos desde a primeira aula assim como estratégias inovadoras na aprendizagem. A indústria é referida como uma causa da escolha deste tipo de paradigmas. No segundo artigo são ainda referidos alguns argumentos para a escolha deste tipo de paradigma, tais como a boa estruturação de código, reutilização de código, e como a transição de uma linguagem de objetos para uma procedimental não é algo tão difícil como no sentido oposto.

Muitos cursos utilizam IDEs para ensinarem linguagens de programação orientadas a objetos, escondendo em parte o código, mostrando esquemas UML, ou apenas um ambiente gráfico [KA16], o que pode complicar a comparação com os alunos ensinados por programação de objetos com código.

De outro modo, há artigos, tais como o de Marcus Crestani e Michael Sperber [CS10], que referem como a escolha da linguagem na introdução à programação influencia o modo de pensar do aluno. Neste artigo, os autores falam da utilização de linguagens funcionais, e como a utilização de DrScheme permite a utilização de vários níveis da linguagem, o que permite a utilização do mesmo tipo de sintaxe com a evolução de novos níveis que contêm o anterior. A criação de vários níveis ou versões na linguagem poderia permitir a utilização de paradigma procedimental e orientado a objetos, e assim o aluno não perderia tempo a aprender uma linguagem nova. A criação de várias versões também podiam obrigar ou proibir a utilização de determinadas funcionalidades da linguagem, o que põe os alunos em pé de igualdade. Desse modo, alunos com mais conhecimentos, não podem utilizar outras ferramentas que facilitem em comparação com os alunos que estão a dar os primei-

ros passos. Por exemplo, utilizar objetos em Python num trabalho na primeira disciplina onde os alunos não aprendem o paradigma de programação orientada a objetos.

De referir ainda os dados observados na Secção 2.2.4 onde, apesar de grande parte dos alunos ter aprendido uma linguagem orientada a objetos, o paradigma seguido na aprendizagem nas aulas não correspondeu aos mesmos valores percentuais, tendo o paradigma procedimental sido mais utilizado nessas linguagens de modo geral (dependendo de linguagem para linguagem, conforme a obrigação ou não de objetos).

Em suma, como Linda Mannila e Michael de Raadt [MdR06] defendem, o objetivo principal deve ser ensinar os alunos a aplicar facilmente os conceitos principais de programação em qualquer linguagem do mesmo modo, e assim é importante escolher uma linguagem que melhor suporte o programa de uma disciplina de introdução à programação.

2.7 Comparação das linguagens mais utilizadas

Como ponto de partida para o desenvolvimento da linguagem de programação, foi realizada uma comparação de estruturas fundamentais das cinco linguagens de programação mais utilizadas mundialmente e referidas anteriormente, e de Pascal, por ser considerada uma linguagem com foco no ensino. A comparação tem como base a documentação oficial existente (*standard* C18 [ISO18], *standard* C++17 [ISO17], *standard* de Pascal [ISO90], documentação oficial de Python3.7 [Fou], documentação oficial de Java12 [Ora] e documentação oficial de Haskell [Has]). As secções seguintes ilustram alguns dos aspetos básicos e fundamentais a ter em conta.

2.7.1 Literais numéricos

Como podemos verificar na Tabela 2.1, existem algumas diferenças na estruturação de literais numéricos nas seis linguagens, mas há casos e aspetos que se mostram idênticos, senão iguais em todas as linguagens. Alguns desses aspetos são:

- Inteiros: A presença de base decimal, octal e hexadecimal é garantida em todas as seis linguagens. Tirando algumas diferenças, as características presentes nas linguagens são:
  - Base decimal: A representação da base decimal em todas as linguagens, com exceção da linguagem Haskell e Pascal, tem os números definidos com pelo menos um algarismo inicial compreendido entre 1 e 9, ainda que o caso de números começados por 0 varie um pouco. No caso do Python, apenas um número variável de algarismos 0 é possível, (ou seja, um número começado por 0 só pode conter algarismos 0), no Java, o número 0 só pode conter um algarismo, pelo que se contiver mais algarismos 0, este número passa a ser considerado octal. No caso do C e C++, qualquer número começado por 0 passa a ser octal, pelo que o próprio número 0 com apenas um algarismo é considerado octal. No caso do Haskell a regra inclui qualquer quantidade de algarismos zero iniciais. Para além do mais Java, C e C++ permitem a utilização de sufixos que determinam o tamanho do literal. No caso de Pascal, o sinal faz parte do literal, e funciona de modo idêntico ao Haskell, e é a única representação para números inteiros em Pascal.
  - Base octal: A representação da base octal depende um pouco de linguagem para linguagem, mas pode-se dividir em dois grupos. Existe o grupo que contém a linguagem Python e Haskell, que usam o prefixo octal 0o ou 0O e depois pelo menos um algarismo de 0 a 7. O outro grupo é constituído pelas linguagens Java, C e C++, nas quais um número octal começa por um algarismo 0 e depois é

seguido de pelo menos um algarismo de 0 a 7. A presença de sufixos encontra-se nas linguagens de Java, C e C++.

- Base hexadecimal: Esta é das poucas regras que é igual em todas as linguagens. Um número em base hexadecimal contém o prefixo 0x ou 0X seguido de pelo menos um algarismo hexadecimal, ou seja, algarismos de 0 a 9, e "a" a "f" (incluindo maiúsculas). Existem sufixos nas linguagens Java, C e C++.
- Base binária: Esta base encontra-se apenas nas linguagens Python, Java e C++. Este é outro caso em que a representação é igual em todas as linguagens, contendo um prefixo 0b ou 0B, seguido de pelo menos um algarismo 0 ou 1. Java e C++ contêm sufixos nesta base.

Como se verifica, as diferenças entre linguagens não são muito grandes, tirando o caso do número 0 e variantes. Na linguagem Python e Java podem-se utilizar *underscores* (_) entre algarismos, existindo a diferença que no Python se pode utilizar apenas um *underscore* entre algarismos, permitindo utilizar também entre o prefixo que determina qual a base do inteiro e o primeiro algarismo, enquanto que, no caso do Java, pode-se utilizar qualquer número de *underscores* entre algarismos, estando-se proibido de utilizar *underscore* entre o prefixo e o primeiro algarismo. O uso do *underscore* permite uma melhor visualização do número em casos de números com muitos algarismos. Existe ainda o caso de sufixos que determinam qual o tipo numérico do literal, sendo que apenas Java, C e C++ os utilizam em todas as bases numéricas. Há ainda o caso de certos tipos de bases que não são definidas pelos standards de C e Pascal, mas que compiladores como GCC e FreePascal suportam. Assim, estes são casos a ter em atenção no desenvolvimento da linguagem.

- Vírgula flutuante: A forma dos literais de vírgula flutuante depende da linguagem. No caso do Python, do Haskell e do Pascal, estas linguagens apenas suportam a utilização de literais de vírgula flutuante em base decimal. As linguagens Java, C e C++ suportam as bases decimal e hexadecimal.

  - Base decimal: A representação em base decimal está definida em todas as linguagens dum modo quase idêntico. Tirando alguns detalhes (diferença de sufixos e a utilização de expoente), o Python, Java, C e C++ contêm quatro maneiras de definir um número decimal de vírgula flutuante. O primeiro caso é um número começado pelo ponto decimal, seguido de um ou mais algarismos decimais; o segundo caso é um número começado por pelo menos um algarismo decimal seguido pelo ponto decimal; o terceiro caso é a existência de pelo menos um algarismo antes e depois do ponto decimal, e o último caso é a existência de pelo menos um algarismo decimal, sem existência de ponto decimal, mas com o expoente. A linguagem Java contém ainda um quinto caso em que existe pelo menos um algarismo decimal, sem existência de ponto decimal, mas com o sufixo. No caso da linguagem Haskell e Pascal, existem duas regras. Na primeira regra, o número é composto pelo menos um algarismo antes e depois do ponto decimal. Na segunda regra, o número é composto por pelo menos um algarismo e o expoente. Em termos de expoente, todas as linguagens dispõem do mesmo tipo de expoente decimal, que é composto pela letra "e" ou "E", seguida opcionalmente por um um sinal positivo (+) ou negativo (-), e depois seguido de pelo menos um algarismo decimal. Existem sufixos opcionais nas linguagens Java, C e C++.
  - Base hexadecimal: A representação em base hexadecimal está definida no Java, C e C++. Em termos de casos, existem essencialmente quatro casos, tal como na base decimal. As diferenças são que os algarismos utilizados são hexadecimais, a existência do prefixo hexadecimal (0x ou 0X), e que o expoente em números hexadecimais é diferente. O expoente binário utilizado, é composto pela letra "p" ou "P", seguida opcionalmente por um sinal positivo (+) ou negativo (-) e depois seguido de pelo menos um algarismo decimal. O expoente diz-se binário pois o seu valor é aplicado como a potência de base 2. Existem sufixos opcionais nas três linguagens que utilizam vírgula flutuante hexadecimal.

Como se verifica, sem contar com o caso do Python, Haskell e Pascal não conterem números hexadecimais, da maior simplicidade do Haskell e Pascal em relação aos números decimais, a representação é praticamente a mesma em todas as linguagens. Sufixos, tal como no caso de números inteiros, apenas se encontram no Java, C e C++, definem o tipo de número de vírgula flutuante que se pretende utilizar.

LITERAIS NUMÉRICOS	INTEIROS	VÍRGULA FLUTUANTE
Python 3.7	**1. Inteiro:** **1.1. Decimal:** 1.1.1. [1-9] [0-9]* 1.1.2. 0+ **1.2. Octal:** 1.2.1. 0[oO] [0-7]+ **1.3. Hexadecimal:** 1.3.1. 0[xX] [0-9a-fA-F]+ **1.4. Binário:** 1.4.1. 0[bB] [0-1]+ **NOTA**: Pode-se usar 1 underscore (_) entre algarismos. (Permite-se entre prefixo e 1º algarismo)	**2. Vírgula Flutuante:** **2.1. Decimal:** 2.1.1. [0-9]* . [0-9]+ exp? 2.1.2. [0-9]+ . exp? 2.1.3. [0-9]+ exp **2.2. exp:** 2.2.1. [eE] [-+]? [0-9]+
Java 12	**1. Inteiro:** **1.1. Decimal:** 1.1.1. [1-9] [0-9]* suf? 1.1.2. 0 suf? **1.2. Octal:** 1.2.1. 0 [0-7]+ suf? **1.3. Hexadecimal:** 1.3.1. 0[xX] [0-9a-fA-F]+ suf? **1.4. Binário:** 1.4.1. 0[bB] [0-1]+ suf? **1.5. suf:** 1.5.1. [lL] **NOTA**: Podem-se usar vários underscores (_), desde que seja entre algarismos. (Não se permite entre prefixo e 1º algarismo)	**2. Vírgula Flutuante:** 2.1. **Decimal:** 2.1.1. [0-9]* . [0-9]+ exp? suf? 2.1.2. [0-9]+ . exp? suf? 2.1.3. [0-9]+ exp suf? 2.1.4. [0-9]+ exp? suf **2.2. Hexadecimal:** 1.1.1. 0[xX] [0-9a-fA-F]* . [0-9a-fA-F]+ expbin suf? 1.1.2. 0[xX] [0-9a-fA-F]+ . expbin suf? 1.1.3. 0[xX] [0-9a-fA-F]+ expbin suf? **2.3. suf:** 2.3.1. [fFdD] **2.4. exp:** 2.4.1. [eE] [-+]? [0-9]+ **2.5. expbin:** 2.5.1. [pP] [-+]? [0-9]+
C 18 C++ 17	**1. Inteiro:** **1.1. Decimal:** 1.1.1. [1-9] [0-9]* suf? **1.2. Octal:** 1.2.1. 0 [0-7]* suf? **1.3. Hexadecimal:** 1.3.1. 0[xX] [0-9a-fA-F]+ suf? **1.4. suf:** 1.4.1. [uU] (l|L|ll|LL)? 1.4.2. (l|L|ll|LL) [uU]? **1.5. Binário:** 1.5.1. 0[bB] [0-1]+ suf?	**2. Vírgula Flutuante:** **2.1. Decimal:** 2.1.1. [0-9]* . [0-9]+ exp? suf? 2.1.2. [0-9]+ . exp? suf? 2.1.3. [0-9]+ exp suf? **2.2. Hexadecimal:** 2.2.1. 0[xX] [0-9a-fA-F]* . [0-9a-fA-F]+ expbin suf? 2.2.2. 0[xX] [0-9a-fA-F]+ . expbin suf? 2.2.3. 0[xX] [0-9a-fA-F]+ expbin suf? **2.3. suf:** 2.3.1. [fFlL] **2.4. exp:** 2.4.1. [eE] [-+]? [0-9]+ **2.5. expbin:** 2.5.1. [pP] [-+]? [0-9]+
Pascal 1990	**1. Inteiro:** **1.1. Decimal:** 1.1.1. [-+]?[0-9]+	**2. Vírgula Flutuante:** **2.1. Decimal:** 2.1.1. [-+]?[0-9]+ . [0-9]+ exp? 2.1.2. [-+]?[0-9]+ exp **2.2. exp:** 2.2.1. [eE] [-+]? [0-9]+
Haskell 2010	**1. Inteiro:** **1.1. Decimal:** 1.1.1. [0-9]+ **1.2. Octal:** 1.2.1. 0[oO] [0-7]+ **1.3. Hexadecimal:** 1.3.1. 0[xX] [0-9a-fA-F]+	**2. Vírgula Flutuante:** **2.1. Decimal:** 2.1.1. [0-9]+ . [0-9]+ exp? 2.1.2. [0-9]+ exp **2.2. exp:** 2.2.1. [eE] [-+]? [0-9]+

Tabela 2.1: Literais numéricos em 6 linguagens [1]

[1] Notação: + (1 ou mais), * (0 ou mais), [..] (1 dos elementos), .-. (intervalo), ? (opcional)

2.7.2 Funções, classes, *structs* e métodos

Nas linguagens analisadas, existem diferentes maneiras de estruturarmos os dados e código e executar ações sobre eles. Tal como referido na Secção 2.2.3, verificamos que determinadas linguagens têm foco imperativo, enquanto Haskell é puramente funcional. Haskell dispõe de diferentes mecanismos organizacionais tais como tuplos, módulos, interfaces e registos e dispõe de funções. Já Java é uma linguagem orientada a objetos que obriga o programador a utilizar objetos e não dispõe de funções. A linguagem C não dispõe de objetos, mas permite a criação de *structs*, que bem utilizadas permitem uma utilização quase idêntica à criação de objetos. C++, como extensão de C, contém todas as opções existentes em C, adicionando objetos e métodos. Python contém funções e, de um modo um pouco menos claro e confuso, classes e métodos. Por último, Pascal inclui a diferença na criação de funções e procedimentos, e como tipo de dados estruturados inclui os `record`. A Tabela 2.3 ilustra mais especificamente o referido.

<table>
<tr><th></th><th>FUNÇÕES / PROCEDIMENTOS</th><th>CLASSES / STRUCTS</th><th>MÉTODOS</th></tr>
<tr><td>Python
3.7</td><td>def nome_função(parâmetros):
  corpo
  return [expressão_opcional];</td><td>class Nome_Classe:
  def __init__(self, parâmetro):
  self.valor = parâmetro</td><td>def nome_metodo(self, parâmetro):
  self.valor = parâmetro</td></tr>
<tr><td>Java
12</td><td>X</td><td>modifier class Nome_Classe {
  tipo nome_variável;
  modifier Nome_Classe(parâmetros){
  corpo
  }
}</td><td>(DENTRO DA CLASSE)
modifier tipo nome_metodo(parâmetros){
  corpo
}</td></tr>
<tr><td>C
18</td><td rowspan="2">tipo nome_função(parâmetros) {
  corpo
}</td><td>struct Nome_estrutura {
  tipo nome_variável;
};</td><td>X</td></tr>
<tr><td>C++
17</td><td>class Nome_Classe {
  public:
  tipo nome_variável;
};</td><td>(DENTRO DA CLASSE)
tipo nome_metodo(parâmetros) {
  corpo
}

(FORA DA CLASSE)
tipo Nome_Classe::nome_metodo(
parâmetros) {
  corpo
}</td></tr>
</table>

Tabela 2.2: Estruturação de código em 6 linguagens (tabela 1 de 2)

	FUNÇÕES / PROCEDIMENTOS	*RECORDS/ TUPLOS*	MÉTODOS
Pascal 1990	**function** *nome_função* **(** *parâmetros* **) :** *tipo*; *declarações_variáveis_locais* **begin** corpo *nome_função* **:=** *expressão* **; end;** **procedure** *nome_proc* **(** *parâmetros* **) ;** *declarações_variáveis_locais* **begin** corpo **end;**	**type** *nome_record* **= record** *nome_variável* **:** tipo**; end;**	X
Haskell 2010	*nome_função* **::** *declaração* *nome_função definição*	**(** elemento1, …, elementoN **)**	X

Tabela 2.3: Estruturação de código em 6 linguagens (tabela 2 de 2)

2.7.3 Instrução condicional if

Este tipo de instruções não muda muito de linguagem para linguagem. Observa-se que existe sintaxe igual em Java, C e C++, em que se a condição avaliada for verdadeira, executa a instrução logo a seguir ao `if`, se não executa imediatamente a instrução a seguir ao `else`. No caso do Python, a indentação determina a posição do código, e as instruções condicionais não são exceção, como tal o modo como o código é representado na Tabela 2.4 é o modo como o código tem de ser escrito, incluindo como referido a indentação. Na linguagem Pascal, existe a palavra reservada *then* que se encontra após a condição. Em termos de Haskell, as instruções condicionais são expressões, e como tal, ao contrário das linguagens imperativas, o `else` é obrigatório, e como tal também, o tipo da instrução se a condição for verdadeira tem de ser o mesmo da instrução se a condição for falsa.

	IF... ELSE	IF... ELSE IF... ELSE
Python 3.7	**if** *condição*: *executa instrução(ões) se condição for True* **else:** *executa instrução(ões) se condição for False*	**if** *condição*: *instrução(ões)* **elif** *condição*: *instrução(ões)* **elif** *condição*: *instrução(ões)* **else:** *instrução(ões)*
Java 12 C 18 C++ 17	**if (***condição***)** *executa instrução se condição for true* **else** *executa instrução se condição for false*	**if (***condição***)** *instrução* **else if (***condição***)** *instrução* **else if (***condição***)** *instrução* **else** *instrução*
Pascal 1990	**if** *condição* **then** *executa instrução se condição for true* **else** *executa instrução se condição for false*	**if** *condição* **then** *instrução* **else if** *condição* **then** *instrução* **else if** *condição* **then** *instrução* **else** *instrução*
Haskell 2010	**if** *condição* **then** *avalia expressão se condição for true* **else** *avalia expressão se condição for false*	**if** *condição1* **then** *avalia expressão* **else if** *condição2* **then** *avalia expressão* **else** *avalia expressão*

Tabela 2.4: Condicionais em 6 linguagens

2.7.4 Ciclos

A utilização de ciclos é fundamental na programação e, sem eles, muitas das coisas utilizadas hoje em dia não seriam possíveis. Das linguagens estudadas, apenas Haskell não possui nenhum tipo de ciclo, devido a ser uma linguagem puramente funcional.

Java, C e C++ dispõem de ciclos `for` que são compostos por inicialização, condição e incremento. A inicialização é feita através de atribuição de um valor a uma variável numérica inteira (não se devem utilizar variáveis numéricas de vírgula flutuante por causa deste tipo de variáveis não ser uma representação exata de números reais). A condição determina quando se deve parar o ciclo, e é normalmente uma expressão com um valor booleano. O incremento representa a forma como se avança no ciclo de elemento para elemento, isto é, podemos avançar num dado conjunto de elementos e alterarmos a variável definida na inicialização e/ou utilizada na inicialização. O ciclo `while` e `do...while` funcionam de modo idêntico ao ciclo `for`, mas mantendo apenas a condição, devendo o utilizador definir a inicialização e incremento noutra parte do código. O Python é um caso especial, pois o ciclo `for` não é um ciclo como os ciclos já referidos, trata-se sim de um iterador, não exis-

tindo a possibilidade de definir o tipo de incremento que se pretende, ou onde inicializar o ciclo. O Java, para além do ciclo `for` já referido, tem também um ciclo `for` como este do Python. O Python tem à sua disposição um ciclo `while` como em Java, C e C++, permitindo assim simular um ciclo `for` dessas linguagens, necessitando o utilizador de definir a inicialização e incremento por si. Na linguagem Pascal, o ciclo `for` funciona de um modo um pouco diferente do das outras linguagens, existindo uma variável de controlo com um valor de início e de fim, que é incrementado em cada iteração do ciclo. O ciclo pode ser definido de duas maneiras diferentes, conforme se queira incrementar ou decrementar o valor da variável de controlo. A instrução `while` do Pascal funciona de modo idêntico às outras linguagens, mas não contém a instrução `do...while`, mas sim a instrução `repeat...until`. Esta instrução executa as instruções dentro do ciclo até se verificar a condição. A Tabela 2.5 ilustra o modo como o código deve ser redigido.

<table>
<tr><th>CICLO</th><th>FOR</th><th>WHILE</th><th>DO ... WHILE /
REPEAT ... UNTIL</th></tr>
<tr><td>Python
3.7</td><td>for variável_iterável in sequência:
instrução(ões)</td><td>while condição:
instrução(ões)</td><td>X</td></tr>
<tr><td>Java
12</td><td>for (variável_iterável : array)
instrução</td><td rowspan="3">while (condição)
instrução</td><td rowspan="3">do
instrução
while (condição);</td></tr>
<tr><td>C
18</td><td rowspan="2">for (inicialização; condição; incremento)
instrução</td></tr>
<tr><td>C++
17</td></tr>
<tr><td>Pascal
1990</td><td>for variável_controlo := valor_inicio to valor_fim do
instrução

for variável_controlo := valor_inicio downto valor_fim do
instrução</td><td>while condição do
instrução</td><td>repeat
instrução(ões)
until condição ;</td></tr>
<tr><td>Haskell
2010</td><td>X</td><td>X</td><td>X</td></tr>
</table>

Tabela 2.5: Ciclos em 6 linguagens

2.8 Problemas encontrados em linguagens de programação

Um dos grandes objetivos desta dissertação é levantar problemas que se encontrem nas linguagens mais utilizadas no ensino da programação e tentar resolver, se não todos, pelo menos uma grande parte destes problemas. Foram encontrados alguns problemas, como se pode ver a seguir em Java, C e C++.

