UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VÍTOR BUJÉS UBATUBA DE ARAÚJO

Faz: uma linguagem funcional didática

Prof. Dr. Lucio Mauro Duarte
Orientador

Prof. Dr. Rodrigo Machado
Co-orientador

Porto Alegre, dezembro de 2013

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Araújo, Vítor Bujés Ubatuba De

Faz: uma linguagem funcional didática / Vítor Bujés Ubatuba De Araújo. – Porto Alegre: PPGC da UFRGS, 2013.

67 f.: il.

Trabalho de conclusão (graduação) – Universidade Federal do Rio Grande do Sul. Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2013. Orientador: Lucio Mauro Duarte; Co-orientador: Rodrigo Machado.

I. Duarte, Lucio Mauro. II. Machado, Rodrigo. III. Título.

“I never think of giving up. What use would that be? We can’t win by giving up.”
— RICHARD STALLMAN

AGRADECIMENTOS

Obrigado.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 7

LISTA DE FIGURAS 8

RESUMO 9

ABSTRACT 10

1 INTRODUÇÃO 11
1.1 Organização do texto 12

2 LINGUAGENS FUNCIONAIS 13
2.1 Conceitos básicos 13
2.2 O uso de Racket no ensino de programação 14

3 AS LINGUAGENS HTDP 16
3.1 Construções sintáticas 16
3.1.1 Tipos de dados básicos 16
3.1.2 Expressões aritméticas 18
3.1.3 Expressões relacionais e lógicas 18
3.1.4 Declaração de variáveis 18
3.1.5 Declaração e chamada de funções 18
3.1.6 Expressões condicionais 19
3.1.7 Estruturas de dados 19
3.1.8 Tipos mistos 19
3.1.9 Listas 20
3.1.10 Definições locais 21
3.1.11 Funções anônimas 22
3.1.12 Análise 22
3.2 Sistema de tipos 25
3.2.1 Análise 25

4 A LINGUAGEM FAZ 26
4.1 Construções sintáticas 26
4.1.1 Tipos de dados básicos 26
4.1.2 Expressões aritméticas 26
4.1.3 Expressões relacionais e lógicas 27
4.1.4 Blocos 27
4.1.5 Definição de variáveis 27

4.1.6 Definição e chamada de funções . . . 27
4.1.7 Condicionais . . . 28
4.1.8 Expressões de bloco . . . 28
4.1.9 Retorno e sinalização de erros . . . 28
4.1.10 O comando `teste` . . . 28
4.1.11 Definição de tipos não paramétricos . . . 28
4.1.12 Enumerações . . . 29
4.1.13 Estruturas de dados . . . 29
4.1.14 Tipos mistos . . . 29
4.1.15 Definição de tipos paramétricos . . . 29
4.1.16 Listas . . . 31
4.1.17 Definições locais . . . 32
4.1.18 Funções anônimas . . . 32
4.1.19 Análise . . . 32
4.2 Sistema de tipos . . . 36
4.2.1 Tipos da linguagem . . . 36
4.2.2 Compatibilidade de tipos e unificação . . . 40
4.3 Semântica e tipos das construções da linguagem . . . 43
4.4 Limitações . . . 47

5 IMPLEMENTAÇÃO . . . 48
5.1 O ambiente DrRacket . . . 48
5.2 A implementação de Faz . . . 49
5.2.1 Análise léxica e sintática . . . 49
5.2.2 Análise semântica . . . 50
5.2.3 Tradução . . . 51
5.2.4 Ambiente padrão . . . 51
5.2.5 Execução interativa . . . 51
5.2.6 Integração com DrRacket . . . 52
5.2.7 Limitações . . . 52

6 EXPERIMENTO DE VALIDAÇÃO . . . 53

7 TRABALHOS RELACIONADOS . . . 57
7.1 Linguagens baseadas no português . . . 57
7.2 Linguagens funcionais de propósito geral . . . 57
7.2.1 Família LISP . . . 57
7.2.2 Família ML . . . 58
7.3 Linguagens funcionais didáticas . . . 59
7.4 Outras linguagens didáticas . . . 60
7.5 Tipagem estática em Racket . . . 60

8 CONCLUSÃO . . . 62

REFERÊNCIAS . . . 64

LISTA DE ABREVIATURAS E SIGLAS

HtDP	How to Design Programs
ISλ	Intermediate Student with Lambda
SICP	Structure and Interpretation of Computer Programs

LISTA DE FIGURAS

Figura 2.1: Cálculo do fatorial em estilo imperativo. 14
Figura 2.2: Cálculo do fatorial em estilo funcional. 14
Figura 2.3: Definição incorreta de uma função que soma dois números. 15

Figura 3.1: Definição e uso de estruturas em ISλ 20
Figura 3.2: Definição e uso de um tipo misto *forma* em ISλ 20
Figura 3.3: Exemplo do uso de listas em ISλ 21
Figura 3.4: Exemplo de código com definições locais (`delta`, `x1`, `x2`) em ISλ . . 21
Figura 3.5: Função que realiza uma operação indicada por um símbolo sobre um valor numérico 23
Figura 3.6: Código com erro de tipo 25

Figura 4.1: Exemplo de condicional em Faz 28
Figura 4.2: Exemplo do uso do comando `teste` em Faz 29
Figura 4.3: Definição e uso de estruturas em Faz 30
Figura 4.4: Definição e uso de um tipo misto *forma* em Faz 30
Figura 4.5: Definição de árvore binária genérica em Faz 31
Figura 4.6: Exemplo do uso de listas em Faz 31
Figura 4.7: Exemplo de código com definições locais (`delta`, `x1`, `x2`) em Faz . . 32
Figura 4.8: Função que recebe um número ou uma string e produz um dado do mesmo tipo 37
Figura 4.9: Exemplo função polimórfica em Faz 39

Figura 5.1: Janela principal do ambiente DrRacket executando a linguagem ISλ . 49
Figura 5.2: Janela principal do ambiente DrRacket executando a linguagem Faz . 50

Figura 6.1: Programa para busca de caminho em grafo acíclico em ISλ 55
Figura 6.2: Programa para busca de caminho em grafo acíclico em Faz 56

RESUMO

A linguagem de programação Racket e o ambiente DrRacket são utilizados em diversos cursos introdutórios de programação. O uso de uma linguagem funcional facilita a exposição de certos conceitos, tais como recursão estrutural, reuso de código através de composição de funções e rotinas genéricas através do uso de funções de alta ordem. Além disso, o ambiente de desenvolvimento interativo propiciado pelo DrRacket facilita a criação, experimentação e teste de programas.

Por outro lado, devido ao fato de Racket ser uma linguagem dinamicamente tipada, declarações de tipos de dados e a noção de domínio dos argumentos de funções e dos campos de estruturas, conceitos frequentemente abordados em disciplinas introdutórias, não possuem uma representação direta na linguagem, sendo frequentemente representados por comentários no código. Além disso, a sintaxe de Racket, baseada nas convenções do LISP, diverge das convenções sintáticas utilizadas na notação matemática tradicional e na maior parte das linguagens de programação. Acreditamos que esse desencontro entre os conceitos apresentados e as expectativas dos alunos de um lado, e os recursos sintáticos e semânticos de Racket de outro, provoca dificuldades de exposição e compreensão do conteúdo de tais disciplinas.

Visando evitar esses problemas, desenvolvemos uma nova linguagem de programação, intitulada Faz. Essa linguagem incorpora um sistema de tipos semi-estático, buscando permitir a expressão dos domínios dos dados manipulados pelo programa e ao mesmo tempo manter a flexibilidade proporcionada pelo sistema de tipos de Racket. Além disso, a linguagem emprega uma sintaxe concebida para facilitar a compreensão dos programas pelos alunos. Para isso, utiliza recursos notacionais da matemática convencional e de outras linguagens de programação, bem como palavras-chave baseadas no português. A linguagem foi implementada como uma extensão do ambiente DrRacket, permitindo o uso de seus recursos interativos no desenvolvimento de programas na nova linguagem.

Palavras-chave: Linguagem de programação funcional, linguagem de programação didática.

Faz: a didactical functional programming language

ABSTRACT

The Racket programming language and the DrRacket environment are used in various courses of introduction to programming. The use of a functional language makes it easier to present certain concepts, such as structural recursion, code reuse through function composition, and generic routines through higher-order functions. Moreover, the interactive development environment provided by DrRacket eases the creation, experimentation and testing of programs.

On the other hand, because Racket is a dynamically typed programming language, data type declarations and the notion of domains of function arguments and structure fields, concepts often studied in introductory courses, do not have a direct representation in the language, being often represented by comments in the code. Moreover, the syntax of Racket, based in the conventions of LISP, diverges from the syntactical conventions used in traditional mathematical notation and most of the other programming languages. We believe that this mismatch between the concepts presented and the students' expectations in addition to the syntactic and semantic resources of Racket cause difficulties in the explanation and comprehension of the subject of such courses.

Aiming to avoid these problems, we have developed a new programming language, named Faz. This language incorporates a semi-static type system, with the intent of enabling the expression of the domains of the data manipulated by the program while keeping the flexibility provided by the type system of Racket. Moreover, the language employs a syntax devised to ease the comprehension of programs by the students, using notational resources from conventional mathematics and other programming languages, as well as Portuguese-based keywords. The language has been implemented as an extension to the DrRacket environment, allowing the use of its interactive features in the development of programs in the new language.

Keywords: functional programming language, introductory programming language.

1 INTRODUÇÃO

Linguagens de programação funcionais são utilizadas na academia e na indústria em diversas aplicações, tais como inteligência artificial (NORVIG, 1992), prova automatizada de teoremas (KAUFMANN; MOORE, 1996) (BOVE; DYBJER; NORELL, 2009) (BARRAS et al., 1997), compiladores (MARLOW; JONES et al., 2012), sistemas de versionamento de software (ROUNDY, 2005), entre outras. Conceitos de programação funcional também possuem aplicação mais ampla, estendendo-se a sistemas escritos em linguagens não funcionais. Exemplos incluem algoritmos de processamento paralelo e distribuído baseados em MapReduce (DEAN; GHEMAWAT, 2004). Há várias linguagens funcionais atualmente em uso. Dentre elas, destacam-se Haskell (MARLOW et al., 2010), OCaml (LEROY et al., 2012), Erlang (ARMSTRONG et al., 1996) e as linguagens da família LISP (MCCARTHY, 1960), tais como Common LISP (STEELE JR., 1990), Clojure (HICKEY, 2012) e Scheme (KELSEY et al., 1998).

No ensino de programação funcional, destaca-se a linguagem Racket[1], uma variante de Scheme. Racket é utilizada em cursos introdutórios de programação na UFRGS e em diversas universidades dos Estados Unidos, tais como a University of Texas[2], a Northeastern University[3] e a Rice University[4] , usualmente em conjunto com o ambiente de programação DrRacket e o livro-texto *How to Design Programs* (FELLEISEN et al., 2001). O uso de uma linguagem funcional facilita a exposição de certos conceitos de programação, tais como recursão estrutural, reuso de código através de composição de funções e rotinas genéricas através do uso de funções de alta ordem (FELLEISEN et al., 2004). Além disso, o ambiente de desenvolvimento interativo propiciado pelo DrRacket facilita a criação, experimentação e teste de programas, permitindo o teste de expressões e funções individuais, a visualização do conteúdo de estruturas de dados e a indicação visual do ponto de ocorrência de erros de sintaxe ou de execução no código.

Por outro lado, a sintaxe de Racket, baseada nas convenções do LISP, diverge das convenções sintáticas utilizadas na matemática tradicional e na maior parte das linguagens de programação. Exemplos de divergências incluem o uso de operadores aritméticos prefixados (e.g., `(+ 2 3)` em vez de `2+3`) e o posicionamento dos parênteses em chamadas de funções (e.g., `(f x y)` em vez de `f(x,y)`). Além disso, devido ao fato de Racket ser uma linguagem dinamicamente tipada, declarações de tipos de dados e a noção de domínio dos argumentos de funções e dos campos de dados estruturados, conceitos frequentemente abordados em disciplinas introdutórias, não possuem uma representação

[1] `http://racket-lang.org`

[2] `http://www.cs.utexas.edu/users/novak/cs307.html`

[3] `http://www.ccs.neu.edu/racket/`

[4] `http://courses.rice.edu/admweb/swkscat.main?p_action=CATALIST&p_acyr_code=2013&p_subj=COMP`

direta na linguagem; o livro-texto recorre ao uso de comentários no código para expressar esses conceitos. Consequentemente, o ambiente de programação ignora a informação de tipos e, portanto, não fornece ao aluno um *feedback* quanto à correção da mesma. Esse desencontro entre os conceitos apresentados e as expectativas dos alunos de um lado, e os recursos sintáticos e semânticos de Racket de outro, provoca dificuldades de exposição e compreensão do conteúdo de tais disciplinas.

Visando a evitar esses problemas, este trabalho propõe uma nova linguagem de programação baseada em Racket, denominada Faz, destinada ao ensino de cursos introdutórios de programação funcional. A motivação para a criação de uma nova linguagem deriva da experiência com o ensino da disciplina de Fundamentos de Algoritmos na UFRGS utilizando o livro-texto *How to Design Programs*. Essa disciplina é ministrada no primeiro ano dos cursos de graduação em Ciência da Computação e Biotecnologia. No curso de Ciência da Computação, a disciplina é ministrada paralelamente com a disciplina de Algoritmos e Programação, que emprega a linguagem imperativa C.

A linguagem desenvolvida emprega uma sintaxe concebida para facilitar a compreensão dos programas pelos alunos, utilizando, para isso, recursos notacionais da matemática convencional e de outras linguagens de programação. Tais recursos incluem operadores aritméticos infixados e o uso de uma notação similar à usada em teoria dos conjuntos para a expressão de tipos. A linguagem também emprega palavras-chave baseadas no português, tornando-a mais bem adaptada à realidade brasileira e facilitando a leitura do código por alunos iniciantes. Além disso, a linguagem incorpora um sistema de tipos semi-estático, buscando permitir a expressão dos domínios de dados manipulados pelo programa e, ao mesmo tempo, manter a flexibilidade proporcionada pelo sistema de tipos de Racket. Com isso, objetiva-se permitir que os tipos de problemas abordados pelo *How to Design Programs* sejam facilmente transponíveis para a nova linguagem.

A linguagem foi implementada como parte do ambiente DrRacket. O código em Faz é analisado sintática e semanticamente e posteriormente traduzido para a linguagem Racket. Isso torna possível o reuso das funcionalidades do ambiente DrRacket a partir da nova linguagem e permite que os recursos interativos do ambiente possam ser utilizados no no desenvolvimento e teste de programas na nova linguagem.

Este trabalho é uma tentativa de, através da elaboração cuidadosa de uma nova linguagem com sintaxe e semântica apropriadas, reduzir as dificuldades identificadas, produzindo uma linguagem mais facilmente compreensível pelos alunos e diminuindo a distância em termos sintáticos entre o paradigma imperativo e o funcional, sem que se perca o caráter funcional da linguagem e as vantagens que o acompanham.

1.1 Organização do texto

O Capítulo 2 apresenta conceitos básicos sobre linguagens funcionais e seu uso no ensino de programação. O Capítulo 3 apresenta um resumo das sublinguagens didáticas de *How to Design Programs*, juntamente com uma análise de suas vantagens e desvantagens em um contexto didático. O Capítulo 4 apresenta a nova linguagem, juntamente com uma discussão das decisões de design tomadas e de alternativas consideradas em seu desenvolvimento. O Capítulo 5 discute a implementação da linguagem no ambiente DrRacket. O capítulo 6 apresenta os resultados de uma enquete realizada de maneira a verificar a opinião dos alunos quanto às linguagens *How to Design Programs* e a nova linguagem. O Capítulo 7 discute trabalhos relacionados. O Capítulo 8 apresenta uma conclusão e discute trabalhos futuros.

2 LINGUAGENS FUNCIONAIS

Este capítulo apresenta conceitos básicos sobre programação funcional. Também será feita uma breve introdução sobre o uso das linguagens funcionais Scheme e Racket no ensino de programação.

2.1 Conceitos básicos

Linguagens funcionais são linguagens em que a computação é expressa primariamente através da aplicação e composição de funções matemáticas. Em uma linguagem imperativa, como Pascal, C ou Java, o resultado de uma chamada de função ou método frequentemente depende do estado do programa, representado por variáveis ou atributos mutáveis de objetos. Em contraste, em um programa puramente funcional, o resultado de uma chamada de função, dados os mesmos argumentos, é sempre o mesmo; não há expressões ou comandos com *efeitos colaterais*, i.e., que possuem algum efeito além de produzir um valor de retorno. Essa propriedade é conhecida como *transparência referencial*, e facilita o raciocínio sobre o comportamento do programa, bem como a elaboração de testes (HUGHES, 1989).

A literatura tradicionalmente classifica linguagens funcionais em puras e impuras. Em linguagens puramente funcionais, tais como Haskell, efeitos colaterais são totalmente abolidos, ou têm seu uso restrito apenas a contextos limitados. Por outro lado, linguagens funcionais impuras, tais como Scheme e ML, permitem o uso livre de efeitos colaterais, mas desencorajam essa prática, fornecendo construções de linguagem que favorecem a elaboração de programas puros.

Uma vez que almejam eliminar ou reduzir o uso de alterações de estado, linguagens funcionais usualmente não empregam comandos de iteração convencionais em linguagens imperativas, tais como *for* e *while*, visto que tais comandos dependem de mudanças de estado; em um programa puramente funcional, uma condição de parada tal como usada em um *while* retornaria sempre o mesmo valor-verdade, por exemplo. Em vez disso, programas funcionais usualmente expressam repetição por meio de funções definidas recursivamente, i.e., funções que chamam a si próprias durante sua execução.

Para demonstrar a diferença entre esses dois paradigmas, considere uma função que calcula o fatorial de um número natural escrita em um estilo imperativo (Figura 2.1) e em um estilo funcional (Figura 2.2) em C. Na versão imperativa, o resultado é calculado através de iteração, usando uma variável mutável como acumulador. Na versão funcional, o resultado é calculado através da composição de chamadas de função (o operador de multiplicação pode ser visto como uma função de dois argumentos).

Outra característica do paradigma funcional é o uso frequente de *funções de alta ordem*, isto é, funções que recebem outras funções como argumento e/ou produzem outras

```
int fac(int n) {
    int result = 1;
    int i;
    for (i=1; i<=n; i++)
        result = result * i;
    return result;
}
```

Figura 2.1: Cálculo do fatorial em estilo imperativo.

```
int fac(int n) {
    if (n==0)
        return 1;
    else
        return n * fac(n-1);
}
```

Figura 2.2: Cálculo do fatorial em estilo funcional.

funções como valor de retorno. Isso permite a definição de algoritmos genéricos, em que uma lacuna no código é preenchida por um argumento funcional. Por exemplo, pode-se definir uma rotina genérica de ordenamento de listas de elementos de tipos quaisquer, passando como argumento para a rotina uma função de comparação que determina a ordem desejada para os elementos.

2.2 O uso de Racket no ensino de programação

Scheme (SUSSMAN; JR., 1975) é uma linguagem funcional impura da família LISP. Assim como outras linguagens dessa família, Scheme emprega uma sintaxe minimalista, baseada em expressões aninhadas delimitadas por parênteses, em que a aplicação de funções e operadores é expressa uniformemente na forma `(`*`operador argumentos...`*`)`. Por exemplo, uma expressão aritmética como $2*3+4*5$ é expressa como `(+ (* 2 3) (* 4 5))` em Scheme; de fato, `+` e `*` são funções em Scheme. O padrão mais difundido de Scheme, o R5RS (KELSEY et al., 1998), define uma linguagem bastante reduzida, oferecendo um mínimo de recursos sobre os quais funcionalidades mais avançadas podem ser definidas. O R5RS é frequentemente utilizado como núcleo para implementações de versões estendidas da linguagem. Racket, previamente conhecido como PLT Scheme (FLATT; PLT, 2010), é uma dessas versões de Scheme, que dispõe de uma grande variedade de bibliotecas e inclui funcionalidades como estruturas, um sistema de módulos e sistemas avançados de macros, que permitem a definição de novas linguagens baseadas em Racket.

O uso de Scheme em cursos introdutórios de Ciência da Computação foi popularizado pelo livro-texto *Structure and Interpretation of Computer Programs* (SICP) (ABELSON; SUSSMAN, 1985), utilizado na disciplina homônima no MIT e adotado por diversas universidades no mundo. O livro-texto *How to Design Programs* (HtDP) (FELLEISEN et al., 2001) foi elaborado de maneira a corrigir o que os autores identificaram como problemas com o uso do SICP como material introdutório (FELLEISEN et al., 2004). Os autores apontam como vantagens do uso de uma linguagem funcional no ensino de programa-

```
(define (soma x y)
  x + y)
```

Figura 2.3: Definição incorreta de uma função que soma dois números.

