André Cassulino Araújo Souza

Codegs: um Tipo Especial de Metaobjetos em Cyan

Sorocaba, SP

01 de Fevereiro de 2018

André Cassulino Araújo Souza

Codegs: um Tipo Especial de Metaobjetos em Cyan

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Teoria Aplicada à Computação.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Prof. Dr. José de Oliveira Guimarães

Sorocaba, SP

01 de Fevereiro de 2018

Cassulino Araújo Souza, André
Codegs: um Tipo Especial de Metaobjetos em Cyan/ André Cassulino Araújo Souza. – 2017.
114 f. : 30 cm.

Dissertação (Mestrado) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
Programa de Pós-Graduação em Ciência da Computação – PPGCC-So.
Orientador: Prof. Dr. José de Oliveira Guimarães
Banca examinadora: Prof. Dr. José de Oliveira Guimarães, Prof. Dr. Mateus Conrad Barcellos da Costa, Prof. Dr. Gustavo Maciel Dias Vieira
Bibliografia

1. Codegs. 2. Metaprogramação em tempo de compilação. 3. Ferramentas visuais para geração de código. I. Orientador. II. Universidade Federal de São Carlos. III. Título

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de dissertação de mestrado do candidato André Cassulino Araújo Souza, realizada em 01/12/2017:

José de Oliveira Guimarães

Prof. Dr. José de Oliveira Guimarães
UFSCar

Prof. Dr. Mateus Conrad Barcellos da Costa
IFES

Prof. Dr. Gustavo Maciel Dias Vieira
UFSCar

Certifico que a sessão de defesa foi realizada com a participação à distância do membro Prof. Dr. Mateus Conrad Barcellos da Costa e, depois das arguições e deliberações realizadas, o participante à distância está de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa do aluno André Cassulino Araújo Souza.

José de Oliveira Guimarães

Prof. Dr. José de Oliveira Guimarães
Presidente da Comissão Examinadora
UFSCar

A minha esposa e filha por sempre estarem ao meu lado, abdicando de momentos juntos, pela compreensão e apoio incondicional em todos os momentos. Aos meu Pais pelo ensinamento de valores e princípios a mim atribuídos, sua valentia e determinação em não desistir são fontes de inspiração para minha vida.

Agradecimentos

Agradeço,

A Deus pelo dom supremo da vida e a oportunidade de trilhar novos caminhos.

Ao meu orientador Dr. José de Oliveira Guimarães sou extremamente grato pela oportunidade, conhecimento, paciência, competência, capacidade e sabedoria. A ele meu profundo agradecimento pelo incentivo e orientação.

Aos professores Dr. Gustavo Maciel Dias Vieira e Dr. Mateus Conrad Barcellos da Costa por terem aceitado o convite para participar da banca, assim como por suas críticas, sugestões e comentários.

Aos demais professores do Programa de Pós Graduação em Ciências da Computação pelo esforço e dedicação.

Ao Roberto Romualdo Marvulle da PPGCCS pelo auxílio e atenção em todo o momento que precisei.

A Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP). Projeto Implementação da Linguagem Cyan, processo nº 2014/01817-3.

A todos que direta ou indiretamente me ajudaram e que de alguma maneira contribuíram para conclusão desse projeto.

A todos vocês, o meu muito obrigado!

"Ensina-nos a contar os nossos dias de tal maneira que alcancemos corações sábios."
(Salmos 90:12)

Resumo

Metaprogramação é amplamente suportada por diversas linguagens de programação. Existe metaprogramação em tempo de edição, compilação e execução. Em muitas linguagens de programação a metaprogramação é definida por um protocolo de metaobjetos que determinam em qual tempo e o que poderá ser feito. A metaprogramação permite ao programador "participar" do processo de compilação. Uma das maneiras de se fazer isto é através de metaobjetos construídos por usuários comuns e que são carregados pelo compilador. Os metaobjetos podem fazer conferências, modificações e geração de código tanto no código-fonte como na Árvore de Sintaxe Abstrata. No entanto, para usar a metaprogramação é necessário ter um conhecimento aprofundado em relação à linguagem tornando tedioso e custoso o seu uso. Por outro lado temos as ferramentas visuais de geração de código. Por exemplo, vários IDEs dão suporte à geração de código para GUI (Graphical User Interface) em várias linguagens de programação, embora não interagindo com o compilador. Neste trabalho é apresentado um plugin para o IDE Eclipse e uma biblioteca de Codegs. Codeg é um tipo especial de metaobjetos de tempo de compilação de Cyan suportado pelo IDE Eclipse, fornecendo recursos de visuais em tempo de edição. O plugin se comunica com o compilador através do Protocolo de Metaobjetos de Cyan.