2.8.1 Problemas em C e C++

Foram encontrados vários problemas em C, como os referidos por R. P. Mody [Mod91], e em C++, por Joe Bergin, Achla Agarwal e Krishna Agarwal [BAA03]. Os problemas das linguagens de programação C e C++ apresentam-se em conjunto devido a alguns desses problemas serem comuns às duas linguagens, devido à criação de C++ ter como base o C e haver alguma compatibilidade de código. Alguns dos problemas de maior gravidade e que serão de algum modo tidos em conta na criação da linguagem Temple são:

- As diretivas `#include`: A necessidade de incluir *header files* da própria linguagem é uma complicação para os alunos. No caso do aluno necessitar de utilizar determinadas bibliotecas como a biblioteca de *input/output* ou de matemática, o aluno tem de saber quais as bibliotecas a importar e os nomes das mesmas. O próprio conceito de bibliotecas pode ser confuso para os alunos inexperientes.

- Conversão de tipos implícita: No caso de se definir uma variável de um tipo com um valor de outro tipo, existe conversão implícita do valor para o tipo da variável, quando deveria dar um erro, pois pode criar resultados inesperados.

- A ordem de processamento de instruções: A ordem de processamento de determinadas partes de instruções depende da implementação do compilador e não da linguagem, o que pode gerar confusão entre os alunos se utilizarem diferentes compiladores.

- Comparação de *strings*: Devido às *strings* em C e C++ serem apontadores para um espaço de memória, a comparação de *strings* compara não o valor das *strings*, mas sim o valor da posição de memória dos apontadores.

- O símbolo da afetação"=" e o símbolo de igualdade "==": A utilização do símbolo de afetação em expressões booleanas da instrução `if`, por exemplo `if(value = 10)`, dá resultados inesperados, pois atribui um valor à variável `value` e é um valor booleano `true` se o valor atribuído a `value` for diferente de `0`, quando o mais certo é o aluno querer comparar o valor `10` com a variável `value`, como em `if(value == 10)`. Ambos os casos são válidos em C e C++, e o primeiro caso devia ser um erro.

- Acesso a uma posição inexistente de um *array*: Em C e C++ é possível aceder a uma posição fora do *array*, acedendo à posição de memória correspondente, o que pode alterar valores externos ao *array* e esconder a existência de erros no programa.

- Chamadas de funções: A chamada de funções e de procedimentos não é diferenciada em C e C++, e desse modo o aluno pode realizar uma chamada de função sem afetar uma variável, descartando o valor devolvido de uma função.

- Expressões com efeitos secundários: Por exemplo, `x = x++ -1`, em que há 3 possíveis resultados, o incremento de `x`, o decremento de `x` ou a não alteração de `x`. O que acontece depende, novamente, do compilador.

- Funções sem devolução de valor: Em ambas as linguagens, é possível ter uma função que supostamente deve retornar um valor de determinado tipo, conforme a definição da função, mas a não existência de retorno de valor não é um erro, apenas um aviso. Assim o código pode ser compilado sem erros, e mais tarde criar problemas.

- Mistura de tipos: `0` pode ser o número inteiro zero, ser o valor booleano falso, ser o terminador de *strings* e pode ainda ser o apontador nulo NULL.

- Espaço de nomes: A utilização de vários espaços de nomes pode criar confusão no caso da utilização do mesmo identificador. Um exemplo dado pelos autores é o seguinte:

```
1  typedef struct list *list;
2
3  struct list {
4    int fld;
5    list list;
6  }
7
8  int length(list l)
9  {
10   int i = 0;
11
12   list: if(l) {
13     i++;
14     l = l->list;
15     goto list;
16   }
17   else return i;
18 }
```

  Neste exemplo, existe a definição de um tipo chamado `list` que corresponde a um apontador para um valor `struct list`. Depois existe a definição de uma estrutura chamada `list` que contém um membro chamado `list` que é um apontador de `list`. Na função `length` existe ainda a *label* `list`. A utilização de vários espaços de nomes acaba por deixar utilizar o mesmo identificador para várias coisas ao mesmo tempo, o que de certeza absoluta pode gerar muita confusão se permitido aos alunos.

- Números octais: um número precedido por 0 passa a ser um número octal e o próprio número 0 acaba por ser octal. Assim o número octal 011 corresponde em decimal a 9. Interessante é que o mesmo não se aplica no caso das constantes de caracteres, em que '\011' é o mesmo que '\11'.

2.8.2 Problemas em Java

Na leitura dos artigos de Jason Hong [Hon98] e de David Clark, Cara MacNish e Gordon F. Royle [CMR98], foram observados alguns pontos que na introdução à programação podem ser um problema:

- O método `main`: A utilização do método `main` é fundamental, e para um aluno que aprende Java, muitas vezes utiliza esse método sem saber o que está por detrás do mesmo, chegando mesmo os professores a dizerem para o utilizarem, apesar dos alunos não saberem o seu significado. O problema está na quantidade de conceitos que este método aborda e utiliza. Exemplo de "Hello World" em Java:

```
1  public class HelloWorld {
2    public static void main(String[] args) {
3      System.out.println("Hello World!");
4    }
5  }
```

  Este método aborda conceitos como modificadores de acesso, instância *versus* classe, valor de retorno, *arrays*, pacotes de classes, variáveis, *output*, e esta grande quantidade de conceitos pode intimidar o aluno.

- *Input* e exceções: A utilização de *input* obriga a utilização de exceções. Ou seja, algo que os alunos utilizam por vezes logo de início depende de um conceito que aprendem só mais tarde, as exceções. No caso do artigo de Hong [Hon98], os alunos eram abstraídos destes problemas através de classes já implementadas pelos professores.
- Diferença entre tipos primitivos e referências: Qual a diferença entre `int` e `Integer` em Java? É que `int` é um tipo primitivo, enquanto `Integer` é uma referência para um objeto da classe `Integer`, e que contém métodos próprios para manipulação de inteiros. Os alunos de início podem não saber qual a diferença entre ambos.
- Classe *versus* instância: A diferença entre variáveis de classe e instância e métodos de classe e instância é uma dificuldade para os alunos. A diferença entre afetar uma variável de uma instância ou afetar uma variável da classe em si, a utilização ou não da palavra reservada `static`, e erros que advêm da utilização ou não da mesma podem gerar erros que os alunos inicialmente não entendem.
- Objeto *versus* classe: Saber que um objeto é uma instância de uma classe e que uma classe é um esquema de objetos pode ser um conceito um pouco complicado de se perceber de início, e ir de encontro à razão porque talvez ensinar programação orientada objetos de início não seja o mais indicado.
- Variáveis locais do método e instâncias do objeto: Tenha-se por exemplo o excerto de código seguinte:

```
1  class Circle {
2
3    Point centre;
4    float radius;
5
6    public boolean smallerThan(Circle other) {
7      if (radius < other.radius) return true;
8      else return false;
9    }
10 }
```

  Neste exemplo, observa-se a utilização de `other.radius` como variável do método `smallerThan` pertencente à variável do objeto `other` e `radius` pertencente à variável de instância da classe `Circle`. A confusão pode-se gerar nos alunos, e muitos chegam a utilizar `this.radius` para se referirem a variáveis de instância do próprio objeto.

2.9 Conclusão

Neste capítulo constata-se que as linguagens de programação utilizadas nas primeiras disciplinas no ensino, tanto a nível mundial como de Portugal, constituem um pequeno leque de linguagens. É observado que as linguagens mais utilizadas são Java, Python, C, C++ e Haskell (por ordem decrescente de utilização). A percentagem de cursos que utilizam linguagens que permite a utilização de objetos é 71,9% das linguagens e das cinco linguagens, todas são linguagens imperativas, à exceção de Haskell que é funcional.

Em termos de características desejadas na linguagem que se pretende criar, deve-se dar atenção à simplicidade no geral, tentando utilizar uma sintaxe simples, poucas maneiras de realizar a mesma ação, tendo atenção às construções básicas de sintaxe.

A linguagem criada deve ter em atenção as maiores dificuldades sentidas por alunos, devendo ter cuidado com conceitos tais como a recursão, apontadores e referências, tipos de dados abstratos, tratamento de erros e uma boa inclusão e utilização de bibliotecas se possível.

Em termos de paradigma, a linguagem, como já referido, deve ter ênfase na simplicidade e atenção às construções básicas de sintaxe (`if...then...else, switch, for, while`), podendo possibilitar a utilização de objetos, mas não tendo o foco neste paradigma, pois foi observado que a utilização do paradigma orientado a objetos numa primeira fase torna a aprendizagem de construções básicas de programação mais difícil.

A comparação das cinco linguagens de programação mais utilizadas e Pascal é um bom ponto de partida para decidir como desenvolver a linguagem de programação, tendo em conta também os problemas levantados ao longo deste capítulo.

Em suma, este projeto pretende dar outra alternativa às linguagens de programação existentes e, assim, tentar resolver casos importantes que ainda não foram resolvidos. Neste caso pretende-se criar uma linguagem que seja mais apelativa para os novos alunos, que seja uma linguagem preparada para educar os alunos e poder ser uma linguagem que esteja dotada para ser uma boa linguagem como ponto de partida no mundo da programação. Pretende-se assim pegar em aspetos bons de várias linguagens e/ou criar novos aspetos se assim for necessário ou benéfico para o trabalho no seu todo. Casos específicos a serem resolvidos são tidos em conta.

3

Desenvolvimento

Durante o trabalho de pesquisa e desenvolvimento da dissertação, pôde-se examinar várias opiniões e reconhecer alguns problemas acerca das linguagens de programação utilizadas atualmente e nos últimos anos. Neste capítulo pretende-se primeiramente verificar quais os problemas a enfrentar e o modo como serão resolvidos. São ilustrados também motivos por detrás de certas decisões.

Na segunda parte deste capítulo é apresentada a definição formal da linguagem criada, a ilustrar como estão definidas todas as instruções pertencentes a esta nova linguagem de programação. A gramática completa da linguagem encontra-se no Anexo C.

O nome escolhido para a linguagem criada foi Temple, como homenagem a um dos mais importantes monumentos da cidade de Évora e de Portugal.

Seguidamente são apresentados detalhes sobre a implementação parcial desta linguagem, ilustrando alguns problemas e soluções encontrados no desenvolvimento da linguagem Temple.

Por fim, são ilustrados exemplos de exercícios possíveis com a linguagem Temple, comparação da linguagem Temple com outras linguagens e também a eficiência da linguagem (apesar de não ser um ponto fulcral deste tipo de linguagens para o ensino).

3.1 Escolhas e decisões

Neste ponto são apresentadas algumas das decisões tomadas no desenvolvimento do trabalho e como tiveram em conta os problemas encontrados noutras linguagens de programação. Também foram tidas em conta as características que se pretendem numa linguagem para o ensino. Como tal, é normal muitas das instruções da linguagem Temple não se diferenciarem muito do que é usual em linguagens utilizadas no ensino, de modo a facilitar a adaptabilidade dos alunos no futuro e para que seja fácil para o aluno entender o significado das palavras que dão o nome às instruções. Outra situação em que se teve muito cuidado foi na verificação de erros, de modo a alertar o aluno para erros comuns e que acontecem com muita frequência. Outros dos cuidados e decisões tomadas serão ilustrados seguidamente mais a fundo.

3.1.1 Tipos primitivos

Um dos pontos referidos na Secção 2.4.2 foi o tipo de dados. Foi dado o exemplo da linguagem C em que existem uma quantidade enorme de tipos primitivos, como o tamanho de cada tipo pode depender da máquina em que o programa corre e ainda a possibilidade de existirem tipos que o aluno pode nem necessitar. Com base nisso foi decidido que a linguagem Temple deveria ter apenas tipos fundamentais e que estes não dependam da máquina onde os programas correm.

Para tal, para inteiros existe o tipo `int` de 32 bits com sinal, para números de vírgula flutuante o tipo `float` com dupla precisão 64 bits com sinal, o tipo `bool` para representar valores lógicos, o tipo `void` para vazio, o tipo `char` para caracteres utilizando *encoding* UTF-8, o tipo `string`, também utilizando o *encoding* UTF-8 e o tipo `file` para ficheiros.

A escolha dos nomes destes tipos deveu-se a serem palavras normalmente associadas a estes tipos. Palavras como `int` para os inteiros foi um decisão rápida e fácil, já no caso dos números de vírgula flutuante, foi decido utilizar a palavra `float` devido à sua similaridade com o tipo que pretende representar, e não palavras como `double` que não têm tanta ligação com o tipo que pretende representar (apesar do tipo ser de dupla precisão, que costuma ser mais utilizado pelos alunos). A decisão do tipo `bool` foi uma decisão fácil também, apesar de uma das palavras possíveis ter sido `boolean`. O tipo `void` foi decidido como um tipo para o vazio e a ser utilizado em procedimentos, para haver um modo de representar funções que não retornam qualquer valor. Já nos tipo `char` e `string`, foi decidido diferenciar um tipo para um carácter e um tipo que represente uma sequência ordenada de caracteres, existindo assim um tipo próprio para as *strings*. No último caso foi decido criar um tipo para os ficheiros, o tipo `file`. Este tipo serve para trabalhar com ficheiros e guardar toda a informação relativa ao ficheiro e tipo de abertura que se realizou com o mesmo. Com este tipo, pretende-se simplificar a manipulação de ficheiros que é por vezes complicado para os alunos.

Um dos problemas que se pretendeu resolver também foi a confusão gerada por se misturarem tipos. Assim sendo, números inteiros não podem ser avaliados como valores booleanos, entre outros.

3.1.2 Literais numéricos

Em termos de representação dos literais numéricos do tipo primitivo `int`, foi decidido que em termos de base, o aluno pode representar os números nas bases decimal, hexadecimal, octal e binária. A Tabela 2.1 foi uma base para a escolha da representação dos números inteiros, com base sobretudo na linguagem Python. Uma decisão tomada foi a utilização de números binários que não existem em algumas das linguagens observadas (apesar de alguns compiladores dessas linguagens incluírem). A utilização de *underscores* entre algarismos foi também considerado algo a mais que não merecia utilização na linguagem produzida por ser uma linguagem

feita com o ensino em mente. Uma das alterações em relação às bases foi que nas bases hexadecimal, octal e binária, o prefixo contém sempre a letra referente à base em minúsculo, ou seja, `0x` para hexadecimal, `0o` para octal e `0b` para binário. Esta decisão foi tomada por motivos visuais, pois na base octal a utilização do prefixo com letra maiúscula (`0O`) poderia causar mais confusão. Assim, foi decido utilizar letra minúscula em todas as bases de modo a ser consistente.

Em termos de representação dos literais numéricos do tipo primitivo `float`, foi decidido que em termos de base, o aluno pode representar os números na base decimal, que para uma linguagem de aprendizagem é suficiente. Uma escolha decidida neste caso foi obrigar o aluno a utilizar um algarismo antes e depois do ponto decimal, para uma representação adequada de números reais. De resto, a representação não é muito diferente da que é utilizada por outras linguagens, com uma definição formal idêntica.

3.1.3 Conversão de tipos implícita

Mais uma vez, simplificação é um dos objetivos desta linguagem, e evitar erros inesperados acaba por ser um pilar fundamental. A conversão de tipos é levada ao mínimo, e a única conversão de tipos implícita na linguagem é a conversão de números inteiros para números de vírgula flutuante em qualquer expressão em que se espera um número de vírgula flutuante e se encontra um número inteiro. De referir ainda que, devido às escolhas feitas para o tipo `int` e para o tipo `float`, não há perda de informação neste tipo de conversão.

Foi decidido não incluir mais conversões implícitas de tipos de modo a não causar conversões inesperadas e tornar os programas mais previsíveis para o aluno. Uma possibilidade pensada foi a conversão implícita de números de virgula flutuante para números inteiros (guardando apenas a parte inteira), mas foi decido que esta funcionalidade poderia ser acrescentada com uma biblioteca e não como algo fundamental nesta fase do trabalho.

3.1.4 Literais booleanos

Os literais booleanos em Temple podem ter dois valores: `true` e `false`. Não existe outra alternativa para representar literais booleanos, tais como inteiros `0` e `1` por exemplo, de modo a evitar misturar tipos e representações de literais de vários tipos. Pretende-se sobretudo evitar gerar confusão entre os alunos.

3.1.5 Literais de char e string

A representação de caracteres em Temple é possível com o *encoding* `UTF-8`. A escolha por detrás deste *encoding* tem a ver com a utilização de caracteres específicos da língua portuguesa, e um modo de poder simplificar operações com caracteres com símbolos de acentuação gráfica ou, por exemplo, o cê cedilhado. A escolha do *encoding* `UTF-8` e não de outro padrão de representação de texto prende-se com a grande utilização do *encoding* `UTF-8` e a grande compatibilidade com `ASCII`.

Na linguagem Temple foi ainda decidido diferenciar os literais de caracteres dos literais de *strings*. Os caracteres encontram-se sempre entre plicas (`'`) e existem ainda caracteres especiais que utilizam a barra invertida (`\`) e que são: inserir nova linha (`\n`), tabulação (`\t`), o carácter plica (`\'`) e ainda a barra invertida (`\\`). No caso das *strings*, estas encontram-se sempre entre aspas (`"`), e assim o carácter plica não necessita do carácter especial de barra invertida, mas para inserir as aspas como carácter em `string` já é necessário inserir o carácter especial (`\"`).

As *strings* têm, por omissão, 255 caracteres, mas é possível definir *strings* com diferentes tamanhos se necessário. Esta decisão tem como base a implementação de *strings* em Pascal, que tem dois tipos de *string*, um com tamanho fixo e um com tamanho arbitrário. Na linguagem Temple, tentou-se simplificar as *strings*, em que um carácter, seja simples ou com acento, é considerado apenas um carácter. Outra situação que se pretendeu evitar foi a necessidade de existir carácter nulo para terminação de uma *string*, não existindo em Temple. Deste modo, tanto o tamanho máximo como o tamanho atual de uma *string* corresponde a um número de caracteres e não de memória necessária para os guardar.

Exemplo para a definição de uma variável de tipo **string** com tamanho predefinido (255 caracteres):

```
1 string uma_string;
```

Exemplo para a definição de uma variável de tipo **string** com tamanho 30:

```
1 string<<30>> uma_string2;
```

Exemplo para a definição de uma variável de tipo **string** com inicialização:

```
1 string<<10>> uma_string3 = "Bom dia";
```

A escolha dos símbolos **<<** e **>>** prende-se com a representação dos *arrays*. Pretendeu-se representar de um modo idêntico aos *arrays*, com o número entre um símbolo de abertura e de fecho para representar a quantidade de caracteres que se pretende. Não são utilizados os símbolos **[** e **]** da representação dos *arrays* para demonstrar que as *strings* não são um caso especial de *arrays* de caracteres (como na linguagem C por exemplo), em que se tem de incluir o carácter nulo.

Existem ainda funções predefinidas que permitem saber o tamanho da *string* a um dado momento (**length**) e o tamanho total possível para a *string* (**maxlength**). Caso o argumento das funções seja um literal, tanto o tamanho atual como o máximo são o mesmo valor.

Exemplo das funções predefinidas:

```
1 string<<10>> uma_string4 = "Exemplo";
2 maxlength(uma_string4);     # Retorna 10
3 length(uma_string4);        # Retorna 7
4
5 string uma_string5;
6 maxlength(uma_string5);     # Retorna 255
```

Caso a *string* não esteja inicializada, o função **length** retorna o valor **0**.

A *string* pode ser alterada carácter a carácter, mas só depois de ter sido inicializada, e tem de ser uma posição com um carácter já inicializado. De referir ainda que a representação de *strings* é de base 0. Um exemplo para alterar um carácter de uma *string* é:

```
1  string<<10>> uma_string6;
2  uma_string6 = "xpto";
3  uma_string6<<0>> = 'a';     # Valor da string é apto
4  uma_string6<<10>> = 'b';    #( Este caso é um erro porque
5                                o tamanho máximo é 10 e a
6                                posição 10 é fora do tamanho
7                                (representação é de base 0) #)
8  uma_string6<<8>> = 'c';     #( Este caso é um erro porque
9                                a posição 8 ainda não foi
10                               inicializada. Só se pode alterar
11                               posições previamente inicializadas #)
```

Assim, por exemplo, numa *string* com tamanho máximo 10 e inicializada com uma *string* de tamanho 4, pode-se alterar caracteres das primeiras 4 posições. A alternativa é alterar a variável com uma nova *string* diretamente. Se o literal da *string* for alterado para um tamanho maior, a quantidade de caracteres que se pode alterar é o mesma que do novo tamanho (tendo em consideração que o tamanho do literal não pode ser superior ao tamanho máximo definido e é um erro aceder fora de um índice válido). É possível também encurtar uma *string* com uma nova inicialização.

Exemplo do caso anterior:

```
1 string<<8>> uma_string7;
2 uma_string7 = "hello";
3 uma_string7<<5>> = 'x';     # Este caso é um erro
4 uma_string7 = "hellow";
5 uma_string7<<5>> = 'x';     # Valor da string é hellox
```

3.1.6 Arrays

Na linguagem Temple foi decidido simplificar as características dos *arrays*. Características definidas foram a possibilidade de definir um *array* com a quantidade de elementos por dimensão, podendo o aluno definir um *array* com qualquer número de dimensões e o tipo dos elementos do *array*. O aluno pode ainda definir um *array* com a quantidade de dimensões e ao mesmo tempo inicializar o *array* com uma sequência de itens que esteja de acordo com a quantidade de elementos por cada dimensão definida.