ção o número reduzido de conceitos a serem apresentados e o fato de que o modelo de computação usado por linguagens funcionais pode ser visto como uma extensão da álgebra elementar, conhecida por alunos egressos do Ensino Médio. Por outro lado, o HtDP dá uma ênfase maior à parte de técnicas de elaboração de programas e utiliza problemas de uma natureza mais familiar a alunos iniciantes em vez de problemas matemáticos avançados e problemas relativos à interpretação e compilação de linguagens, tais como abordados no SICP.

A criação do HtDP é parte de um projeto maior dos autores denominado *Program by Design* (anteriormente *TeachScheme!*)[1]. O objetivo desse projeto é promover o ensino de programação em instituições de ensino médio e superior utilizando uma abordagem bem estruturada para a construção de programas. Além do livro-texto, o projeto produziu a linguagem Racket e o ambiente gráfico de programação DrRacket. O HtDP não utiliza a linguagem Racket completa; em vez disso, o livro define subconjuntos didáticos da linguagem completa, progredindo de uma linguagem mais limitada (*Beginner Student*) para linguagens com mais funcionalidades (*Intermediate Student*, *Advanced Student*). A motivação por trás disso é impedir que erros de programadores principiantes sejam interpretados como programas válidos com um comportamento inesperado. Por exemplo, um usuário sem experiência com a linguagem Racket poderia tentar definir (incorretamente) uma função como a da Figura 2.3. Na linguagem Racket completa, em vez de realizar a soma esperada pelo programador, o corpo da função seria interpretado como três expressões separadas, e o resultado da função seria o valor da última expressão, i.e., `y`. Nas sublinguagens didáticas, apenas uma expressão é permitida no corpo da função, produzindo uma mensagem de erro apropriada para o usuário no caso de um equívoco desse tipo.

As sublinguagens didáticas, bem como o Racket padrão, apresentam outras divergências com relação ao Scheme padrão visando torná-las mais didáticas, tais como a exibição de todos os tipos de valores com a mesma sintaxe baseada em construtores que é usada para criá-los, e o uso dos nomes `first` e `rest` em vez de `car` e `cdr` para as funções de acesso aos campos de listas encadeadas.

[1] `http://www.programbydesign.org/`

3 AS LINGUAGENS HTDP

As linguagens HtDP são subconjuntos didáticos da linguagem Racket. Essas linguagens estão organizadas em uma progressão (*Beginner Student, Beginner Student with List Abbreviations, Intermediate Student, Intermediate Student with Lambda, Advanced Student*), cada linguagem adicionando recursos à anterior. Como visto, a motivação para o uso de subconjuntos progressivamente maiores da linguagem completa é evitar que erros comuns de alunos sejam interpretados como programas válidos. A linguagem *Advanced Student* introduz operações de atribuição; todas as linguagens anteriores são puramente funcionais, exceto por algumas funções para manipulação de gráficos fornecidas por bibliotecas, que produzem efeitos colaterais, e de algumas funções como `random` e `current-seconds`, cujo valor de retorno varia a cada chamada. A linguagem Advanced Student não é abordada na disciplina de Fundamentos de Algoritmos.

As seções seguintes apresentam os recursos sintáticos e semânticos da linguagem *Intermediate Student with Lambda* (ISλ). Também será apresentada uma análise das vantagens e desvantagens do uso dessa linguagem em um contexto didático. Essa análise baseia-se na experiência do autor deste trabalho como monitor da disciplina de Fundamentos de Algoritmos por seis semestres, durante os quais observou-se a recorrência de uma série de erros e dúvidas entre os alunos.

3.1 Construções sintáticas

3.1.1 Tipos de dados básicos

Há cinco tipos de dados básicos que podem aparecer como constantes no código de um programa: números, booleanos, caracteres, strings e símbolos.

3.1.1.1 Números

ISλ suporta diversos tipos de números, tais como:

- Inteiros: `42`, `-1`;
- Números racionais: `355/113`, `-1/2`;
- Números em ponto flutuante: `3.141592`;
- Números complexos, cujas partes podem ser de qualquer um dos tipos acima: `1-2i`, `0+355/113i`.

ISλ faz uma distinção entre números exatos e inexatos: números inteiros, racionais e números complexos com componentes inteiras ou racionais são considerados exatos,

enquanto números em ponto flutuante são considerados inexatos e são prefixados com `#i`. Ao contrário do que ocorre na linguagem Racket completa, em ISλ números escritos em notação decimal no código (e.g., `1.5`) são convertidos internamente para os números racionais correspondentes. O objetivo disso é trabalhar com números exatos sempre que possível, evitando arrendondamentos que ocorrem com operações sobre números em ponto flutuante e que podem provocar confusão entre os alunos. Algumas operações, tais como `sqrt` (raiz quadrada), podem produzir valores inexatos, indicados pelo prefixo `#i` (e.g., `#i1.4142135`). O ambiente DrRacket pode ser configurado para exibir números racionais em notação decimal, tornando seu uso mais transparente.

A maior parte das operações matemáticas tratam os diversos tipos numéricos de maneira transparente, permitindo a combinação de tipos distintos de números e realizando conversões automaticamente quando necessário (e.g., é possível somar um inteiro e um número em ponto flutuante). Algumas funções, tais como `even?`, que testa se um número inteiro é par, produzem um erro de execução se utilizadas com um número do tipo inapropriado.

3.1.1.2 Booleanos

Os dois valores-verdade da linguagem são `true` e `false`. Ao contrário do que acontece em outras linguagens, como C e a linguagem Racket completa, em ISλ apenas esses dois valores podem ser usados em contextos que esperam valores-verdade, tais como expressões condicionais e operadores lógicos. Isso garante que se uma expressão não-booleana for usada em tal contexto, o usuário será informado do erro em tempo de execução.

3.1.1.3 Caracteres

Caracteres são representados através da sintaxe `#\`*c*, onde *c* é um caracter literal ou, no caso de caracteres especiais, o nome do caracter (e.g., `#\a`, `#\Newline`).

3.1.1.4 Strings

Strings são sequências de caracteres representadas, sintaticamente, pelos caracteres que as compõem entre aspas (e.g., `"hello, world"`). Caracteres especiais podem ser indicados utilizando uma sintaxe similar à da linguagem C, usando uma sequência de caracteres iniciada por `\` (e.g., `\n` para indicar uma quebra de linha).

3.1.1.5 Símbolos

Símbolos são um tipo de dados comumente encontrado nas linguagens da família LISP. Símbolos são constantes simbólicas, representadas sintaticamente em ISλ como `'`*nome*, onde *nome* é o nome dado ao símbolo. A propriedade de implementação que distingue símbolos de strings é que a linguagem garante que duas ocorrências de um símbolo de mesmo nome no código de um programa referenciam o mesmo objeto, e não meramente dois objetos com o mesmo conteúdo, como é o caso com strings idênticas. Isso significa que uma comparação de dois símbolos por igualdades pode ser implementada mais eficientemente como uma mera comparação de ponteiros, em vez de uma comparação dos caracteres que os compõem. Para os fins com que são usados ao longo de *How to Design Programs*, símbolos e strings são usualmente intercambiáveis.

3.1.2 Expressões aritméticas

Expressões aritméticas são escritas em notação prefixada, na forma:

```
(operador operandos...)
```

A maior parte das operações aritméticas aceita dois ou mais argumentos. A exigência dos parênteses em torno de cada operação elimina a necessidade de regras de precedência.

```
(+ 2 3)
(+ (* 2 3) (* 4 5))
(* (+ 2 3) (+ 4 5))
```

Uma expressão do tipo $ax^2 + bx + c$ pode ser escrita como

```
(+ (* a (expt x 2)) (* b x) c)}
```

3.1.3 Expressões relacionais e lógicas

ISλ utiliza funções de comparação distintas para cada tipo de dados. Por exemplo, a função = realiza um teste de igualdade entre dois números, `string=?` entre strings, `symbol=?` entre símbolos, e assim por diante. De maneira análoga, há funções tais como `<`, `string<?` e assim por diante para a comparação da ordem relativa de números reais, caracteres e strings.

A linguagem suporta os operadores lógicos convencionais (`and`, `or`, `not`). Assim como os operadores aritméticos, os operadores `and` e `or` são n-ários, computando a conjunção ou a disjunção de todos os seus argumentos. Assim como na linguagem C, os operadores `and` e `or` realizam avaliação em curto-circuito: seus argumentos são avaliados da esquerda para a direita, encerrando a avaliação assim que o resultado da expressão puder ser determinado. Por exemplo, em uma expressão como:

```
(or (> 5 2) (< 0 7))
```

apenas o primeiro argumento da disjunção (`(> 5 2)`) é avaliado, já que se qualquer um dos argumentos da disjunção for verdadeiro, o resultado será verdadeiro independentemente dos demais.

A linguagem não possui funções explícitas para testar se dois valores são diferentes. Para isso, utilizam-se os operadores de igualdade em conjunto com a negação lógica.

3.1.4 Declaração de variáveis

Variáveis podem ser declaradas usando a forma:

```
(define nome expressão)
```

Na linguagem ISλ, assim como em muitas linguagens funcionais, variáveis são imutáveis. Isto é, uma vez que um valor tenha sido associado à variável, não é possível atribuir um novo valor à mesma.

3.1.5 Declaração e chamada de funções

Funções podem ser declaradas usando a forma:

```
(define (nome parâmetros...)
  expressão)
```

Chamadas de funções são escritas com uma sintaxe análoga às operações aritméticas (de fato, os operadores aritméticos são funções em ISλ): `(nome argumentos...)`.

3.1.6 Expressões condicionais

Expressões condicionais são aquelas cujo resultado depende do resultado de um ou mais testes, representados por expressões booleanas. Em ISλ, expressões condicionais são escritas utilizando a forma `cond`:

```
;; sinal: número -> símbolo
;; Retorna um símbolo representando o sinal do número
;; passado como argumento.
(define (sinal n)
  (cond [(< n 0) 'negativo]
        [(> n 0) 'positivo]
        [else 'neutro]))
```

O corpo da expressão condicional é constituído de uma ou mais cláusulas, compostas por um par teste-resultado entre colchetes. Para computar o resultado da expressão condicional, o teste de cada cláusula é avaliado em sequência, até que seja encontrado um teste cujo valor-verdade seja `true`, ou que a cláusula `else` seja atingida. O resultado de tal cláusula é utilizado como o valor da expressão.

Quando se deseja realizar apenas um teste, a forma mais simples (`if` *`teste valor-se-verdadeio valor-se-falso`*) pode ser utilizada. Ao contrário do que ocorre na maior parte das linguagens imperativas, as formas `cond` e `if` são consideradas expressões, e podem ser utilizadas em qualquer contexto em que outras expressões possam ser usadas. Isso é análogo ao operador ternário *`teste`*`?` *`valor-se-verdadeiro`* `:` *`valor-se-falso`* da linguagem C. A forma `if` é usualmente evitada, por não possuir um delimitador explícito tal como `else`.

3.1.7 Estruturas de dados

Tipos de dados estruturados podem ser declarados usando a forma:

```
(define-struct nome-do-tipo (campos...))
```

Essa declaração cria um tipo estruturado com o nome e os campos especificados, bem como uma função construtora de valores do novo tipo (`make-`*`tipo`*), uma função predicado que testa se um dado valor pertence ao tipo (*`tipo`*`?`) e funções de projeção que retornam o valor de um dos campos dado um valor do tipo estruturado (*`tipo-campo`*). A Figura 3.1 apresenta a definição de um tipo estruturado `pessoa` com os campos `nome`, `idade` e `sexo` e exemplos das operações citadas sobre valores desse tipo.

3.1.8 Tipos mistos

Embora o fato de ISλ ser uma linguagem dinamicamente tipada impeça que o ambiente detecte erros de tipo em tempo de compilação, isso dá uma maior flexibilidade à linguagem, permitindo escrever funções que trabalham com dados de diversos tipos facilmente. Visando a abordar essa ideia de maneira mais estruturada, *How to Design Programs* utiliza o conceito de *tipos mistos*, i.e., tipos compostos pela união de outros tipos. Devido à linguagem não suportar declarações formais de tipo, tipos mistos são definidos informalmente por meio de comentários no código, como pode ser visto na Figura 3.2. Nesse trecho de código, são definidos os tipos estruturados `retângulo` e `círculo` e um tipo misto, na forma de um comentário, formado pela união dos dois tipos estruturados. A seguir, é definida uma função `área`, que aceita uma forma de qualquer um

```
; Definição do tipo de dados
(define-struct pessoa (nome idade sexo))

; Criação de uma instância do tipo
(define p (make-pessoa "Anna" 18 'f))

; Seleção de campos individuais
(pessoa-nome p)      ; devolve "Anna"
(pessoa-idade p)     ; devolve 18
(pessoa-sexo p)      ; devolve 'f

; Predicado de pertinência
(pessoa? p)          ; devolve true
(pessoa? 5)          ; devolve false
```

Figura 3.1: Definição e uso de estruturas em ISλ

```
(define-struct retângulo (lado altura)
(define-struct círculo (raio))

;; Uma forma é:
;; - um retângulo; ou
;; - um círculo.

;; área: forma -> número
;; Retorna a área de uma forma.
(define (área f)
  (cond
    [(retângulo? f) (* (retângulo-lado f) (retângulo-altura f))]
    [(círculo? f)   (* PI (expt (círculo-raio f) 2))]
    [else           (error "Forma desconhecida")]))
```

Figura 3.2: Definição e uso de um tipo misto *forma* em ISλ

dos tipos e computa sua área, utilizando as funções-predicado `retângulo?` e `círculo?` para determinar qual tipo foi passado como argumento em tempo de execução.

3.1.9 Listas

Listas são um tipo de dados estruturado nativo da linguagem ISλ. Em ISλ, listas são definidas indutivamente como:

- `empty` é uma lista (vazia);
- Se *x* é um valor qualquer e *L* é uma lista, então `(cons x L)` é uma lista.

Em termos de implementação, trata-se de listas simplesmente encadeadas. `cons` é o construtor dos elos da lista. Cada elo possui dois componentes: um valor da lista e um ponteiro para o restante da lista. Um elo é um valor estruturado cujos campos podem ser obtidos pelas funções de projeção `first` e `rest`. Os predicados `empty?` e `cons?` permitem testar se um valor é uma lista vazia ou um elo (i.e., uma lista não-vazia), respectivamente. O predicado `list?` permite testar se algo é uma lista (vazia ou

```
; Cria uma de uma lista de três elementos e associa-a
; a uma variável.
(define lista (cons 1 (cons 2 (cons 3 empty))))

(first lista)   ; devolve 1
(rest lista)    ; devolve (cons 2 (cons 3 empty))

(cons? lista)   ; devolve true
(cons? empty)   ; devolve false
(empty? lista)  ; devolve false
(empty? empty)  ; devolve true

(list? lista)   ; devolve true
(list? empty)   ; devolve true
(list? 42)      ; devolve false
```

Figura 3.3: Exemplo do uso de listas em ISλ

```
;; bhaskara: número número número -> lista-de-números
;; Dados os coeficientes de uma equação do segundo grau,
;; retorna uma lista com suas raízes reais.
(define (bhaskara a b c)
  (local ((define delta (- (* b b) (* 4 a c))))
    (cond
      [(< delta 0) empty]
      [else
       (local ((define x1 (/ (- (- b) (sqrt delta)) (* 2 a)))
               (define x2 (/ (+ (- b) (sqrt delta)) (* 2 a))))
         (cond
           [(= delta 0) (list x1)]
           [else (list x1 x2)]))])))
```

Figura 3.4: Exemplo de código com definições locais (`delta`, `x1`, `x2`) em ISλ

não). `(list? x)` é equivalente a `(or (empty? x) (cons? x))`. A Figura 3.3 demonstra a criação de uma lista contendo os valores `1`, `2` e `3` e o uso das funções citadas.

Para facilitar a criação de listas, a linguagem oferece uma função `list`, que recebe um número arbitrário de argumentos e retorna uma lista cujos elementos são os argumentos em sequência, evitando uma sequência de construtores `cons` aninhados.

3.1.10 Definições locais

A linguagem utiliza o operador `local` para introduzir definições de variáveis e funções com escopo limitado. Sua sintaxe é:

```
(local ( sequência de formas define )
  expressão na qual as definições são visíveis)
```

A Figura 3.4 mostra um exemplo de função com definições locais.

3.1.11 Funções anônimas

A forma `lambda` é uma construção sintática que permite expressar funções sem que seja necessário dar-lhes um nome, da mesma maneira que é possível construir valores estruturados sem que seja necessário atribuí-los a uma variável. Sua sintaxe é:

```
(lambda (parâmetros da função)
  corpo da função)
```

O conceito de função anônima usualmente não é abordado em disciplinas introdutórias, mas é aqui apresentado para fins de comparação com a nova linguagem no capítulo seguinte.

3.1.12 Análise

Durante o ensino da disciplina de Fundamentos de Algoritmos utilizando a linguagem $IS\lambda$ e seus subconjuntos, pôde-se observar diversos pontos positivos e negativos da sintaxe da linguagem Racket de um ponto de vista didático. Segue uma enumeração desses pontos.

Tipos de dados primitivos. O uso transparente dos diversos tipos numéricos é um ponto positivo de ISλ em comparação a outras linguagens, pois permite que o aluno trabalhe com dados numéricos de maneira similar às entidades matemáticas que representam. Da mesma forma, o uso de números exatos sempre que possível, e a representação interna de números decimais como racionais, evita preocupações com questões de arredondamento. Por outro lado, a distinção sintática entre números exatos e inexatos através de um prefixo, embora teoricamente interessante, é uma distração desnecessária em uma disciplina introdutória.

A sintaxe peculiar para representação de caracteres e o uso de uma sintaxe distinta para representação de caracteres especiais em caracteres e strings (`#\Newline` vs `"\n"`) são aspectos desnecessariamente confusos da linguagem.

A distinção entre símbolos e strings é relativamente arbitrária, e não há uma regra bem definida de quando cada um dos tipos deve ser usado. O tipo símbolo é uma herança do LISP, em que símbolos são usados, entre outras coisas, para representar identificadores no código em contextos de metaprogramação, em que programas manipulam código como dados. No contexto de uma disciplina introdutória, não há uma necessidade para um tipo simbólico distinto.

Símbolos são também frequentemente usados em situações em que outras linguagens utilizam enumerações. A linguagem ISλ não provê nenhum meio para declarar enumerações. A introdução de tal mecanismo eliminaria o principal caso de uso em que símbolos poderiam ser considerados mais apropriados do que strings. O uso de enumerações declaradas tem ainda a vantagem de permitir ao ambiente indicar para o usuário usos de constantes não definidas, ao contrário do que acontece em ISλ, onde qualquer símbolo pode ser usado sem declaração prévia e erros de digitação podem provocar falhas silenciosas em um programa.

Expressões aritméticas. O uso da notação prefixada delimitada por parênteses torna o agrupamento das operações explícito, evitando a necessidade de regras de precedência, e unifica a sintaxe dos operadores aritméticos e das chamadas de função. Por outro lado, essa notação diverge bastante da notação matemática convencional e é uma frequente fonte de confusão entre os alunos, especialmente em expressões maiores nas quais o aninhamento dos parênteses não é evidente.

```
; Realiza a operação indicada sobre o valor. Se a operação
; for desconhecida, retorna o valor intacto.
(define (calcula operação valor)
  (cond [(symbol=? operação 'quadrado) (* valor valor)]
        [(symbol=? operação 'módulo) (abs valor)]
        [else valor]))

(calcula 'abs -3)   ; Chamada correta
(calcula -3 'abs)   ; Chamada incorreta
```

Figura 3.5: Função que realiza uma operação indicada por um símbolo sobre um valor numérico

Expressões relacionais e lógicas. O uso de funções distintas para comparações de valores de diferentes tipos é benéfico em uma linguagem dinamicamente tipada, pois ajuda a detectar erros de tipo em tempo de execução. Considere o código na Figura 3.5, que realiza a operação indicada por um símbolo sobre um número. Se, ao chamar essa função, a ordem dos argumentos for invertida, a função `symbol=?` detectará o número passado no lugar do símbolo, produzindo um erro de execução que será notificado ao usuário. Por outro lado, se uma função de comparação genérica fosse utilizada, os testes de igualdade produziriam `false` sem provocar erros (`-3` é diferente de `'quadrado` e de `'módulo`), dificultando a detecção da chamada incorreta. Efetivamente, as funções de comparação de ISλ introduzem verificações dinâmicas de tipo no programa.

Por outro lado, essas verificações são úteis primariamente para suprir a ausência de um sistema de tipos estático em ISλ. Como será visto na Seção 3.2.1, essa ausência de verificações estáticas de tipo dificultam a detecção precisa do ponto em que se encontram os erros de tipo do programa. Na presença de um sistema de tipos estático, o uso de funções distintas de comparação para tipos diferentes torna-se desnecessário.