Palavras-chaves — Metaprogramação. Ferramentas visuais de geração de código. Integração metaprogramação e ferramentas visuais.

Abstract

Metaprogramming is widely supported by several programming languages. There exist metaprogramming at time of editing, compilation, and execution. In many languages metaprogramming is defined by a protocol that determines in what time and what can be done. Metaprogramming allows a programmer to "participate" in the build process. One of the ways to do this is through meta-objects built by ordinary users that are loaded by the compiler. Meta-objects can make conferences, modifications, and code generation either in the source code or in the Abstract Syntax Tree. However, to use metaprogramming it is necessary to have an in-depth knowledge of the programming language, thus making it tedious and costly to use. On the other hand, we have the visual generation of code. For example, several IDEs support code generation for GUI (Graphical User Interface) in several programming languages although they do not interact directly with the compiler. In this work, we present a plugin for the Eclipse IDE and a Codegs library. Codegs are a special type of compile-time metaobject in Cyan supported by the plugin to the Eclipse IDE, providing visual resources at edit time. The plugin communicates with the compiler through the of Cyan Metaobject Protocol.

Keywords — Metaprogramming. Visual code generation tools. Metaprogramming integration with visual tools.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

ASA	Árvore de Sintaxe Abstrata
AST	*Abstract Syntax Three* – O mesmo que ASA.
CASE	*Computer-Aided Software Engineering* - Engenharia de Software Assistida por Computador
CLP	Controlador Lógico Programável
DSL	*Domain Specific Language* - Linguagem Específica de Domínio
ETMOP	*Edit Time Metaobject Protocol* - Protocolo Metaobjeto de Tempo de Edição
IDE	*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado
MOP	*MetaObject Protocol* - Protocolo Metaobjeto

Sumário

1 Introdução

Segundo Damasevicius e Stuikys (DAMASEVICIUS; STUIKYS, 2008) a metaprogramação refere-se a métodos e processos de escrever programas de alto nível que constroem metaprogramas. Apesar da metaprogramação ser utilizada amplamente há muito tempo em ciência da computação, em muitos casos o termo é usado para definir conceitos diferentes de software isto quando não são empregados os conceitos de metaprogramação sem reconhecer o seu uso.

Guimarães (GUIMARÃES, 2014) diz que metaprogramação pode ter inúmeros sentidos: programa que manipula outros programas, programa que se auto examina, programa que modifica a si mesmo e geração de código em tempo de compilação.

A metaprogramação tem diferentes usos, sendo que, por exemplo, podemos usar para a pré-geração de tabelas de dados para serem usadas em compilação. Vamos supor que você esteja criando um jogo e quer uma pesquisa rápida para todos os senos dos números inteiros de 8 bits, você pode calculá-los e codificar manualmente ou pode pedir ao seu programa para construir a tabela na inicialização no tempo de execução ou pode criar um programa que gere o código da tabela antes da compilação. Tarefas deste tipo feitas em execução podem tornar a inicialização do programa extremamente lento e nestes casos escrever um programa que escreva tabelas de dados estáticas é a melhor opção (BARTLETT, 2005).

O uso de metaprogramação para geração de código em tempo de compilação permite ao programador escrever um código-fonte com menas linhas de código em uma programação de alto nível.

A metaprogramação pode ser feita também em tempo de execução, neste caso as alterações feitas no programa podem ser refletidas em tempo real sem a necessidade de recompilar todo o programa, porém isto tem um custo de processamento em tempo de execução.

Escrever metaprogramas é denominado de metaprogramação (BARTLETT, 2005). Metaprogramas podem ser definidos como programas geradores de códigos que encapsulam o desenvolvimento de programas (DAMASEVICIUS, 2006). Os metaprogramas podem aumentar a eficiência, a produtividade, a confiabilidade e qualidade dos sistemas de software (TAHA; SHEARD, 2000).

Damasevicius e Stuikys (DAMASEVICIUS; STUIKYS, 2008) determinaram 8 conceitos principais de metaprogramação a partir da analise de 41 fontes selecionadas do banco de dados on-line da IEEE Xplore, fontes estas que foram publicados entre os anos

de 1965 e 2007 e são relacionados a metaprogramação conforme a tabela 1.