Exemplos de definição de um *array*:

```
1 array[3,6,2] int novo_array;
2 array[2,2] int novo_array2 = {{1,0},{5,3}};
```

A afetação dos *arrays* diretamente não é possível (devido à afetação neste caso não ser de complexidade constante e a possibilidade dos *arrays* serem de tamanhos diferentes), assim o aluno tem de afetar posição a posição. Para afetar uma posição, o aluno tem de indicar o índice em todas as dimensões do *array* que pretende afetar. A afetação de uma posição fora do *array* gera um erro durante a execução do programa, tendo em conta um dos problemas das linguagens C e C++, na Secção 2.8.1. Os *arrays* são de base `0`, e assim o primeiro índice do *array* será o índice `0` em todas as dimensões definidas no *array*.

Exemplo de afetação de um *array*:

```
1 array[2,1]float novo_array3;
2 novo_array3[0,0] = 3.0;
3 novo_array3[2,0] = 4.3;     #( Neste caso é um erro porque os arrays
4                                 são de base 0, e assim o valor máximo
5                                 acessível na primeira dimensão é 1
6                                 (um valor a menos do que o usado).
7                                 É um erro de acesso fora do array. #)
```

De referir ainda que podem existir *arrays* de *arrays*.

Exemplo de *arrays* de *arrays* e acesso:

```
1 array[2,3]array[3]char var_arr_char;
2 var_arr_char[1,0][2] = 'ç';
```

Uma função que foi criada para os *arrays* foi a função `size`, que tem dois argumentos, um *array* e a dimensão que o aluno pretende saber o tamanho. Esta função retorna o tamanho de uma dimensão de um *array*,

tendo utilidade por exemplo, na utilização de *arrays* em funções que têm tamanho da dimensão arbitrário. As dimensões são de base 1, considerando a 1ª dimensão como número 1.

Exemplo da função **size**:

```
1 array[2,3]int array1;
2 int x = size(array1, 1);
3 # valor de x será 2, primeira dimensão do array
```

Em termos de sintaxe, foi decido incluir a palavra **array** na definição de um *array* de modo a simplificar. Um dos problemas resolvidos foi que em determinadas linguagens os parêntesis retos são inseridas após o tipo do *array*, por exemplo, **tipo[] nome_array**, ou noutras linguagens são inseridas após o nome do *array*, por exemplo, **tipo nome_array[]**. Deste modo não existe a confusão de onde colocar o tipo do *array*, pois o aluno começa a definição com a palavra **array** e sabe que o tipo do *array* vem após os parêntesis retos onde determina as dimensões e os tamanhos das mesmas. De referir por fim que a quantidade de elementos de uma dimensão pode ser definido através de uma variável para além de valores constantes.

3.1.7 Typedef

Na linguagem Temple podem-se criar novos tipos com base nos tipos já existentes. A declaração **typedef** cria um sinónimo para um tipo já definido, criando uma diferenciação personalizada pelo programador. No caso do programador criar um tipo **Y** com base no tipo **X**, o programador pode usar o tipo **Y** em lugares que utilizaria o tipo **X** e vice versa.

Exemplo da criação de um novo tipo:

```
1 typedef idade int;
```

Esta possibilidade de sinónimos é uma mais valia para o aluno. Caso utilize um sinónimo de um tipo e a determinada altura queira alterar o tipo utilizado em várias partes do programa, basta alterar na declaração **typedef** e não tem de alterar em todas as posições do programa. Há ainda o caso do aluno criar um sinónimo com base em outro sinónimo, e nesse caso o tipo originário do primeiro sinónimo é tido como sinónimo do tipo original.

Exemplo deste caso:

```
1 typedef idade int;      # criação do tipo idade equivalente a int
2 typedef anos idade;     #( criação do tipo anos equivalente a idade,
3                            portanto equivalente a int #)
```

Em termos de sintaxe, foi decidido utilizar a palavra-chave **typedef** que significa literalmente definir um tipo. Em termos da ordem da instrução, foi decidido ter a ordem da palavra-chave **typedef** seguido do nome do novo tipo e só então o tipo que o origina. A razão por detrás desta escolha prende-se que o aluno define um novo tipo baseado num que existia anteriormente. Outra razão foi, como se pode ver no ponto seguinte, em tipos estruturados, que também utiliza a instrução **typedef**, que o aluno define o nome da nova estrutura (um novo tipo) e depois define a organização deste novo tipo. Estas decisões foram tomadas para harmonizar a definição de um tipo simples e de um tipo estruturado, apesar de no caso de definir um tipo simples ter uma ordem trocada em relação à linguagem C, ao manter-se um modo idêntico em ambos os casos simples e estruturado, pensamos que seja uma mais valia para o aluno.

3.1.8 Structs

Através do **typedef** o aluno pode criar um tipo composto por valores de vários tipos, uma estrutura. O aluno dá o nome à estrutura e a todos os membros da estrutura na sua definição, e no caso de *arrays*, estes têm de ter definidos todos os tamanhos das dimensões.

Exemplo de uma definição de uma estrutura:

```
1 typedef ponto {
2    int x;
3    int y;
4 };
```

Para inicializar uma variável de uma estrutura há três possibilidades. Uma possibilidade é definir uma variável e inicializar alguns ou todos os membros e pela ordem que pretenda (indicando explicitamente os membros), através de chavetas. Exemplo:

```
1 ponto pontoA = {.y = 10, .x = 5};  # definição e inicialização do ponto
```

A segunda possibilidade é a inicialização da variável membro a membro (de notar que o acesso a um membro não inicializado é um erro). Exemplo:

```
1 ponto pontoB;
2 pontoB.x = 5;
3 pontoB.y = 10;
```

Num terceiro caso, através da afetação de outra variável da mesma estrutura. Exemplo:

```
1 ponto pontoC = {.x = 2, .y = 12};
2 ponto pontoD = pontoC;
```

O caso dos *arrays* como membro é um caso especial. A utilização de *arrays* dentro de estruturas obriga a uma total definição das dimensões e quantidade de elementos de cada uma das dimensões, pelo que todas as variáveis da estrutura terão o mesmo tamanho. Os *arrays* que fizerem parte de uma estrutura e não se inicializarem juntamente na definição de uma variável da estrutura, têm de ser inicializados posição a posição.

3.1.9 Memória dinâmica

A linguagem Temple permite o utilizador obter memória para valores no decorrer dum programa em tempo de execução, dinamicamente. O utilizador pode criar uma referência de um determinado tipo (já existente) para uma variável. É necessário ao utilizador alocar a memória para um valor através do operador **new**. Para o utilizador inicializar a variável de uma referência, este necessita de desreferenciar o nome da variável através do operador **$**. O utilizador pode ainda afetar uma referência criada com outra referência já existente. As referências podem também ser apagadas com a palavra-chave **delete**.

Exemplos de memória dinâmica:

```
1  ref int ref_var1;
2  ref_var1 = new int;
3
4  ref int ref_var2 = ref_var1;
5  int abc = 10;
6  ref_var2$ = abc;
7
8  #ref_var1$ == 10
9
10 typedef structP {int x};
11 ref structP varX = new structP;
12 varX$.x = 15;
13
14 delete varX;
```

Em termos sintáticos, foi decidido utilizar a palavra **ref** de modo ao aluno saber que é uma referência, e não a utilização de símbolos que podem complicar a representação das referências. Do mesmo modo, a alocação de memória foi simplificada quando comparada com certas linguagens de programação, não necessitando o aluno de indicar a quantidade de *bytes* de memória que necessita para com base no tamanho do valor para onde a referência aponta. O aluno utiliza o operador **new** de modo a alocar o espaço para um valor de um determinado tipo, tratando a implementação das questões da memória necessária. Este tipo de instruções também teve influência de linguagens como Java onde se criam objetos, tendo uma representação em parte idêntica. Já a desreferenciação quis-se simples e com a utilização de um símbolo para a sua representação. Foi decidido utilizar o símbolo **$** devido a ser um símbolo que é utilizado para representar a moeda corrente em muitos países, um modo de representar valor dado a algo, e assim, na linguagem Temple, é um símbolo para devolver o valor de uma referência. Por fim, o identificador **delete** foi escolhido para opor o identificador **new** para libertar o espaço alocado para onde a referência aponta.

3.1.10 Identificadores

Os identificadores na linguagem Temple (nomes de constantes, variáveis, funções, *labels* e novos tipos) são definidos de modo semelhante a linguagens como C e Java, em que podem começar por uma letra minúscula ou maiúscula, seguida por uma sequência opcional composta por letras, *underscores* e algarismos. Para além disso, a linguagem Temple diferencia nomes de variáveis entre caracteres minúsculos e maiúsculos, e assim o identificador **ABC** é diferente de **abc**. A diferença para linguagens como C e Java é a não utilização de *underscores* para iniciar o nome de um identificador, isto porque os *underscores* estão reservados para o primeiro símbolo das variáveis da implementação de Temple e, assim, não existir alguma possibilidade de gerar problemas.

3.1.11 Variáveis e constantes

Em Temple podem-se declarar variáveis e constantes.

As constantes são visíveis em todo o programa (com atenção no contexto global, em que a ordem das constantes interessa, por exemplo, quando se cria uma constante com base em outra constante) e são definidas apenas no contexto global do programa. As constantes têm a obrigatoriedade de serem inicializadas na sua definição e podem ser inicializadas através de outras constantes, valores literais e/ou operações com expressões de literais ou constantes. As constantes não contêm explicitamente o tipo na definição.

Exemplo de uma constante:

```
1 const MAX_VALUE = 10;
```

As variáveis podem ser definidas em qualquer contexto (global ou local) e não necessitam de ser inicializadas na sua definição (apesar de poderem ser). Pode-se criar uma variável com um nome já utilizado por outra variável, desde que esta nova variável seja definida num contexto mais interno, e que ainda não tenha sido definida nesse contexto mais interno. A utilização de uma variável que não esteja inicializada gera um erro.

Exemplo de variáveis:

```
1 int x = 10;
2 void funcao_n() {
3     int x = 2;      # nova definição de x num novo contexto de função
4     int y = x;      #( valor de y é do x do contexto mais interno,
5                       neste caso y = 2 #)
6 }
```

Foi decidido utilizar a palavra-chave `const` para definir uma constante e diferenciar da definição de variáveis. Na definição das constantes o aluno não insere o tipo porque este é inferido do valor da constante. Na definição de variáveis foi ainda pensado em poder-se definir várias variáveis na mesma instrução, utilizando o nome do tipo apenas uma única vez, e foi pensado ainda poder definir desse modo e inicializar os valores dessas mesmas variáveis, mas de modo a evitar criar mais confusão, foi decidido manter a linguagem simples, e assim por instrução só se pode definir e inicializar uma variável.

3.1.12 Funções e passagem de argumentos

As definições de funções (que incluem as funções que retornam um valor e procedimentos) são compostas pelo tipo de retorno, o nome da função, seguido de uma sequência de argumentos (que pode ser vazia) e o corpo (que também pode ser vazio). Os argumentos podem ser de qualquer tipo existente no programa, e são constituídos pelo tipo do argumento, seguido de um nome dado pelo utilizador.

Os argumentos das funções são passados por referência, e assim o aluno pode afetar variáveis diretamente com o nome da variável do argumento. O aluno pode usar valores constantes e literais como argumentos ainda assim, que são tratados na implementação da linguagem Temple, criando uma variável temporária e dando essa variável como argumento da função (algo que o aluno nem chega a ver).

Exemplo de argumentos por referência:

```
1 int a = 10;
2 void altera_valor(int x, int y) {
3     x = y;
4     #( valor do argumento x é alterado
5        por referência com valor do argumento y #)
6 }
7
8 altera_valor(a,30);
9 # após chamada de função, o valor da variável a é 30
```

O único caso a ter em conta é o caso dos *arrays*. Foi decidido que o tamanho das dimensões não faz parte do tipo dos *arrays*, e assim apenas a quantidade de dimensões fazem parte do tipo do *array*. Uma função pode aceitar um array com 3 dimensões, independentemente do tamanho das mesmas. Esta solução poderia gerar problemas no retorno de funções, pois a função poderia retornar *arrays* de tamanhos variados. Uma alternativa

seria fazer com que a linguagem obrigasse a definir o tamanho das dimensões na definição da função, o que faria com que a função apenas aceitasse *arrays* com uma quantidade definida de dimensões e tamanhos por cada dimensão. Esta decisão limitaria muito uma função que, por exemplo, ordenasse os elementos de um `array`, não podendo ordenar arrays de tamanho variável por dimensão. Mesmo com o beneficio de se poder aceitar um *array* de determinado tamanho e de se poder retornar esse *array* com a função, foi decidido que seria melhor o aluno utilizar uma função para vários *arrays* com as mesmas dimensões, e assim afetar os *arrays* por referência. Apesar da não obrigação da definição do tamanho de cada dimensão nos argumentos de uma função, o aluno pode ainda assim definir o tamanho das dimensões que pretender, não estando obrigado a definir o tamanho de todas as dimensões. Desse modo uma função não pode retornar *arrays*.

Exemplo de uma definição de função com um *array* como argumento:

```
1 char nova_funcao(array[,3,,2]char um_array){
2     # array de char com 4 dimensões, 2 com tamanho definido
3     corpo da função
4 }
```

Em termos da forma de uma função, pretendeu-se manter uma forma idêntica à linguagem C, indicando o tipo de retorno da função, o nome da função e os argumentos, seguido do corpo. No corpo, o aluno tem a liberdade de inserir variáveis locais no local que pretender, não sendo obrigado a colocá-las no início do corpo. Do mesmo modo, o aluno pode colocar a instrução `return` no lugar que pretender, não sendo obrigado a colocar no final da função.

3.1.13 Chamadas de função

Na chamada de funções, a quantidade e a ordem dos argumentos tem de ser a mesma da definição formal da função. Existem dois tipos de chamada de função, a função que retorna um valor de determinado tipo e o procedimento que não retorna nada (`void`). Se a função retornar algo, a chamada de função tem de aparecer obrigatoriamente num contexto onde possa aparecer uma expressão. Caso seja um procedimento, a chamada de procedimento não pode ser utilizada num contexto onde possa aparecer uma expressão, pois é uma instrução. Deste modo, o valor retornado de uma função nunca pode ser descartado, algo referenciado como problema na Secção 2.8.1.

Exemplo de chamadas de função:

```
1  void teste1() {}              # definição do procedimento teste1
2  int teste2() { return 10; }   # definição da função teste2
3
4  teste1();                    # chamada do procedimento sem afetação
5  int a = teste2();            #( chamada de função com afetação na
6                                   definição de uma variável #)
7
8  int b = teste1();            #( procedimentos não devolvem valor,
9                                  portanto é um erro #)
10
11 teste2();                    #( chamada de função obriga
12                                 a que o valor retornado
13                                 seja utilizado,
14                                 como tal é um erro #)
```

3.1.14 Espaço de nomes e *scopes*

Na linguagem Temple, o espaço de nomes foi algo muito tido em conta. Uma das propriedades é a existência de um espaço de nomes com determinadas características. Existe um espaço de nomes que inclui os nomes de funções, novos tipos, variáveis, constantes, campos de uma estrutura, argumentos de funções e ainda de nomes das *labels* para ciclos.

O espaço de nomes é regido pelas seguintes regras:

- Funções, novos tipos e constantes só podem ser definidos no *scope* mais global do programa.
- Um nome pode ser reutilizado numa nova definição noutro *scope*, desde que não sejam de entidades da linguagem diferentes, ou seja, um nome utilizado por uma função não pode ser utilizado por uma variável, mesmo que num *scope* mais interno (mesmo que de outra função, por exemplo). Isto faz com que o utilizador possa utilizar entidades globais em qualquer ponto do programa e nunca os possa redefinir, sendo a única exceção as variáveis globais, que podem ser redefinidas noutro *scope* mais interno. Isto foi uma decisão para ajudar os alunos, tendo também como base um problema encontrado que é a definição de uma variável no *scope* mais interno com o nome da própria função, algo possível em C. Esse problema faz com que a recursão seja impossível de ser executada nesse *scope*, e sendo a recursão um dos maiores problemas dos alunos, como visto na Secção 2.5.1, foi algo que foi muito pensado.
- Com base no ponto anterior, uma variável pode ser definida com o mesmo nome já utilizado por outra variável de *scope* superior. Exemplo:

  ```
  1 int a = a;
  ```

  Este exemplo em C iria definir a variável **a** e ir buscar o valor no seu novo local de memória, indo buscar qualquer informação que estivesse nesse local de memória. O objetivo nesta linguagem é criar uma variável **a** com base numa variável **a** dum *scope* superior sem alterar o valor no *scope* superior.
- Novos *scopes* são criados nos seguintes casos: Corpo de uma função (que inclui a declaração dos argumentos), no ciclo **while..do**, ciclo **repeat..until**, ciclo **for**, **switch..case** e **if..then..else**. Exemplo:

  ```
  1 int a = 10;
  2 void uma_funcao(int a) {
  3 while true do
  4     int a = 20;
  5 endwhile
  6 }
  ```

- Os nomes de *labels* são um caso especial, em que o nome de uma *label* só pode aparecer uma vez dentro de uma função, mesmo que em *scopes* diferentes. Logo uma *label* **a** só pode ser utilizada uma vez numa função **x**, mesmo que se tente definir num *scope* mais interno. Isto é de modo ao aluno poder ter, por exemplo, vários *scopes* cada um com uma *label* diferente e poder sair com a instrução **break** de vários *scopes* de uma só vez. Se o aluno pudesse ter uma *label* **a** e, dentro de um *scope* mais interno, outra *label* **a**, e existisse a instrução **break a**, não se saberia qual a *label* **a** a que o **break** se referiria, ou apenas poderia sair do *scope* mais interno.
- Os nomes de *labels* têm em atenção que não podem ser utilizados por qualquer outro tipo de entidades dentro de uma função, independentemente se encontrarem-se no mesmo *scope* ou não.
- O único local em que podem ser utilizadas *labels* é na definição de ciclos (**for**, **repeat..until** e **while..do**) e na instrução **break** dentro dos ciclos (único local onde se pode utilizar a instrução **break**).

Com estas regras, tenta-se aliviar um pouco o problema dos espaços de nomes.

3.1.15 Outras instruções

As instruções que foram incluídas na linguagem Temple são consideradas instruções fundamentais numa linguagem de programação. Instruções de afetação, condicionais e ciclos que de certo modo não se diferenciam muito das instruções semelhantes em outras linguagens mas que, nas suas diferenças, tentam ajudar os alunos a evitar certos erros que costumam acontecer.

Afetação

O utilizador pode afetar variáveis ao longo do programa, copiando o valor de uma expressão. Tem de existir compatibilidade de tipos na afetação. O único tipo que o utilizador não pode afetar diretamente são os *arrays* (apesar de poder afetar posição a posição). Uma estrutura pode ser afetada por outra, desde que estas sejam do mesmo tipo, não permitindo afetação por uma estrutura que seja de um tipo diferente, mesmo que a sua constituição seja totalmente idêntica. A afetação é ainda possível entre tipos sinónimos criados através da instrução `typedef`. O utilizador não pode afetar funções, constantes, nem afetar a variável de controlo do ciclo `for`. De referir ainda que o aluno pode afetar posições de *strings*, afetar referências e ainda afetar o valor referenciado pela referência através de desreferenciação. Em termos sintáticos foi ainda pensado utilizar uma seta (←) a indicar que o valor da expressão da afetação é copiado para uma variável, muitas vezes utilizado em pseudo-código, mas devido à não existência do símbolo nos teclados comuns, foi decido manter o que é habitualmente utilizado em outras linguagens, e assim o símbolo de igualdade (=) foi utilizado na afetação nesta linguagem.

Exemplos de afetação:

```
1  int a;
2  a = 3;
3
4  array[2,1]float b;
5  b[0,0] = 4.21;
6
7  ref char c = new char;
8  c$ = 'x';
9
10 string d = "uma string";
11 d<<2>> = 'o';
```

If..then..else

A instrução `if..then..else` é uma instrução condicional da linguagem Temple. A instrução contém obrigatoriamente uma expressão booleana que é avaliada. Caso o valor da expressão avaliada seja `true`, é executada a instrução ou sequência de instruções após a palavra-chave `then`, caso o valor da expressão avaliada seja `false`, é executada a instrução ou instruções a seguir da palavra-chave `else`. O utilizador pode ainda inserir mais que uma alternativa caso a avaliação da expressão booleana seja `false`, ou seja, o utilizador pode inserir a instrução `elif` e avaliar outra expressão booleana caso a expressão avaliada no `if` seja `false`. Do mesmo modo, o utilizador pode inserir a instrução `elif` as vezes que pretender, desde que se encontrem após o `if` inicial. A cláusula `else` não é obrigatória. O utilizador não é obrigado a executar instruções após as cláusulas `if`,

elif nem **else**. A instrução **if..then..else** tem de terminar obrigatoriamente com o terminador **endif**. Cada bloco de instruções, seja este após a palavra-chave **then** ou **else**, cria um novo *scope*, onde novas variáveis podem ser criadas com nomes utilizados anteriormente por outras variáveis (ter em atenção as regras dos espaços de nomes referidas anteriormente).

Em termos sintáticos, foi decido utilizar as palavras-chave **if** e não palavras como **cond**, por exemplo, de modo a manter a similaridade com outras linguagens e não causar mais confusão entre os alunos. Após a palavra-chave **if** encontra-se a expressão que é avaliada, e foi decido não utilizar parêntesis porque a expressão já se encontra entre duas palavras-chave, as palavras-chave **if** e **then**. A palavra-chave **then** existe para demonstrar que a seguir à avaliação da expressão booleana se encontram as instruções a executar caso a expressão avaliada seja verdadeira. A palavra-chave **then** encontra-se após expressões que são avaliadas, e assim aparece associado a cada **if** e **elif**. A palavra-chave **elif** foi escolhida com base na linguagem Python, também pela representação idêntica da instrução num todo, mantendo uma aparência idêntica. A instrução pode ainda conter a palavra-chave **else** que contém as instruções a executar caso nenhuma das expressões booleanas avaliadas anteriormente tenham sido verdadeiras. Por fim, a instrução termina sempre com a palavra-chave **endif**. Esta escolha prendeu-se com a utilização em demasia de chavetas em linguagens como Java, C ou C++ e assim foi escolhido de modo a manter uma linguagem mais "limpa", um pouco à imagem de Python. A utilização da palavra-chave **endif** ajuda, também quando existem instruções **if..then..else** encadeadas, a identificar onde termina a instrução, sem ter uma quantidade enorme de chavetas em linguagens como Java, C e C++, tanto de abertura como de fecho. Outro motivo para utilizar o terminador **endif** é eliminar ambiguidades no caso de **ifs** encadeados.