Declaração e chamada de funções. A colocação dos parênteses em torno da expressão completa (e.g., `(f x y)`, em vez de apenas em torno dos argumentos (e.g., `f(x,y)`) provoca uma série de confusões. Primeiro, a sintaxe diverge da convenção matemática e da maior parte das linguagens de programação, que utilizam a segunda forma. Segundo, a sintaxe para definição de funções, análoga à sintaxe da chamada, é inconsistente com a sintaxe para definição de estruturas, que mantém o nome do tipo estruturado a ser definido isolado dos demais parâmetros, como será visto a seguir. Além disso, o uso de `define` para definir tanto variáveis quanto funções, ainda que simples, causa certa confusão. A oposição entre os operadores `define` (sem qualificadores indicando de que tipo de definição se trata) e `define-struct` (que indica explicitamente que trata-se de uma definição de estrutura) é inconsistente e também provoca dúvidas.

A linguagem não possui nenhum mecanismo para a declaração formal dos tipos dos parâmetros e de retorno das funções. Uma vez que esse é um conceito importante para fins de organização e documentação do código, o livro-texto HtDP supre essa falta por meio de comentários no código indicando os tipos de entrada e saída de cada função, denominados *contratos*. O fato de que tais comentários não são analisados pela linguagem implica que os alunos não têm como testar a correção dessas declarações por meio da execução do programa, além de desmotivá-los a escrevê-las.

Expressões condicionais. A ausência de um delimitador explícito entre o teste e o resultado de cada cláusula do `cond` são uma causa comum de confusão entre os alunos em

programas maiores, onde tanto o teste quanto o resultado são expressões mais complexas e é difícil visualizar claramente que expressão está sendo delimitada por cada par de parênteses.

O fato de que a linguagem é baseada em expressões e o valor da expressão que constitui o corpo da função é implicitamente o valor de retorno da função, sem o uso de uma palavra explícita como o `return` de linguagens imperativas, torna menos evidente que o valor do resultado de uma cláusula da expressão condicional é também o valor de retorno da função como um todo.

O uso de uma única expressão condicional, em vez de uma cadeia de `if`s e `else`s, para a realização de múltiplos testes, evita aninhamento excessivo no código e lembra a notação comumente usada na matemática para indicar funções definidas por partes.

Estruturas de dados. A sintaxe para declaração de estruturas apresenta uma inconsistência em relação à sintaxe para declaração de funções quanto à colocação dos parênteses, como mencionado anteriormente. Além disso, analogamente às funções, não é possível declarar os tipos dos campos das estruturas, sendo necessário recorrer a comentários.

A sintaxe usada para as funções de projeção é uma fonte notória de confusão entre os alunos. Erros frequentes incluem: esquecer o argumento sobre o qual a projeção deve ser realizada, utilizar o argumento como prefixo em vez do nome do tipo (i.e., `p-nome` em vez de `(pessoa-nome p)` para extrair o campo `nome` de uma pessoa `p`), esquecer os parênteses em torno da expressão (por não ser claro que os seletores são funções), esquecer o prefixo `make-` na chamada do construtor, confusão entre o nome do tipo e uma variável contendo um valor do tipo, entre outros.

Tipos mistos. A flexibilidade proporcionada pelos tipos mistos é explorada em diversos exercícios de *How to Design Programs*. Essa funcionalidade é um ponto forte da linguagem ISλ. Por outro lado, ISλ sacrifica toda a segurança estática para atingir essa flexibilidade, o que não é estritamente necessário.

Listas. Listas são apenas um tipo particular de dado estruturado, mas a linguagem não as trata de maneira consistente com os tipos de dados definidos pelo usuário com a forma `define-struct`; ISλ utiliza nomes especiais para os operadores de construção e projeção que não seguem as mesmas convenções utilizadas para outros tipos estruturados, tais como `cons` em vez de `make-cons` e `first` em vez de `cons-first`. Se por um lado os nomes utilizados são mais concisos, por outro lado eles obscurecem a relação entre estruturas e listas. Além disso, um argumento em favor de operadores mais concisos reforça o fato de que os operadores para manipulação de estruturas definidas pelo usuário são demasiadamente verbosos. Se uma sintaxe mais simples fosse adotada para a manipulação de estruturas em geral, a mesma sintaxe poderia ser usada para listas sem prejuízo à legibilidade do código.

Definições locais. Definições locais são frequentemente úteis para simplificar expressões complexas em um trecho de código, evitando o aninhamento excessivo de expressões e permitindo dar nomes significativos a subexpressões. ISλ utiliza uma sintaxe desnecessariamente complexa para definições locais, em que a colocação do parênteses e a delimitação entre as definições e a expressão-resultado é frequentemente confusa. Por ser uma forma distinta das definições globais, a apresentação do conceito de definição local é usualmente postergada para o final da disciplina, o que impede que os alunos se beneficiem desse recurso no desenvolvimento de programas ao longo da disciplina, dificultando a compreensão e estruturação de programas maiores. Idealmente, a sintaxe para definições locais deveria ser o mais próxima possível da sintaxe para definições globais, permitindo a introdução do conceito mais cedo na disciplina.

```
(define (dobro x)
  (* 2 x))

(define (f x y)
  (+ (dobro x)
     (dobro y)))

(f 1 'a)
```

Figura 3.6: Código com erro de tipo

3.2 Sistema de tipos

ISλ é uma linguagem dinamicamente tipada, i.e., erros de tipo, tais como desencontro entre os tipos e número de parâmetros esperados por uma função e os tipos e número de argumentos utilizados em uma chamada da função só são detectados em tempo de execução, quando uma operação primitiva da linguagem recebe um argumento de tipo inválido.

Considere o código na Figura 3.6. Esse código define uma função que calcula o dobro de um número, uma função `f` que calcula a soma do dobro de dois números e faz uma chamada a `f` usando um número e um símbolo como argumentos. Ao avaliar essa chamada de função, o argumento `'a` será passado para a função `f`, que o passará à função `dobro`, que por sua vez o passará à função de multiplicação `*`. Somente ao tentar avaliar a multiplicação entre `2` e `'a` a linguagem detectará o erro.

3.2.1 Análise

Se por um lado a tipagem dinâmica dá uma certa flexibilidade à linguagem, permitindo a escrita de funções que trabalham com argumentos de tipos diversos de maneira mais livre, como visto anteriormente, por outro lado isso limita a capacidade do ambiente de indicar com precisão trechos de código incorretos.

Considere o código da Figura 3.6. Embora o erro se encontre na chamada final à função `f`, que foi escrita de maneira a trabalhar com números, e não símbolos, a linguagem somente detectará o erro ao tentar realizar a operação de multiplicação entre `2` e `'a`, indicando o trecho de código no corpo da função `dobro` como o ponto de erro. Cabe então ao usuário determinar de que maneira o valor `'a` chegou à função `dobro`. Outro problema da tipagem dinâmica é que, como já mencionado na Seção 3.1.12, a linguagem não permite a declaração formal dos tipos dos parâmetros e de retorno das funções e dos campos de estruturas.

Um sistema de tipos estático evitaria esse problema: uma vez que os tipos dos parâmetros das funções `dobro` e `f` tenham sido declarados e que as funções tenham sido consideradas corretas pelo sistema de tipos, a linguagem seria capaz de identificar a chamada a `f` como o ponto de erro, evitando a propagação do erro para outras funções.

4 A LINGUAGEM FAZ

Neste trabalho, foi desenvolvida uma nova linguagem de programação, intitulada Faz. A linguagem foi desenvolvida visando a evitar os problemas com as linguagens didáticas HtDP identificados no capítulo anterior, mas ao mesmo tempo manter suas vantagens em um contexto didático.

As seções seguintes apresentam os recursos sintáticos e semânticos da nova linguagem, bem como uma discussão das decisões de *design* tomadas e alternativas consideradas.

4.1 Construções sintáticas

4.1.1 Tipos de dados básicos

Há quatro tipos de dados básicos que podem aparecer como constantes no código de um programa em Faz.

Números. A linguagem possui sintaxe para números inteiros (e.g., `42`, `-1`), números em ponto flutuante (e.g., `3.141592`) e números imaginários (e.g., `1.5i`). Números racionais podem ser obtidos através do operador de divisão (e.g., `1/2`). Números complexos podem ser obtidos através da soma ou subtração de um número real e um imaginário (e.g., `1+2i`). Diferentemente de ISλ, não há uma distinção sintática entre números exatos e inexatos.

Caracteres. Caracteres são representados através da sintaxe `'c'`, de maneira análoga à linguagem C e derivadas (C++, Java, etc.). Caracteres especiais também são indicados utilizando uma sintaxe análoga a C, utilizando uma sequência de caracteres iniciada por `\` (e.g., `'\n'` para indicar o caractere de quebra de linha).

Strings. A sintaxe para strings é análoga à de ISλ e C: os caracteres que compõem a string são escritos entre aspas (e.g., `"hello, world"`). Caracteres especiais em strings são indicados utilizando a mesma sintaxe usada para caracteres especiais individuais (e.g., `"\n"` representa uma string contendo uma quebra de linha).

Booleanos. As constantes `verdadeiro` e `falso` constituem os valores booleanos da linguagem.

4.1.2 Expressões aritméticas

Expressões aritméticas são escritas de maneira similar à maioria das linguagens de programação, utilizando operadores infixados com as precedências convencionais da notação algébrica. Parênteses podem ser usados para alterar a ordem de avaliação das expressões. Os operadores aritméticos `+`, `-`, `*` (multiplicação), `/` (divisão) e `^` (exponenciação) são suportados.

```
2+3
2*3 + 4*5
(2+3) * (4+5)
```

Uma expressão do tipo $ax^2 + bx + c$ pode ser escrita como

```
a*x^2 + b*x + c
```

4.1.3 Expressões relacionais e lógicas

Os operadores relacionais de Faz são análogos aos da linguagem C: == (igualdade), != (diferença), <=, <, >, >=. Os mesmos operadores podem ser usados para comparar dados dos diferentes tipos, quando aplicáveis.

Os operadores lógicos são `e`, `ou` e `não`. Assim como em ISλ, os operandos de `e` e `ou` são avaliados em curto-circuito da esquerda para a direita.

4.1.4 Blocos

O código em Faz é estruturado em blocos, que definem o escopo das definições de variáveis e funções da linguagem. Um bloco consiste de zero ou mais definições de variáveis e funções locais, seguidas de um comando final, que é avaliado no escopo das declarações. O comando final produz um valor, que é tomado como o valor do bloco. O comando final pode ser um comando `devolve` *`expressão`*, análogo ao `return` de linguagens imperativas, ou um comando condicional. Alternativamente, o comando final pode ser da forma `erro` *`expressão`*, que não produz um valor de retorno, mas sinaliza um erro de execução. As subseções seguintes descrevem cada um desses elementos.

4.1.5 Definição de variáveis

Variáveis podem ser declaradas usando a forma:

```
seja nome = expressão
```

Por padrão, o tipo da variável é inferido a partir do tipo da expressão. Alternativamente, o tipo pode ser declarado explicitamente usando a forma:

```
seja nome ∈ tipo = expressão
```

Assim como em ISλ, variáveis são imutáveis.

4.1.6 Definição e chamada de funções

Funções podem ser declaradas usando a forma:

```
função nome(param1 ∈ tipo1, param2 ∈ tipo2, ...) -> tipo de retorno
    corpo da função
```

As declarações de tipos dos parâmetros e de retorno da função são obrigatórias. O corpo da função é um bloco. Quando a função é chamada, o bloco é avaliado e o valor produzido pelo comando final do bloco é retornado pela função.

Chamadas de função possuem a forma *`função`*`(`*`arg1`*`, `*`arg2`*`, ...)`.

```
função sinal(n ∈ Números) -> Strings
    se n<0 devolve "negativo"
    se n==0 devolve "neutro"
    senão devolve "positivo"
```

Figura 4.1: Exemplo de condicional em Faz

4.1.7 Condicionais

Condicionais são comandos em Faz, e não expressões, como é o caso em ISλ. Condicionais possuem a forma:

```
se expressão-teste1 bloco1
se expressão-teste2 bloco2
...
senão bloco
```

Se o condicional é o comando final de um bloco, o valor produzido pelo condicional é tomado como o valor do bloco. Ao contrário de ISλ, em Faz a cláusula `senão` é obrigatória. Isso ocorre porque a cláusula `senão` implicitamente delimita o comando condicional. A Figura 4.1 apresenta um exemplo de condicional.

4.1.8 Expressões de bloco

Além de poderem ocorrer como o corpo de uma função ou de uma cláusula de condicional, um bloco pode ser escrito entre parênteses. Nesse caso, o valor produzido pelo bloco é capturado e pode ser utilizado como uma expressão. Por exemplo, `1 + (se 5>2 devolve 10 senão devolve 20)` é uma expressão válida e produz o valor `11`.

4.1.9 Retorno e sinalização de erros

O comando `devolve` *`expressão`* computa o valor da expressão e o usa como o valor de retorno do bloco. O comando `erro` *`expressão`* computa o valor da expressão, que deve produzir uma string, e encerra a execução do programa, emitindo ao usuário uma mensagem de erro contendo a string e o nome da função onde ocorreu o erro.

4.1.10 O comando `teste`

Além de definições de variáveis, funções e tipos, o corpo do programa também pode conter expressões a serem avaliadas quando o programa for executado. Essas expressões possuem um papel similar a uma função principal como `main` em C, com a diferença de que o valor computado pelas expressões é automaticamente impresso. Na linguagem Faz, essas expressões são introduzidas pelo comando `teste` *`expressão`*. A Figura 4.2 demonstra a definição de uma função e uma chamada à mesma a ser avaliada quando da execução do programa.

4.1.11 Definição de tipos não paramétricos

A palavra `tipo` é usada para introduzir novos tipos de dados. A sintaxe é:

```
tipo nome-do-tipo = componentes
```

Onde *`componentes`* é uma das seguintes alternativas:

```
função fac(n ∈ Números) -> Números
    se n==0
        devolve 1
    senão
        devolve n * fac(n-1)

teste fac(5)
```

Figura 4.2: Exemplo do uso do comando `teste` em Faz

- O nome de um tipo já existente. Nesse caso, define-se um sinônimo para esse tipo.
- Uma lista de *construtores* entre chaves, separados por vírgulas. Construtores podem ser nomes simples, permitindo a definição de *enumerações*, ou da forma *nome*`(`*param1* `∈` *tipo1*`,` *param2* `∈` *tipo2*`...)`, permitindo a definição de *estruturas*.
- Uma união de outros componentes, escritos na forma *componente1* `U` *componente2*, permitindo a definição de tipos mistos.

4.1.12 Enumerações

Definindo-se um tipo em termos de construtores simples, efetivamente criam-se tipos enumerados. Por exemplo, a seguinte declaração define uma enumeração de cores:

```
tipo Cores = { vermelho, verde, azul }
```

Além do tipo em si, a declaração também define as constantes `vermelho`, `verde` e `azul` no programa.

4.1.13 Estruturas de dados

Definindo-se um tipo em termos de construtores com parâmetros, criam-se tipos de dados estruturados. Uma vez definido o tipo de dados, pode-se criar novos valores do tipo invocando-se o construtor definido, extrair os campos de um valor estruturado através do operador `de` e testar se um valor pertence ao tipo através do operador ∈. A Figura 4.3 demonstra a definição de um tipo de dados para representar pessoas, análogo ao visto na Figura 3.1, e exemplos de operações sobre valores desse tipo.

4.1.14 Tipos mistos

Tipos mistos são suportados diretamente em Faz na forma de uniões de tipos. A Figura 4.4 demonstra a declaração de um tipo misto `Formas`, constituído de retângulos e círculos, e a definição de uma função `área`, que aceita uma forma qualquer e computa sua área, de maneira análoga ao código da Figura 3.2.

4.1.15 Definição de tipos paramétricos

É possível definir tipos genéricos parametrizados por outros tipos. A sintaxe é análoga à declaração de tipos não paramétricos, mas inclui uma lista de variáveis de tipo introduzidas pela palavra `de`:

```
tipo nome-do-tipo de parâmetros = componentes
```

```
# Definição dos tipos de dados

tipo Sexos = { masculino, feminino }

tipo Pessoas = { pessoa(nome ∈ Strings,
                        idade ∈ Números,
                        sexo ∈ Sexos) }

# Criação de uma instância do tipo
seja p = pessoa("Anna", 18, feminino)

# Seleção de campos individuais
nome de p               # devolve "Anna"
idade de p              # devolve 18
sexo de p               # devolve feminino

# Teste de pertinência
p ∈ Pessoas            # devolve verdadeiro
5 ∈ Pessoas            # devolve falso
```

Figura 4.3: Definição e uso de estruturas em Faz

```
# Definição de tipos para cada forma
tipo Retângulos = { retângulo(lado ∈ Números, altura ∈ Números) }
tipo Círculos = { círculo(raio ∈ Números) }

# Definição do tipo misto
tipo Formas = Retângulos U Círculos

função área(f ∈ Formas) -> Números
    # Retorna a área de uma forma.
    se f ∈ Quadrados
        devolve lado de f * altura de f
    se f ∈ Círculos
        devolve pi * (raio de f)^2
    senão
        erro "Forma desconhecida"
```

Figura 4.4: Definição e uso de um tipo misto *forma* em Faz

```
tipo Árvores de ?X = { vazia, nó(valor ∈ ?X,
                                 esquerda ∈ Árvores de ?X,
                                 direita ∈ Árvores de ?X) }
```

Figura 4.5: Definição de árvore binária genérica em Faz

```
# Cria uma de uma lista de três elementos e
# associa-a a uma variável.
seja lista = elo(1, elo(2, elo(3, vazio)))

primeiro de lista          # devolve 1
resto de lista             # devolve elo(2, elo(3, vazio))

lista == vazio             # devolve falso

resto de resto de resto de lista == vazio
                           # devolve verdadeiro

lista ∈ Listas de Números  # devolve verdadeiro
vazio ∈ Listas de Números  # devolve verdadeiro
42 ∈ Listas de Números     # devolve falso
```

Figura 4.6: Exemplo do uso de listas em Faz

Onde *parâmetros* pode ser uma variável de tipo, na forma `?var`, ou uma lista de variáveis de tipo entre parênteses separadas por vírgula. Por exemplo, um tipo genérico de árvores binárias pode ser definida como na Figura 4.5. A semântica das variáveis de tipo e tipos paramétricos serão vistos em maior detalhe na Seção 4.2.1.

4.1.16 Listas

Listas são um tipo de dados paramétrico estruturado pré-definido em Faz. Esse tipo poderia ser definido em Faz como:

```
tipo Listas de ?X = { vazio,
                      elo(primeiro ∈ ?X, resto ∈ Listas de ?X) }
```

Isto é, uma lista pode ser vazia (representada pela constante `vazio`) ou um elo (análogo ao `cons` de ISλ) constituído por dois componentes: `primeiro`, o primeiro elemento da lista; e `resto`, uma lista contendo os elementos restantes, que novamente pode ser vazia ou outro elo.

Por ser definida da mesma maneira que os outros dados estruturados, listas em Faz podem ser manipuladas utilizando os mesmos operadores válidos para estruturas definidas pelo usuário. A Figura 4.6 demonstra a criação de uma lista contendo os valores `1`, `2` e `3` e o uso de algumas operações sobre listas.

Assim como ISλ, Faz oferece uma sintaxe abreviada para a criação de listas: listas podem ser descritas listando-se seus elementos em ordem entre colchetes, separados por vírgulas. Por exemplo, `[1,2,3]` é equivalente a `elo(1, elo(2, elo(3, vazio)))`.

```
função bhaskara(a ∈ Números, b ∈ Números, c ∈ Números) -> Listas de Números
    # Dados os coeficientes de uma equação do segundo grau,
    # retorna uma lista com suas raízes reais.
    seja delta = b^2 - 4*a*c
    se delta < 0
        devolve []
    senão
        seja x1 = (-b - raiz(delta)) / (2*a)
        seja x2 = (-b + raiz(delta)) / (2*a)
        se delta == 0
            devolve [x1]
        senão
            devolve [x1, x2]
```

Figura 4.7: Exemplo de código com definições locais (`delta`, `x1`, `x2`) em Faz

4.1.17 Definições locais

Definições locais são feitas através da mesma sintaxe usada para definições globais; o bloco em que a definição ocorre define seu escopo. A Figura 4.7 mostra um exemplo de função com definições locais.