Classe do Conceito	Conceito	No.	Total no.	Percentual
Transformação	Manipulação	11	29	70%
	Transformação	7		
	Modificação	5		
	Adaptação	4		
	Tradução	1		
	Pré-processamento	1		
Geração	Geração de código	14	17	41%
	Instanciação	2		
	Weaving	1		
Metaprograma	Modelo	6	16	39%
	Componente genérico	5		
	Macro	2		
	metaprograma	2		
	Meta-especificação	1		
Níveis de abstração	Representação	6	11	27%
	Abstração	4		
	Encapsulamento	1		
Generalização	Construção	5	8	20%
	Generalização	2		
	Parametrização	1		
Separação de conceitos	Análise	5	8	20%
	Separação de conceitos	3		
Reflexão	Reflexão	6	7	17%
	Introspecção	1		
Metadados	Metadados	3	5	12%
	Parâmetros	2		
Outros conceitos	Protocolo metaobjeto	2	11	27%
	Traits	2		
	Prova do teorema	1		
	Avaliação parcial	1		
	Inspeção	1		
	Especialização	1		
	Execução do tempo de execução	1		
	Otimização	1		
	Interpretação	1		

Tabela 1: Conceitos relacionados à metaprogramação (DAMASEVICIUS; STUIKYS, 2008)

Estes conceitos são detalhados nas seções a seguir.

1.1 Transformação

A transformação do programa é uma manipulação da sua representação, resultando na mudança da forma (sintaxe) do programa. Sua semântica pode ser alterada ou não no processo. Podemos então definir transformação como sendo o processo de mudar uma forma de um programa (código-fonte, especificação ou modelo) para outro, bem como uma descrição formal ou abstrata de um algoritmo que implementa essa transformação (DAMASEVICIUS; STUIKYS, 2008).

1.2 Geração

A geração de código é usado para produzir códigos de maneira automática, reduzindo assim a necessidade de programadores a escrever código manualmente.

Em Cyan (GUIMARÃES, 2017, p. 103) podemos gerar métodos e outras estruturas através do uso de metaobjetos em tempo de compilação.

Exemplo de metaobjeto em Cyan:

```
1 package main
2
3 object Person
4   public String name
5   public Int age
6   @init(name, age)
7 end
```

Código 1.1: Protótipo Person em Cyan

No código acima, `@init(name, age)` é chamado de "anotação de metaobjeto". Os métodos do metaobjeto serão responsáveis pela geração de um método `'init:'` que inicializa os parâmetros `name` e `age`, conforme demonstrado no código a seguir:

```
1 package main
2
3 object Person
4   public String name
5   public Int age
6   func init: String name, Int age {
7     self.name = name;
8     self.age = age;
9   }
10 end
```

Código 1.2: Classe Java equivalente ao protótipo Person em Cyan

1.3 Metaprograma

Metaprogramas são programas geradores de código que tem por finalidade abstrair o desenvolvimento de programas (DAMASEVICIUS; STUIKYS, 2008). Os metaprogramas geralmente são descritos usando uma estrutura genérica de linguagens de alto nível, como por exemplo os templates de `C++`.

Conforme demonstrado na Figura 1, as linguagens que suportam metaprogramas podem ser chamadas de metalinguagens enquanto que o programa gerado pode ser denominado de programa objeto e a linguagem na qual foi gerada o programa objeto pode ser chamada de linguagem objeto (CZARNECKI; EISENECKER, 2000) (ATTARDI; CISTERNINO, 2001).

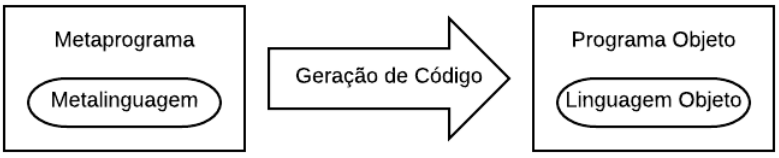


Figura 1: Metaprograma.

1.4 Níveis de Abstração

Abstração é a maneira fundamental de organizar conhecimento em agrupamento de fatos semelhantes (JACKSON, 2006) (STEPANOV, 2002). A abstração esconde detalhes menos importantes do ponto de vista funcional, mas essenciais para o desenvolvimento do programa, e enfatiza a informação que é importante para um desenvolvedor ou usuário final (DAMASEVICIUS; STUIKYS, 2008).