Exemplos da instrução **if..then..else**:

```
1  int a = 2;
2  if a == 2 then
3     int a = 3;
4  elif a == 3 then
5     int a = 2;
6  elif a == 4 then
7     int a = 4;
8  else
9     a = 0;
10 endif
11
12 bool x = true;
13 if x then
14    x = false;
15 elif !x then
16 endif
```

Switch..case

A instrução **switch..case** é uma instrução condicional da linguagem Temple. A instrução **switch..case** contém uma expressão do tipo **int** ou **char** após a palavra-chave **switch**. O utilizador pode depois inserir a quantidade pretendida de **cases**, onde inclui um literal **int** ou **char** que é comparado com o valor avaliado após a instrução **switch**. A palavra-chave **case** pode ser ainda seguida de uma sequência de valores literais (do tipo **int** ou **char**, todos do mesmo tipo), ou ainda um intervalo de valores literais (também do tipo **int** ou **char**). Os valores literais não podem ser repetidos, mesmo nos casos de uma sequência ou do intervalo de valores. Se um valor de um **case** for igual ao valor avaliado após o **switch**, o conjunto de instruções (opcionais) são executadas e não é executada mais nenhuma instrução (de qualquer outro **case**). Existe ainda a

palavra-chave **default**, em que as instruções são executadas caso nenhum **case** tenha dado *match* com o valor avaliado no **switch**. A palavra-chave **default**, que acaba por ser um **case** sem valor literal, só pode existir após todos os **cases**, mas não é uma instrução obrigatória. Cada bloco de instruções de cada **case** cria um novo *scope*, onde novas variáveis podem ser criadas com nomes utilizados anteriormente por outras variáveis (ter em atenção as regras dos espaços de nomes referidas anteriormente). A instrução **switch..case** tem de terminar obrigatoriamente com o terminador **endswitch**.

Foi decidido incluir a instrução **switch..case** na linguagem Temple como uma alternativa possível à instrução **if..then..else**. A instrução teve como base a instrução da linguagem C, apesar de algumas alterações terem sido feitas. Uma das grandes alterações foi não se necessitar de utilizar a instrução **break** no final de cada **case**. Isto foi decidido para que, caso haja um *match* entre a expressão do **switch** e de um **case**, se executem apenas as instruções desse **case**, não podendo executar instruções de outro **case**. Um dos grandes motivos por detrás desta decisão é que muitas vezes os alunos podem esquecer-se da instrução **break** (que quererão utilizar de certeza) e o programa poderá executar mais que um **case**. Isto obriga também a que o aluno faça uma instrução mais correta, caso pretenda que mais que um **case** seja executado, juntando os valores como uma sequência ou intervalo, algo também possibilitado nesta linguagem. Para além disso, uma das decisões tomadas foi que um valor de um **case** não se pode encontrar em mais que um **case**. Deste modo, um valor contido num intervalo não pode estar noutro **case**, por exemplo.

Exemplos de **switch..case**:

```
1  int x = 20;
2  int y;
3  switch x
4      case 0:
5          y = 0;
6      case 1,2,3:
7          y = 1;
8      case 4..10:
9          y = 2;
10     default:
11         y = 3;
12 endswitch
13
14 char a = 'a';
15 int b = -1;
16 switch a
17     case 'a'..'z':
18         b = 0;
19     case 'A'..'Z':
20         b = 1;
21     case '0'..'9':
22         b = 2;
23 endswitch
```

While..do

A instrução **while..do** é uma instrução de ciclo da linguagem Temple. A instrução contém obrigatoriamente uma expressão booleana que é avaliada. Caso o valor da expressão avaliada seja **true**, é executada a instrução ou bloco de instruções após a palavra-chave **do**, e avalia-se novamente a expressão booleana e executam-se as instruções a seguir da palavra-chave **do** até que o valor da expressão seja **false**. Quando o valor da expressão avaliada for **false**, as instruções seguidas da palavra-chave **do** não são executadas e a execução

passa para a instrução seguinte à instrução **while..do**. O bloco de instruções desta instrução cria um novo *scope*, onde novas variáveis podem ser criadas com nomes utilizados anteriormente por outras variáveis (ter em atenção as regras dos espaços de nomes referidas numa secção anterior). De referir ainda que pode ser dado um nome a uma instrução **while..do** através de uma *label*, de modo a mais tarde, com a instrução **break**, poder sair do ciclo (sem ser através da avaliação da expressão booleana). A instrução **while..do** tem de terminar obrigatoriamente com o terminador **endwhile**.

Em termos sintáticos foi decido utilizar a palavras-chave **while** de modo a manter a similaridade com outras linguagens. Após a palavra-chave **while** encontra-se a expressão que é avaliada e foi decidido não utilizar parêntesis porque a expressão já se encontra entre duas palavras-chave, as palavras-chave **while** e **do**. A palavra-chave **do** existe para delimitar o sítio onde começa o corpo do ciclo, que contém as instruções a executar caso a expressão avaliada seja verdadeira. Por fim, a instrução termina sempre com a palavra-chave **endwhile**. Novamente, a razão por detrás desta escolha é a utilização de muitas chavetas em outras linguagens, como Java, C e C++. A utilização da palavra-chave **endwhile** ajuda também quando existem instruções **while** encadeadas, ajudando a identificar onde termina a instrução, sem ter uma quantidade enorme de chavetas, tanto de abertura como de fecho. Por outro lado também, a diferenciação da palavra-chave **endwhile** e de **endif** que em outras linguagens seriam apenas chavetas, ajuda à diferenciação de onde termina uma instrução e outra, para além de mais palavras-chave em outras instruções nos pontos seguintes.

Exemplos da instrução **while..do**:

```
1 int x = 0;
2 while x<10 do
3    x = x + 1;
4 endwhile
5
6 uma_label: while true do
7    break uma_label;
8 endwhile
```

Repeat..until

A instrução **repeat..until** é uma instrução de ciclo da linguagem Temple. A instrução contém obrigatoriamente uma expressão booleana que é avaliada. A instrução ou instruções (opcionais) a seguir da palavra-chave **repeat** são executadas uma vez até se chegar à expressão booleana após a palavra-chave **until**. Caso o valor da expressão seja **false**, é executada a instrução ou conjunto de instruções após a palavra-chave **repeat** novamente, e volta-se a avaliar a expressão **booleana** e executam-se as instruções a seguir da palavra-chave **repeat** até que o valor da expressão seja **true**. Quando o valor da expressão avaliada for **true**, as instruções seguidas da palavra-chave **repeat** não são executadas, e a execução passa para a instrução seguinte à instrução **repeat..until**. O bloco de instruções desta instrução cria um novo *scope*, onde novas variáveis podem ser criadas com nomes utilizados anteriormente por outras variáveis (ter em atenção as regras dos espaços de nomes referidas anteriormente). De referir ainda que pode ser dado um nome a uma instrução **repeat..until**, de modo a mais tarde, com a instrução **break**, poder sair do ciclo (sem ser através da avaliação da expressão booleana). A instrução **repeat..until** tem de terminar obrigatoriamente com o terminador **endrepeat**.

Foi decidido utilizar a instrução **repeat..until** como uma alternativa à instrução **while..do** em que as instruções do corpo do ciclo são executadas pelo menos uma vez. Em termos sintáticos, mantém-se a simplicidade das instruções anteriores e, do mesmo modo, evitam-se chavetas como noutras linguagens. A expressão avaliada nesta instrução não se encontra entre parêntesis (apesar de poderem ser utilizadas como em qualquer outra expressão). Foi decido utilizar a palavra-chave **endrepeat** de modo a manter a consistência com

as outras instruções, apesar de ter sido colocada a hipótese desta instrução não utilizar esse terminador. De referir ainda que ao contrário da instrução **while..do**, em que as instruções do ciclo são executadas caso a expressão seja verdadeira, na instrução **repeat..until**, as instruções (para além da primeira vez que as instruções são sempre executadas) só são executadas enquanto a expressão for falsa. A instrução foi escolhida com base na definição na linguagem Pascal.

Uma das escolhas tomadas foi assim utilizar a instrução **repeat..until** ao invés de uma instrução como **do..while** de linguagens como C ou Java. O grande motivo por detrás desta decisão foi incorporar uma instrução que executa as instruções do corpo do ciclo pelo menos uma vez, mas que seja de certo modo diferente da instrução **while..do**. Assim, pretende-se dar uma outra visão aos alunos, com um ciclo que é executado quando a expressão avaliada é falsa e incorporar um instrução utilizada na linguagem Pascal.

Exemplos da instrução **repeat..until**:

```
1 int x = 0;
2 repeat
3    x = x + 1;
4 until x < 10 endrepeat
5
6 int y = 0;
7 repeat
8    y = y + 1;
9 until y == 10 endrepeat
```

For

A instrução **for** é uma instrução de ciclo da linguagem Temple. A instrução contém obrigatoriamente um identificador de uma variável de controlo do ciclo do tipo **int** que é definida e inicializada dentro do ciclo **for**, que pode ser utilizada nas instruções (opcionais) do ciclo **for** como uma variável local, mas que não pode ser alterada. A variável é inicializada com uma expressão do tipo **int** e é o valor de partida no ciclo **for**. É também declarada uma expressão em que a variável irá terminar a execução, também do tipo **int**, sendo o valor final inclusivo para execução. O utilizador utiliza a palavra-chave **to**, caso o valor inicial da variável seja menor que o valor final, ou a palavra-chave **downto**, caso o valor inicial da variável seja maior que o valor final. Existe ainda a palavra-chave **step** seguida de uma expressão do tipo **int** que pode indicar o incremento da variável caso se pretenda ter um incremento diferente de uma unidade, com valor sempre positivo, tendo em conta que as palavras-chave **to** e **downto** definem se é um incremento ou decremento. Uma particularidade da linguagem Temple é que as expressões referidas anteriormente são todas avaliadas uma única vez no início do ciclo **for** e, assim, caso uma expressão (ou parte de expressão) das expressões referidas anteriormente seja alterada (por exemplo, uma expressão pode ser uma variável definida anteriormente), essa expressão não é alterada no contexto do ciclo **for**, apesar das partes constituintes dessa expressão serem devidamente alteradas. Desse modo, se um ciclo **for** tiver de início de realizar 10 iterações, não será possível alterar as expressões para que sejam realizadas mais do que 10 iterações, ou alterar a expressão do **step** e a uma dada altura o incremento ser diferente.

O bloco de instruções desta instrução cria um novo *scope*, onde novas variáveis podem ser criadas com nomes utilizados anteriormente por outras variáveis (ter em atenção às regras dos espaços de nomes referidas anteriormente). De referir ainda que pode ser dado um nome a uma instrução **for**, de modo a que mais tarde, com a instrução **break**, poder sair do ciclo (sem ter de realizar todas as iterações). A instrução **for** tem de terminar obrigatoriamente com o terminador **endfor**.

Estas decisões foram tomadas com o intuito de criar um ciclo **for** que permite verificar se o ciclo pode terminar

(com valor inicial menor ou igual que o final no caso de incremento, e com valor inicial maior ou igual que o final no caso de decremento). Pretendia-se também criar um ciclo que não podia ser alterado durante a execução do mesmo, e para isso, os valores iniciais, finais e de `step` são guardados no início da execução do ciclo. O valor inicial não pode ser alterado, pois a variável, como foi referido, é uma variável especial, pois o utilizador não pode alterar o seu valor, mas ao mesmo tempo o seu valor vai sendo alterado com o ciclo (como uma variável). A variável pode assim ser utilizada numa afetação, mas não pode ser afetada. A utilização de `step` é opcional, porque muitas das vezes o incremento ou decremento que se pretende é de uma unidade, e assim o aluno não é obrigado a utilizar o `step` para esses casos (mesmo que o possa fazer). A grande influência para estas escolhas foi a linguagem Pascal.

Exemplos de ciclo `for`:

```
1  int a = 0;
2  for i = 0 to 10 do
3     a = a + i;
4  endfor
5
6  int x = 1;
7  for i = 20 downto 1 step x do
8     x = x + 1;
9  endfor
10
11 #(Nesta instrução o valor de i é decrementado 1 unidade por ciclo
12   O ciclo é executado 20 vezes
13   O valor de x é alterado durante a execução do ciclo,
14   apesar disso o valor de step no início do ciclo era 1,
15   e o step mantém-se 1 até ao fim do ciclo #)
```

3.1.16 Expressões e operações

Na linguagem Temple, as expressões são formadas por variáveis, literais dos tipos básicos e chamadas de funções. Em termos de operadores entre expressões, a linguagem dispõe de operadores lógicos (`or`, `and`, `not`), de operadores de comparação (`<` (menor que), `<=` (menor ou igual que), `>` (maior que), `>=` (maior ou igual que), `==` (igual a) e `!=` (diferente de)) e de operadores aritméticos (`+` (adição), `-` (subtração), `*` (multiplicação), `/` (divisão de números de vírgula flutuante), `//` (divisão inteira) e `%` (resto da divisão inteira)). Existem ainda operadores unários para números positivos (`+`) e o simétrico de um número (`-`). Os operadores binários são escritos de maneira infixa, e os operadores unários de maneira prefixa. As operações têm o nível de precedência utilizado matematicamente. Em termos de verificação de erros no sistema de tipos, neste caso nas operações, o tipo de cada operando tem de ser compatível um com o outro. Existe o caso da conversão do tipo `int` para `float` em determinados casos. Na comparação de referências há ainda o cuidado de verificar se as referências são para o mesmo tipo, ou ainda no caso de existirem referências de referências, que estas contêm o mesmo nível de profundidade. Existe ainda verificação de *overflow* de modo a evitar que resultados inesperados aconteçam, enviando uma mensagem de erro caso isso aconteça, ao contrário de certas linguagens em que problemas de *overflow* acontecem e o programa executa normalmente.

3.1.17 Input e output para ecrã

Para o *input* e *output* para ecrã, o utilizador tem disponíveis várias funções predefinidas da linguagem Temple. Para o *input* o utilizador tem uma função chamada `input` que tem como argumento uma sequência de

variáveis onde o utilizador pretende guardar os valores que vai inserir. Os valores inseridos são separados por espaços de modo a que o programa saiba separar os valores inseridos.

Para o *output*, o utilizador tem uma função chamada **output** que tem como argumento uma sequência de expressões. Quando imprimidas, os valores são separados por um espaço.

Exemplo de *input* e *output*:

```
1 int a;
2 char b;
3 input(a, b);      #guarda os valores inseridos nas variáveis
4                   #exemplo: 10 s
5
6 int c = 30;
7 char d = 'y';
8 output(c, d, 10);      # imprime a mensagem: 30 y 10
```

3.1.18 Input e output para ficheiros

Em termos de *input* e *output* de ficheiros, existem dois tipos de operações que o aluno pode fazer, um é o modo de texto para ficheiro e outro é o modo de estruturas.

Para trabalhar com ficheiros, o aluno dispõe de várias funções predefinidas na linguagem Temple, que trabalham com o tipo primitivo **file**.

Para inicializar ou afetar uma variável do tipo **file**, o aluno dispõe de duas funções, a função **open** e a função **create**. A função **open** abre um ficheiro que já exista numa dada localização, caso exista, senão cria o ficheiro nessa localização. No caso do ficheiro ser aberto, este mantém a informação que existia anteriormente no ficheiro, podendo acrescentar ou alterar a informação posteriormente. No caso do ficheiro ser criado, este é criado sem qualquer informação. A função **create** cria um ficheiro numa dada localização, independentemente de já existir um ficheiro nessa localização ou não. Caso já exista um ficheiro nessa localização, este é apagado e substituído por um novo ficheiro vazio, com o mesmo nome.

Em ambos os tipos de abertura de ficheiros, o aluno começa a trabalhar do início do ficheiro, necessitando de alterar a posição caso pretenda trabalhar noutra posição do ficheiro.

Em termos de sintaxe, ambas as funções funcionam de modo idêntico, sendo ambas uma função com três argumentos, uma *string* com a localização do ficheiro, uma *string* com a operação que se pretende fazer com o ficheiro (de escrita, leitura ou ambas) e uma *string* com o tipo de abertura que se faz (se é uma abertura para ficheiros de texto ou de estruturas). Para os tipos de leitura, foi decidido incluir as *strings*: **"read"** para fazer apenas leitura, **"write"** para fazer apenas escrita e **"readwrite"** para fazer leitura e escrita. Para os tipos de abertura, foi decidido incluir as *strings*: **"text"** para trabalhar com texto e **"binary"** para trabalhar com ficheiros que contêm apenas um tipo de estrutura. O tipo de estrutura é gerido pelo aluno, que pode utilizar o tipo de estrutura que pretender com o ficheiro (com tipos definidos pelo aluno inclusive).

Exemplo de abertura e criação de ficheiros:

```
1 file novo_ficheiro;
2 novo_ficheiro = open("../pasta/ficheiro1", "readwrite", "text");
3
4 file novo_ficheiro2 = create("../pasta/ficheiro2", "read", "binary");
```

Para fazer *output* o aluno utiliza um procedimento chamado **write**. Este procedimento tem como argumentos

o ficheiro do tipo **file** e uma sequência de expressões entre vírgulas.

Para fazer *input* o aluno utiliza um procedimento chamado **read**. Este procedimento tem como argumentos o ficheiro do tipo **file** e uma sequência de variáveis entre vírgulas.

Existe ainda outro procedimento para *input* que é **readline**. Este procedimento funciona de modo idêntico ao procedimento **read**, mas este procedimento lê uma frase do ficheiro como uma *string* de texto até encontrar o carácter **\n** que representa o fim de linha. O carácter é consumido (apesar de não ser guardado na variável), de modo a que a leitura seguinte comece após o carácter **\n**.

Para fechar o ficheiro, o utilizador utiliza o procedimento **close** com a variável do ficheiro como argumento.

Exemplo de *input* e *output* de ficheiros:

```
1  novo_ficheiro3 = open("../pasta/ficheiro3", "readwrite", "text");
2
3  write(novo_ficheiro3, "Uma história para se contar.\n");
4  # procedimento que escreve no ficheiro
5
6  string abc;
7  read(novo_ficheiro3, abc);
8  # procedimento que lê uma palavra apenas para a variável abc
9
10 string def;
11 readline(novo_ficheiro3, def);
12 # procedimento que lê uma frase para a variável def
```

Existem ainda algumas ideias que não ficaram bem definidas para outras funções ou procedimentos para o aluno utilizar com ficheiros. Um desses procedimentos é o procedimento **goto** que tem dois argumentos, o ficheiro do tipo **file** e uma expressão do tipo **string**. A ideia por detrás deste procedimento é o aluno poder ir para o início do ficheiro ou para o fim, com os comandos **"begin"** e **"end"**.

Outro procedimento pensado foi o procedimento **seek** que tem dois argumentos, o ficheiro do tipo **file** e uma expressão do tipo **int**. A ideia é permitir o aluno andar uma quantidade (indicada por um valor inteiro no segundo argumento) que pretende avançar ou recuar a partir da posição atual. Este procedimento seria utilizado em modo de estruturas.

Ainda foram pensadas algumas funções, tais como **eof** e **eoline**, com um argumento que é o ficheiro do tipo **file**. Estas funções retornariam um valor booleano **true** quando, na leitura de um ficheiro (**eof** para ambos os tipos de abertura, e **eoline** apenas para ficheiros de texto) se chegasse ao fim do ficheiro ou ao fim de uma linha, respetivamente.

Por fim, uma função que ficou mais em aberto é a função **error**, que tem apenas um argumento que é o ficheiro do tipo **file**. A ideia seria retornar um valor inteiro correspondente a um tipo de erro (ou junção de erros) que possa ocorrer no trabalho com ficheiros (problemas de acesso a ficheiro, abertura, escrita ou leitura, entre outros). Esta função e as anteriores ficaram para trabalho futuro, necessitando de mais ajustes e/ou trabalho junto de alunos para ver quais os melhores passos a dar.

Exemplos possíveis para estas funções:

```
1  novo_ficheiro4 = open("../pasta/ficheiro4", "readwrite", "text");
2
3  goto(novo_ficheiro4, "begin");
4  string abc;
5  while !eoline(novo_ficheiro4) do
6     read(novo_ficheiro4, abc);
7     output(abc, "\n");
8  endwhile
9
10 #(o aluno abre um ficheiro de texto e vai para o início do ficheiro
11   enquanto não chegar ao fim de uma linha, lê string a string
12   para a variável abc e imprime por linha cada palavra para o ecrã #)
13
14
15 novo_ficheiro5 = open("../pasta/ficheiro5", "readwrite", "binary");
16
17 goto(novo_ficheiro5, "end");
18 if eof(novo_ficheiro5) then
19    write(novo_ficheiro5, 10);
20    seek(novo_ficheiro5, -1);
21 endif
22
23 #(o aluno abre um ficheiro binário de inteiros e vai para
24   o fim do ficheiro. Se estiver no fim do ficheiro (que está),
25   escreve o valor 10 no ficheiro e move o apontador de escrita
26   para a posição onde escreveu o valor 10, ou seja,
27   escreveu um valor numa posição X,
28   o apontador moveu para a posição X+1, e com o procedimento seek,
29   o apontador voltou para a posição anterior, X #)
30
31 int x = error(novo_ficheiro5);
32
33 #(variável x fica com um valor inteiro que corresponde
34   a um determinado erro conforme uma lista de erros
35   que possa ser definida posteriormente #)
```

De notar que uma possível lista com os erros não foi definida e fica como uma possibilidade de trabalho futuro.