4.1.18 Funções anônimas

Funções anônimas podem ser expressas utilizando a forma `função`, de maneira similar à declaração de funções, omitindo-se o nome da função e envolvendo-se a forma entre parênteses:

```
(função (param1 ∈ tipo1, param2 ∈ tipo2, ...) -> tipo de retorno
     corpo da função)
```

4.1.19 Análise

Tipos de dados básicos. Faz elimina símbolos como um tipo de dados da linguagem, removendo assim a ambiguidade no uso de símbolos e strings presente em ISλ. A sintaxe para caracteres foi alterada de modo a torná-la análoga à das strings e mais próxima da sintaxe usada em outras linguagens de programação. Assim como ISλ, a linguagem possui um único tipo numérico, que unifica inteiros, racionais e complexos, tornando o comportamento dos números mais próximo das entidades matemáticas que representam. Por exemplo, não há uma divisão inteira distinta da divisão usual.

Expressões aritméticas. Faz introduz operadores infixados na linguagem, tornando a sintaxe mais similar à notação algébrica e evitando aninhamento excessivo de parênteses em expressões maiores. Buscou-se introduzir apenas operadores suficientemente familiares a alunos egressos do Ensino Médio. Por exemplo, ao contrário do que ocorre em linguagens como C, não há um operador infixado para o cômputo do resto de uma divisão; para isso, a função `resto` é utilizada. Com isso, buscou-se facilitar a compreensão de código escrito na linguagem e evitar a introdução de regras de precedência além das já usuais na álgebra.

Expressões relacionais e lógicas. Uma vez que as verificações de tipos providas pelo uso de operadores de comparação distintos para cada tipo, como ocorre em Racket, tornam-se desnecessárias em uma linguagem com um sistema de tipos estático, Faz usa os mesmos operadores para comparar dados dos diversos tipos, o que facilita a escrita de testes de comparação.

Comandos e blocos. A maior parte das linguagens imperativas faz uma distinção entre expressões, i.e., construções que produzem um valor, e comandos, ou *statements*, construções que possuem apenas um efeito colateral e não produzem um valor. Por exemplo, na linguagem C, operações aritméticas e construções com o operador ternário `?` são expressões, enquanto comandos como `if`, `for`, `while` e `return` são utilizados apenas por seus efeitos colaterais (controle de fluxo) e não produzem valores que possam ser usados como parte de expressões maiores.

Linguagens funcionais, por outro lado, geralmente eliminam essa distinção, uma vez que nessas linguagens tende-se a escrever programas por meio de composição de valores e limita-se o uso de efeitos colaterais. Por exemplo, na maior parte das linguagens funcionais, a construção condicional `if` ou `cond` é uma expressão que produz um valor, podendo ser utilizada como parte de uma expressão maior (similar ao operador ternário `?` em C). Da mesma forma, variáveis locais frequentemente são introduzidas por meio de uma construção sintática do tipo `let` *`var = valor`* `in` *`expressão`*, em que as variáveis são visíveis apenas na expressão, e a forma `let` como um todo produz como valor o resultado da expressão, podendo ser utilizada em uma expressão maior (e.g., `1 + (let x=3 in x*x)` é uma expressão válida em Haskell e produz o valor `10`). Uma vez que esse tipo de construção naturalmente permite aninhamento e as variáveis são visíveis apenas dentro da subexpressão, o conceito de bloco, frequentemente encontrado em linguagens imperativas como uma maneira de delimitar o escopo dos identificadores introduzidos, torna-se desnecessário em uma linguagem funcional desse tipo.

Essa divergência entre linguagens imperativas e funcionais é de natureza puramente sintática; embora o paradigma funcional favoreça uma eliminação da distinção entre comandos e expressões, isso não é estritamente necessário para a manutenção do caráter funcional de uma linguagem. Durante o projeto da linguagem desenvolvida neste trabalho, foi constatado que a introdução de comandos na linguagem possui uma série de consequências benéficas para a clareza do código:

- A introdução de um comando explícito para retorno de valores (`devolve`) torna mais evidente que o valor produzido por uma expressão é o valor de retorno da função como um todo, especialmente em casos em que o corpo da função é uma construção complexa, como um condicional.

- A organização de comandos em blocos permite o uso da mesma construção para introduzir tanto definições globais como locais (i.e., não é necessária uma forma especial como `let` para introduzir variáveis locais).

- O uso de comandos permite que as próprias palavras-chave que introduzem cada comando sirvam como delimitador de comandos e expressões, eliminando a necessidade de pontuação explícita para delimitar o corpo de funções e de outras construções, sem perda de clareza, como será visto mais adiante.

Com base nisso, optou-se por uma sintaxe baseada em comandos e blocos, em vez de uma sintaxe puramente baseada em expressões. Porém, esses conceitos foram adaptados de forma a melhor adequá-los ao caráter funcional da linguagem. Diferentemente do que ocorre em uma linguagem imperativa convencional, e similarmente às construções de linguagens funcionais, todo bloco em Faz produz um valor. Esse valor é normalmente usado no cômputo do valor de retorno de funções, mas pode ser usado como parte de uma expressão, envolvendo o bloco entre parênteses, como visto na Seção 4.1.8.

Outra adaptação realizada advém do fato de que, diferentemente de linguagens imperativas, Faz não possui comandos de atribuição ou outros comandos que sejam invocados apenas por seu efeito colateral, e não por seu valor. Consequentemente, cada bloco possui, além das declarações de variáveis e funções locais, exatamente um comando, que computa o valor de retorno do bloco. Em Faz, essa limitação é incorporada na sintaxe da linguagem. Dessa forma, o comando final indica implicitamente o fim do bloco, uma vez que não pode haver mais comandos após o comando final. Isso elimina a necessidade de delimitadores explícitos de bloco, tais como palavras especiais de início e fim, chaves ou indentação significativa[1].

Definição de variáveis. O uso da palavra `seja` para introduzir variáveis é usual em textos matemáticos e é prontamente compreensível.

Definição e chamada de funções. A linguagem introduz declarações dos tipos dos parâmetros e de retorno das funções, substituindo as anotações de tipo na forma de comentários frequentemente empregadas em ISλ. O uso da palavra `função` para introduzir as definições de função evita ambiguidades com outros tipos de definição. A sintaxe das chamadas de função é mais próxima à da notação algébrica.

Diferentemente das definições de variáveis, em que o tipo da variável é inferido se não for declarado, definições de funções exigem a declaração explícita dos tipos dos parâmetros e de retorno. Há dois motivos principais por trás dessa decisão. Em primeiro lugar, no contexto de uma disciplina introdutória, é interessante enfatizar a necessidade de documentar as interfaces das funções. Em segundo lugar, a presença de uniões de tipos faz com que expressões que seriam detectadas como erros em um sistema de tipos convencionais possuam um tipo válido em Faz. Por exemplo, uma construção como

```
se x<0 devolve 1
senão devolve "a"
```

seria detectada como um erro em uma linguagem estaticamente tipada convencional, mas seria aceita em Faz, onde a construção produz um valor do tipo `Números U Strings`. Com declarações de tipos explícitas nas funções, tais erros podem ser detectados no momento em que o tipo do corpo (e.g., `Números U Strings`) não for compatível com o tipo de retorno da função (e.g., `Números`).

Condicionais. A organização do código em blocos, com uma palavra explícita para o retorno de um valor (`devolve`), provê uma separação mais explícita entre o teste e a resposta de cada cláusula do condicional. O uso da palavra `se` para introduzir cada cláusula também contribui para a legibilidade. A obrigatoriedade da cláusula `senão` dá a essa cláusula a função de delimitador do condicional. Isso, combinado com a delimitação implícita dos blocos de cada cláusula e a exigência de apenas um comando de retorno por bloco, permite que múltiplas cláusulas sejam postas em sequência sem a necessidade de criar aninhamentos de `se/senão` e sem provocar ambiguidade com uma possível sequência de condicionais distintos.

Expressões de bloco. Como visto, a estruturação em comandos e blocos traz uma série de benefícios à linguagem. Por outro lado, a distinção entre expressões e comandos de certa forma reduz a ortogonalidade das construções da linguagem. Expressões de bloco foram introduzidas na linguagem de modo a compensar essa perda, permitindo

[1] Indentação significativa é um problema quando o código é copiado de *slides* de aula ou postado em outros meios que perdem a indentação do texto. Por outro lado, em uma linguagem em que a estrutura do código pode ser reconstruída sem auxílio da indentação, é possível ao ambiente de programação realizar indentação automática do código.

o uso de comandos como expressões, e podem ser vistas como uma construção dual ao comando `devolve`, que permite usar uma expressão como o comando final de um bloco. Embora essa dualidade possa ser vista como uma complicação desnecessária, considerou-se-a aceitável na elaboração da linguagem tendo em vista os benefícios trazidos pela introdução de comandos e blocos.

Retorno e sinalização de erros. O uso da palavra `devolve` torna mais explícito que a expressão que a segue é o valor de retorno do bloco, especialmente em um comando mais complexo, como um condicional, diferentemente do que ocorre em ISλ, onde a ausência de uma palavra explícita para indicação de retorno provoca dúvidas entre alunos.

Avaliação de expressões de teste. A ausência de delimitadores explícitos de bloco dá aos comandos da linguagem um papel secundário de delimitadores. Um artefato disso é a necessidade de uma palavra especial para a introdução de expressões a serem avaliadas na execução do programa: diferentemente do que ocorre em ISλ, em que expressões podem ser escritas diretamente no corpo do programa, Faz requer um comando, para evitar que a expressão seja tomada como continuação do bloco anterior. Se por um lado isso exige a adição de um comando extra à linguagem, por outro lado pode-se considerar que a palavra-chave contribui para a clareza do código.

Tipos definidos pelo usuário. A linguagem tenta explorar a analogia entre tipos e conjuntos, uma vez que o conceito de conjunto é familiar a estudantes egressos do Ensino Médio. Assim, a descrição dos tipos segue uma sintaxe similar à descrição de conjuntos por extensão e compreensão. Da mesma forma, a linguagem usa o operador $\in$ para indicar pertinência a um tipo e o operador `U` para criar uniões de tipos.[2]

A introdução de enumerações compensa a eliminação de símbolos da linguagem, permitindo a introdução de constantes de maneira mais estruturada, evitando o problema existente em ISλ de que todo símbolo existe sem necessidade de declaração prévia, o que ocasionalmente faz com que erros de digitação passem desapercebidos no código. A introdução de tipos de união permite a definição formal de tipos mistos de maneira análoga ao que é feito informalmente em ISλ, conferindo a Faz uma flexibilidade análoga à de ISλ de maneira mais estruturada. O uso da palavra `de` para a extração de campos de um dado estruturado aproxima-se da linguagem natural (e.g., `nome de p` para extrair o campo `nome` de uma instância `p`).

O operador $\in$ possui uma função dupla, sendo usado tanto na declaração do tipo de uma variável ou campo de estrutura quanto em testes de pertinência. Embora o contexto torne em princípio evidente a função do operador (e.g., $\in$ é um teste de pertinência em `se x ∈ Números` e uma declaração em `seja x ∈ Números = 42`), essa duplicidade pode vir a ser um problema. Da mesma forma, o operador `de` atua tanto na seleção de campos de estruturas (e.g., `nome de p`) quanto na introdução de tipos paramétricos (e.g., `Listas de Números`). Novamente, cada uma das funções é em tese evidente pelo contexto em que o operador ocorre.

Listas. A linguagem trata listas da mesma maneira que estruturas de dados definidas pelo usuário, tornando a relação entre listas e estruturas mais evidente e unificando os operadores usados para a manipulação de ambos.

Definições locais. A organização do código em blocos elimina a necessidade de um comando especial para a introdução de definições locais: o escopo de uma definição é

[2] O ambiente DrRacket possui suporte ao Unicode, permitindo o uso de caracteres especiais, como $\in$. Pretende-se no futuro introduzir um atalho de teclado no ambiente para a inserção desse símbolo. Alternativamente, o caractere `:` pode ser usado no lugar de $\in$. O operador `U` é não é um caractere especial, mas sim a letra `U` maiúscula.

dado pelo bloco em que ela ocorre. Assim, a declaração de variáveis locais é simplificada, facilitando a apresentação do conceito em uma disciplina introdutória.

4.2 Sistema de tipos

Como visto no capítulo anterior, a ausência de um sistema de tipos estático nas linguagens HtDP provoca uma série de inconvenientes, tais como a impossibilidade de expressar formalmente os tipos dos argumentos e de retorno de funções e os tipos dos campos de estruturas e a impossibilidade de indicar ao usuário o ponto apropriado do código em que ocorrem erros de tipo. Por outro lado, a tipagem dinâmica oferece uma flexibilidade às linguagens HtDP que não é normalmente encontrada em linguagens estaticamente tipadas, permitindo a definição de funções que trabalham com dados de tipos diversos sem a necessidade de introduzir uniões etiquetadas (*tagged unions*). Essa flexibilidade é explorada em diversos exercícios do livro-texto HtDP e é um ponto positivo dessas linguagens.

Visando a permitir a declaração formal dos tipos de dados manipulados pelo programa sem perder a flexibilidade oferecida por um sistema de tipos dinâmico, a linguagem Faz emprega um sistema de tipos semi-estático. A linguagem exige que os tipos dos argumentos e de retorno de funções e os tipos dos campos de estruturas sejam declarados pelo usuário e o uso correto desses tipos é verificado em tempo de compilação. Por outro lado, a linguagem introduz *uniões de tipos* (PIERCE, 2002), permitindo a elaboração de programas que trabalham com dados de tipos diversos de maneira intuitiva.

Esta seção descreve o sistema de tipos de Faz. A introdução de um sistema de tipos semi-estático é a principal divergência semântica, e não puramente sintática, entre Faz e as linguagens HtDP. Assim, sua elaboração e implementação constituem uma das principais contribuições do presente trabalho.

4.2.1 Tipos da linguagem

Tipos básicos. Os tipos de dados básicos da linguagem são `Números`, `Booleanos`, `Caracteres` e `Strings`. Esses tipos contêm os valores expressáveis como constantes no código por meio da sintaxe descrita na Seção 4.1.1.

Universo e vazio. O tipo `Tudo` representa o universo de valores da linguagem e é supertipo de todos os tipos. O tipo `Nada` é um tipo vazio e é subtipo de todos os tipos. O tipo `Nada` não pode ocorrer por si só, mas pode ocorrer como parâmetro de um tipo paramétrico quando a instanciação do tipo não impõe qualquer restrição sobre o parâmetro. Por exemplo, o tipo da lista vazia `[]` é `Listas de Nada`.

Tipos funcionais. Há infinitos tipos da forma `Funções(`t_1`, ..., `t_n`) -> `t_R, onde t_i são tipos, representando o tipo das funções que recebem n argumentos dos tipos t_1, ..., t_n e retornam um valor do tipo t_R.

Uniões de tipos. Se s e t são tipos, então `s U t` é um tipo constituído por todos os elementos de s e de t. Por exemplo, `Números U Strings` é um tipo que contém todos os números e todas as strings. O emprego de uniões de tipos torna possível escrever funções que trabalham com dados de tipos diversos de maneira intuitiva.

Na presença de uniões de tipos, surge a questão de quais operações são válidas sobre um dado cujo tipo é uma união. Em linguagens com uniões de tipo que empregam sistemas de tipos estáticos convencionais, só são permitidas as operações aplicáveis a ambos os tipos que compõem a união. Por exemplo, se `AB` é um tipo de estrutura com os campos `a` e `b`, `BC` é um tipo de estrutura com os campos `c` e `d`, e `x` é uma variável do tipo `AB U BC`, então uma expressão que extraia o campo `b` de `x` é bem tipada, já que o campo `b`

```
função f(x ∈ Números U Strings) -> Números U Strings
    se x ∈ Números
        devolve x + x
    se x ∈ Strings
        devolve concatena_strings(x, x)
    senão
        erro "Tipo inválido"
```

Figura 4.8: Função que recebe um número ou uma string e produz um dado do mesmo tipo

garantidamente existe em `x`, mas não uma que extraia o campo `a` ou `c`, pois não é possível garantir que tal campo exista em `x`. Tais sistemas de tipos são seguros, pois permitem apenas as operações cuja execução sem erros possa ser garantida, mas impedem que certas operações potencialmente corretas sejam tipadas.

A linguagem Faz, por outro lado, permite que uma operação seja aplicada a um dado cujo tipo é uma união se ela for aplicável a qualquer um dos tipos que compõem a união, emitindo apenas um aviso ao usuário caso a operação não seja aplicável a ambos os tipos. Consequentemente, o sistema de tipos de Faz não oferece garantias tão fortes quanto as de um sistema de tipos estático convencional, mas confere uma maior flexibilidade à linguagem, permitindo que certos programas que seriam permissíveis em uma linguagem dinamicamente tipada, como Racket, possam ser expressos em Faz.

Por exemplo, considere a função `f` definida na Figura 4.8. Essa função recebe um número ou uma string e produz um dado do mesmo tipo. Em uma linguagem com um sistema de tipos estático convencional, uma expressão do tipo `f(1) + 2` seria rejeitada, pois ao tipo de retorno de `f`, `Números U Strings`, não pode ser aplicada a operação de soma, já que strings não podem ser somadas. Em Faz, por outro lado, tal expressão é permitida, produzindo apenas um aviso ao usuário.

Para evitar a emissão do aviso, o usuário pode envolver o código que opera sobre o tipo misto em um condicional: sempre que o teste de uma cláusula de um condicional é um teste de pertinência da forma *`x ∈ tipo`*, o sistema de tipos considera que no bloco correspondente a essa cláusula a variável *`x`* possui o tipo especificado, uma vez que o bloco só será executado se o teste de pertinência for verdadeiro em tempo de execução. Assim, a soma `x + x` no corpo da função `f` na Figura 4.8 não produz um aviso, já que essa soma ocorre dentro de uma cláusula de um condicional cujo teste garante que `x` é um número. Também são aceitas como testes de especificação de tipo conjunções de expressões booleanas em que uma ou mais das expressões sejam testes de pertinência. Por exemplo, em uma cláusula do tipo:

```
se x ∈ Números e y ∈ Números e x > y
    devolve x-y
```

as variáveis `x` e `y` possuem o tipo `Números`. Isso se aplica tanto à expressão `x>y`, pois a conjunção é avaliada em curto-circuito e, portanto, a comparação só será realizada se os testes de pertinência forem verdadeiros, quanto ao bloco `devolve x-y`, pois este só será avaliado se o teste como um todo for verdadeiro.

Incidentalmente, esse tipo de análise só é permissível sem restrições em Faz devido ao caráter puramente funcional da linguagem, que garante que os valores das variáveis não sofrerão alterações após os testes de pertinência.

Tipos não paramétricos definidos pelo usuário. Cada construção da forma `tipo` *t* `= componentes` introduz um novo tipo não paramétrico *t*. Se *componentes* inclui uma lista de construtores entre chaves, então cada construtor sem parâmetros (i.e., item de enumeração) produzirá uma constante do tipo *t* e cada construtor da forma *nome*`(`c_1 `∈` t_1`, ..., `c_n `∈` t_n`)` produzirá uma função do tipo `Funções(`t_1`, ..., `t_n`) -> `*t*. Um mesmo construtor não pode ocorrer em mais de uma declaração de tipo ou mais de uma vez em uma mesma declaração de tipo.

Variáveis de tipo e tipos paramétricos. Assim como diversas outras linguagens funcionais estaticamente tipadas, Faz suporta tipos paramétricos, isto é, tipos parametrizados por outros tipos. Por exemplo, `Listas` é um tipo paramétrico: para ser utilizado em um programa, é necessário fornecer um outro tipo como parâmetro, através da palavra `de`, produzindo tipos como `Listas de Números`, `Listas de Strings`, etc. Tipos paramétricos permitem a definição de estruturas de dados genéricas, capazes de conter valores de tipos variados sem a necessidade de redefinir a estrutura para cada tipo.

Além disso, Faz suporta polimorfismo paramétrico em funções. Isto é, é possível definir funções cujo tipo contém variáveis universalmente quantificadas. Por exemplo, uma função como `elo` possui o tipo `Funções (?X, Listas de ?X) -> Listas de ?X`, indicando que a função recebe um valor de um tipo qualquer e uma lista de valores do mesmo tipo e produz uma lista do mesmo tipo. Quando a função é chamada, as variáveis são instanciadas com os tipos concretos dos argumentos utilizados, através de um algoritmo de *unificação*. Por exemplo, em uma chamada como `elo(1, [2,3])`, `?X` é unificado com `Números` e `Listas de ?X` com `Listas de Números`, de onde se conclui que `?X` deve ser instanciado para `Números`. O tipo do resultado da chamada, portanto, é `Números`.

A presença de uniões de tipos introduz uma complexidade maior no algoritmo de unificação ausente em outras linguagens funcionais. Por exemplo, enquanto em uma linguagem como Haskell uma chamada como `elo(1, ["a"])` produziria um erro de tipo, em Faz a unificação de `?X` com `Números` e `Listas de ?X` com `Listas de Strings` é válida e produz a instanciação `?X = Números U Strings`. O tipo do resultado da chamada, portanto, é `Listas de (Números U Strings)`, refletindo o fato de que a lista resultante contém tanto números quanto strings.