3.1.19 Comentários

Na linguagem Temple existem duas opções para a inclusão de comentários. O utilizador pode querer comentar apenas uma linha, utilizando o carácter **#**, comentando toda a linha a partir da utilização do símbolo de comentário. Se o utilizador pretender comentar mais que uma linha, o comentário começa pelos caracteres **#(** e termina quando encontrar os caracteres **#)**. De referir que uma questão que se quis ter em atenção foi o utilizador poder ter comentários de uma linha dentro de comentários de várias linhas, isto é, caso o utilizador tenha um comentário de uma linha dentro do comentário de várias linhas, não é gerado um erro (e assim, caso o utilizador pretenda retirar os comentários de várias linhas, permite manter uma linha em específico comentada). Do mesmo modo, o utilizador pode ter comentários de várias linhas dentro de outros comentários de várias linhas (sendo aplicado o comentário mais abrangente, e assim pode ter a abertura de um comentário de várias linhas a terminar com primeiro fecho que encontra, independentemente se tiver mais aberturas durante o comentário mais abrangente).

3.1.20 Bibliotecas

Em termos de bibliotecas da linguagem Temple, o que ficou decidido na realização do trabalho foi que a linguagem contém funções predefinidas para alguns casos como *input* e *output* para ecrã, e algumas funções específicas para *arrays*, *strings*, entre outras. O utilizador não tem de incluir essas funções, pois estas estão disponíveis de imediato, evitando o problema referido na Secção 2.8.1 do caso do C e C++ conterem as diretivas `#include`. A possibilidade de a linguagem vir mais tarde a incluir outras bibliotecas ficou para trabalho futuro e no momento o aluno implementa o código apenas em um ficheiro.

3.2 Definição formal da linguagem

Nesta secção é apresentada a definição formal da linguagem Temple, com atenção aos casos específicos e detalhes não apresentados na secção anterior. Pretende-se dar assim uma visão mais técnica da linguagem criada.

3.2.1 Tipo inteiro, literais e operações

Inteiros

O tipo `int` é inteiro de 32 bits com sinal e, como tal, tem como valor literal possível de `-2,147,483,648` a `2,147,483,647`, de $-(2^{31})$ a $2^{31} - 1$

Operações de inteiros

As operações têm verificação de *overflow* e divisão por 0, e são as seguintes:

Operadores Binários	Operações
`==`, `!=`, `<`, `<=`, `>`, `>=`	comparações
`+`, `-`	adição e subtração
`*`, `//`, `%`	multiplicação, divisão inteira e resto da divisão inteira

Operadores Unários	Operações
`+`	positivo
`-`	simétrico

Literais de inteiros

- Inteiro decimal

  Os literais inteiros decimais são definidos pela expressão regular:

  `[0-9]+`

- Inteiro hexadecimal

Os literais inteiros hexadecimais são definidos pela expressão regular:

`0x[0-9a-fA-F]+`

- Inteiro octal

  Os literais inteiros octais são definidos pela expressão regular:

  `0o[0-7]+`

- Inteiro binário

  Os literais inteiros binários são definidos pela expressão regular:

  `0b[0-1]+`

3.2.2 Tipo de números de vírgula flutuante, literais e operações

Números de vírgula flutuante

O tipo `float` é número de vírgula flutuante com dupla precisão, seguindo o *standard* IEEE 754-2008 e, como tal, tem um formato 64-bit de base 2 chamado binary64. O *standard* especifica binary64 com: 1 bit de sinal, 11 bits de expoente e 52 bits de fração. Uma alteração ao *standard* é a não existência de *NaN* e infinito.

Operações de números de vírgula flutuante

Operadores Binários	Operações
==, !=, <, <=, >, >=	comparações
+, −	adição e subtração
*, /	multiplicação, divisão de vírgula flutuante

Operadores Unários	Operações
+	positivo
−	simétrico

Literais de números de vírgula flutuante

- Número de vírgula flutuante decimal

  Os literais de vírgula flutuante decimal são definidos pela expressão regular:

  `[0-9]+.[0-9]+([eE][-+]?[0-9]+)?`

3.2.3 Tipo booleano, literais e operações

Booleanos

O tipo `bool` é um tipo lógico, e como tal só pode ter dois valores (`true` e `false`).

Operações de booleanos

Operadores Binários	Operações
`or`	OU lógico
`and`	E lógico
`==`, `!=`	comparações

Operadores Unários	Operações
`not`	negação

Literais de booleanos

- Booleano

  Os literais de booleanos têm dois valores possíveis:

  `true`

  `false`

3.2.4 Tipo de caracteres, literais e operações

Caracteres

O tipo `char` é um tipo para os caracteres. O *encoding* utilizado é o UTF-8 devido à sua grande utilização.

Literais de caracteres

- Caracteres

  Os literais de caracteres seguem o *encoding* utilizado, UTF-8, e como tal todos os caracteres pertencentes a este *encoding* são válidos como caracteres desta linguagem. Os literais encontram-se entre plicas (`'`), e apenas um carácter é possível encontrar-se entre elas. O carácter barra invertida (`\`) é utilizado para inserir caracteres especiais, entre eles uma nova linha (`\n`), tabulação (`\t`), o carácter plica (`\'`) e no caso de querer representar o carácter de barra invertida (`\\`).

3.2.5 Tipo de strings, literais e operações

Strings

O tipo `string` é um tipo para uma sequência ordenada de caracteres. O *encoding* utilizado é o UTF-8 devido à sua grande utilização. As *strings* têm por *default* 255 caracteres como no exemplo seguinte:

```
string variavel;
```

Caso se pretenda ter *strings* com menos ou mais caracteres, a declaração de uma variável *string* tem a seguinte forma:

```
string<<tamanho_maximo>> variavel;
```

O utilizador consegue saber a quantidade de caracteres de uma determinada string com uma função da linguagem, a função `length(variavel)`, que retorna um inteiro com a quantidade de caracteres no momento. A função `maxlength(variavel)` retorna a quantidade máxima de caracteres que a *string* pode conter.

Para além destas funções, o utilizador pode ainda afetar caracteres de uma *string*, desde que essa posição contenha já um carácter sendo o primeiro carácter a posição 0. Exemplo:

```
variavel<<posicao>> = carácter;
```

Literais de string

- Strings

  Os literais de `string` seguem o *encoding* utilizado, UTF-8, e como tal todos os caracteres pertencentes a este *encoding* são válidos para *strings* desta linguagem (com exceção de uma nova linha que só se pode encontrar no final de uma *string*). Os literais têm de estar de acordo com a definição da variável e ter tamanho compatível com a mesma. Os literais encontram-se entre aspas (`"`). O carácter barra invertida (\) é utilizado para inserir caracteres especiais, entre eles uma nova linha (`\n`), tabulação (`\t`), o carácter aspas (`\"`) e no caso de querer representar o carácter de barra invertida (`\\`).

3.2.6 Tipo void, literais e operações

O tipo `void` é utilizado em funções que não retornam qualquer valor, ou seja, procedimentos.

3.2.7 Arrays

A linguagem Temple suporta a utilização de *arrays*. Os *arrays* podem ser de vários tipos, definidos pelo tipo dos valores que o *array* contém e a quantidade de elementos do *array*. A sequência de itens é um conjunto de expressões que se encontram entre vírgulas delimitadas por chavetas, que por sua vez podem ser delimitados por mais conjuntos de chavetas, conforme a quantidade de dimensões do *array*.

Os *arrays* são definidos de uma das seguintes formas:

- Declarar um *array*

  `array[`$dimensão_1$`,...,`$dimensão_N$`] tipo nome_array;`
  $N \geq 1$

- Declarar e inicializar *array* com valores

  `array[`$dimensão_1$`,...,`$dimensão_N$`] tipo nome_array = {` *sequência de itens* `};`
  $N \geq 1$

Para se aceder a uma posição do *array*, acede-se do seguinte modo:

`nome_array[`K_1`,...,`K_N`]`
$0 \leq K <$ *tamanho da dimensão*

Pode-se utilizar a função função **`size(variavel, numero_dimensao)`**, que tem como argumentos a variável **`array`** e um inteiro correspondente à dimensão. A função retorna o tamanho da dimensão inserida.

Existem ainda *arrays* de qualquer tipo, que incluí *arrays* de *arrays*:

`array[`$dimensão_1$`,...,`$dimensão_N$`]...array[`$dimensão_1$`,...,`$dimensão_M$`]tipo nome_array;`
$N \geq 1$ e $M \geq 1$

Os *arrays* são de base 0; não é possível afetar *arrays*; a passagem de *arrays* em argumentos de função é por referência; uma função não pode devolver um *array*.

3.2.8 Novos tipos e estruturas

Typedef

Na linguagem Temple podem-se criar novos tipos com bases nos tipos já existentes, de modo a poder existir uma diferenciação personalizada pelo programador, criando um sinónimo para o tipo já existente. Como tal, é possível criar novos tipos da seguinte forma:

```
typedef novo_tipo tipo;
```

Para além desta definição, também existe a possibilidade de criar novos tipos com base em *arrays* de outros tipos da seguinte forma:

`typedef novo_tipo array[`$dimensão_1$`,...,`$dimensão_N$`] tipo;`

A utilização destes tipos funciona de modo idêntico à utilização dos tipos originários.

Structs

A linguagem Temple suporta a declaração de tipos estruturados com **`typedef`**, tal como nos tipos primitivos.

Os membros da estrutura podem ser de tipos primitivos, tipos criados com base em outros tipos, **`arrays`** com tamanho definido, referências para outras estruturas e ainda estruturas, desde que estas estejam definidas anteriormente.

No caso do **`typedef`**, é criada uma estrutura e dado um nome a esse tipo. As estruturas criadas definem um novo tipo, tal como no exemplo seguinte:

```
typedef novo_tipo {
    tipo membro1;
    tipo membro2;
    ....
}
```

O novo tipo é definido ao mesmo tempo que a estrutura deste é definida.

Caso se criem duas estruturas com a mesma composição (tanto tipos como nomes dos mesmos), estas são consideradas estruturas diferentes. Exemplo:

Estrutura `struct1`:

```
typedef struct1{
    int x;
    int y;
}
```

Estrutura `struct2`:

```
typedef struct2{
    int x;
    int y;
}
```

Como se verifica no exemplo, são criadas duas estruturas idênticas, mas são consideradas estruturas diferentes. Assim `struct1` e `struct2` são tipos diferentes.

Existem três maneiras de inicializar uma variável deste tipo criado pelo programador.

A primeira maneira é criar uma nova variável e inicializar os membros da variável que pretendemos ao mesmo tempo. A ordem de inicialização das variáveis é definida pelo utilizador, indicando qual o membro que pretende inicializar.

```
novo_tipo nome_var = { .membro2=expressão_membro2,
                       .membro1=expressão_membro1,...};
```

A segunda maneira é declarar a nova variável sem inicializar os membros que podem ser inicializados posteriormente, mas ao declarar a variável, a memória é imediatamente alocada. Membros que não podem ser inicializados deste modo direto são os *arrays*. Se não forem inicializados da primeira maneira, estes terão de ser inicializados elemento a elemento.

```
novo_tipo nome_var;
nome_var.membro1 = expressão_membro1;
```

`nome_var.membro2[posicao] = expressão;` (No caso de *array* com uma dimensão)

A terceira e última maneira é afetar a variável de estrutura com outra variável de estrutura.

De notar que, caso um membro não seja inicializado, o acesso a esse membro gera um erro.

3.2.9 Memória dinâmica

Em Temple, o utilizador pode criar dinamicamente variáveis no decorrer dum programa em tempo de execução. Como tal, o utilizador pode criar referências para novas variáveis e alocar memória para a mesma mais tarde, ou criar referências e alocar memória para os tipos das mesmas ao mesmo tempo, como se pode ver seguinte exemplo:

```
ref nome_tipo nome_var;
nome_var = new nome_tipo;
```

```
ref nome_tipo nome_var = new nome_tipo;
```

De modo a inicializar os valores posteriormente, considerando o `membro1` parte constituinte deste novo tipo, o exemplo seguinte ilustra essa possibilidade:

```
nome_var$.membro1 = expressão_membro1;
```

Deste modo a memória é alocada para a variável em *runtime*, funcionando de modo idêntico à criação de variáveis de estruturas normalmente, mas criando uma referência utilizando a palavra-chave `ref` e `new`. Para se aceder ao valor dessa variável é necessário desreferenciar o apontador da forma: `nome_var$`.

Uma referência que não refere nada é representada por `NULL`.

A memória dinâmica também se aplica para tipos não compostos do seguinte modo:

```
ref nome_tipo nome_var = new nome_tipo;
```

Pode-se libertar memória para onde uma referência aponta, para tal apenas é necessário fazer `delete`:

```
delete nome_var;
```

Existe verificação para não se alocar nem libertar memória múltiplas vezes, gerando um erro, assim como no caso de tentar libertar memória que não chegou a ser alocada.

3.2.10 Identificadores

Os nomes de variáveis e funções funcionam de modo idêntico ao Java e C. Os nomes podem começar por letras (maiúscula ou minúscula), seguido de uma sequência opcional composta por letras, *underscore* e algarismos. A linguagem é sensível a maiúsculas e minúsculas, pelo que `ABC` e `abc` são identificadores distintos. O primeiro carácter não pode ser um *underscore* pois está reservado para os identificadores da implementação da linguagem.

Expressão regular de um identificador: `[a-zA-Z][_a-zA-Z0-9]*`

3.2.11 Variáveis e constantes

Variáveis

Uma declaração de variável inicia-se com o tipo da variável e o seu nome e termina com ponto e vírgula. Se a declaração incluir a sua inicialização, a seguir ao nome da variável aparecerá o sinal de igual, seguido de uma expressão compatível com o tipo da variável.

Declaração de uma variável:

```
tipo nome_variavel;
```

Declaração e inicialização de variável

```
tipo nome_variavel = expressão;
```

Na linguagem Temple existem variáveis globais e locais. As variáveis globais podem ser utilizadas em qualquer parte do programa (com exceção no espaço global, onde só podem ser utilizadas após a definição).

Constantes

As constantes são definidas de modo idêntico às variáveis, sendo que o valor das mesmas não pode ser alterado e mantém-se durante toda a execução do programa. É obrigatório definir um valor quando é definida a constante. A expressão genérica é:

```
const nome_const = expressão_constante;
```

As constantes podem ser definidas através de literais, de outras constantes, ou expressões com literais e/ou constantes.

3.2.12 Instruções

A sintaxe das instruções em Temple é muito idêntica à sintaxe do C e do Java, com algumas pequenas diferenças ou adições. As instruções da linguagem Temple são:

- A afetação: A afetação faz com que a variável tome o valor da expressão. Os tipos têm de ser compatíveis, senão ocorre um erro. A afetação é possível em todos os tipos (primitivos e definidos pelo programador). Não é possível para *arrays*, apenas podendo fazer afetação de posições de um *array*.

  ```
  variavel = expressão;
  ```

- A chamada de um procedimento:

  ```
  nome(argumentos);
  ```

(onde **argumentos** é uma sequência de expressões separadas por vírgula (em caso de conter mais do que um argumento). Caso a sequência seja vazia, a utilização de parêntesis continua a ser obrigatória.)

- A instrução condicional **if..then..else**:

```
if <expressão> then <instruções>
[elif <expressão> then <instruções>]*
[else <instruções>]
endif
```

(Nestas instruções, a **<expressão>** é do tipo **bool**. As **<instruções>** podem ser bloco de instruções sem instruções. A utilização de **elif** é opcional e quantas vezes o utilizador assim o pretender. A utilização de **else** é também opcional.)

- A instrução condicional **switch...case**:

```
switch <expressão>
    [case <cases>: <instruções>]*
    [default: <instruções>]
endswitch
```

(Os valores possíveis para **<cases>** são literais do tipo **int**, **char** ou um intervalo de inteiros, da seguinte forma:

```
literal1..literal2
```

e ainda podem ser dados vários valores para o mesmo **<cases>**, da forma:

```
literal1,literal2,...,literalN
```

tendo os literais de ser todos do mesmo tipo, **char** ou **int**.)

- O ciclo **while**:

```
[label:] while <expressão> do
    <instruções>
endwhile
```

(Nestas instruções, a **<expressão>** é do tipo **bool**. As **<instruções>** podem ser bloco de instruções sem instruções. A *label* é opcional, podendo ser utilizada em conjunto com a instrução **break** para determinar de qual ciclo sair.)

- O ciclo **repeat...until**:

```
[label:] repeat
    <instruções>
until <expressão> endrepeat
```

(Nestas instruções, a **<expressão>** é do tipo **bool**. As **<instruções>** podem ser bloco de instruções sem instruções. A *label* é opcional, podendo ser utilizada em conjunto com a instrução **break** para determinar de qual ciclo sair.)

- O ciclo `for`:

```
[label:] for identificador = <expressão> to|downto <expressão>
[step <expressão>] do
    <instruções>
endfor
```

(O `identificador` é declarado no ciclo `for` e não pode ser alterado em qualquer parte. O `identificador` pode ser utilizado dentro do ciclo e é tratado como uma variável local ao ciclo. O `step` é opcional e tem de ser do tipo inteiro. A *label* é opcional, podendo ser utilizada em conjunto com a instrução `break` para determinar de qual ciclo sair.)

- `break`:

```
break [label];
```

(Sai do ciclo em que se encontra no caso de não ser indicado uma *label* do ciclo a sair. No caso de ser indicada a *label*, sai do ciclo a que pertence a *label* indicada.)

- `return`:

```
return [<expressão>];
```

(Esta instrução é utilizada para sair da função, podendo retornar uma expressão de acordo com o tipo da função.)

- `delete`:

```
delete <variável>;
```

(Esta instrução é utilizada para libertar memória para onde uma variável de uma referência aponta.)

3.2.13 Expressões

As expressões da linguagem Temple são formadas a partir de:

- Literais dos tipos básicos;
- Variáveis;
- Chamadas de funções;

Expressões mais complexas são construídas utilizando operadores binários da linguagem que são os seguintes, por ordem crescente de precedência:

Operadores Binários	Operações
`or`	OU lógico
`and`	E lógico
`==`, `!=`, `<`, `<=`, `>`, `>=`	comparações
`+`, `-`	adição e subtração
`*`, `//`, `/`, `%`	multiplicação, divisão inteira, divisão de vírgula flutuante e resto da divisão inteira
`.`	acesso a membro de estrutura

Os operadores unários da linguagem são:

Operadores Unários	Operações
`+`	positivo
`-`	simétrico
`not`	negação
`$`	desreferenciação
`new`	alocação de memória

Os operadores binários são escritos de maneira infixa, entre os seus operandos, e todos associam à esquerda. A expressão `2-3+1` equivale a `(2-3)+1`.

Os operadores unários `not` (negação), `+` (positivo) e `-` (simétrico) são escritos de maneira prefixa. Os operadores unários têm precedência sobre os operadores binários, com exceção do acesso a membro de estruturas que é o operador que tem mais precedência de todos.

A ordem da avaliação dos operadores pode ser alterada com a utilização de parêntesis curvos, como normalmente.

3.2.14 Funções

Definição de funções

A definição de uma função é composta pelo tipo de retorno (qualquer um dos tipos, incluindo `void`), o nome de função que segue as regras referidas anteriormente, declaração dos parâmetros entre parêntesis (pode ser vazia), seguido da abertura de chavetas que compõe o corpo (sequência de instruções) e fecho de chavetas.

Exemplo para definição de função:

```
tipo nome_função( parâmetros ) {
    corpo da função
}
```

Argumentos formais

Os argumentos de uma função são declarados com o tipo e o seu nome. Quando a sequência de argumentos numa função tem mais do que um argumento, estes são separados por vírgulas. Se a função não contiver argumentos, a função contém os parêntesis de abertura e fecho à mesma, sem nada entre eles.

Chamadas de funções

O número de argumentos numa chamada de função tem de ser o mesmo número de argumentos da definição da função. A ordem é a mesma que na definição e o tipo do argumento na chamada de função tem de ser o mesmo tipo do argumento na definição formal correspondente, para cada argumento. O tipo de uma chamada de função é o tipo do valor devolvido pela função.

Passagem de argumentos nas chamadas de funções

A passagem de argumentos das funções é feita por referência. Qualquer alteração ao argumento na função irá alterar o valor do argumento efetivo, exceto se o argumento efetivo for constante.

3.2.15 Visibilidade de nomes

Os nomes declarados fora de qualquer função são nomes globais. Os nomes declarados na definição de uma função são locais.

Redeclarar um nome num contexto em que esse nome já foi declarado é um erro.

Nomes globais

São globais os nomes das funções, de variáveis globais, de constantes e de novos tipos criados com `typedef`. Os nomes são visíveis em qualquer parte do programa, com atenção ao contexto global em que a ordem de definição interessa e só podem ser utilizados no contexto global após a definição. O único caso em que esta regra não se verifica é nas referências dentro de estruturas para tipos ainda não definidos. No corpo de uma função pode-se utilizar qualquer nome definido globalmente.

Nomes locais

São locais a uma função os nomes das variáveis declaradas no seu corpo, dos argumentos e das *labels*. O nome de uma variável local é visível no corpo da função desde a sua declaração até ao fim do contexto onde foi definida na função. Se for declarada uma variável local com o mesmo nome que uma variável global, qualquer referência à variável com esse nome é referência à variável local, tornando a variável global invisível. A criação de novas variáveis locais a ciclos e instruções condicionais `if..then..else` e `switch` é também possível, podendo a variável ter o mesmo nome de uma variável local à função e qualquer referência a essa variável é uma referência à do contexto mais interno, tornando a outra variável invisível. No caso das *labels*, estas não podem ter nome de qualquer nome global e nome utilizado dentro da função em questão, independentemente se os nomes se encontrarem no mesmo contexto ou não.

3.2.16 Inicialização de variáveis

O valor inicial de uma variável cuja declaração não contenha a sua inicialização é indefinido. A utilização de uma variável que não tenha sido inicializada gera um erro.

3.2.17 Sistema de tipos

Temple é uma linguagem fortemente tipificada, e portanto o tipo de uma expressão tem de corresponder ao tipo esperado no contexto.

Existe um caso de conversão de tipos, que é o caso de numa expressão do tipo `float` existir uma expressão do tipo `int` que é convertido para `float`.