Variáveis de tipo se comportam de maneira diferente no interior da função polimórfica que as introduzem e em uma chamada da função. No interior da função, uma expressão cujo tipo é uma das variáveis introduzidas pela função pode assumir qualquer tipo concreto, dependendo dos argumentos com os quais a função for chamada. Assim, a análise de tipo trata essas variáveis como tipos concretos sobre os quais não se tem nenhuma informação. Na chamada da função, por outro lado, busca-se substituir as variáveis de tipos pelos tipos concretos dos argumentos utilizados. Em outras palavras, as variáveis introduzidas pela função são não instanciáveis no corpo da função, mas são instanciáveis quando na chamada da função.

A Figura 4.9 demonstra a definição de uma função polimórfica, `compose`, que recebe duas funções de um argumento, `f` e `g`, e produz sua função composta, `g∘f`. A assinatura da função introduz três variáveis de tipo, `?X`, `?Y` e `?Z`. A assinatura indica que o tipo do retorno de `f` deve ser o mesmo do parâmetro de `g` (`?Y`) e que os tipos do parâmetro e de retorno da função resultante são os mesmos do parâmetro de `f` (`?X`) e do retorno de `g` (`?Z`), respectivamente. A função `compose` define localmente uma função, `composta`, que recebe um argumento `x` do tipo `?X` e produz um resultado do tipo `?Z`, através da aplicação `f(g(x))`. Na definição de `composta`, `?X` e `?Z` referem-se às mesmas variáveis

```
função compose(f ∈ Funções(?X)->?Y, g ∈ Funções(?Y)->?Z)
       -> Funções(?X)->?Z
    função composta(x ∈ ?X) -> ?Z
        devolve g(f(x))
    devolve composta
```

Figura 4.9: Exemplo função polimórfica em Faz

definidas na assinatura de `compose`: o escopo de uma variável de tipo introduzida na assinatura de uma função estende-se por todo o corpo da função, incluindo definições internas. Do ponto de vista da função `composta`, tais variáveis são tipos concretos como quaisquer outros, uma vez que uma variável de tipo é não instanciável no corpo da função que a introduz. A função `compose` retorna a função `composta` como resultado.

Uma vez definida essa função, considere a existência das funções `bool_para_num`, do tipo `Funções (Booleanos) -> Números`, e `num_para_string`, do tipo `Funções (Números) -> Strings`. Em uma chamada como `compose (bool_para_num, num_para_string)`, os tipos concretos dos argumentos serão unificados com os tipos abstratos dos parâmetros de `compose`. O processo de unificação determinará que, para que a chamada da função seja válida, `?X` deverá ser substituído por `Booleanos`, `?Y` por `Números` e `?Z` por `Strings`. Consequentemente, a chamada produzirá como resultado uma função do tipo `Funções (Booleanos) -> Strings`, como esperado para a operação de composição.

Tipos paramétricos definidos pelo usuário. Cada construção da forma `tipo` *t* `de (`p_1`, ..., `p_n`) = componentes`, onde p_1, ..., p_n são variáveis de tipo, introduz um novo tipo paramétrico. Se *componentes* inclui uma lista de construtores entre chaves, então cada construtor sem parâmetros produzirá uma constante do tipo *t* `de (Nada, ..., Nada)` e cada construtor da forma *nome*`(`c_1 `∈` t_1`, ...,` c_n `∈` t_n`)` produzirá uma função do tipo `Funções(`t_1`, ...,` t_n`) ->` *t* `de (`x_1`, ...,` x_n`)`, onde x_i é igual a p_i se p_i ocorre nos tipos dos campos do construtor e `Nada` caso contrário. Por exemplo, uma definição como:

```
tipo T de (?X, ?Y) = { a(p1 ∈ ?X),
                       b(p2 ∈ ?Y),
                       c(p3 ∈ ?X, p4 ∈ ?Y),
                       d }
```

produz os seguintes construtores e respectivos tipos:

```
a ∈ Funções(?X) -> T de (?X, Nada)
b ∈ Funções(?Y) -> T de (Nada, ?Y)
c ∈ Funções(?X, ?Y) -> T de (?X, ?Y)
d ∈ T de (Nada, Nada)
```

Esse emprego do tipo `Nada` para indicar variáveis de tipo não utilizadas no construtor tem base no fato de que `Nada` é o subtipo de todos os tipos. Por exemplo, o fato de que a lista vazia `[]` possui o tipo `Listas de Nada` permite que uma lista vazia seja utilizada em contextos que esperam `Listas de Números`, uma vez que `Nada` é um subtipo de `Números`. Os parâmetros de um tipo só podem ser anulados dessa forma se for possível construir valores do tipo resultante. Por exemplo, `Listas de Nada` é um tipo válido, pois contém o valor `[]`. Por outro lado, se um tipo `Inanulável` for definido como:

```
tipo Inanulável de ?X = { foo(x ∈ ?X) }
```

então uma expressão do tipo `Inanulável de Nada` não é aceita pelo verificador de tipos, pois não é possível construir valores desse tipo, uma vez que isso exigiria chamar o construtor `foo` com um argumento do tipo `Nada` e não existem valores de tal tipo. Em outras palavras, para que uma expressão seja bem tipada, ela deve possuir um tipo não vazio.

Algumas outras linguagens, como Haskell, atribuem um tipo polimórfico à lista vazia (i.e., análogo a `Listas de ?X` ao invés de `Listas de Nada`). Em Faz, optou-se pelo uso do tipo `Nada` nessas situações por dois motivos. Em primeiro lugar, Faz, diferentemente de Haskell, possui uma relação de subtipagem em que o tipo `Nada` é naturalmente aplicável a essas situações. Esse emprego do tipo `Nada` é análogo ao uso do tipo `Nothing` na linguagem Scala (LAUSANNE, 2013). Em segundo lugar, isso garante que apenas tipos funcionais contenham variáveis de tipos abstratas, o que simplifica o algoritmo de unificação.

4.2.2 Compatibilidade de tipos e unificação

Para que uma expressão possa ser usada como argumento de uma função, operador ou no lado direito de uma definição de variável ou para que um bloco possa ser usado como corpo de uma função, é necessário que o tipo da expressão ou bloco seja *compatível* com o tipo esperado pela função, operador ou definição em que ocorre. Dois fatores devem ser levados em conta ao se definir compatibilidade entre tipos. Em primeiro lugar, a presença de uniões de tipos induz uma relação de subtipagem baseada na continência ($\subseteq$) de um tipo por outro. Por exemplo, `Números` é um subtipo de `Números U Strings`, pois todo valor do primeiro tipo é também um valor do segundo. Assim, uma expressão do tipo `Números` é *compatível* com um contexto que espera o tipo `Números U Strings`. Além disso, Faz introduz a noção de *compatibilidade parcial* entre tipos, permitindo que uma expressão cujo tipo não é um subtipo do tipo do contexto em que ocorre possa ser usada em tal contexto, desde que os tipos possuam uma intersecção não vazia, emitindo um aviso ao usuário, como descrito na seção anterior. Por exemplo, uma expressão do tipo `Números U Strings` pode ser utilizada como argumento de uma função que espera `Números`, produzindo um aviso, pois *alguns* valores de `Números U Strings` são compatíveis com `Números`.

Em segundo lugar, a compatibilidade de tipos polimórficos está sujeita à escolha dos tipos para os quais as variáveis de tipo serão instanciadas. Por exemplo, se `f` é uma função do tipo `Funções (?X, ?X) -> ?X`, a chamada `f(5, "foo")` será bem tipada se a escolha do valor concreto de `?X` for tal que o tipo de cada argumento seja um subtipo do parâmetro correspondente, ou seja, se `Números` $\subseteq$ `?X` e `Strings` $\subseteq$ `?X`. A análise de tipos busca encontrar uma atribuição de tipos às variáveis que satisfaça todas essas *restrições* impostas pelos tipos dos argumentos utilizados. No exemplo citado, uma atribuição que satisfaz todas as restrições é `?X = Números U Strings`. Se a análise produzir um conjunto de restrições que não podem ser todas satisfeitas simultaneamente, ocorre um erro de tipo.

Em resumo, a verificação de compatibilidade verifica se o tipo de uma construção é compatível, parcialmente compatível ou incompatível com o tipo do contexto em que ocorre. Na presença de variáveis de tipo, determina-se ainda quais são as restrições sobre as possíveis instanciações concretas das variáveis. Por fim, substituem-se as variáveis por tipos concretos que satisfaçam as restrições, se houverem. Seguem as regras utilizadas na verificação.

Universo e vazio. Todos os tipos são compatíveis com `Tudo`. O tipo `Nada` é compatível com todos os tipos.

Variáveis instanciáveis. Uma variável instanciável `?X` é compatível com qualquer tipo `t`, produzindo a restrição `?X ⊆ t`. Qualquer tipo `t` é compatível com `?X`, produzindo a restrição `t ⊆ ?X`. Dois tipos contendo variáveis quantificadas são considerados incompatíveis entre si, i.e., apenas um dos lados de uma relação de compatibilidade pode ser polimórfico.

Tipos primitivos. Tipos primitivos (`Números`, `Strings`, `Caracteres`, `Booleanos`) só são compatíveis entre si se forem iguais.

Uniões de tipos. Um tipo `S` é compatível com `X U Y` se `S` for compatível com `X` ou com `Y`. Por exemplo, `Números` é compatível com `Números U Strings`. No caso de os graus de compatibilidade de `S` com `X` e com `Y` serem diferentes, opta-se pela opção mais compatível. Se a verificação produzir restrições sobre as variáveis de tipo, opta-se pelo conjunto de restrições da opção mais compatível.

`X U Y` é compatível com `S` se tanto `X` quando `Y` forem compatíveis com `S`. Nesse caso, como ambas as alternativas devem ser compatíveis, as restrições produzidas por cada alternativa devem ser combinadas. O grau de compatibilidade é o menos compatível das duas alternativas.

A união das restrições se dá da seguinte maneira. Para toda variável `?X`, substituem-se todas as restrições da forma[3] `Si ⊆ ?X ⊆ Ti` por uma única restrição `S ⊆ ?X ⊆ T`, onde `S` é a união de todos os `Si` e `T` é a intersecção de todos os `Ti`. Isto é, na restrição resultante, busca-se reduzir a faixa de valores que a variável pode assumir, escolhendo-se limites inferiores e superiores mais estreitos do que os das restrições originais, de modo que todas as restrições sejam satisfeitas.

Variáveis não instanciáveis. Uma variável não instanciável (i.e., uma variável definida na assinatura de uma função, quando usada no interior da função) é compatível consigo mesma. Para os fins das demais regras aqui citadas, uma variável não instanciável se comporta como um tipo simples qualquer. Por exemplo, se `?N` é uma variável não instanciável e `?X` é uma variável instanciável, então `?N` é compatível com `?X`, produzindo a restrição `?N ⊆ ?X` sobre a variável `?X`.

Tipos funcionais. Dois tipos funcionais são compatíveis se possuem o mesmo número de parâmetros, o tipo de retorno do primeiro for compatível com o do segundo e os tipos dos parâmetros do segundo forem compatíveis com os do primeiro. Isto é, a relação de subtipagem entre funções é covariante com relação ao resultado e contravariante com relação aos parâmetros. As restrições produzidas pelas sub-relações são combinadas e o grau de compatibilidade é o menos compatível produzido por qualquer uma das sub-relações.

Tipos definidos pelo usuário. Um tipo definido pelo usuário é substituído pelo lado direito de sua definição. Por exemplo, dada a definição:

```
tipo T1 = Números U Strings
tipo T2 = Strings U Caracteres
```

verificar a compatibilidade entre `T1` e `T2` é equivalente a verificar a compatibilidade entre `Números U Strings` e `Strings U Caracteres`. Além de outros tipos e uniões, o lado direito de uma definição de tipo pode conter listas de construtores. Duas listas de

[3] Uma restrição da forma `T ⊆ ?X` pode ser vista como uma abreviação de `T ⊆ ?X ⊆ Tudo`. Analogamente, `?X ⊆ T` pode ser vista como uma abreviação de `Nada ⊆ ?X ⊆ T`. Assim, todas as restrições têm um formato uniforme.

construtores produzidas por tipos `S de (S1, ... Sn)` e `T de (T1, ... Tn)` são compatíveis se `S = T` e cada `Si` for compatível com o `Ti` correspondente. Por exemplo, dada a definição:

```
tipo Pares de (?X, ?Y) = { par(p1 ∈ ?X, p2 ∈ ?Y) }
```

o tipo `Pares de (Números, Booleanos)` é compatível com `Pares de (Números U Strings, Caracteres U Booleanos)`, pois o nome do tipo é o mesmo (`Pares`) e os parâmetros de tipo correspondentes são compatíveis. Assim como na chamada de função, quaisquer restrições produzidas são combinadas e o grau de compatibilidade é o menos compatível de qualquer uma das alternativas.

Concretização das restrições. Como visto, a verificação de compatibilidade entre uma função polimórfica e seus argumentos em uma chamada produz um conjunto de restrições. Essas restrições são utilizadas para atribuir valores concretos às variáveis quantificadas do tipo da função. A atribuição ocorre da seguinte maneira. Para cada restrição da forma `S ⊆ ?X ⊆ T`:

- Se `S = T`, então `?X = S`. Isto é, se os limites superior e inferior da variável forem iguais, só há uma atribuição possível, que é utilizada.
- Se `T = Tudo`, então `?X = S`. Isto é, se a restrição não impõe qualquer limite superior, o limite inferior é utilizado.
- Sse `S = Nada`, então `?X = T`. Isto é, se a restrição não impõe qualquer limite inferior, o limite superior é utilizado.
- Caso contrário, a atribuição é considerada falha e um erro de compilação é produzido.

A última cláusula acima implica que restrições que aceitam mais de uma solução (e.g., `Números ⊆ ?X ⊆ Números U Strings`, em que tanto `?X = Números` quanto `?X = Números U Strings` são soluções válidas) ou não aceitam qualquer solução (e.g., `Números U Strings ⊆ ?X ⊆ Números`) são rejeitadas pela análise semântica. Outra implicação das cláusulas é que, na ausência de restrições sobre uma variável, ela adquire o tipo `Nada`.

Limitações. A integração de uniões de tipos e polimorfismo paramétrico introduz dificuldades ausentes nos sistemas de tipos de linguagens funcionais estaticamente tipadas convencionais, como Haskell. Para os fins do presente trabalho, foram adotadas algumas restrições no uso de tipos paramétricos de maneira a evitar essas dificuldades. Como visto, dois tipos polimórficos são considerados incompatíveis entre si. Não é possível passar uma função polimórfica como argumento para outra função polimórfica, por exemplo. Além disso, faz-se a exigência de que as restrições impliquem uma atribuição única para cada variável. Finalmente, faz-se a exigência de que uniões contendo variáveis de tipos devem ser disjuntas para quaisquer valores atribuídos às variáveis de tipo, evitando assim restrições com múltiplas soluções. Por exemplo, um tipo como `(Listas de Números) U (Listas de ?X)` não é válido, pois a união não é disjunta quando `?X = Números`; em um tipo como `(Listas de Números) U (Listas de ?X)`, tanto `?X = Nada` quanto `?X = Números` produziriam o mesmo tipo.

4.3 Semântica e tipos das construções da linguagem

A semântica das construções de Faz é dada por sua tradução para construções equivalentes em Racket. A tradução de cada construção é descrita nesta seção. A notação `T[x] = y` será utilizada para indicar que a tradução de uma construção `x` em Faz é a construção `y` em Racket. Também serão descritas as regras de tipo de cada construção. A expressão "tipo esperado"será utilizada para indicar que o tipo de uma construção deve ser compatível ou parcialmente compatível com o tipo especificado.

Expressões aritméticas. A avaliação dos operadores aritméticos (`+`, `-`, `*`, `/`, `^`) se dá da esquerda para a direita, respeitando as precedências convencionais da notação algébrica. O tipo esperado dos operandos é `Números`, e o tipo do resultado é `Números`. A tradução das expressões aritméticas para Racket é trivial:

```
T[e1+e2]   =  (+ T[e1] T[e2])
T[e1-e2]   =  (- T[e1] T[e2])
T[e1*e2]   =  (* T[e1] T[e2])
T[e1/e2]   =  (/ T[e1] T[e2])
T[e1^e2]   =  (expt T[e1] T[e2])
T[+e1]     =  (+ T[e1])
T[-e1]     =  (- T[e1])
```

Operações relacionais e lógicas. Como visto, os operadores relacionais de Faz são aplicáveis a quaisquer tipos de dados. O operador `==` possui um operador equivalente em Racket, `equal?`, que opera sobre quaisquer tipos. O operador `!=` é traduzido para a negação de uma expressão com `equal?`. Os demais operadores são traduzidos para uma função polimórfica especial do *runtime* de Faz[4] Para que a operação seja permitida, os operandos devem possuir o mesmo tipo ou tipos distintos cuja intersecção seja não vazia; nesse último caso, um aviso é emitido ao usuário. O resultado dessas operações é sempre do tipo `Booleanos`.

```
T[e1==e2]   =   (equal? T[e1] T[e2])
T[e1!=e2]   =   (not (equal? T[e1] T[e2]))

T[e1<e2]    =   (faz-cmp <  char<?  string<?  T[e1] T[e2])
T[e1<=e2]   =   (faz-cmp <= char<=? string<=? T[e1] T[e2])
T[e1>=e2]   =   (faz-cmp >= char>=? string>=? T[e1] T[e2])
T[e1>e2]    =   (faz-cmp >  char>?  string>?  T[e1] T[e2])
```

Onde `faz-cmp` é definida como:

```
(define (faz-cmp number-op char-op string-op e1 e2)
  (cond [(and (number? e1) (number? e2)) (num-op e1 e2)]
        [(and (char? e1) (char? e2))     (char-op e1 e2)]
        [(and (string? e1) (string? e2)) (string-op e1 e2)]
        [else (error "Valores não podem ser comparados")]))
```

Expressões lógicas são avaliadas da esquerda para a direita em curto-circuito. Sua tradução para Racket é trivial. O tipo esperado dos operandos e o tipo do resultado da expressão são `Booleanos`.

[4]Em uma linguagem sem uniões de tipos, seria possível analisar os tipos dos operandos e emitir a função de comparação adequada em Racket na tradução. A presença de uniões faz com que nem sempre os tipos concretos sejam conhecidos em tempo de compilação, impedindo essa abordagem.

```
T[e1 e e2]     =  (and T[e1] T[e2])
T[e1 ou e2]    =  (or T[e1] T[e2])
```

Blocos. Blocos são traduzidos para uma construção `local` de Racket.

```
T[<definição1>                (local (T[<definição1>]
   ...                  =             ...
  <definiçãon>                       T[<definiçãon>])
  <comando final>]            T[<comando final>])
```

A tradução de cada um dos componentes do bloco será vista adiante. A avaliação se dá seguinte maneira. É criado um novo ambiente de definições contendo cada um dos identificadores definidos pelos comandos de definição do bloco, inicialmente sem valor. Em seguida, cada um dos comandos de definição é avaliado em sequência e o valor atribuído pela definição (lado direito da igualdade no comando `seja` ou o corpo da função no comando `função` passa a valer para aquela definição. A expressão que computa o valor de uma variável não pode se referir a variáveis que ainda não estejam definidas no momento da avaliação (i.e., cuja definição ocorre depois no bloco). O corpo de uma função, por outro lado, pode se referir a funções definidas posteriormente, permitindo assim a definição de funções mutuamente recursivas. Finalmente, o comando final do bloco é avaliado no escopo do ambiente de definições criado pelo bloco. O tipo do bloco é o tipo do comando final.

Variáveis. A tradução do comando `seja` é trivial:

```
T[seja x = e1]          =  (define T[x] T[e1])
T[seja x ∈ tipo = e1]   =  (define T[x] T[e1])
```

A declaração de tipo, se houver, é descartada na tradução, uma vez que Racket é dinamicamente tipada. A avaliação consiste em avaliar o valor do lado direito da igualdade e atribuí-lo à variável correspondente no ambiente de identificadores local. Se a definição inclui o tipo da variável, o tipo esperado para *e1* é o tipo especificado; caso contrário, o tipo de *e1* é tomado como o tipo da variável.

A ocorrência de um identificador em uma expressão em Faz é traduzida para um identificador equivalente em Racket, modificado para evitar conflitos com identificadores pré-definidos de Racket e do *runtime* de Faz. O tipo do identificador é obtido do ambiente de definições.

```
T[id]  =  "id"
```

Funções. A tradução de uma definição de função é trivial. Assim como nas definições de variáveis, a informação de tipos é perdida na tradução.

```
T[função f(x1∈t1, ..., xn∈tn) -> tR = (define (T[f] T[x1] ... T[xn])
     <bloco>]                            T[<bloco>])
```

O tipo esperado para o bloco é t_R. Dentro do bloco, os identificadores `x1, ..., xn` estão definidos com os tipos especificados.