Operadores

Operadores	Tipo do(s) operando(s)	Tipo do resultado
`or` e `and`	bool	bool
`not`	bool	bool
`==` e `!=`	$\tau \in$ {bool, int, float, char, ref τ}	bool
`<`, `<=`, `>`, `>=`	$\tau \in$ {int,float}	bool
`+`,`-`, `*`, `/`	$\tau \in$ {int,float}	τ
`%`, `//`	int	int
`/`	float	float
`+` (positivo), `-` (simétrico)	$\tau \in$ {int, float}	τ
`new`	τ	ref τ
`$`	ref τ	τ

Instruções

Uma instrução não tem valor.

No caso das condições das instruções condicionais, estas devem ser expressões booleanas.

Avaliação

A avaliação das expressões é feita da esquerda para a direita.

A avaliação dos operadores `and` e `or` é *short-circuited*. Ou seja, o segundo operando de `or` só é avaliado se o valor do primeiro operando for `false`, e o segundo operando de `and` só é avaliado se o valor do primeiro operando for `true`.

3.2.18 Comentários e espaços

Na linguagem Temple, os comentários são inicializados com o carácter `#` estendendo-se até ao fim da linha. A opção para comentários em uma ou mais linhas é a utilização de `#(` para inicializar e `#)` para terminar os comentários.

3.2.19 Input/output

O *input/output* de Temple pode ser divido em 3 modos. O modo de texto para o ecrã, o modo de texto para ficheiro, e leitura e escrita em ficheiro de elementos de um tipo.

Input/Output - Ecrã

Em termos do modo de texto para ecrã, o programador pode fazer `input` do seguinte modo:

```
input(sequência_variáveis);
```

No caso do `output` para ecrã, pode ser feito do seguinte modo:

```
output(sequência_expressões);
```

Input/Output - Ficheiro

Neste tipo de *input* e *output*, o utilizador trabalho com o tipo primitivo `file` que contém informação acerca do ficheiro.

Para abertura de ficheiros, o utilizador dispõe de duas funções, `open` e `create`, que funcionam do seguinte modo:

Funções	ficheiro não existe	ficheiro já existe
`open`	cria ficheiro	abre ficheiro
`create`	cria ficheiro	substitui ficheiro

Estas funções afetam uma variável do tipo `file` do seguinte modo:

```
file var_ficheiro = create(localização, tipo_de_leitura, tipo_de_abertura);
```

Todos os argumentos da função são do tipo `string` e `localização` (ou nome do ficheiro) é o local onde se encontra o ficheiro, `tipo_de_leitura` é o modo com que se pretende utilizar o ficheiro (`"read"` para leitura apenas, `"write"` para escrita apenas e `"readwrite"` para leitura e escrita), e o `tipo_de_abertura` é o modo como se quer abrir o ficheiro (`"text"` para ficheiros de texto e `"binary"` para ficheiros de elementos de um tipo).

Para `input` e `output` de ficheiros, a linguagem Temple dispõe dos procedimentos `write` para escrever no ficheiro, `read` para ler do ficheiro e `readline` para ler frases de texto do ficheiro, sendo este último unicamente utilizado no modo de texto.

```
write(var_ficheiro, expressões);
read(var_ficheiro, variáveis);
readline(var_ficheiro, expressões);
```

Existem ainda outras funções, como o procedimento `goto` que pode mover o apontador para o início do ficheiro com o argumento `"begin"` ou para o fim `"end"`, o procedimento `seek` que anda uma quantidade de estruturas para a frente ou para trás com um argumento do tipo `int`, a função `eof` que retorna um valor booleano `true` se encontrar-se no fim do ficheiro, a função `eoline` que retorna um valor booleano `true` se encontrar-se no final de uma frase de texto e a função `error` que retorna um valor do tipo `int` correspondente a um erro que tenha acontecido no manuseamento do ficheiro (lista de erros não definida).

```
goto(var_ficheiro, begin ou end);
seek(var_ficheiro, expressão inteira);
var_booleana = eof(var_ficheiro);
var_booleana = eoline(var_ficheiro);
var_inteira = error(var_ficheiro);
```

3.3 Implementação

Um dos objetivos da tese, para além da definição de uma linguagem de programação, foi a possibilidade de implementar a linguagem definida. Para a criação da linguagem Temple, não seria possível definir a linguagem sem saber como determinadas decisões iriam influenciar a linguagem na sua utilização. Com a informação adquirida no estado da arte, e reconhecendo que a linguagem não se deve criar como um todo, mas sendo desenvolvida, a implementação acabou por influenciar determinados aspetos da definição da linguagem, assim como existiam determinadas características que condicionaram a implementação. Pode-se dizer que ao longo do desenvolvimento da linguagem Temple, o tempo de desenvolvimento foi-se dividindo entre definição formal e a implementação de modo alternado.

Para se implementar a linguagem Temple, foi necessário criar uma listagem das palavras reservadas e de expressões regulares para literais e nomes de identificadores como `tokens` utilizando o construtor de analisadores lexicais `flex`. Após essa listagem, foi criada a gramática da linguagem Temple, assim como estruturas na linguagem C para cada elemento que se encontra na gramática. Essas estruturas foram utilizadas para se criar uma árvore da gramática, com ajuda do construtor de analisadores sintáticos `bison` de modo a não existirem ambiguidades na gramática.

Quando se compila código Temple, é observado se o código cumpre as regras estabelecidas na gramática, e com a criação das estruturas pertencentes à gramática, são avaliados todos os elementos pertencentes à gramática, tanto de modo sintático como semântico. Por fim, se a verificação terminar com sucesso, o código Temple é convertido para código C, que será depois compilado para código máquina.

Neste capítulo são discutidos alguns casos que foram considerados de alguma importância ou de destaque na implementação da linguagem Temple.

3.3.1 Overflow

Um dos grandes objetivos da linguagem Temple é ter cuidado com *overflow*. Existem dois tipos de verificação de *overflow* em Temple para o tipo inteiro.

O primeiro tipo de verificação de *overflow* é no caso da utilização de um literal. Quando um valor de literal é inserido, o valor é guardado como uma *string* e, na análise semântica, o valor é convertido com a função `strtol` para um valor inteiro de 64 bits. O valor, agora inteiro, é comparado o valor máximo e mínimo de 32 bits com sinal. Caso o valor esteja fora do limite definido, ou que seja um valor de tamanho muito maior que 64 bits, é gerado um erro de *overflow*.

O segundo tipo de verificação de *overflow* ocorre nas operações aritméticas. Este tipo de verificação ocorre em tempo de execução. Foi criado um ficheiro *header* de C que é incluído em todos os ficheiros de Temple compilados em C. Nesse ficheiro *header* encontram-se definidas duas funções, a função `over_int1` que verifica se existe um *overflow* na negação (possível devido a não existir simétrico do menor valor negativo) e a função `over_int2` que verifica *overflow* nas operações aritméticas de adição, subtração, divisão inteira e resto da divisão.

No caso da adição, se os operandos forem positivos, o resultado tem de ser positivo, assim como se ambos forem negativos, o resultado tem de ser negativo, caso contrário, encontramo-nos num caso de *overflow*.

No caso da subtração, se o 1º operando for negativo e o 2º positivo, o resultado tem de ser negativo, assim como se o 1º operando for positivo e o 2º operando negativo, o resultado tem de ser positivo. Existe ainda o caso especial em que o 1º operando é igual a zero e o 2º operando é igual ao menor valor negativo, resultando no simétrico do menor valor negativo (o mesmo problema verificado com a função `over_int1`).

No caso da multiplicação, considerando que o 1º operando é diferente de zero, o 2º operando é igual a dividir o resultado pelo 1º operando. Não se verificando esta regra, existe erro de *overflow*. Caso o 1º operando seja 0, o resultado é 0 e, portanto, não há erro de *overflow*.

No caso da divisão, assim como do resto da divisão, não se pode dividir por zero, e existe ainda um caso especial que é dividir o valor máximo negativo por -1, o que gera novamente o erro observado com a função `over_int1`.

Deste modo, foram implementadas as duas funções referidas, `over_int1` e `over_int2`, que substituem as operações aritméticas em código C.

3.3.2 Funções

Em Temple, existem vários tipos de verificações que têm de existir no caso das funções, entre elas a verificação da passagem de argumentos na chamada de função (que é por referência) e ainda o caso de não existir retorno nas funções. Nos pontos seguintes são detalhados estes problemas e como foram resolvidos na implementação.

Passagem de argumentos

Na linguagem Temple, a passagem de argumentos na chamada de função é sempre por referência. Esta escolha recaiu na hipótese de se poderem alterar valores de *arrays*, mas isso criou outro tipo de problemas. Caso o utilizador chame uma função em que o argumento é uma variável, na implementação é dada uma referência da variável (posição de memória) como argumento, mas no caso em que o argumento é um literal ou uma constante, isso não é possível. Desse modo, antes de uma chamada de função, são criadas variáveis temporárias onde serão guardados os valores de constantes e literais e utilizadas na chamada da função.

Uma das dificuldades na implementação foi ter em consideração casos como argumentos de chamadas de função que são também chamadas de função. Outra das dificuldades encontradas foi a necessidade de se criarem variáveis temporárias distintas para todos os literais encontrados (de modo a não se perderem os valores) e saber o nome da variável temporária a utilizar e o local correto na chamada da função (os nomes das variáveis temporárias seguem o nome `fcall_tempN` em que `N` é um valor positivo, incrementado por cada novo argumento em todo o programa).

Outro dos problemas encontrados foi que em todos os locais que podem aparecer chamadas de função (como instrução ou como expressão), tem de se verificar com antecedência se vai existir alguma chamada de função, para se criarem as variáveis temporárias no local adequado (por exemplo, numa afetação, tem de se verificar se na afetação de uma variável não se encontra nenhuma chamada de função, porque caso se encontre, tem de se definir as variáveis temporárias necessárias antes de converter a afetação de Temple para C).

Funções sem retorno

Numa função, o valor de retorno tem de estar de acordo com o valor da definição formal da função. Em Temple existem 2 verificações para que exista sempre retorno de valor.

A primeira verificação ocorre em tempo de compilação e verifica se no corpo de uma função existe `return` ou não. É verificado se instruções como o `if..then..else` e `switch case` contêm a instrução `return` em todos os casos. No caso de instruções de ciclo há duas possibilidades. Na instrução `repeat..until` é sempre executado um caso e, como tal, a inclusão de valor de retorno é avaliada. No caso das instruções de ciclo

`while..do` e ciclo `for`, a possibilidade de não executar nenhuma vez o ciclo gera a possibilidade de não haver retorno de valor, e como tal é enviada uma mensagem de *warning* a avisar o utilizador a possibilidade de não existir retorno.

A segunda verificação ocorre em tempo de execução. Esta verificação ocorre com uma chamada de uma função definida na implementação da linguagem, chamada `return_error`. Esta chamada de função ocorre no código compilado em C que o aluno executa após a transformação da linguagem Temple em C. Esta chamada de função é invisível para o utilizador, e ocorre em funções que retornem qualquer tipo (que não sejam procedimentos). Esta chamada ocorre como última instrução numa função e o programa só chega a essa posição se não tiver retornado nada anteriormente. Se o programa chegar a esta instrução, é enviada uma mensagem de erro a demonstrar que a função chegou ao fim e que não retornou um valor que devia. Espera-se assim resolver um dos problemas encontrados na Secção 2.8.1.

3.3.3 Ciclo for

Uma das decisões tomadas na linguagem Temple foi ter um ciclo `for` em que os valores do ciclo não podem ser alterados durante a execução do ciclo. Desse modo, tanto o valor de início como o valor final e o `step` são guardados antes da execução do ciclo `for` em variáveis temporárias utilizadas no ciclo em si. Mas o ciclo só é executado (assim como as instruções que fazem parte do ciclo) se os valores do ciclo forem válidos.

Para se verificar se os valores do ciclo `for` são válidos ou não, quando o código de Temple é convertido para C, ao encontrar um ciclo `for`, é chamada uma função chamada `check_for_values` implementada em C (incluída no ficheiro `header` utilizado na execução de código compilado) que verifica se os valores são válidos, e que, caso não sejam, salta para a instrução após o ciclo `for`. Os valores são válidos caso o valor inicial seja menor que o final e utilizar-se a palavra-chave `to` ou caso o valor inicial seja maior que o final e utilizar-se a palavra-chave `downto`. Para além destes casos, tem-se sempre em consideração que, caso exista `step`, este seja um valor positivo (considerado como um passo, seja um passo positivo no caso da utilização da palavra-chave `to` ou um passo negativo no caso da utilização da palavra-chave `downto`).

3.3.4 Labels e breaks

Uma das possibilidades existentes na linguagem Temple é a possibilidade do aluno sair de múltiplos ciclos com uma instrução `break`, indicando o nome da *label* do ciclo de que pretende sair. Foi decidido que os nomes das *labels* não se podem repetir ao longo de uma função (posteriormente, foi também decidido que o nome de uma *label* não pode ser utilizado por qualquer outra entidade dentro da função). O modo como a solução foi implementada foi que existe uma *label* logo a seguir à instrução do ciclo e a instrução `break` da linguagem Temple foi convertida em uma instrução `goto` para a linha onde se encontra a *label*. Este tipo de conversão só existe no caso do ciclo conter uma *label*, caso contrário a instrução `break` da linguagem Temple mantém-se na compilação para linguagem C.

3.3.5 Conclusão

Nesta secção encontram-se referidas algumas decisões tomadas na implementação da linguagem Temple para resolver determinados problemas ou tornar possíveis decisões tomadas na definição da linguagem. Estes foram os problemas que geraram alguma dificuldade na implementação ou ocuparam grande parte do tempo de implementação da linguagem. Apesar de tudo, há muitos casos que não foram contemplados devido à

implementação não ter sido terminada ou ainda por serem de pouca importância no desenvolvimento deste trabalho.

3.4 Eficiência

Apesar da eficiência não ser um dos objetivos da linguagem Temple, convém que a linguagem não demore muito tempo a compilar, de modo a que o aluno possa obter *feedback* em tempo útil para um bom desenvolvimento de código.

Como tal, foram implementados os algoritmos de Fibonacci em 3 linguagens: Python, C e Temple. Os algoritmos foram implementados de modo iterativo e de modo recursivo, de modo a comparar valores que supostamente são de complexidade proporcional e exponencial, respetivamente. Estas linguagens foram escolhidas devido à linguagem Temple ser compilada para linguagem C e de modo a podermos também comparar uma linguagem que é compilada (C) com uma linguagem que é interpretada (Python). Pretende-se avaliar sobretudo o impacto da tradução de Temple para C na compilação, mas também ter uma ideia de como o tempo de execução é afetado devido às verificações que ocorrem em tempo de execução. Todos os valores que foram registados foram observados na mesma máquina virtual e foi registado o tempo real para compilação e execução (execução apenas no caso da linguagem Python).

Para comparar as linguagens, tentou-se implementar o algoritmo Fibonacci de modo a ser o mais idêntico possível nas 3 linguagens. O código das linguagens encontra-se nas seguintes Listagens:

```
1  #include<stdio.h>
2
3  int VALUE = 10;
4
5  int fib(int x){
6      int result = 0;
7      int temp1 = 1;
8      for(int i = 0; i<x; i++){
9          int temp2 = temp1;
10         temp1 = result;
11         result = temp1 + temp2;
12     }
13     return result;
14 }
15
16 int main() {
17    int x = fib(VALUE);
18 }
```

Listagem 3.1: Fibonacci iterativo em C

```
1  #include<stdio.h>
2
3  int VALUE = 10;
4
5  int fib(int x){
6      if(x == 0)
7          return 0;
8      else if(x == 1)
9          return 1;
10     else
11         return fib(x-1) + fib(x-2);
12 }
13
14
15
16 int main() {
17    int x = fib(VALUE);
18 }
```

Listagem 3.2: Fibonacci recursivo em C

```
1  VALUE = 10;
2
3  def fib(x):
4      result = 0
5      temp1 = 1
6      for i in range(x):
7          temp2 = temp1
8          temp1 = result
9          result = temp1 + temp2
10     return result
11
12 if __name__== "__main__":
13     x = fib(VALUE)
```

Listagem 3.3: Fibonacci iterativo em Python

```
1  VALUE = 10;
2
3  def fib(x):
4      if x == 0:
5          return 0
6      elif x == 1:
7          return 1
8      else:
9          return fib(x-1) + fib(x-2)
10
11
12 if __name__== "__main__":
13     x = fib(VALUE)
```

Listagem 3.4: Fibonacci recursivo em Python

```
1  const VALUE = 10;
2
3  int fib(int x){
4      int result = 0;
5      int temp1 = 1;
6
7      for i = 0 to x-1 do
8          int temp2 = temp1;
9          temp1 = result;
10         result = temp1 + temp2;
11     endfor
12     return result;
13 }
14
15 void main() {
16    int x = fib(VALUE);
17 }
```

Listagem 3.5: Fibonacci iterativo em Temple

```
1  const VALUE = 10;
2
3  int fib(int x){
4      if x == 0 then
5          return 0;
6      elif x == 1 then
7          return 1;
8      else
9          return fib(x-1) + fib(x-2);
10     endif
11 }
12
13
14
15 void main() {
16    int x = fib(VALUE);
17 }
```

Listagem 3.6: Fibonacci recursivo em Temple

Para verificar a eficiência, tanto em modo iterativo como recursivo, foram observados tempos para a compilação e execução do algoritmo Fibonacci para os valores 36, 37 e 38. Foram observados três tempos para cada valor e para cada linguagem. Foram calculadas as médias e a razão entre os tempos das linguagens C e Python para a linguagem Temple, de modo a poder-se comparar a diferença de tempos. Os tempos de compilação foram registados nas Tabelas 3.1, 3.2, 3.3 e 3.4, e os tempos de execução nas Tabelas 3.5, 3.6, 3.7 e 3.8. Com base nas médias dos tempos observados, foram ainda criados os gráficos das Figuras 3.1, 3.2, 3.3 e 3.4.

3.4.1 Tempos de compilação

Iterativo

Iterativo	36			37			38		
	t1	t2	t3	t1	t2	t3	t1	t2	t3
`C`	0.058	0.057	0.063	0.061	0.062	0.064	0.060	0.062	0.064
`Temple`	0.098	0.098	0.088	0.099	0.096	0.098	0.101	0.095	0.099

Tabela 3.1: Tempos de compilação de Fibonacci iterativo

Iterativo	36		37		38	
	Média	Razão	Média	Razão	Média	Razão
`C`	0.059	1.610	0.062	1.580	0.062	1.580
`Temple`	0.095	1	0.098	1	0.098	1

Tabela 3.2: Média e razão de compilação de Fibonacci iterativo

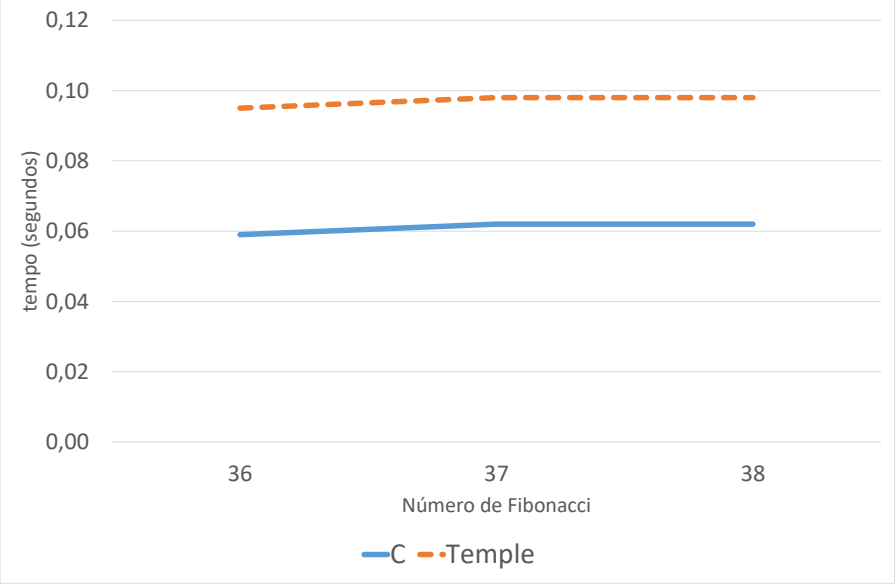

Figura 3.1: Tempos de compilação de Fibonacci iterativo

Recursivo

Recursivo	36			37			38		
	t1	t2	t3	t1	t2	t3	t1	t2	t3
`C`	0.068	0.063	0.064	0.064	0.065	0.062	0.065	0.063	0.064
`Temple`	0.098	0.095	0.099	0.102	0.099	0.100	0.096	0.097	0.102

Tabela 3.3: Tempos de compilação de Fibonacci recursivo

Recursivo	36		37		38	
	Média	Razão	Média	Razão	Média	Razão
`C`	0.065	1.492	0.064	1.563	0.064	1.531
`Temple`	0.097	1	0.100	1	0.098	1

Tabela 3.4: Média e razão de compilação de Fibonacci recursivo

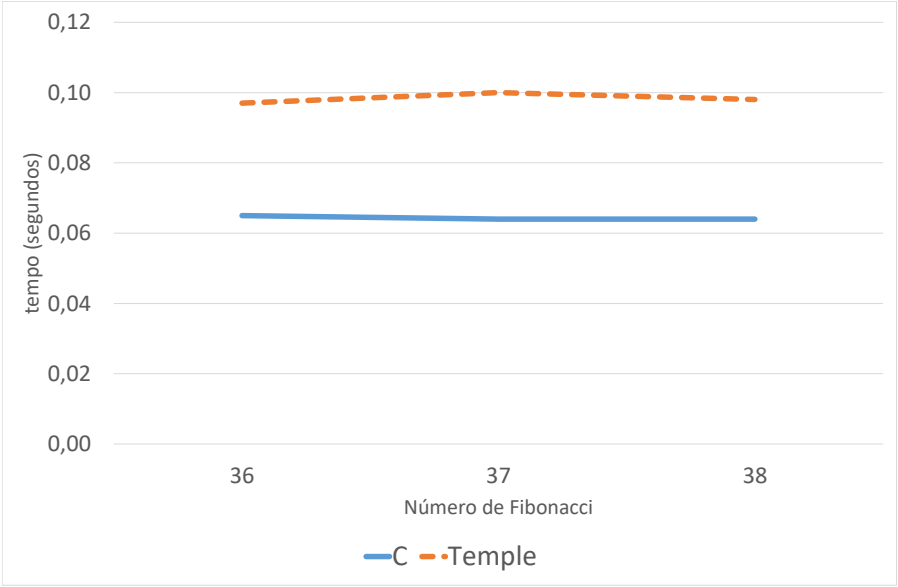

Figura 3.2: Tempos de compilação de Fibonacci recursivo

Como se pode observar nas Figuras 3.1 e 3.2, tanto os tempos de compilação do algoritmo iterativo como do algoritmo recursivo mantêm-se muito idênticos. Pode-se reconhecer que, em média, Temple é 1,5 a 1,6 vezes mais lento que C a compilar os programas, o que faz todo o sentido, tendo em conta que Temple tem ainda de ser convertido para C. Tanto os tempos de C como de Temple (tanto com o algoritmo iterativo como com o algoritmo recursivo) não se alteram muito, estando os tempos de C, em média, perto dos 0,060 segundos e os tempos de Temple, em média, perto dos 0,100 segundos.