A tradução de uma chamada de função também é trivial. Para avaliar a expressão, a função e os argumentos são avaliados da esquerda para a direita. Em seguida, o bloco que compõe o corpo da função é avaliado dentro de um ambiente de definições em que

cada um dos parâmetros é associado ao valor avaliado para os argumentos. Esse ambiente é subordinado ao ambiente em que a definição foi avaliada, i.e., a linguagem usa escopo estático ou léxico. O resultado da avaliação do bloco é o resultado da aplicação da função. Os tipos esperados para os argumentos são os tipos dos parâmetros tais como especificados na definição da função. O tipo da aplicação é o tipo de retorno da função.

```
T[f(e1, ..., en)]        =  (T[f] T[e1] ... T[en])
```

Condicionais. O comando condicional é traduzido para um `cond` de Racket.

```
T[se e1 bloco1           = (cond [T[e1] T[bloco1]]
  ...                              ...
  senão blocon]                   [else T[blocon]])
```

Cada um dos testes é avaliado em sequência. O bloco correspondente ao primeiro teste que retornar `verdadeiro` é avaliado e o valor produzido pelo bloco é o valor do comando condicional. Os tipos esperados para as expressões de teste é `Booleanos`. O tipo do condicional é a união dos tipos de cada um dos blocos.

Expressões de bloco. A tradução e a avaliação são idênticas às de um bloco comum; como visto, a presença de comandos de bloco serve apenas para contornar a distinção entre expressões e comandos em Faz. O tipo da expressão é o tipo do bloco.

Retorno e sinalização de erros. A tradução do comando `devolve` *`e1`* é idêntica à tradução de *`e1`*. Como visto, o comando `devolve` é o dual das expressões de bloco e serve apenas como ponte entre o mundo das expressões e dos comandos.

```
T[devolve e1]        =  T[e1]
```

Comandos de sinalização de erro são traduzidos como:

```
T[erro e1]           =  (error 'nome T[e1])
```

Onde *`nome`* é o nome da função em que o comando ocorre. A avaliação consiste em avaliar *`e1`* e gerar um erro de execução contendo a mensagem produzida por *`e1`* e o nome da função em que o comando ocorre. O tipo esperado para *`e1`* é `Strings`. O tipo do comando é `Nada`, o subtipo de todos os tipos; uma vez que o valor do comando nunca será usado, pois o comando termina a execução do programa, o comando pode ocorrer no contexto de qualquer tipo. O comando `erro` é um caso especial em que um tipo vazio é aceito pela linguagem.

O comando `teste`. Assim como o comando `devolve`, a tradução do comando `teste` *`e1`* é idêntica à de *`e1`*.[5] O tipo de *`e1`* é irrestrito, desde que a expressão seja bem tipada.

Definições de tipos. O comando `tipo` cria novos tipos e construtores tal como descrito na seção anterior. Definições de construtores sem argumentos (i.e., itens de enumeração) são traduzidas para variáveis cujo nome é o nome do construtor, devidamente transformado para evitar conflito com identificadores internos, e cujo valor é o nome do construtor representado como um símbolo. Definições de construtores com argumentos (i.e., estruturas) são traduzidos para funções que recebem um argumento para cada campo do construtor e produzem um vetor cujo primeiro elemento é o nome do construtor como um símbolo e os elementos restantes são, alternadamente, o nome de um campo e seu valor.

[5] Uma implementação alternativa seria traduzir o comando `teste e1` para código que imprimisse a expressão *`e1`* e seu valor, permitindo identificar na saída do programa que valor corresponde a que expressão.

```
T[tipo ... = { ..., x, ... }] = (define T[x] 'x)

T[tipo ... = { ..., s(c1∈t1, ..., cn∈tn), ... }]
   = (define (T[s] T[c1] ... T[cn])
       (vector 's 'c1 T[c1] ... 'cn T[cn]))
```

Seletores campos de estruturas. Expressões da forma *campo* `de` *x* são traduzidas para uma chamada a uma função `select` do runtime de Faz:

```
T[c de x] = (select T[x] 'c)
```

A função `select` recebe o nome de um campo e uma estrutura, representada como um vetor, e retorna o valor do campo correspondente, se houver. Por questões de compatibilidade com Racket, listas de Faz não são representadas como vetores, e sim como listas de Racket. Assim, listas são tratadas separadamente pela função `select`.

Uma vez que é possível dar o mesmo nome a campos de tipos estruturados distintos e é possível criar uniões desses tipos, o tipo de uma expressão como *campo* `de` *x* depende dos tipos que *x* pode assumir. Especificamente, o tipo de *campo* `de` *x* é a união dos tipos de quaisquer campos *campo* definidos pelos construtores do tipo de `x` ou qualquer um de seus subtipos. Por exemplo, dadas as definições:

```
tipo Livros = { livro(código ∈ Números) }
tipo Carros = { carro(código ∈ Strings) }
tipo Ambos = Livros U Carros

seja a ∈ Livros = livro(42)
seja b ∈ Carros = carro("ABC-1234")
seja c ∈ Ambos = (se random(0,1)==0 devolve a senão devolve b)
```

o tipo de `código de a` é `Números`, o tipo de `código de b` é `Strings` e o tipo de `código de c` é `Números U Strings`.

Testes de pertinência. Além dos construtores, a definição de um tipo *t* produz uma função ∈-*t*. Ao ser invocada, essa função produz um predicado (i.e., outra função) que recebe um argumento e testa se ele pertence ao tipo *t*. Se *t* é um tipo paramétrico, então ∈-*t* recebe um argumento para cada parâmetro do tipo. Cada argumento deve ser um teste de pertinência. O predicado produzido pela chamada testa se seu argumento é um símbolo correspondente a um construtor sem argumentos de *t*, um vetor correspondente a um construtor estruturado de *t*, ou um subtipo de *t*, através de chamadas a outros predicados. No caso dos construtores estruturados de tipos paramétricos, o predicado ainda testa se os campos com tipos paramétricos possuem os tipos especificados pelos argumentos da função ∈-*t*. Por exemplo, a definição do tipo `Árvores de ?X` da Figura 4.5 produz a função:

```
(define (∈-Árvores ?X)
  (lambda (x)
    (or (eq? x 'vazia)
        (and (vector? x)
             (eq? (vector-ref x 0) 'nó)
             (?X (select x 'valor))
             ((∈-Árvores ?X) (select x 'esquerda))
             ((∈-Árvores ?X) (select x 'direita))))))
```

Além dos geradores de predicados gerados para tipos definidos pelo usuário, o *runtime* da linguagem contém geradores de predicados para os tipos nativos da linguagem.

Testes de pertinência da forma `x ∈ t` são traduzidas para chamadas de `∈-t`. Devido a limitações da implementação, o tipo `t` não pode ser funcional e não pode conter variáveis de tipo. O tipo esperado de `x` é `Tudo`, i.e., qualquer expressão bem tipada é aceita. O tipo do resultado da expressão é `Booleanos`.

```
T[x ∈ t] = (Tpred[t] T[x])
```

onde:

```
Tpred[t] = (∈-t)
Tpred[t de (t1, ..., tn)] = (∈-t Tpred[t1] ... Tpred[tn])
```

Listas. `vazio` é uma variável pré-definida no *runtime* da linguagem cujo valor é `empty`, a lista vazia em Racket. `elo`, o construtor de listas de Faz, é equivalente à função `cons` de Racket. Listas da forma `[e1, ..., en]` são traduzidas para uma chamada à função `list` de Racket. Os tipos de `vazio` e `elo` são como esperado dada a definição do tipo `Listas` na Seção 4.1.16, i.e., `vazio` tem o tipo `Listas de Nada` e `elo` tem o tipo `Funções (?X, Listas de ?X) -> Listas de ?X`. O tipo da construção `[e1, ..., en]` é o mesmo da construção equivalente empregando os construtores `elo` e `vazio`.

```
T[[e1, ..., en]] = (list T[e1] ... T[en])
```

Funções anônimas. Funções anônimas são traduzidas para formas `lambda` em Racket. A tipagem de funções anônimas é análoga à de funções declaradas.

```
T[(função f(x1∈t1, ..., xn∈tn) -> t   =  (lambda (T[x1] ... T[xn])
    <bloco>]                                 T[<bloco>])
```

4.4 Limitações

Tanto ISλ quanto Faz não possuem mecanismos próprios para a execução de operações com efeitos colaterais. Isso dificulta o tratamento de operações de entrada e saída. ISλ não conta com nenhuma função convencional de entrada e saída, uma vez que a linguagem foi concebida visando à interação por meio da avaliação de expressões no ambiente DrRacket. Faz herda de ISλ essa limitação. ISλ dá acesso a uma biblioteca gráfica, cujas funções possuem o efeito colateral de desenhar formas geométricas na tela. Cada uma dessas funções retorna um booleano indicando sucesso ou falha da operação, o que permite o uso do operador lógico `and` para descrever sequências de operações. Faz permite a mesma abordagem.

Tanto ISλ quanto Faz permitem a sinalização de erros, mas não permitem seu tratamento, encerrando a execução do programa no caso de um erro. No contexto de uma disciplina introdutória, essa limitação não é significativa.

Quanto ao sistema de tipos, há algumas limitações no uso de tipos polimórficos de maneira a evitar problemas com sua interação com uniões de tipos, tal como descrito na Seção 4.2.2, bem como nos tipos que podem ser empregados em testes de pertinência, como descrito na Seção 4.3.

5 IMPLEMENTAÇÃO

A linguagem Faz foi implementada como parte do ambiente DrRacket. Implementações de novas linguagens para o ambiente DrRacket podem ser escritas na linguagem Racket completa, que conta com uma diversidade de bibliotecas, tais como geradores de analisadores léxicos e sintáticos no estilo Lex (LESK; SCHMIDT, 1975) e Yacc (JOHNSON, 1975). Usualmente, novas linguagens são implementadas no ambiente DrRacket por meio da tradução do código-fonte na nova linguagem para código em Racket, que é então compilado e executado utilizando a mesma infraestrutura usada pela linguagem Racket.

Este capítulo descreve, em linhas gerais, o ambiente DrRacket e a implementação da linguagem Faz no mesmo.

5.1 O ambiente DrRacket

DrRacket (anteriormente DrScheme) é um ambiente de desenvolvimento integrado que permite a edição, compilação e execução de programas em uma variedade de linguagens. A linguagem principal do ambiente é o Racket, mas também são suportadas outras linguagens baseadas em Scheme, tais como como as linguagens didáticas HtDP e o padrão R5RS, bem como uma versão de Algol 60 e algumas linguagens experimentais. O ambiente foi escrito de maneira a permitir a incorporação de novas linguagens de maneira modular.

A janela principal do ambiente é dividida em duas áreas. A área superior, denominada *Definitions*, é usada para a edição de código, onde podem ser escritas definições de variáveis, funções e tipos de dados, bem como expressões a serem avaliadas quando o programa for executado. Uma vez escrito o programa, o usuário pode acionar o botão *Run* na barra de ferramentas da janela, fazendo com que o programa seja compilado e, se a compilação for bem sucedida, executado. A área inferior da janela, denominada *Interactions*, exibe os resultados produzidos pelas expressões avaliadas na execução do programa e fornece um *prompt* onde o usuário pode entrar com novas expressões e observar seus resultados. Isso permite ao usuário testar as funções definidas na janela *Definitions* com os argumentos que desejar e observar se os resultados correspondem ao esperado, sem a necessidade de alterar e recompilar o código para cada teste. Não é necessário escrever um programa completo com um ponto de partida tal como a função `main` em C. Isso permite ao usuário testar suas funções à medida em que as escreve, facilitando sua depuração.

As Figuras 5.1 e 5.2 apresentam *screenshots* de interação com o ambiente DrRacket utilizando as linguagens ISλ e Faz, respectivamente.. A área *Definitions* contém a definição de uma função que computa o fatorial de um número natural. A área *Interactions* apresenta a expressão digitada pelo usuário (`fac(5)`), seguida do resultado de sua avalia-

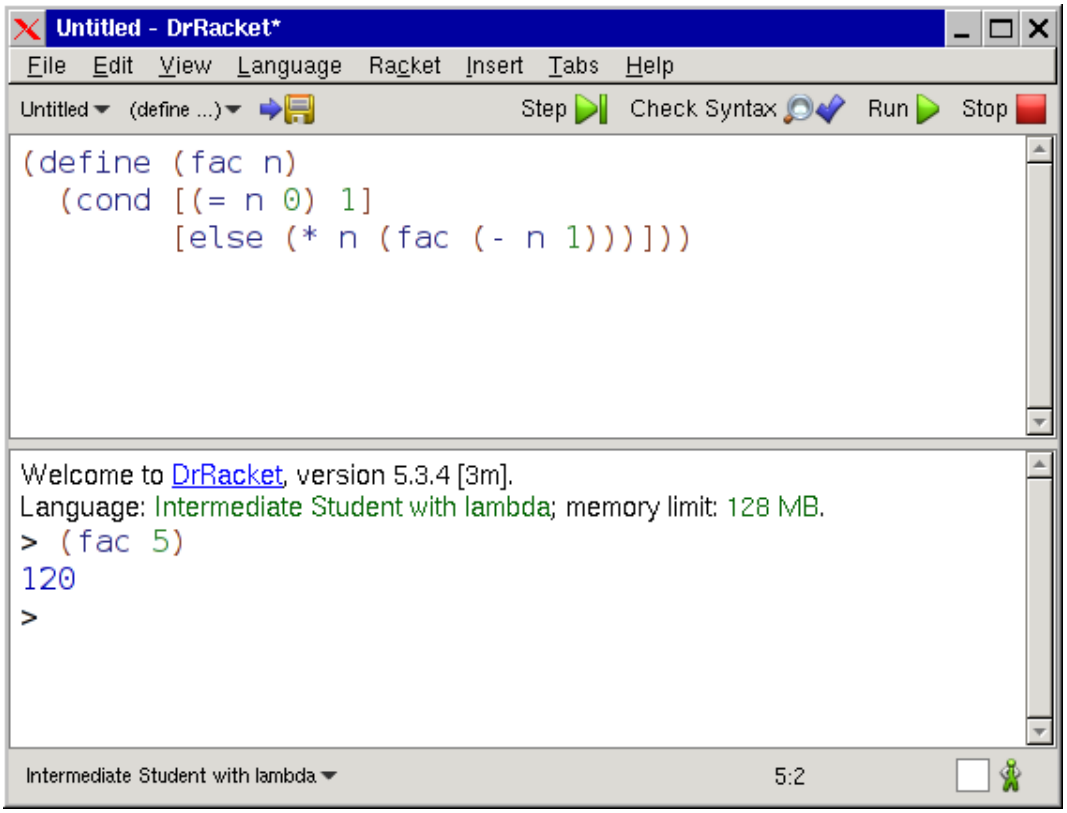

Figura 5.1: Janela principal do ambiente DrRacket executando a linguagem ISλ

ção (`120`). No caso da linguagem Faz, também é exibido o tipo da expressão (`Números`).

5.2 A implementação de Faz

A implementação de Faz é constituída por um conjunto de arquivos na linguagem Racket que implementam os diversos passos da compilação de código, bem como a integração da linguagem com o ambiente DrRacket. Esta seção descreve os principais componentes da implementação.

5.2.1 Análise léxica e sintática

O primeiro passo da compilação consiste da separação do texto que compõe o código-fonte em *tokens*, seguida da geração de uma árvore de sintaxe abstrata. Para isso, são utilizadas as bibliotecas `parser-tools/lex` e `parser-tools/yacc` da linguagem Racket, que provêem funcionalidade equiparável às ferramentas Lex (LESK; SCHMIDT, 1975) e Yacc (JOHNSON, 1975). A árvore sintática é anotada com informação suficiente para determinar o trecho do código a que corresponde cada nó, de maneira a permitir a notificação de erros para o usuário. A análise léxica e sintática está implementada no arquivo `parse.rkt`, que define as estruturas de dados que representam a árvore sintática e as regras da análise léxica e da gramática da linguagem. O arquivo também define a função `parse-faz-from-port`, principal ponto de entrada do módulo, que produz a árvore sintática de um programa lido de uma *stream*, ou *porta* na terminologia de Racket, representando um arquivo ou o conteúdo da janela *Definitions* ou *Interactions*.

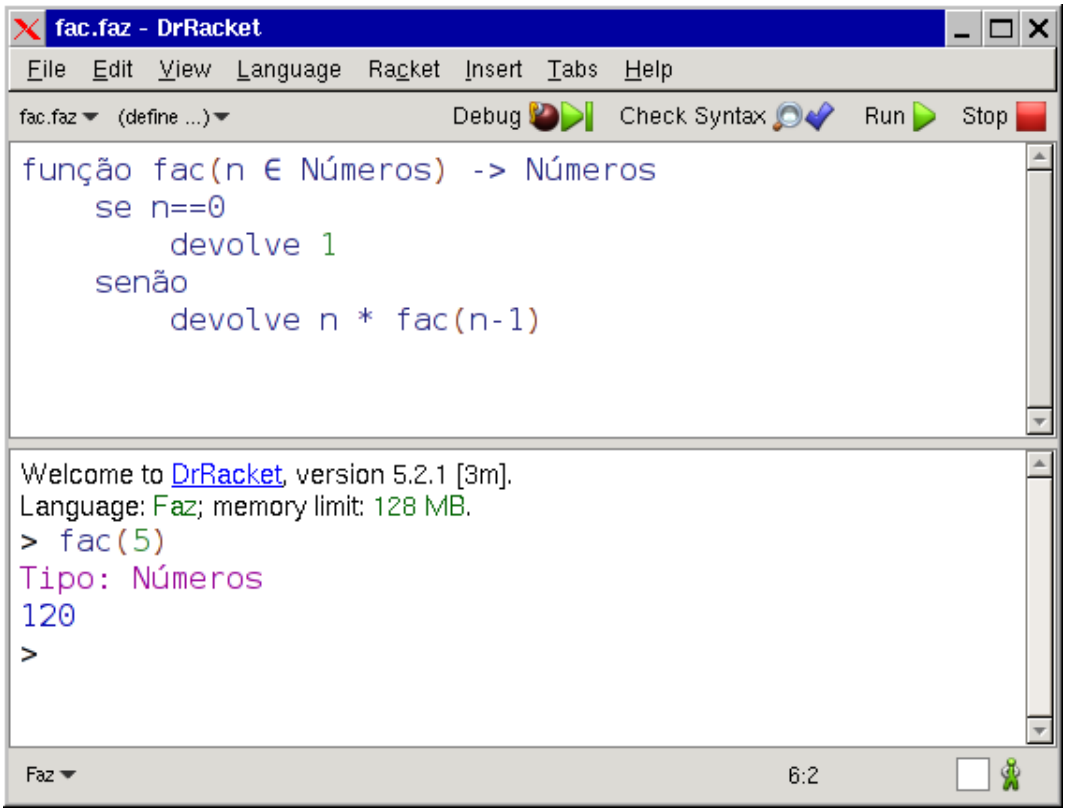

Figura 5.2: Janela principal do ambiente DrRacket executando a linguagem Faz

5.2.2 Análise semântica

Uma vez gerada a árvore sintática, é realizada a análise semântica do código. O nível mais alto do código de um programa é composto por declarações de variáveis, funções e tipos e por comandos de avaliação de expressões. Em um primeiro passo, são processadas as declarações de tipos. É verificada a consistência das declarações, isto é, a ausência de declarações duplicadas de tipos e construtores. Registram-se os novos tipos de dados definidos pelo usuário em uma lista de tipos conhecidos, bem como os nomes e os tipos dos campos de tipos estruturados, e o ambiente de definições de identificadores é acrescido dos construtores de tipos. A seguir, processa-se o corpo do programa, que é tratado como o bloco de nível mais externo. Para cada bloco do programa, são coletadas as declarações de variáveis e funções. Para declarações de variáveis, o valor atribuído à variável sofre uma análise de tipos. Se bem tipado, verifica-se a compatibilidade entre o tipo declarado para a variável e o tipo computado para o seu valor. Se a declaração não inclui explicitamente o tipo da variável, o tipo do valor é tomado como tipo da variável. Para declarações de funções, o bloco que compõe o corpo é analisado recursivamente e, se bem tipado, é verificada a compatibilidade entre o tipo do corpo e o tipo de retorno da função.

Uma vez que a linguagem possui tipos de união, a verificação de compatibilidade de tipos considera as relações de subtipagem entre os tipos. A análise de tipos se dá primariamente por meio de duas funções: `typecheck`, que computa o tipo de um nó da árvore sintática, e `compatible`, que determina se o tipo computado para um nó é compatível com o tipo esperado. Por exemplo, na análise de uma expressão do tipo `x+y`, os tipos de `x` e `y` são computados e, em seguida, verifica-se sua compatibilidade como o tipo esperado para os argumentos do operador `+`, isto é, `Números`. A função `compatible` produz uma tupla `(`*`grau`*`, `*`restrições`*`)`, onde *`grau`* é o grau de compatibilidade entre os dois tipos e *`restrições`* é o conjunto de restrições sobre as variáveis de tipos para

que haja a compatibilidade, tal como descrito na Seção 4.2.2. Outras funções importantes envolvidas na análise semântica são `solve-constraints`, que realiza a combinação de restrições sobre variáveis, e `apply-solution`, que substitui as variáveis de tipo por tipos concretos baseados em uma solução. A análise semântica está implementada no arquivo `typecheck.rkt`.

5.2.3 Tradução

Se a análise semântica for bem sucedida, o próximo passo a ser realizado é a tradução da árvore sintática para código Racket. Assim como nas demais linguagens da família LISP, o código de um programa em Racket é uma estrutura de dados baseada em listas aninhadas e outros elementos sintáticos, tais como símbolos, usados para representar os identificadores da linguagem, bem como números, strings e outras constantes que aparecem no código. Em Racket, esses elementos sintáticos são acrescidos de anotações que permitem indicar a linha, coluna e arquivo onde o elemento foi encontrado, de maneira a possibilitar mensagens de erro mais significativas, e o contexto em que o elemento deve ser avaliado, de maneira a evitar conflitos entre identificadores de mesmo nome que ocorram em partes diversas do programa. Esse objeto contendo elementos sintáticas e contexto é denominado *objeto sintático*. A etapa de tradução consiste em gerar estruturas sintáticas de Racket a partir da árvore sintática de Faz e das informações de tipos coletadas durante a análise semântica.

A etapa de tradução está implementada no arquivo *translate.rkt*, cujo ponto de entrada é a função `ast->scheme`, que recebe um nó da árvore sintática abstrata de Faz e produz um objeto sintático correspondente à tradução do nó para Racket. A tradução é tal como descrito na Seção 4.3.

5.2.4 Ambiente padrão

A linguagem conta com um ambiente padrão de funções, constantes e tipos predefinidos. O tipo `Booleanos` e seus construtores `verdadeiro` e `falso`, bem como o tipo `Listas` e seus construtores `vazio` e `elo`, são definidos diretamente como um trecho de código em Faz que é automaticamente incluído no começo de cada programa antes da compilação. Essas definições são utilizadas apenas para a análise semântica, uma vez que, para manter compatibilidade com Racket, esses tipos são tratados especialmente pela etapa de tradução, traduzindo-os para os valores equivalentes de Racket. Outros elementos do ambiente padrão incluem funções matemáticas tais como `resto`, `sen` e `cos` e funções para manipulação dos tipos de dados primitivos, tais como `concatena_listas` e `concatena_strings`, bem como funções utilizadas internamente pelo *runtime* da linguagem, como a função `select`, utilizada para selecionar campos de estruturas. O ambiente padrão é definido no arquivo `stdenv.rkt`.

5.2.5 Execução interativa

Como visto, o ambiente DrRacket permite a entrada de expressões para avaliação na área *Interactions*. O código digitado nessa área passa pelos mesmos passos de análise léxica, sintática e semântica, tradução para código Racket e subsequente compilação. Diferentemente da análise sintática de um programa completo, a análise de código interativo utiliza um símbolo inicial alternativo, que permite que expressões apareçam diretamente no código sem a necessidade de serem introduzidas por uma palavra-chave como `devolve` ou `teste`, mas não permite a entrada de definições de variáveis, funções

e tipos, que devem ser definidos exclusivamente na área *Definitions*. Os outros passos de compilação são idênticos aos da compilação de um programa completo.

5.2.6 Integração com DrRacket

Para adicionar suporte a uma nova linguagem no ambiente DrRacket, é necessário criar uma *collection*, i.e., uma coleção de módulos Racket. A coleção deve estar organizada de maneira a implementar uma ferramenta do DrRacket, conforme descrito na documentação do ambiente (PLT DESIGN, 2013). Para tal, a coleção deve conter um arquivo `info.rkt`, que descreve o nome e outros metadados da ferramenta e os arquivos que a implementam. Convencionalmente, uma ferramenta é implementada em um arquivo `tool.rkt`, que importa os outros arquivos da implementação, se houver. A ferramenta deve definir uma *unit* `tool@`, que contém funções a serem executadas para carregar a ferramenta. No caso de uma linguagem, a *unit* deve definir uma classe que implementa a `drracket:language:language<%>`. Essa interface contém uma diversidade de métodos que são invocados pelo DrRacket na edição, compilação e execução de código na linguagem, bem como na configuração de parâmetros da linguagem. Desses métodos, os mais importantes são `front-end/complete-program` e `front-end/interaction`, invocados na compilação do código na área *Definitions* e *Interactions*, respectivamente. Esses métodos recebem, como um de seus argumentos, uma *stream* de onde o código a ser compilado pode ser lido e devem retornar um *thunk*, i.e., uma função sem argumentos, que é invocada sucessivamente pelo DrRacket. Cada invocação do *thunk* deve retornar um objeto sintático representando o código em Racket que será compilado e avaliado pelo DrRacket e terá seu resultado apresentado ao usuário.

No caso da linguagem Faz, tanto `front-end/complete-program` quanto `front-end/interaction` invocam a função `compile-faz-from-port`, que recebe uma *stream*, realiza todos os passos de compilação já citados, armazena o resultado do passo de tradução em uma variável e retorna um *thunk* que, cada vez que é invocado, retorna o trecho de código Racket traduzido correspondente a um comando do programa em Faz. Depois que todos trechos foram retornados, o *thunk* retorna `eof`, que sinaliza o fim do programa.

5.2.7 Limitações

A implementação atual da linguagem Faz apresenta algumas limitações. O analisador léxico gerado pela biblioteca `parser-tools/yacc` é relativamente limitado quanto a tratamento de erros, permitindo identificar a linha e a coluna do código em que ocorreu um erro sintático, mas não fornecendo contexto suficiente para informar ao usuário a natureza exata do erro, o que dificulta a depuração de erros de sintaxe. Esse problema pode ser resolvido reescrevendo-se o analisador sintático manualmente, em vez de usar um analisador gerado mecanicamente. Outra possibilidade seria procurar uma biblioteca ou programa alternativo para a geração do analisador. Neste trabalho, por simplicidade, optou-se por usar o gerador que acompanha a linguagem Racket.

A implementação atual exige que os tipos utilizados em testes de pertinência não contenham variáveis de tipos ou tipos funcionais. No caso de tipos funcionais em particular, a tradução de Faz para Racket não preserva informação de tipo suficiente para que um teste de pertinência em tempo de execução possa verificar os tipos dos argumentos e de retorno de um valor funcional; na prática, entretanto, tais testes sobre funções são incomuns, especialmente no contexto de uma disciplina introdutória.

6 EXPERIMENTO DE VALIDAÇÃO

O presente trabalho foi desenvolvido com base na observação da recorrência de certas dificuldades entre os alunos no ensino da disciplina de Fundamentos de Algoritmos utilizando as linguagens didáticas HtDP, conforme exposto nos Capítulos 1 e 3. Como forma de validação, realizou-se uma enquete entre os alunos de Ciência da Computação da UFRGS de maneira a verificar se essas observações refletem as experiências do alunos. A enquete foi realizada por meio de um formulário *online* divulgado através da lista de discussão da graduação do Instituto de Informática da UFRGS. Foram realizadas seis perguntas, divididas em três categorias:

Identificação.

- "Em que semestre você fez a disciplina de Fundamentos de Algoritmos?"
- "Com que outras linguagens de programação você tem experiência (se houver)?"

Opinião.

- "Quais foram as maiores dificuldades que você encontrou com a linguagem Scheme/Racket na disciplina? [Responda em ordem de dificuldade (mais difícil primeiro). É recomendado mas não obrigatório preencher todos os três campos.]"
- "Quais foram as maiores facilidades / pontos positivos que você encontrou com a linguagem Scheme/Racket na disciplina? [Responda em ordem de "positividade" (mais positivo primeiro). É recomendado mas não obrigatório preencher todos os três campos.]"

Comparação de linguagens. Foram apresentados dois programas, cada um escrito nas linguagens Racket (especificamente, ISλ) e Faz: um programa para o cálculo das raízes reais de uma equação do segundo grau a partir de seus coeficientes (Figuras 3.4 e 4.7), escolhido por ser um programa simples empregando uma fórmula conhecida; e um programa maior para encontrar um caminho entre dois nós em um grafo acíclico (Figuras 6.1 e 6.2), escolhido por ser um programa visto na disciplina de Fundamentos de Algoritmos e por empregar diversas construções das linguagens, tais como condicionais, estruturas, listas e definições locais. Para cada programa, fez-se a pergunta: "Em qual linguagem o programa é mais fácil de compreender e seria mais fácil de escrever?", com as opções "Scheme/Racket"[1], "A nova linguagem" e "Tanto faz".

[1] O nome "Scheme/Racket" foi utilizado por serem os nomes convencionalmente usados ao se referir às linguagens didáticas HtDP na disciplina e devido ao fato de a linguagem só ter passado a se chamar Racket em 2010.

Tabela 6.1: Preferência dos alunos pelas linguagens Faz e Racket em diferentes programas

	Faz	Racket	Tanto faz
Raízes reais	55 (92%)	2 (3%)	3 (5%)
Caminho em grafo	49 (82%)	7 (12%)	4 (7%)

Além das perguntas, deixou-se na enquete um espaço para comentários e observações. Os dados brutos coletados estão disponíveis *online*[2]. 60 alunos responderam à enquete. Para ambos os programas, a maioria dos alunos preferiu a sintaxe de Faz (Tabela 6.1).

As principais dificuldades reportadas pelos alunos com a linguagem ISλ podem ser classificadas em quatro grupos:

- *Sintaxe*, por 36 alunos (60%). Além da sintaxe no geral, foram mencionados alguns problemas específicos, tais como dificuldades com o uso excessivo de parênteses, notação prefixada e definições locais.
- *Paradigma funcional*, por 32 alunos (53%). Problemas específicos mencionados incluem o uso de recursão e a ausência de variáveis mutáveis.
- *Tipos de dados*, por 8 alunos (13%). Problemas específicos mencionados incluem a definição dos contratos como comentários e a ausência de um sistema de tipos estático.
- *Ambiente de desenvolvimento*, por 3 alunos (5%). O único problema específico mencionado foi a dificuldade de encontrar os erros no código. Como discutido na seção 3.2.1, a ausência de tipagem estática é em parte responsável por essa dificuldade.

As principais facilidades ou pontos positivos de ISλ citados pelos alunos podem ser classificados em:

- *Facilidade para escrever algoritmos recursivos*, por 20 alunos (33%).
- *Facilidade para trabalhar com estruturas de dados*, por 13 alunos (22%). Um aluno mencionou a facilidade do uso de estruturas em comparação à linguagem C.
- *Sintaxe*, por 7 alunos (12%).
- *Contato com o paradigma funcional*, por 5 alunos (8%).
- *Simplicidade*, por 5 alunos (8%).
- *Expressividade*, isto é, a habilidade de descrever algoritmos com pouco código, por 4 alunos (7%).

Cabe notar que esta pesquisa não tem o intuito de provar uma hipótese de maneira estatisticamente significativa, mas sim apenas de dar uma ideia geral da experiência dos alunos com as linguagens HtDP e a disciplina de Fundamentos de Algoritmos e avaliar a aceitação da nova linguagem desenvolvida neste trabalho pelos mesmos.

[2] `http://inf.ufrgs.br/~vbuaraujo/tcc/`

```
;; Um nó é uma estrutura
;;   (make-nó nome vizinhos)
;; onde nome é um símbolo,
;;      vizinhos é uma lista-de-símbolos.
(define-struct nó (nome vizinhos))

;; Um grafo é uma lista-de-nós.

;; Um caminho é:
;; 1. false (caminho inválido); ou
;; 2. uma lista-de-símbolos.

;; vizinhos: símbolo grafo -> lista-de-símbolos
;; Retorna uma lista dos vizinhos de um nó.
(define (vizinhos nome grafo)
  (cond [(empty? grafo) empty]
        [(symbol=? nome (nó-nome (first grafo))) (nó-vizinhos (first grafo))]
        [else (vizinhos nome (rest grafo))]))


;; encontra-caminho: símbolo símbolo grafo -> caminho
;; Dado um grafo e os nomes de dois nós, encontra um caminho entre
;; os nós no grafo.
(define (encontra-caminho origem destino grafo)
  (cond [(symbol=? origem destino) (list origem)]
        [else
         (local (
             (define tentativa (percorre-vizinhos (vizinhos origem grafo) destino grafo)) )
           (cond [(false? tentativa) false]
                 [else (cons origem tentativa)]))]))

;; percorre-vizinhos: lista-de-símbolos símbolo grafo -> caminho
;; Encontra um caminho entre qualquer uma das origens e o destino no grafo.
(define (percorre-vizinhos origens destino grafo)
  (cond [(empty? origens) false]
        [else (local (
                  (define tentativa (encontra-caminho (first origens) destino grafo)) )
                (cond [(false? tentativa) (percorre-vizinhos (rest origens) destino grafo)]
                      [else tentativa]))]))

;; Exemplo de grafo.
(define exemplo
  (list
    (make-nó 'a (list 'b 'c))
    (make-nó 'b (list 'd 'e))
    (make-nó 'c (list 'e))
    (make-nó 'd empty)
    (make-nó 'e empty)))

;; Teste.
(encontra-caminho 'a 'e exemplo)
```

Figura 6.1: Programa para busca de caminho em grafo acíclico em ISλ

Das quatro principais classes de problemas mencionados com a linguagem ISλ, os problemas com a sintaxe e com tipos de dados foram diretamente abordados no desenvolvimento de Faz. Os problemas com o ambiente ainda são passíveis de melhorias, embora a presença da verificação estática de tipos reduza a dificuldade de encontrar erros no código. A dificuldade de adaptação ao paradigma por alunos com experiência com linguagens imperativas é um problema inerente à proposta de se utilizar uma linguagem funcional no ensino de programação. Porém, com a redução dos problemas com a sintaxe e tipos de dados, é possível que haja uma melhor compreensão do paradigma funcional por parte dos alunos, amenizando a transição entre os paradigmas. A nova linguagem, ao mesmo tempo em que reduz os problemas mencionados com ISλ, mantém seus aspectos positivos, derivados em grande parte do emprego do paradigma funcional.

```
tipo Nós = { nó(nome ∈ Strings, vizinhos ∈ Listas de Strings) }
tipo Grafos = Listas de Nós
tipo Caminhos = Booleanos U Listas de Strings

função vizinhos(nome ∈ Strings, grafo ∈ Grafos) -> Listas de Strings
    # Devolve a lista de vizinhos de um nó.
    se grafo == []
        devolve []
    se nome == nome de primeiro de grafo
        devolve vizinhos de primeiro de grafo
    senão
        devolve vizinhos(nome, resto de grafo)

função encontra_caminho(origem ∈ Strings, destino ∈ Strings, grafo ∈ Grafos) -> Caminhos
    # Dado um grafo e os nomes de dois nós, encontra um caminho entre
    # os nós no grafo.
    se origem == destino
        devolve [origem]
    senão
        seja tentativa = percorre_vizinhos(vizinhos(origem, grafo), destino, grafo)
        se tentativa == falso
            devolve falso
        senão
            devolve Elo(origem, tentativa)

função percorre_vizinhos(origens ∈ Listas de Strings, destino ∈ Strings, grafo ∈ Grafos) -> Caminhos
    # Encontra um caminho entre qualquer uma das origens e o destino no grafo.
    se origens == []
        devolve falso
    senão
        seja tentativa = encontra_caminho(primeiro de origens, destino, grafo)
        se tentativa == falso
            devolve percorre_vizinhos(resto de origens, destino, grafo)
        senão
            devolve tentativa

# Exemplo de grafo.
seja exemplo = [
                nó("a", ["b", "c"]),
                nó("b", ["d", "e"]),
                nó("c", ["e"]),
                nó("d", []),
                nó("e", [])
               ]

# Teste.
teste encontra_caminho("a", "e", exemplo)
```

Figura 6.2: Programa para busca de caminho em grafo acíclico em Faz

7 TRABALHOS RELACIONADOS

Faz é uma linguagem funcional didática baseada no português com tipagem semi-estática. Este capítulo discute outros trabalhos com cada uma dessas características para fins de comparação com o presente trabalho. Serão discutidas linguagens baseadas no português, linguagens funcionais de propósito geral e didáticas e outras linguagens didáticas não funcionais, bem como sistemas de tipos estáticos para Racket.

7.1 Linguagens baseadas no português

Diversos cursos e materiais didáticos empregam variantes do que se conhece por Português Estruturado, ou Portugol (MANZANO, 2006). Em sua forma original, Portugol foi concebido como uma forma de pseudo-código, utilizado apenas como ferramenta notacional para descrever algoritmos, e não como uma linguagem executável por computador. Posteriormente à sua popularização, surgiram implementações para variantes mais rigorosamente definidas da linguagem, tais como o G-Portugol (SILVA, 2010), o Portugol Viana (VIANA, 2008), o Portugol IDE (MANSO; OLIVEIRA; MARQUES, 2009) e o Visualg (TONET, 2012).

Todas as variantes de Portugol são linguagens imperativas reminiscentes da linguagem Pascal (WIRTH, 1971), não apresentando, portanto, as características que tornam o paradigma funcional vantajoso no ensino de programação citadas no Capítulo 2 deste trabalho. Não é do conhecimento do autor outras linguagens funcionais baseadas no português.

7.2 Linguagens funcionais de propósito geral

7.2.1 Família LISP

LISP (MCCARTHY, 1960) introduziu o paradigma de programação funcional, baseado na composição e aplicação de funções. Desde sua criação, surgiram diversos dialetos ou novas linguagens baseadas em LISP, de tal maneira que hoje em dia entende-se LISP como uma família de linguagens relacionadas. Os principais membros dessa família são: o Scheme (KELSEY et al., 1998), que por sua vez deu origem à linguagem Racket e suas variantes, abordadas no Capítulo 3 deste trabalho; o Common LISP (STEELE JR., 1990), uma linguagem criada com o intuito de unificar os diversos dialetos de LISP criados nas décacas de 1970 e 1980, tais como o MACLISP (MOON, 1974), o Interlisp (TEITELMAN; GOODWIN; BOBROW, 1978) e o Lisp Machine Lisp (WEINREB; MOON, 1981); e Clojure (HICKEY, 2012), uma linguagem que executa sobre a máquina virtual da linguagem Java (ARNOLD; GOSLING; HOLMES, 2005) e possui funcionalidades de integração com esse ambiente. Destas, Scheme e derivados têm sido frequentemente

utilizados no ensino de programação. Todas essas linguagens são dinamicamente tipadas e empregam uma sintaxe baseada em expressões aninhadas delimitadas por parênteses (e, no caso de Clojure e algumas variantes de Scheme, colchetes e chaves), tal como visto no Capítulo 3. Common LISP é uma linguagem funcional impura, incluindo diversas funções e macros para manipulação de estado. Scheme é uma linguagem impura, mas dá uma ênfase maior à construção de programas puros. Clojure, por sua vez, possui elementos impuros, mas dá uma ênfase ainda maior à construção de programas puros, fornecendo diversas estruturas de dados imutáveis em sua biblioteca padrão.

As vantagens e desvantagens das linguagens da família LISP em um contexto didático são similares aos pontos citados no Capítulo 3.

7.2.2 Família ML

Outra família importante de linguagens funcionais originou-se com a linguagem ML. Dentre as linguagens dessa família, destacam-se Standard ML (MILNER; TOFTE; MACQUEEN, 1997) e OCaml (LEROY et al., 2012). ML teve uma influência no desenvolvimento da linguagem Miranda (TURNER, 1985), que por sua vez inspirou as linguagens Haskell (MARLOW et al., 2010) e Clean (PLASMEIJER; EEKELEN, 2011). São linguagens estaticamente tipadas com polimorfismo paramétrico. Utilizam-se de uma sintaxe equacional: funções são geralmente definidas por partes na forma de equações que dizem como uma chamada da função com diferentes argumentos deve ser avaliada. Por exemplo, a função fatorial em Haskell pode ser definida como:

```
fatorial 0 = 1
fatorial n = n * fatorial (n-1)
```

Nessas linguagens, a aplicação de funções é escrita sem parênteses, justapondo o nome da função e seus argumentos. Funções de múltiplos argumentos são tratadas como funções *curried*: uma função de dois argumentos, por exemplo, é tratada como uma função que recebe um argumento e produz uma nova função, que é então aplicada ao segundo argumento. Isso permite a aplicação parcial de funções. Por exemplo, dada a seguinte definição em Haskell:

```
soma x y = x+y
```

é possível escrever a seguinte definição:

```
incrementa = soma 1
```

Ou seja, `soma`, aplicada a apenas um argumento, `1`, produz uma função que espera o próximo argumento e soma `1` ao mesmo.