3.4.2 Tempos de execução

Iterativo

Iterativo	36			37			38		
	t1	t2	t3	t1	t2	t3	t1	t2	t3
`C`	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
`Temple`	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
`Python`	0.033	0.034	0.031	0.033	0.032	0.035	0.034	0.032	0.031

Tabela 3.5: Tempos de execução de Fibonacci iterativo

Iterativo	36		37		38	
	Média	Razão	Média	Razão	Média	Razão
`C`	0.001	1	0.001	1	0.001	1
`Temple`	0.001	1	0.001	1	0.001	1
`Python`	0.033	0.033	0.034	0.034	0.032	0.032

Tabela 3.6: Média e razão de execução de Fibonacci iterativo

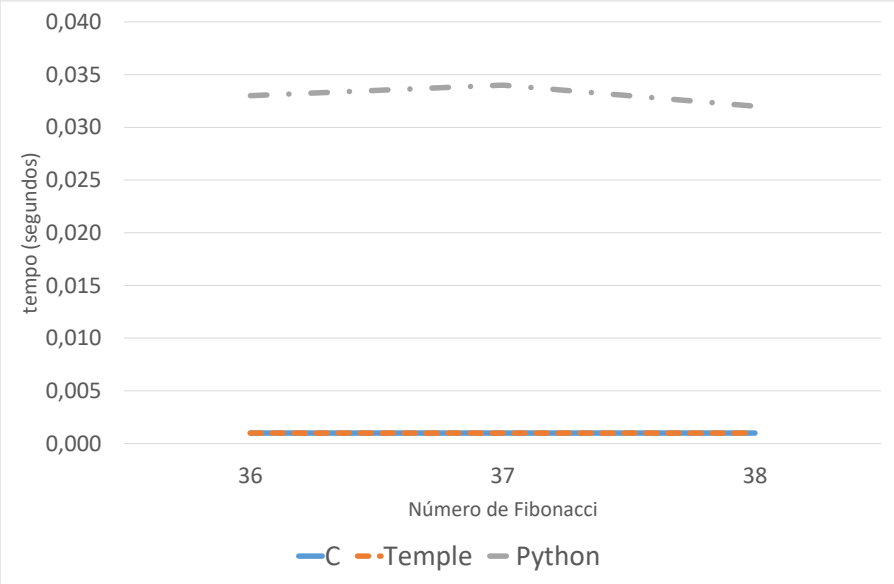

Figura 3.3: Tempos de execução de Fibonacci iterativo

Como se pode observar na Figura 3.3, devido à quantidade de operações a executar ser bastante pequena (devido ao algoritmo ser iterativo), os tempos de C e de Temple são muito idênticos. Já Python, tem um tempo de execução um pouco maior, demorando 33 vezes mais que os programas de Temple e de C. Neste tipo de algoritmos pode-se pensar que os valores de Temple serão sempre iguais aos de C, mas o mesmo não se comprova, como se pode ver no caso seguinte.

Recursivo

Recursivo	36			37			38		
	t1	t2	t3	t1	t2	t3	t1	t2	t3
`C`	0.304	0.301	0.305	0.481	0.487	0.474	0.781	0.778	0.781
`Temple`	1.718	1.704	1.715	2.794	2.790	2.813	4.514	4.525	4.528
`Python`	19.907	20.071	19.939	31.508	31.414	31.466	51.064	51.567	51.526

Tabela 3.7: Tempos de execução de Fibonacci recursivo

Recursivo	36		37		38	
	Média	Razão	Média	Razão	Média	Razão
`C`	0.303	5.650	0.481	5.819	0.780	5.797
`Temple`	1.712	1	2.799	1	4.522	1
`Python`	19.972	0.085	31.463	0.089	51.386	0.88

Tabela 3.8: Média e razão de execução de Fibonacci recursivo

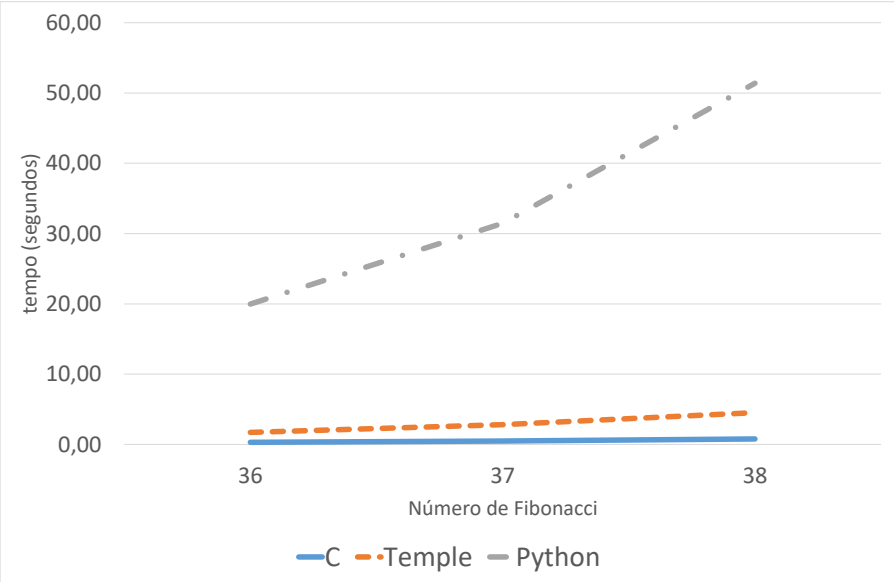

Figura 3.4: Tempos de execução de Fibonacci recursivo

Como se pode observar na Figura 3.4, num tipo de algoritmo recursivo, onde existe uma quantidade maior de operações, já existe uma diferença entre Temple e C, apesar de não ser uma grande diferença. Neste caso, a execução do algoritmo em Temple chega a ser aproximadamente 5,6 a 5,8 vezes mais lento que a execução em C. Já Python é outro caso à parte, em que os tempos são muito maiores, com valores aproximadamente 11,2 a 11,7 vezes mais lentos que Temple. É de esperar que a curva exponencial se mantenha no caso de valores superiores, e desse modo, a linguagem Temple mantém-se viável em mais valores que Python.

3.4.3 Conclusão

Em suma, apesar de não ser um dos focos da linguagem, observou-se que a linguagem Temple pode ser uma boa linguagem. Não é a linguagem mais rápida, mas como se observou no caso do algoritmo recursivo, um algoritmo mais exigente, a linguagem Temple surpreendeu bastante pela positiva, e demonstrou que é uma linguagem que está preparada para algoritmos que poderiam facilmente ser executados em C.

3.5 Exemplos de código

De modo a demonstrar os progressos atingidos com a linguagem Temple, nesta secção encontram-se alguns exemplos com exercícios que poderiam ser utilizados na primeira disciplina de programação.

`Exemplo 1`: Programa que verifica se um ano é bissexto ou não (sabendo que são bissextos todos os anos múltiplos de 4 com exceção dos anos múltiplos de 100, que não são bissextos, mas se forem múltiplos de 400 são bissextos).

```
1  void anoBissexto() {
2      output("Insira um ano:\n");
3      int ano;
4      input(ano);
5
6      if ano%4 != 0 then
7          output("não é bissexto\n");
8      elif ano%100 != 0 then
9          output("é bissexto\n");
10     elif ano%400 != 0 then
11         output("não é bissexto\n");
12     else
13         output("é bissexto\n");
14     endif
15 }
16
17 void main() {
18     anoBissexto();
19 }
```

Listagem 3.7: Ano bissexto

Exemplo 2: Programa que, dado um número, calcula o seu fatorial.

```
1  int fatorial(int x) {
2      if x < 0 then
3          output("Erro: número negativo.\n");
4          return -1;
5      elif x == 0 then
6          return 1;
7      else
8          return x * fatorial(x-1);
9      endif
10 }
11
12 void main() {
13     output("Insira um número:\n");
14     int valor;
15     input(valor);
16     int resutaldo = fatorial(valor);
17 }
```

Listagem 3.8: Fatorial recursivo

Exemplo 3: Programa que, dados dois números e um sinal de operação, devolve o resultado.

```
1  int calcula(int x, int y, char op) {
2      switch op
3      case '+':
4          return x+y;
5      case '-':
6          return x-y;
7      endswitch
8  }
9
10 void main() {
11     output("Insira os dois números e o sinal:\n");
12     int x;
13     int y;
14     char op;
15     input(x, y, op);
16     output("Resultado:", calcula(x,y,op), "\n");
17 }
```

Listagem 3.9: Operação com sinal

Exemplo 4: Programa que, dados dois vetores de inteiros com o mesmo tamanho, compara os valores posição a posição, e guarda o menor valor no 1º *array*, e o maior valor no 2º *array*.

```
1  void menor_maior(array[]int array_1, array[]int array_2) {
2      for i = 0 to size(array_1, 1) do
3          if array_1[i] > array_2[i] then
4              int temp = array_2[i];
5              array_2[i] = array_1[i];
6              array_1[i] = temp;
7          endif
8      endfor
9  }
10
11 void main() {
12     array[3]int lista1 = {5,1,4};
13     array[3]int lista2 = {2,4,3};
14
15     menor_maior(lista1, lista2);
16
17     # lista1 = {2,1,3}
18     # lista2 = {5,4,4}
19 }
```

Listagem 3.10: Array menor e maior

Exemplo 5: Implementação de Listas Ligadas

```
1  typedef No{
2      int valor;
3      ref No seguinte;
4   };
5
6  ref No novoNo() {
7      ref No temp = new No;
8      temp$.seguinte = NULL;
9      return temp;
10 }
11
12 bool addNo(ref No cabeca, int valor) {
13     ref No novo = novoNo();
14     novo$.valor = valor;
15
16     ref No temp = cabeca;
17     while temp$.seguinte != NULL do
18         temp = temp$.seguinte;
19     endwhile
20
21     temp$.seguinte = novo;
22 }
23
24
25
```

```
26
27
28
29
30 int removerUltimo(ref No cabeca) {
31     int resultado = 0;
32
33     if cabeca$.seguinte == NULL do
34         resultado = cabeca$.valor;
35         delete cabeca;
36         return resultado;
37     endif
38
39
40     ref No temp = cabeca;
41     while temp$.seguinte$.seguinte != NULL do
42         temp = temp$.seguinte;
43     endwhile
44
45     resultado = temp$.seguinte$.valor;
46     delete temp$.seguinte;
47     temp$.seguinte = NULL,
48     return resultado;
49 }
```

Listagem 3.11: Listas ligadas

4

Conclusões e trabalho futuro

Nesta dissertação foi apresentado o estado atual do ensino de programação na primeira disciplina de programação. De modo a compreender melhor o estado atual das universidades, foi realizado um estudo em que se observaram quais as linguagens de programação atualmente utilizadas. Tentou-se também compreender os motivos que possam estar por detrás dessas escolhas (também observando outro estudo que foi realizado na Austrália, um pouco mais aprofundado nas escolhas). Uma das escolhas por detrás da linguagem em que este trabalho se debateu inicialmente foi a influência da indústria e como muitas vezes a escolha da linguagem por parte das universidades é influenciada pelas linguagens que no momento são mais utilizadas fora do ambiente educacional. Em termos educacionais, foram avaliadas quais seriam as boas características de uma linguagem de programação ideal para o ensino de programação. Foi observado que a linguagem tem de estar preparada paras as dificuldades normalmente encontradas pelos alunos, com mensagens de erro que ajudem o aluno a ir na direção correta. No estado da arte tinha sido observado num estudo que os alunos preferiam realizar os trabalhos e exercícios sozinhos do que nas aulas, o que mais uma vez demonstra que a linguagem tem de estar preparada para ajudar os alunos a ir na direção certa. Para além de ser uma linguagem que envie mensagens de erros adequadas para o aluno, deve ser uma linguagem simples, mas que ao mesmo tempo não se diferencie muito das outras linguagens de modo a que a adaptação do aluno a novas linguagens não seja muito difícil. A eficiência não é algo fundamental nesta fase da aprendizagem do aluno, apesar de ser benéfico que o tempo de compilação e execução não seja muito superior a outras linguagens. Em relação ao paradigma escolhido,

de um modo geral, as linguagens utilizadas em universidades eram na maioria linguagens procedimentais ou orientadas a objetos, pelo que nos vários artigos visualizados, os autores não entraram em consenso se uma linguagem orientada a objetos deveria ser ensinada como primeira linguagem ou se apenas mais tarde, após o aluno ter aprendido as instruções mais simples numa linguagem procedimental, e existia ainda o caso de criar-se uma linguagem com vários *subsets* de modo a integrar vários paradigmas na mesma linguagem ou ter uma linguagem que acrescenta novas funcionalidades à medida que o aluno progride na disciplina.

Com base no estado da arte, começou-se a desenvolver a linguagem. Muitas das características da linguagem Temple foram definidas de início, mas muitas características foram sendo alteradas com a implementação da linguagem e observação de resultados da execução de excertos de código. A sintaxe da linguagem foi definida totalmente (apesar de muitas coisas poderem ser mais tarde acrescentadas com bibliotecas, algo referido no estado da arte, em que deve tentar-se chegar a um equilíbrio do que faz parte da linguagem e o que pode ser acrescentado mais tarde). Com a linguagem Temple foi decidido manter uma quantidade limitada de tipos primitivos, de modo a simplificar a linguagem (tal como em muitas outras escolhas), e fazer com que os tipos não dependam da máquina onde a linguagem corra, como linguagens como C. Os literais pretenderam-se similares às linguagens observadas no estado da arte, e a conversão de tipos tentou-se manter no mínimo para evitar erros inesperados. Uma questão que se pretendeu resolver foi a verificação de `overflow`, tanto para operações com literais como dos próprios literais, enviado mensagem de erro para o terminal. Para além dos tipos primitivos, a linguagem dispõe ainda de outros tipos como *arrays*, estruturas (criadas através da instrução `typedef` que cria sinónimos entre palavras e uma construção), de referências (para se criarem variáveis dinamicamente) e de um tipo especial para os ficheiros (onde se pretende simplificar o manuseamento de ficheiros, que as vezes pode parecer muito complicado para os alunos). Em termos de instruções, não se pretendeu diferenciar muito das linguagens existentes e assim podem-se encontrar ciclos e condicionais idênticos a linguagens como Java, C ou Python, apesar de se ter tido cuidado em relação à instrução `for` em que se evita criar ciclos infinitos, ou alteração do ciclo enquanto este é executado (evitando resultados inesperados). Para além disto, foi dada atenção às funções que têm argumentos passados por referência, de modo ao aluno poder afetar um *array* diretamente, aos *arrays* que podem ter várias dimensões com tamanhos variados e ainda em relação às regras do espaço de nomes.

Deste modo, a linguagem Temple pode ser utilizada por alunos que estejam a aprender a programar, como se pretendia de início, apesar de alguns dos benefícios da linguagem só se verificarem numa implementação completa da linguagem (tais como verificações de erros). A dissertação acaba também por dar uma visão global sobre o estado atual do ensino da programação e deixar em aberto a possibilidade de alterações à linguagem Temple e à sua evolução, senão mesmo um ponto de partida para a criação de novas linguagens para o ensino de programação.

4.1 Trabalho futuro

Apesar do grande progresso no desenvolvimento da linguagem Temple, muitas coisas ficaram por ser trabalhadas e desenvolvidas. Uma dessas coisas foi a utilização do *encoding* UTF8 nas `strings` e `char`, em que foi decidido utilizar este *encoding*, mas não chegou a ser implementado na linguagem Temple, estando apenas o *encoding* ASCII implementado neste momento. Outra questão levantada acerca das `strings` e `char` foi que, utilizando o *encoding* UTF8, os caracteres não ocupam todos a mesma quantidade de memória. Os *arrays* ficaram também por implementar, assim como verificação de acessos a *arrays*. A memória dinâmica foi outro aspeto que necessita de mais trabalho, sobretudo com o caso de libertação de memória.

Outra parte da linguagem Temple que ficou mais em aberto foi a questão do `input` e `output` de ficheiros e as instruções que fazem parte do manuseamento de ficheiros. Seria necessário a implementação estar completa e observar se os alunos iriam conseguir funcionar com este tipo de instruções neste momento.

Um dos pontos que foi falado foi a criação de subconjuntos da linguagem e, com isso, a criação de objetos (com possibilidade de encapsulamento) ou ainda a possibilidade de modularização, seja isso com bibliotecas ou poder separar programas por vários ficheiros.

De um modo geral, a linguagem necessitaria de ser utilizada por alunos e, com base na sua utilização e *feedback*, corrigir ou alterar funcionalidades da linguagem. Tal como referido no estado da arte, em que uma linguagem não se deve criar, mas ir sendo criada, só com a ajuda dos alunos e também de professores que, com anos de experiência no ensino, se poderia criar uma linguagem que apesar de não poder suprir todos os problemas existentes (por variadas razões, e em certos casos contradições), seja uma linguagem que mais facilmente ajude o aluno a dar os primeiros passos no mundo da programação.

A

Lista de universidades mundiais

AMÉRICAS

BRASIL

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Universidade de São Paulo USP	Ciências de Computação	C		
2	Universidade Estadual de Campinas UNICAMP	Engenharia de Computação	C		
3	Universidade Estadual de Campinas UNICAMP	Bacharelado em Ciência da Computação	C		
4	Universidade Federal do Rio de Janeiro	Engenharia de Computação e Informação	C	C++	JAVA
5	Universidade Federal do Rio de Janeiro	Bacharelado em Ciência da Computação	C		

ESTADOS UNIDOS DA AMÉRICA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	California Institute of Technology Caltech	Computer Science	PYTHON		
2	Carnegie Mellon University	Computer Science	C	PYTHON	
3	Columbia University	BS in Computer Science (SEAS)	JAVA		
4	Cornell University	Computer Science	PYTHON	MATLAB	
5	Georgia Institute of Technology	Computer Science	PYTHON	MATLAB	
6	Harvard University	Computer Science	C	PYTHON	JAVASCRIPT
7	Massachusetts Institute of Technology (MIT)	Engineer in Computer Science	PYTHON		
8	New York University - Courant	Computer Science	JAVA	PYTHON	
9	New York University - Tandon School of Engineering	Computer Science	PYTHON		
10	Pennsylvania State University	Computer Science	C++		
11	Princeton University	Computer Science	JAVA		
12	Purdue University	Computer Science	JAVA		
13	Stanford University	Computer Science	JAVA	JAVASCRIPT	PYTHON
14	State University of New York (SUNY) - Binghamton University	Computer Science	JAVA		
15	State University of New York (SUNY) - Stony Brook University	Computer Science	JAVA		
16	State University of New York (SUNY) - University at Albany	Computer Science	VISUAL BASIC.NET		
17	State University of New York (SUNY) - University at Buffalo	Computer Science & Engineering	JAVA		
18	University of California Berkeley	Computer Science	PYTHON	SCHEME	
19	University of California Los Angeles UCLA	Computer Science	C++		
20	University of California--San Diego	Computer Science and Engineering	JAVA		
21	University of Illinois--Urbana-Champaign	Computer Science (Engineering)	JAVA		
22	University of Maryland--College Park	Computer Science	JAVA		
23	University of Michigan--Ann Arbor - COLLEGE OF LITERATURE, SCIENCE, AND THE ARTS (LSA)	Computer Science	C++	PYTHON	
24	University of Michigan--Ann Arbor - College of Engineering	Computer Science	C++	PYTHON	
25	University of Pennsylvania	Computer Science	JAVA		
26	University of Southern California - Viterbi	Computer Science	C++		
27	University of Southern California - Viterbi	Computer Science (Games)	C++		
28	University of Texas Austin	Computer Science	JAVA		
29	University of Washington	Computer Science Undergrad	JAVA		
30	Yale	Computer Science	C	PYTHON	JAVASCRIPT

CANADÁ

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	McGill University	Major Computer Science	JAVA		
2	McGill University	Major Computer Science - Computer Games	JAVA		
3	McGill University	Major Software Engineering	JAVA		
4	University of Alberta	Specialization in Computing Science	PYTHON		
5	University of British Columbia	BS Computer Science	STUDENT LANGUAGE		
6	University of Toronto	Computer Science Specialist	PYTHON		
7	University of Waterloo	Computer Science	RACKET		

Tabela A.1: Américas

EUROPA

REINO UNIDO

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Imperial College London	Computing	HASKELL		
2	King's College London	Computer Science BSc	JAVA		
3	University College London UCL	BSc Computer Science	C	HASKELL	
4	University of Bristol	BSc Computer Science	C	HASKELL	
5	University of Cambridge	Computer Science	ML	JAVA	
6	University of Edinburgh	Computer Science	HASKELL		
7	University of Glasgow	Computing Science	PYTHON		
8	University of Manchester	Computer Science	JAVA		
9	University of Oxford	Computer Science	HASKELL		
10	University of Southampton	BEng Software Engineering	JAVA		
11	University of Southampton	BSc Computer Science	JAVA		