As linguagens diferem na maneira como tratam polimorfismo, especialmente com relação aos tipos numéricos. Em ML, constantes possuem um tipo fixo (e.g., `1` é do tipo `int`, enquanto `1.0` é do tipo `real`). Os operadores aritméticos são polimórficos, permitindo usar os mesmos operadores com operandos dos diferentes tipos numéricos, desde que ambos os operandos sejam do mesmo tipo; não há conversão implícita entre os tipos. OCaml utiliza operadores distintos para os diferentes tipos numéricos (e.g., `+` para inteiros e `+.` para reais). Em Haskell, tipos polimórficos são atribuídos a constantes e a operadores por meio de um sistema de *typeclasses*. Por exemplo, `1` é do tipo `Num a => a`, i.e., a constante pode ser instanciada para qualquer tipo `a` que implemente a

typeclass `Num`. Miranda utiliza um único tipo numérico `num` que engloba tanto inteiros quanto números em ponto flutuante, evitando a questão do polimorfismo das constantes e operadores aritméticos.

ML e OCaml são linguagens funcionais impuras, permitindo o uso de estado mutável através de referências e de funções de entrada e saída irrestritas. Haskell e Clean são linguagens puramente funcionais, restringindo a propagação de efeitos colaterais no programa.

A sintaxe equacional para definição de variáveis e funções nas linguagens da família ML, em especial Miranda, Haskell e Clean, é bastante clara. Por outro lado, a sintaxe de Faz busca minimizar as diferenças sintáticas entre o paradigma imperativo e o funcional, permitindo um foco maior nas diferenças semânticas entre os paradigmas. Haskell utiliza mais operadores infixados do que Faz, tais como `:` para construir listas (análogo às funções `cons` em LISP e `elo` em Faz), `!!` para a indexação de listas e `.` para a composição de funções. Isso torna o código mais compacto, mas por outro lado introduz novas regras de precedência à linguagem e pode tornar o código menos legível para um usuário que não esteja familiarizado com esses operadores. Os sistemas de tipos das linguagens da família ML são mais restritivos do que o de Faz. Por um lado, isso confere a essas linguagens maiores garantias estáticas de segurança. Por outro lado, conforme visto na Seção 4.2.1, a presença de tipos de união em Faz dá uma maior flexibilidade à linguagem, permitindo que certos programas possíveis em linguagens dinâmicas também possam ser escritos em Faz sem que para isso seja necessário abandonar completamente a tipagem estática. Miranda e Faz unificam os tipos numéricos, permitindo que operações combinem números inteiros e reais de maneira transparente. Isso é benéfico em uma linguagem introdutória, pois permite trabalhar com números como as entidades matemáticas que representam, sem que o usuário tenha que se preocupar com sua representação interna. Por outro lado, tipos inteiros e reais distintos permitem a geração de código mais eficiente.

7.3 Linguagens funcionais didáticas

LOGO (FOUNDATION, 2013) é uma linguagem didática baseada em LISP. Trata-se de uma linguagem multiparadigma, que incorpora elementos de programação funcional com comandos imperativos. A linguagem possui um enfoque em manipulação de texto e no desenho de gráficos através de comandos que controlam um cursor, usualmente na forma de uma tartaruga, capaz de se movimentar pela tela e traçar linhas. LOGO emprega uma sintaxe baseada em LISP. Em LOGO, entretanto, a maior parte dos parênteses são opcionais: o número de argumentos esperado por um operador é usado para determinar a interpretação correta de um comando. Por exemplo, uma vez que os operadores `sum` e `product` aceitam dois argumentos por padrão, um comando como

```
print product sum 2 3 sum 4 5
```

é interpretado como:

```
print (product (sum 2 3) (sum 4 5))
```

Além disso, LOGO suporta o uso de operadores aritméticos infixados. LOGO utiliza uma sintaxe peculiar para o acesso e atribuição de variáveis. No acesso, variáveis são escritas na forma `:`*`nome`*. A atribuição de um valor a uma variável é feita através do comando `make "`*`nome valor`*, onde `"`*`nome`* é uma *palavra*, um tipo de dados equivalente

ao símbolo de Racket. LOGO utiliza escopo dinâmico para as variáveis: variáveis definidas em uma função são visíveis em outras funções chamadas por ela. Assim como as demais linguagens da família LISP, LOGO é dinamicamente tipada.

A linguagem LOGO é uma tentativa interessante de tornar a sintaxe de LISP mais acessível a programadores iniciantes, eliminando a maior parte dos parênteses. Porém, a interpretação das expressões dependente do número de argumentos esperado por cada comando pode dificultar a compreensão de quais argumentos pertencem a cada comando. Além disso, LOGO herda diversas peculiaridades dos LISPs clássicos abandonadas em Scheme, tais como escopo dinâmico, a atribuição de valores a variáveis por meio de símbolos e a manipulação de funções por meio de símbolos e listas e não como valores de primeira classe. Assim como os demais LISPs, LOGO é dinamicamente tipado, o que incorre nos mesmos problemas com a tipagem dinâmica mencionados no Capítulo 3.

Helium (HEEREN; LEIJEN; IJZENDOORN, 2003) é um compilador e interpretador para a linguagem Haskell desenvolvido especificamente para o ensino dessa linguagem. Helium elimina certas funcionalidades de Haskell, tais como polimorfismo baseado em *typeclasses*, o que permite ao compilador detectar certos erros comuns entre iniciantes e emitir avisos e mensagens de erro mais apropriadas. A ideia é análoga à das sublinguagens didáticas de *How to Design Programs*, como visto no Capítulo 3 deste trabalho. As vantagens e desvantagens da linguagem em um contexto didática são análogas às de Haskell, discutidas na seção anterior.

7.4 Outras linguagens didáticas

Pascal (WIRTH, 1971) foi uma das primeiras linguagens criadas tendo o ensino de programação como um de seus propósitos. Trata-se de uma linguagem imperativa, com tipagem estática forte, que enfatiza os princípios da programação estruturada.

Scratch (MALONEY et al., 2010) e Etoys (KAY, 2005) são linguagens e ambientes didáticos baseados em Squeak (INGALLS et al., 1997), uma variante de Smalltalk (GOLDBERG; ROBSON, 1983). Em Scratch, programas são construídos por meio da composição visual de expressões e estruturas de controle. A linguagem tem um foco na criação de software multimídia. Etoys permite a criação e controle de objetos em um mundo virtual. Ambas as linguagens têm como público alvo crianças no Ensino Fundamental e Médio.

Assim como as variantes de Portugol, Pascal é uma linguagem imperativa, não apresentando os benefícios do paradigma funcional. Por outro lado, Pascal tem a vantagem de poder ser utilizado como uma linguagem de propósito geral. Scratch e Etoys, além de não serem linguagens funcionais, possuem um público alvo diferente: enquanto Scratch e Etoys são voltadas a estudantes do Ensino Fundamental e Médio, com um enfoque em aplicações multimídia e interativas, Faz é voltada a estudantes de nível superior, com um enfoque em programação mais convencional para manipulação de dados.

7.5 Tipagem estática em Racket

Typed Racket, anteriormente conhecido como Typed Scheme (TOBIN-HOCHSTADT; FELLEISEN, 2008), é uma tentativa de elaborar um sistema de tipos estático suficientemente flexível para permitir o estilo de programação convencionalmente utilizado em Racket, em que tipos são combinados de forma mais livre do que em linguagens estaticamente tipadas convencionais, à maneira dos tipos mistos

de HtDP, descritos na Seção 3.1.8. Assim como Faz, Typed Racket permite o uso de uniões de tipos e emprega condicionais com predicados de tipos para selecionar entre os múltiplos tipos de uma variável cujo tipo é uma união. Por outro lado, diferentemente de Faz, Typed Racket não possui a noção de compatibilidade parcial de tipos, o que faz com que a linguagem rejeite programas que seriam aceitos por Faz. Typed Racket permite a descrição de tipos mais elaborados do que Faz, tais como intersecções de tipos e funções com número variável de argumentos, o que confere uma maior flexibilidade à linguagem e permite dar tipos mais precisos a algumas funções. Por outro lado, os tipos de Typed Racket podem ser bastante complexos e de difícil compreensão, especialmente no contexto de uma disciplina introdutória. Typed Racket possui limitações similares às de Faz no que diz respeito a polimorfismo paramétrico.

8 CONCLUSÃO

Neste trabalho foi elaborada a linguagem Faz, uma linguagem de programação visando a facilitar o ensino e a aprendizagem de cursos introdutórios de programação empregando o paradigma funcional. A linguagem foi desenvolvida visando a evitar os problemas encontrados no ensino de programação com as linguagens didáticas *How to Design Programs*, baseadas em Racket. Diversas decisões de projeto foram consideradas na elaboração da linguagem, buscando torná-la mais familiar e mais facilmente compreensível a alunos brasileiros egressos do Ensino Médio. Para isso, optou-se por adotar uma sintaxe baseada no português e na notação matemática e de outras linguagens de programação. Também decidiu-se adotar uma organização do código baseada em blocos e uma distinção entre comandos e expressões, o que, além de tornar a linguagem mais legível e tornar definições globais e locais de variáveis e funções mais similares, reduz a distância sintática entre Faz e linguagens imperativas populares, como C. Isto torna linguagem mais familiar a alunos com experiência prévia nessas linguagens e permite focar o aprendizado nas diferenças semânticas entre as linguagens. Por outro lado, o uso de convenções da matemática e de comandos em português torna a linguagem mais acessível a alunos sem experiência prévia em programação. Finalmente, introduziu-se à linguagem um sistema de tipos semi-estático com uniões de tipos, permitindo à linguagem incorporar declarações formais dos tipos de funções e estruturas de dados e, ao mesmo tempo, manter a flexibilidade proporcionada pelo sistema de tipos dinâmico de Racket.

A linguagem foi implementada e integrada ao ambiente de desenvolvimento DrRacket. Para isso, foram implementados os passos de análise léxica e sintática do código, utilizando para isso as bibliotecas de geração de analisadores que acompanham a linguagem Racket, análise semântica, incluindo verificação de tipos e coleta de informações sobre as variáveis, funções e tipos definidos pelo usuário, e a tradução do código para a linguagem Racket a partir dos dados gerados pelos passos anteriores, permitindo assim reaproveitar a infraestrutura da linguagem Racket na implementação da nova linguagem.

A linguagem Faz e sua atual implementação apresentam algumas limitações, conforme visto nas Seções 4.4 e 5.2.7. Trabalhos futuros incluem a eliminação dessas limitações. No que diz respeito à linguagem, poderia ser interessante a adição de mecanismos sintáticos para a execução de efeitos colaterais em sequência, o que permitiria a incorporação de funções convencionais de entrada e saída, cuja ausência é uma limitação herdada da linguagem ISλ. Quanto à interface, melhorias incluem indicações mais precisas de erros de sintaxe encontrados no código e uma melhor adaptação do editor de código do ambiente DrRacket às convenções sintáticas da nova linguagem, especialmente com relação à indentação automática e coloração de sintaxe. Para isso, seria necessário reescrever o analisador sintático da linguagem e integrar ao ambiente DrRacket funções para a determinação da indentação e coloração corretas.

A implementação atual do sistema de tipos da linguagem também é passível de melhorias. Testes de pertinência não admitem tipos contendo variáveis. Testes de pertinência a tipos funcionais em tempo de execução (e.g., `se f ∈ Funções (Números) -> Números`) atualmente não são possíveis, pois a informação dos tipos dos argumentos e de retorno das funções, presente em tempo de compilação, não é preservada em tempo de execução pela etapa de tradução do código Faz para Racket. Possíveis soluções incluem alterar a representação das funções em tempo de execução de maneira a incluir essa informação ou manter uma tabela relacionando cada função com seu tipo. Na prática, esse tipo de teste é raro, especialmente no contexto de uma disciplina introdutória. A implementação atual do cômputo de intersecções de tipos na combinação de restrições é limitada, retornando `Nada` caso os tipos não sejam idênticos. Essa restrição só afeta algumas funções polimórficas com parâmetros de tipos funcionais quando a relação de subtipagem é utilizada.

O sistema de tipos em si apresenta limitações no que diz respeito à interação entre tipos paramétricos e uniões de tipos. Tipos polimórficos são considerados incompatíveis entre si. Isso implica que funções polimórficas, como `compose` (Figura 4.9), não admitem argumentos polimórficos. Além disso, no caso geral, na presença de uniões de tipos nem sempre há uma solução única para um conjunto de restrições sobre variáveis de tipos. A implementação atual rejeita conjuntos de restrições com essa propriedade. Uma integração mais completa entre tipos paramétricos e uniões de tipos é uma possível melhoria. Outro trabalho futuro seria formalizar o sistema de tipos por meio de regras de inferência lógica mais precisamente definidas, visando a verificar formalmente quais propriedades são garantidas ou não pelo sistema em comparação a sistemas de tipos mais convencionais.

Finalmente, seria interessante experimentar a linguagem na prática, empregando-a em uma disciplina de ensino de programação funcional, de maneira a verificar se as modificações realizadas em relação às linguagens HtDP são efetivas em facilitar a exposição e compreensão dos conceitos pelos alunos.

REFERÊNCIAS

ABELSON, H.; SUSSMAN, G. **Structure and Interpretation of Computer Programs**. Cambridge, Mass., USA: MIT Press, 1985.

ARMSTRONG, J. et al. **Concurrent Programming in ERLANG**. [S.l.]: Prentice Hall, 1996.

ARNOLD, K.; GOSLING, J.; HOLMES, D. **The Java Programming Language**. 4th.ed. [S.l.]: Addison-Wesley Professional, 2005.

BARRAS, B. et al. **The Coq proof assistant reference manual**: Version 6.1. Acesso em setembro de 2013, `http://hal.archives-ouvertes.fr/docs/00/06/99/68/PDF/RT-0203.pdf`.

BOVE, A.; DYBJER, P.; NORELL, U. A brief overview of Agda–a functional language with dependent types. In: **Theorem Proving in Higher Order Logics**. [S.l.]: Springer, 2009. p.73–78.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: SYMPOSIUM ON OPERATING SYSTEMS DESIGN & IMPLEMENTATION - VOLUME 6, 6., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2004. p.10–10. (OSDI'04).

FELLEISEN, M. et al. **How to Design Programs**. [S.l.]: MIT Press, 2001.

FELLEISEN, M. et al. The Structure and Interpretation of the Computer Science Curriculum. **Journal of Functional Programming**, West Sussex, v.10, n.11–13, Sept./Nov. 2004.

FLATT, M.; PLT. **Reference**: Racket. Acesso em outubro de 2013, `http://racket-lang.org/tr1/`.

FOUNDATION, L. **Logo Foundation**. Acesso em outubro de 2013, `http://el.media.mit.edu/logo-foundation/index.html`.

GOLDBERG, A.; ROBSON, D. **Smalltalk-80**: the language and its implementation. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1983.

HEEREN, B.; LEIJEN, D.; IJZENDOORN, A. van. Helium, for learning Haskell. In: ACM SIGPLAN WORKSHOP ON HASKELL, 2003. **Proceedings...** [S.l.: s.n.], 2003. p.62–71.

HICKEY, R. **Clojure documentation**. Acesso em setembro de 2013, `http://clojure.org/documentation`.

HUGHES, J. Why Functional Programming Matters. **The Computer Journal**, [S.l.], v.32, n.2, p.98–107, 1989.

INGALLS, D. et al. Back to the future: the story of Squeak, a practical Smalltalk written in itself. **ACM SIGPLAN Notices**, [S.l.], v.32, n.10, p.318–326, 1997.

JOHNSON, S. C. **Yacc**: Yet Another Compiler-Compiler. [S.l.]: Bell Laboratories Murray Hill, NJ, 1975. v.32.

KAUFMANN, M.; MOORE, J. S. ACL2: An industrial strength version of Nqthm. In: COMPUTER ASSURANCE, 1996. COMPASS'96,'SYSTEMS INTEGRITY. SOFTWARE SAFETY. PROCESS SECURITY'. PROCEEDINGS OF THE ELEVENTH ANNUAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 1996. p.23–34.

KAY, A. **Squeak Etoys, Children & Learning**. Acesso em outubro de 2013, `ftp://debian.ofset.org/speeches/jrfernandez/malaga08/doc/etoys_n_learning.pdf`.

KELSEY, R. et al. Revised^5 Report on the Algorithmic Language Scheme. **ACM SIGPLAN Notices**, New York, NY, USA, v.33, n.9, p.26–76, Sept. 1998.

LAUSANNE École Polytechnique Fédérale de. **Scala documentation**. Acesso em outubro de 2013, `http://www.scala-lang.org/documentation/`.

LEROY, X. et al. **The OCaml system release 4.00**: Documentation and user's manual. Acesso em setembro de 2013, `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

LESK, M. E.; SCHMIDT, E. **Lex**: a lexical analyzer generator. [S.l.]: Bell Laboratories Murray Hill, NJ, 1975.

MALONEY, J. et al. The Scratch Programming Language and Environment. **ACM Transactions on Computing Education (TOCE)**, [S.l.], v.10, n.4, p.16, 2010.

MANSO, A.; OLIVEIRA, L.; MARQUES, C. G. Ambiente de Aprendizagem de Algoritmos – Portugol IDE. In: VI CONFERÊNCIA INTERNACIONAL DE TIC NA EDUCAçãO – CHALLENGES 2009. **Actas...** Braga: Centro de Competência da Universidade do Minho, 2009.

MANZANO, J. LPP–Linguagem de Projeto de Programação: proposta de padronização da estrutura sintática de uma linguagem de projeto de programação a ser definida para a área de desenvolvimento de software para países com idioma português. **THESIS**, São Paulo, v.6, p.44–58, Setembro 2006.

MARLOW, S. et al. **Haskell 2010 Language Report**. Acesso em setembro de 2013, `http://www.haskell.org/onlinereport/haskell2010`.

MARLOW, S.; JONES, S. P. et al. The Glasgow Haskell Compiler. In: **The Architecture of Open Source Applications**. [S.l.]: Lulu Press, 2012. v.2.

MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, Part I. **Communications of the ACM**, New York, NY, USA, v.3, n.4, p.184–195, Apr. 1960.

MILNER, R.; TOFTE, M.; MACQUEEN, D. **The Definition of Standard ML**. Cambridge, MA, USA: MIT Press, 1997.

MOON, D. A. **MACLISP Reference Manual**. [S.l.]: Massachusetts Institute of Technology, 1974.

NORVIG, P. **Paradigms of artificial intelligence programming**: case studies in Common LISP. [S.l.]: Morgan Kaufmann, 1992.

PIERCE, B. C. **Types and Programming Languages**. [S.l.]: The MIT Press, 2002. p.206–207.

PLASMEIJER, R.; EEKELEN, M. van. **Clean Language Report version 2.2**. Acesso em outubro de 2013, `http://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf`.

PLT DESIGN, I. **Racket Documentation**: drracket plugins. Acesso em outubro de 2013, `http://docs.racket-lang.org/tools/`.

ROUNDY, D. Darcs: distributed version management in Haskell. In: ACM SIGPLAN WORKSHOP ON HASKELL, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.1–4.

SILVA, T. **G-Portugol – Manual**. Acesso em outubro de 2013, `http://gpt.berlios.de/manual_big/manual.html`.

STEELE JR., G. L. **Common LISP**: the language (2nd ed.). Newton, MA, USA: Digital Press, 1990.

SUSSMAN, G. J.; JR., G. L. S. Scheme: an interpreter for extended lambda calculus. In: MEMO 349, MIT AI LAB. **Anais...** [S.l.: s.n.], 1975.

TEITELMAN, W.; GOODWIN, J.; BOBROW, D. G. **Interlisp Reference Manual**. [S.l.]: Xerox Palo Alto Research Centers, 1978.

TOBIN-HOCHSTADT, S.; FELLEISEN, M. The design and implementation of Typed Scheme. In: ACM SIGPLAN NOTICES. **Anais...** [S.l.: s.n.], 2008. v.43, n.1, p.395–406.

TONET, B. **Software Visualg 2.0**. Acesso em outubro de 2013, `http://www.cefetsp.br/edu/adolfo/disciplinas/lpro/materiais/visualg.pdf`.

TURNER, D. A. Miranda: a non-strict functional language with polymorphic types. In: FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE. **Anais...** [S.l.: s.n.], 1985. p.1–16.

VIANA, P. **Melhoria do Processo de Ensino-Aprendizagem nas disciplinas de Programação e Algoritmos**. Acesso em outubro de 2013, `http://portugolviana.estg.ipvc.pt/`.

WEINREB, D.; MOON, D. The Lisp Machine Manual. **ACM SIGART Bulletin**, [S.l.], n.78, p.10–10, 1981.

WIRTH, N. The programming language Pascal. **Acta informatica**, [S.l.], v.1, n.1, p.35–63, 1971.