ALEMANHA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Karlsruhe Institute of Technology (Universität Karlsruhe)	Computer Science	JAVA		
2	Ludwig-Maximilians-Universitat Munchen	Ciências da computação Bacharelado	JAVA		
3	Ludwig-Maximilians-Universitat Munchen	Ciência da Computação mais Linguística Computacional (Bacharelado)	JAVA		
4	RWTH Aachen University	Computer Science	JAVA	HASKELL	PROLOG
5	Technical University of Berlin	Computer Science	C		
6	Technische Universität München	Informatics	JAVA		

ESPANHA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Polytechnic University of Valencia - Campus de Alcoy	Informatics Engineering	JAVA		
2	Polytechnic University of Valencia - Campus de Vera (Valencia)	Informatics Engineering	JAVA		
3	Universidad Carlos III de Madrid (UC3M)	Ingeniería Informática	JAVA		
4	Universidad Complutense de Madrid	Ingeniería Informática	C++		
5	Universidad Complutense de Madrid	Ingeniería del Software	C++		
6	Universidad de Extremadura - Centro Universitario De Mérida	Ingeniería Informática En Tecnologías de la Información	C++	C	
7	Universidad de Extremadura - Escuela Politécnica	Ingeniería Informática En Ingeniería De Computadores	C++		
8	Universidad de Extremadura - Escuela Politécnica	Ingeniería Informática En Ingeniería Del Software	C++		
9	Universidad Politécnica de Madrid - Escuela Técnica Superior de Ingeniería de Sistemas Informáticos	Ingeniería de Computadores	C		
10	Universidad Politécnica de Madrid - Escuela Técnica Superior de Ingeniería de Sistemas Informáticos	Ingeniería del Software	C		
11	Universidad Politécnica de Madrid - Escuela Técnica Superior de Ingeniería de Sistemas Informáticos	Sistemas de Información	C		
12	Universidad Politécnica de Madrid - Escuela Técnica Superior de Ingeniería de Sistemas Informáticos	Tecnologías para la Sociedad de la Información	C		
13	Universidad Politécnica de Madrid - Escuela Técnica Superior de Ingenieros Informáticos	Ingeniería Informática	JAVA		
14	Universitat Autònoma de Barcelona	Computer Engineering	C		
15	Universitat de Barcelona	Ingeniería Informática	JAVA		
16	Universitat Politècnica de Catalunya BarcelonaTech - Barcelona School of Informatics (FIB)	Informatics Engineering	C++		
17	Universitat Politècnica de Catalunya BarcelonaTech - Vilanova i la Geltrú School of Engineering	Informatics Engineering	C++		
18	University of Granada	Ingeniería Informática	C++		
19	University of Granada - Ceuta	Ingeniería Informática	C++		

ITÁLIA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Politecnico di Milano - Como	Engineering Of Computing Systems	C	C++	
2	Politecnico di Milano - Milano Leonardo, Cremona	Engineering Of Computing Systems	C	C++	
3	Politecnico di Torino	Computer Engineering	C		
4	Università degli Studi di Milano - Crema	Computer Science	C	JAVA	
5	Università degli Studi di Milano - Crema	Computer Systems and Networks Security	C	JAVA	
6	Università degli Studi di Milano - Milano	Computer Science	JAVA		
7	Università degli Studi di Milano - Milano	Computer Science for New Media Comunications	JAVA		
8	Università degli Studi di Padova	Computer Engineering	C		
9	Università degli Studi di Roma La Sapienza	Informatics	PYTHON		
10	Università degli Studi di Roma La Sapienza	Computer and System Engineering	PYTHON		
11	Università di Bologna (Università degli Studi di Bologna)	Computer Engineering	C		
12	Università di Pisa (Università degli Studi di Pisa)	Computer Science	C		

Tabela A.2: Europa (Tabela 1 de 2)

EUROPA

FRANÇA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Université d'Orléans (comUE Université confédérale Léonard de Vinci)	Licence Informatique	JAVA		
2	Université de Bordeaux (ComUE d'Aquitaine)	Licence Informatique	PYTHON		
3	Université de Lyon I / Université Claude Bernard (comUE Université de Lyon)	Licence Informatique	C	C++	
4	Université de Paris VI / Université Pierre et Marie Curie (comUE Sorbonne Universités)	Computer Science	PYTHON		
5	Université de Paris VII / Université Paris Diderot (comUE Université Sorbonne Paris Cité)	Licence Informatique	JAVA		

SUIÇA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	École Polytechnique Fédérale de Lausanne	Computer Science	JAVA		
2	Eidgenössische Technische Hochschule ETH Zürich / Swiss Federal Institute of Technology Zurich	Computer Science	JAVA		

BÉLGICA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Catholic University of Leuven - Kortrijk	Bacharel em Informática	PYTHON		
2	Catholic University of Leuven - Leuven	Bacharel em Informática	PYTHON		
3	Université Catholique de Louvain	sciences informatiques	JAVA		

HOLANDA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Delft University of Technology TU Delft	Computer Science and Engineering	JAVA		
2	Eindhoven University of Technology	Computer Science and Engineering	JAVA		
3	University of Amsterdam / Universiteit van Amsterdam	Informatica	JAVA		
4	Utrecht University / Universiteit Utrecht	Bachelor Informatica	C#		
5	VU University of Amsterdam / Vrije Universiteit Amsterdam	Computer Science (Informatica)	C++		

SUÉCIA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Chalmers University of Technology	Engenharia de Computação 180 Licenciatura	C		
2	Chalmers University of Technology	Engenharia de Computação 300 Mestrado Integrado	HASKELL		
3	Chalmers University of Technology	Tecnologia da informação 300 Mestrado Integrado	JAVA		
4	Royal Institute of Technology / Kungliga Tekniska Högskolan	Engenharia de informática, Master of Science 300	JAVA		
5	Royal Institute of Technology / Kungliga Tekniska Högskolan	Tecnologia da informação, Master of Science 300	JAVA		
6	Royal Institute of Technology / Kungliga Tekniska Högskolan - FLEMINGSBERG	Engenharia Informática, Ensino Superior 180	C		
7	Royal Institute of Technology / Kungliga Tekniska Högskolan - KISTA	Engenharia Informática, Ensino Superior 180	JAVA		
8	Uppsala University / Uppsala Universitet	Bacharel em Ciência da Computação	HASKELL		

DINAMARCA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Aarhus University / Aarhus Universitet	Informática	JAVA		
2	Aarhus University / Aarhus Universitet	Desenvolvimento de Produtos IT	JAVA		

FINLÂNDIA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Aalto University	tecnologia da informação	SCALA		
2	University of Helsinki	Informática	JAVA		
3	University of Turku / Turun yliopisto	Licenciatura em Informática (LuK)	PYTHON		
4	University of Turku / Turun yliopisto	Graduação em Ciência da Computação (FM)	C#		

ÁUSTRIA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Johannes Kepler Universität Linz	Computer Science Bachelor	JANA (JAVA)		
2	Technische Universität Wien	Bacharelado Informática de Mídia e Computação Visual	JAVA		
3	Technische Universität Wien	Bacharelado Software e Engenharia de Informação	JAVA		
4	Technische Universität Wien	Bacharelado em Engenharia Informática	JAVA		

Tabela A.3: Europa (Tabela 2 de 2)

OCEÂNIA

AUSTRÁLIA

	Universidade	Curso	Linguagem	alternativa	2ª alternativa
1	Australian National University	Bachelor of Advanced Computing (Honours)	HASKELL		
2	Australian National University	Bachelor of Software Engineering (Honours)	HASKELL		
3	Australian National University	Bachelor of Information Technology	HASKELL		
4	Monash University	Bachelor Information Technology	JAVA	PYTHON	C++
5	Monash University	Bachelor Computer Science	PYTHON		
6	Monash University	Bachelor Computer Science Advanced	PYTHON		
7	University of Melbourne	Major Computing and Software Systems	PYTHON		
8	University of New South Wales	Computer Science	C		
9	University of New South Wales	Software Engineering	C		
10	University of Queensland Australia	Computer Science	PYTHON		
11	University of Queensland Australia	Information Technology	PYTHON		
12	University of Sydney	Bachelor of Advanced Computing	PYTHON		
13	University of Sydney	Bachelor of Engineering Honours (Software)	PYTHON		

Tabela A.4: Oceânia

B

Lista de universidades portuguesas

Portugal

	Universidades Públicas	Curso	Grau Académico	Linguagem
1	ISCTE - Instituto Universitário de Lisboa	Engenharia Informática	Licenciatura	JAVA
2	Universidade da Beira Interior	Engenharia Informática	Licenciatura	C
3	Universidade da Madeira	Engenharia Informática	Licenciatura	PYTHON
4	Universidade de Aveiro	Engenharia Informática	Licenciatura	PYTHON
5	Universidade de Coimbra - Faculdade de Ciências e Tecnologia	Engenharia Informática	Licenciatura	PYTHON
6	Universidade de Évora - Escola de Ciências e Tecnologia	Engenharia Informática	Licenciatura	PYTHON
7	Universidade de Lisboa - Faculdade de Ciências	Engenharia Informática	Licenciatura	JAVA
8	Universidade de Lisboa - Instituto Superior Técnico	Engenharia Informática e de Computadores	Licenciatura	PYTHON
9	Universidade de Lisboa - Instituto Superior Técnico (Tagus Park)	Engenharia Informática e de Computadores	Licenciatura	PYTHON
10	Universidade de Trás-os-Montes e Alto Douro - Escola de Ciências e Tecnologia	Engenharia Informática	Licenciatura	C
11	Universidade do Algarve - Faculdade de Ciências e Tecnologia	Engenharia Informática	Licenciatura	C
12	Universidade do Minho	Engenharia Informática	Mestrado Integrado	HASKELL
13	Universidade do Porto - Faculdade de Ciências	Ciência de Computadores	Licenciatura	C
14	Universidade do Porto - Faculdade de Engenharia	Engenharia Informática e Computação	Mestrado Integrado	SCHEME
15	Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia	Engenharia Informática	Mestrado Integrado	JAVA

	Politécnicos Públicos	Curso	Grau Académico	Linguagem
16	Instituto Politécnico de Coimbra - Instituto Superior de Engenharia de Coimbra	Engenharia Informática	Licenciatura	C
17	Instituto Politécnico de Lisboa - Instituto Superior de Engenharia de Lisboa	Engenharia Informática e de Computadores	Licenciatura	JAVA
18	Instituto Politécnico de Setúbal - Escola Superior de Tecnologia de Setúbal	Engenharia Informática	Licenciatura	JAVA
19	Instituto Politécnico do Porto - Instituto Superior de Engenharia do Porto	Engenharia Informática	Licenciatura	JAVA

Tabela B.1: Portuguesas

C

Gramática

Neste anexo encontra-se a gramática da linguagem Temple. Cada produção da gramática pode incluir não terminais, que têm a sua produção de modo a gerar uma árvore de gramática e se encontram em itálico. Os *tokens* ou terminais encontram-se a negrito de modo a se distinguirem melhor, a expressão `/*empty*/` representa uma regra vazia e existem ainda as expressões regulares, que encontram-se com a palavra-chave "ER" antes da expressão, que representam terminais mais complexos. De referir ainda que existem funções pré-definidas da linguagem Temple que não fazem parte da gramática da linguagem, mas sim da tabela de símbolos da linguagem.

program: global_declarations

global_declarations: /*empty*/
| *global_declaration global_declarations*

global_declaration: consts
| *function_def*
| *type_def*
| *struct_def*
| *var_declaration*

consts: **const** *id* **=** *expression* **;**

*function_def: type id***(***formal_args***) {***decls_stats***}**

type: id
| **int**
| **float**
| **void**
| **bool**
| **char**
| **string**
| **string<<***expression***>>**
| **ref** *type*
| **file**
| **array[***expressions***]** *type*

id : ER [a-zA-Z][_a-zA-Z0-9]*

formal_args: /* empty */
| *multiple_args*

multiple_args: single_arg
| *single_arg* **,** *multiple_args*

single_arg: type id
| **array** *args_arrays type id*

args_arrays: **[***commas***]**
| **[***commas***]** *args_arrays*

commas: /*empty*/
| *expression*
| *commas* **,**
| *commas* **,** *expression*

decls_stats: local_declaration decls_stats
| *statement decls_stats*
| /*empty*/

type_def: **typedef** *id type* **;**
| **typedef** *struct_def* **;**

struct_def: id **{***struct_members***} ;**

struct_members: struct_member struct_members
| *struct_member*

struct_member: type ids **;**

ids: id **,** *ids*
| *id*

local_declaration: vars_local
| *arrays*

var_declaration: vars
| *arrays*

vars_local: type id **;**
| *type id* **=** *expression* **;**
| *type id* **= {***ini_struct_members***} ;**

vars: type id **;**
| *type id* **=** *expression* **;**
| *type id* **= {***ini_struct_members***} ;**

ini_struct_members: ini_struct_member **,** *ini_struct_members*
| *ini_struct_member*

ini_struct_member: **.***id* **= {***item_seq***}**
| **.***id* **= {***ini_struct_member***}**

arrays: type id **= {***item_seq***} ;**

braces_seq: **{***item_seq***},** *braces_seq*
| **{***item_seq***}**

item_seq: braces_seq
| *expressions*

multiple_lits: int_float_lits
| *char_lits*

int_float_lits: int_lit1
| *int_lit1*, *int_float_lits*
| *float_lit1*
| *float_lit1*, *int_float_lits*

char_lits: *char_lit1*
| *char_lit1*, *char_lits*

single_lits: *int_lit1*
| *float_lit1*
| *char_lit1*
| *string_lit1*
| *bool_lit1*

string_lit1: " *string_lit2* "
| " "
| "\""

string_lit2: *char_lit2* *string_lit2*
| *char_lit2*

char_lit1: ' *char_lit2* '
| ' '
| '\''

char_lit2: ER [**a**-**z**A-**Z**0-**9**]
| ER [**!** | **#** | **$** | **%** | **&** | **␣** | **(** | **)**]
| ER [***** | **+** | **-** | **.** | **/** | **:** | **;**]
| ER [**<** | **=** | **>** | **?** | **@** | **[** | **]** | **^** | **_**]
| ER [**´** | **`** | **{** | **}** | **|** | **~**]
| ER [**\n** | **\t** | \\ ]]

float_lit1: ER [**0**-**9**]+**.**[**0**-**9**]+([**eE**](**+**|**-**)?[**0**-**9**]+)?

int_lit1: ER [**0**-**9**]+
| ER **0x**[**0**-**9a**-**fA**-**F**]+
| ER **0o**[**0**-**7**]+
| ER **0b**[**0**-**1**]+

bool_lit1: **true**
| **false**

statement: *l_value* **=** *expression* **;**
| *function_call* **;**
| *if_then_else*
| *switch_case*
| *label_cycles*
| **break** *id* **;**
| **break ;**
| **return ;**
| **return** *expression* **;**
| *input_output*
| *files_operation*
| **delete** *id* **;**

l_value: *id*
| *l_value***$**
| *l_value***.***id*
| *l_value***[***expressions***]**
| *l_value***<<***expression***>>**

expressions: *expression* **,** *expressions*
| *expression*

expression: *expression* **or** *expression*
| *expression* **and** *expression*
| *expression* **==** *expression*
| *expression* **!=** *expression*
| *expression* **<** *expression*
| *expression* **<=** *expression*
| *expression* **>** *expression*
| *expression* **>=** *expression*
| *expression* **+** *expression*
| *expression* **-** *expression*
| *expression* ***** *expression*
| *expression* **//** *expression*
| *expression* **%** *expression*
| *expression* **/** *expression*
| *value*

value: *single_lits*
| **new** *type*
| *l_value*
| *function_call*
| **(***expression***)**
| **-** *value*
| **+** *value*
| **not** *value*
| *open_file*
| *create_file*

function_call: *id*(*fcall_args*)

fcall_args: /*empty */
| *fcall_mult_args*

fcall_mult_args: *expression*
| *expression*, *fcall_mult_args*

open_file: **open(***expression* , *expression* , *expression***)**

create_file: **create(***expression* , *expression* , *expression***)**

if_then_else: **if** *expression* **then** *decls_stats elif* **endif**

elif: *else*
| **elif** *expression* **then** *decls_stats elif*

else: /*empty*/
| **else** *decls_stats*

switch_case: **switch** *expression case_list* **endswitch**

case_list: **case** *cases* **:** *decls_stats case_list*
| **default :** *decls_stats*
| **case** *cases* **:** *decls_stats*

cases: *multiple_lits*
| *int_lit1* **..** *int_lit1*

label_cycles: *cycles*
| *id* **:** *cycles*

cycles: *for_cycle*
| **while** *expression* **do** *decls_stats* **endwhile**
| **repeat** *decls_stats* **until** *expression* **endrepeat**

for_cycle: **for** *id* **=** *expression to_downto expression step* **do** *decls_stats* **endfor**

to_downto: **to**
| **downto**

step: **step** *expression*
| /*empty */

input_output: **input(***l_values***) ;**
| **output(***expressions***) ;**

files_operation: *write_file*
| *read_file*
| *read_line*
| *close_file*
| *goto_file*
| *seek_file*

write_file: **write(***l_value* **,** *expressions***) ;**

read_file: **read(***l_value* **,** *l_values***) ;**

l_values: *l_value* **,** *l_values*
| *l_value*

read_line: **readline(** *l_value* **,** *l_value***) ;**

close_file: **close(***l_value***) ;**

goto_file: **goto(***l_value* **,** *expression***) ;**

seek_file: **seek(***l_value* **,** *expression***) ;**

Bibliografia

[BAA03] Joe Bergin, Achla Agarwal, and Krishna Agarwal. Some deficiencies of C++ in teaching cs1 and cs2. *SIGPLAN Not.*, 38(6):9–13, June 2003.

[BBN10] Andrew Black, Kim B. Bruce, and James Noble. Panel: Designing the next educational programming language. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 201–204, New York, NY, USA, 2010. ACM.

[BCK08] Jens Bennedsen, Michael E Caspersen, and Michael Kölling. *Reflections on the teaching of programming: Methods and implementations, LNCS*, volume 4821. Springer, 2008.

[CMR98] David Clark, Cara MacNish, and Gordon F. Royle. Java as a teaching language — opportunities, pitfalls and solutions. In *Proceedings of the 3rd Australasian Conference on Computer Science Education*, ACSE '98, pages 173–179, New York, NY, USA, 1998. ACM.

[CS10] Marcus Crestani and Michael Sperber. Experience report: Growing programming languages for beginning students. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 229–234, New York, NY, USA, 2010. ACM.

[dRWT03] Michael de Raadt, Richard Watson, and Mark Toleman. Language tug-of-war: Industry demand and academic choice. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 137–142, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[DSB+00] Roger Duke, Eric Salzman, Jay Burmeister, Josiah Poon, and Leesa Murray. Teaching programming to beginners - choosing the language is just the first step. In *Proceedings of the Australasian Conference on Computing Education*, ACSE '00, pages 79–86, New York, NY, USA, 2000. ACM.

[Fou] Python Software Foundation. Python 3.7.3 documentation.

[GG12] Desmond Wesley Govender and Irene Govender. Are students learning object oriented programming in an object oriented programming course?: Student voices. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 395–395, New York, NY, USA, 2012. ACM.

[Gup04] Diwaker Gupta. What is a good first programming language? *Crossroads*, 10(4):7–7, 2004.

[Has] Haskell. Haskell documentation.

[Hon98] Jason Hong. The use of Java as an introductory programming language. *XRDS*, 4(4):8–13, May 1998.

[IBRS15] Mirjana Ivanović, Zoran Budimac, Miloš Radovanović, and Miloš Savić. Does the choice of the first programming language influence students' grades? In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, CompSysTech '15, pages 305–312, New York, NY, USA, 2015. ACM.

[ISO90] ISO/IEC 7185:1990 information technology — programming languages — Pascal. Standard, International Organization for Standardization, Geneva, CH, 1990.

[ISO17] ISO/IEC 14882:2017 information technology — programming languages — C++. Standard, International Organization for Standardization, Geneva, CH, 2017.

[ISO18] ISO/IEC 9899:2018 information technology — programming languages — C. Standard, International Organization for Standardization, Geneva, CH, 2018.

[JLTS11] Ambikesh Jayal, Stasha Lauria, Allan Tucker, and Stephen Swift. Python for teaching introductory programming: A quantitative evaluation. *Innovation in Teaching and Learning in Information and Computer Sciences*, 10(1):86–90, 2011.

[KA16] Wanda M. Kunkle and Robert B. Allen. The impact of different teaching approaches and languages on student learning of introductory programming concepts. *ACM Trans. Comput. Educ.*, 16(1):3:1–3:26, January 2016.

[KKR95] Michael Kölling, Bett Koch, and John Rosenberg. Requirements for a first year object-oriented teaching language. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '95, pages 173–177, New York, NY, USA, 1995. ACM.

[LAMJ05] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *SIGCSE Bull.*, 37(3):14–18, June 2005.

[LBBO16] Mark C. Lewis, Douglas Blank, Kim Bruce, and Peter-Michael Osera. Uncommon teaching languages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 492–493, New York, NY, USA, 2016. ACM.

[MC96] Linda McIver and Damian Conway. Seven deadly sins of introductory programming language design. In *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*, pages 309–316. IEEE, 1996.

[MdR06] Linda Mannila and Michael de Raadt. An objective comparison of languages for teaching introductory programming. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06, pages 32–37, New York, NY, USA, 2006. ACM.

[Mod91] R. P. Mody. C in education and software engineering. *SIGCSE Bull.*, 23(3):45–56, September 1991.

[MZ11] George McMaster and Michael Zastre. More concepts for teaching introductory programming. In *Proceedings of the 16th Western Canadian Conference on Computing Education*, WCCCE '11, pages 7–11, New York, NY, USA, 2011. ACM.

[Ora] Oracle. Oracle jdk 12 documentation.

[Ros00] Rocky Ross. Going backwards: Introductory programming languages. *SIGACT News*, 31(4):65–73, December 2000.

[RRR03] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[SC12] Michael Sperber and Marcus Crestani. Form over function: Teaching beginners how to construct programs. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, pages 81–89, New York, NY, USA, 2012. ACM.