O programador pode escrever código em alto nível, conforme mostra a Figura 2, e o metaprograma irá gerar o programa-objeto utilizando as informações encapsuladas nos níveis de abstração.

1.5 Generalização

A generalização pode ser entendida como uma transformação de um componente de domínio específico para um componente genérico (metaprograma) que é mais amplamente utilizável e reutilizável do que o original. Um exemplo de generalização são as classes genéricas.

Classes genéricas são classes que encapsulam operações que não são específicas de um determinado tipo de dados. Pode ser usado com tabelas `hash`, filas, árvores, listas

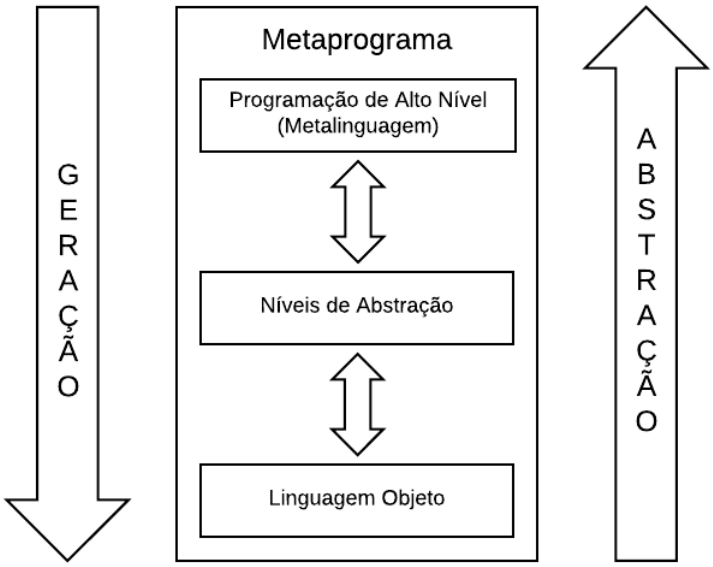


Figura 2: Níveis de Abstração de Metaprograma.

encadeadas, pilhas e afins. Independente do tipo de dados utilizados as operações são tratadas de forma similar (DAMASEVICIUS; STUIKYS, 2008).

Um exemplo de funções genéricas são os Templates[1] de C++:

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)  {
   return a < b ? b:a;
}

int main () {
   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

```

[1] Disponível em: <https://www.tutorialspoint.com/cplusplus/cpp_templates.htm>

```
20    string s1 = "Hello";
21    string s2 = "World";
22    cout << "Max(s1, s2): " << Max(s1, s2) << endl;
23
24    return 0;
25 }
```

Código 1.3: Template em C++

Neste caso, independente do tipo de dados passado, o template permite que seja feita a comparação. Após compilado e executado a saída ficaria assim:

```
1     Max(i, j): 39
2     Max(f1, f2): 20.7
3     Max(s1, s2): World
```

Código 1.4: Uso do Template em C++

1.6 Separação de conceitos

O termo "separação de conceitos" foi introduzido por Dijkstra (DIJKSTRA, 1982) o qual ele propôs que "dispõe-se a estudar profundamente um aspecto do sujeito isolado para sua própria consistência, sempre sabendo que se ocupa apenas com um dos aspectos".

Separação de conceitos no nível conceitual é geralmente considerado como principal meio para gerenciar a complexidade do domínio. As partes do programa relacionadas às preocupações são implementados separadamente e integrados de volta para formar um programa completo (DAMASEVICIUS; STUIKYS, 2008). Em resumo podemos dizer que separar conceitos é fazer uma aplicação em módulos com o foco de cada módulo em resolver um único problema e caso precise realizar algo que não faz parte de sua tarefa, solicita a outro módulo para colaborar quando se fizer necessário.

1.7 Reflexão

O termo "reflexão computacional" foi introduzido por Maes (MAES, 1987) que define reflexão computacional como a atividade realizada por um sistema computacional ao fazer computação sobre (e por isso possivelmente afetando) sua própria computação.

Damasevicius e Stuikys (DAMASEVICIUS; STUIKYS, 2008) diz que reflexão pode ser definida como a capacidade de um programa para manipular o seu próprio estado (por exemplo, sua semântica) como dado durante a sua própria execução.

A reflexão pode ser classificada como introspectiva ou comportamental. A reflexão introspectiva é quando um programa examina-se a si mesmo, já a reflexão comportamental permite alterar o próprio código do programa durante sua execução (GUIMARÃES, 2014).

Linguagens como Java, Smalltalk, Ruby e Groovy dão suporte à reflexão compor-tametal e introspectiva.

No exemplo abaixo exemplificamos o uso de reflexão introspectiva em Java.

Classe `HelloWorld`:

```
1 package main;
2
3 public class HelloWorld {
4
5     public void hello() {
6         System.out.println("Hello World!");
7     }
8 }
```

Código 1.5: Classe HelloWorld em Java

Chamada ao método `hello()` sem o uso de Reflexão:

```
1 package main;
2
3 public class Reflection {
4
5     public static void main(String[] args) {
6         HelloWorld helloWorld = new HelloWorld();
7         helloWorld.hello();
8     }
9 }
```

Código 1.6: Classe Java sem reflexão

Com o uso da reflexão:

```
1 package main;
2
3 import java.lang.reflect.Method;
4
5 public class Reflection {
6
7     public static void main(String[] args) {
8
9         try {
```

```
10              Object helloWorld = Class.forName("main.HelloWorld")
                    .newInstance();
11              Method m = helloWorld.getClass().getDeclaredMethod(
                   "hello", null);
12              m.invoke(helloWorld);
13          } catch (Exception e) {
14              e.printStackTrace();
15          }
16      }
17  }
```

Código 1.7: Classe Java com reflexão

Na linha 10 instanciamos o objeto `main.HelloWorld` e na linha 11 procuramos dentre os métodos declarados na classe deste objeto, pelo método `hello` que na sequência, o invocamos na linha 12.

Ou seja, neste caso, o programa é capaz de se examinar e invocar métodos das classes.

1.8 Metadados

Os metadados são dados estruturados e codificados que descrevem características e informações para auxiliar na identificação, descoberta, avaliação e gerenciamento de dados que possuem estes respectivos metadados (DAMASEVICIUS; STUIKYS, 2008).

As anotações (*annotations*) de Java são uma forma de metadados, as quais fornecem informações a respeito da própria classe mas não interferem ou alteram o funcionamento da classe em si (ORACLE, 2017). Dentre as formas de uso, podemos citar:

- informações para o compilador detectar erros ou suprimir avisos;
- processamento em tempo de compilação podendo gerar código-fonte ou outros arquivos de suporte como por exemplo XML;

Exemplo de anotações em Java:

```
1 myString = (@NonNull String) string;
```

Código 1.8: Anotação em Java

A anotação `@NonNull` indica que aquela variável não poderá conter um valor nulo, não fazendo nenhum tipo de alteração no código.

1.9 Outros Conceitos - Protocolo de metaobjetos

Um protocolo de metaobjetos (MOP) é uma interface que permite ao programador personalizar o comportamento e implementação de linguagens de programação e outros sistemas de softwares (CHIBA, 1995). Linguagens como CLOS (STEELE, 1990), Smalltalk (GOLDBERG; ROBSON, 1983) e Cyan (GUIMARÃES, 2017) implementaram protocolos de metaobjetos.

Uma das formas de metaprogramação se dá através de objetos que denominamos de metaobjetos. O conceito de metaobjetos pode variar dependendo da linguagem utilizada, portanto, vamos usar a definição da linguagem Cyan.

Em Cyan um metaobjeto é um objeto de tempo de compilação que pode adicionar código a um programa e fazer verificações no código-fonte além daquelas feitas pelo compilador (GUIMARÃES, 2017).

Podemos dividir os metaobjetos em metaobjetos de tempo de compilação e em metaobjetos de tempo de execução. Os metaobjetos em tempo de execução são objetos cujos métodos são executados durante a execução do programa sendo uma forma de implementarmos reflexão.

Com os metaobjetos de tempo de execução podemos adiar para a execução a decisão de fazer alterações no programa, tais como: inserir e remover variáveis de instância e métodos em classes, mudar a herança, criar e remover classes, entre outros, dando uma maior flexibilidade ao programador (GUIMARÃES, 1998).

Já os metaobjetos em tempo de compilação são objetos cujos métodos são executados em compilação. O compilador inclui o metaobjeto ao seu próprio código e os métodos do metaobjeto podem fazer conferências adicionais, inserir métodos e variáveis de instância em classes e protótipos, adicionar, modificar e/ou remover métodos existentes (GUIMARÃES, 2014).

1.10 Geração de Código por Ferramentas Visuais

Um tipo de metaprogramação é a geração de código usando interfaces gráficas. A geração de código pode ser feita através dos ambientes de desenvolvimento (IDE) como por exemplo o Adobe Dreamweaver[2] que permite criar páginas web apenas arrastando os elementos e posicionando-os na página, gerando de forma automatizada o código fonte em HTML.

Outros ambientes, como o NetBeans,[3] também oferecem ao programador estas funcionalidades, permitindo a criação de Interfaces Gráficas apenas arrastando os elementos

[2] Ambiente de desenvolvimento Integrado (ADOBE SYSTEMS INCORPORATED, 2017)
[3] Ambiente de desenvolvimento Integrado (NETBEANS, 2017).

e os posicionando. Também é gerado de forma automática o código fonte daquela interface.

As ferramentas CASE[4] também são ferramentas visuais para a programação desde a fase de projeto até a implementação do programa. Estas ferramentas possuem um IDE completo, aonde o programador não necessita escrever códigos extensos e na maioria das vezes precisa apenas posicionar os objetos e indicar quais são as funcionalidades que terão e todo o restante da programação é feito pelo IDE.

O uso destas ferramentas visuais trazem algumas vantagens como:

- maior produtividade: a maior parte do código é feita pela ferramenta e os códigos escritos pelo programador podem ter recursos de autocompletar;
- facilidade no uso: não é necessário conhecer os detalhes da linguagem e sim apenas arrastar os elementos para o local desejado;
- depuração: o IDE pode fornecer valores de variáveis em determinados pontos e conectar-se a diferentes repositórios de dados.
- posicionamento: no caso das interfaces gráficas, o programador não precisa se preocupar com o posicionamento, aonde ele arrastar e soltar o item, o IDE já determina o posicionamento correto;
- menor quantidade de código: o programador irá escrever apenas os códigos essenciais ao funcionamento dos elementos.

Porém o uso destas ferramentas também podem trazer algumas desvantagens, como:

- curva de aprendizado: os IDEs podem ser ferramentas complicadas necessitando de tempo para o programador aproveitar ao máximo todos seus benefícios;
- incompatibilidade: não é possível usar em sistemas que não possuam interfaces gráficas, como por exemplo em servidores Linux[5];
- treinamento: muitas ferramentas necessitam de treinamento específico para que se possa utilizar;
- custo/benefício: o investimento em algumas destas ferramentas não compensam em muitos casos os benefícios adquiridos (LEBLANG; CHASE, 1984);

[4] CASE: *Computer Aided Software Engineering* (Engenharia de Software Assistido por Computador) são ferramentas integradas para desenvolvimento de aplicações.

[5] Servidores "Linux" por padrão são instalados sem interface gráfica sendo sua total configuração e administração por linha de comando. O uso de interfaces gráficas desperdiçam recursos de processamento e memória que poderiam ser alocados a outras tarefas.

1.11 Motivação

Dmitriev e Watson (DMITRIEV; WATSON, 2004) diz que ser programador deveria significar que podemos fazer qualquer em um computador, ter controle completo. No entanto, ao contrário disto os programadores são restritos porque dependem de uma infraestrutura de programação que não se pode mudar facilmente. A programação depende das linguagens de programação e dos ambientes de desenvolvimento (IDE's). Caso precise de alguma alteração ou extensão da linguagem, é necessário aguardar até que o fornecedor do IDE adicione os novos recursos. Esta dependência é que limita a total liberdade de programar. O uso de editores de texto para a construção de programas, devido às suas limitações, dá uma falsa liberdade ao programador.

Quando um compilador compila o código-fonte, ele analisa o texto em uma estrutura semelhante a uma árvore, chamada árvore de sintaxe abstrata. Os programadores fazem essencialmente a mesma operação mentalmente quando lêem o código fonte. É por isso que temos colchetes e chaves e parênteses e precisamos formatar e recuar o código para seguir as convenções de codificação, de modo que torna mais fácil ler a fonte (DMITRIEV; WATSON, 2004).

A execução de tarefas comuns de programação, como criar métodos e variáveis de instância, ler e gravar dados em arquivos de texto ou em banco de dados, entre outros, é necessário conhecer específicos da linguagem de programação. Abstrair estes detalhes tornaria a programação menos tediosa e mais eficiente. Uma programação em alto nível permite ao programador focar em elementos essenciais ao seu desenvolvimento e se preocupar menos com detalhes específicos da linguagem de programação.

Por exemplo, vamos criar um classe `Employee` com variáveis de instância e criar a incialização desta variáveis:

```
1  public class Employee {
2      private String name;
3      private int age;
4
5      public void setName(String name){
6          this.name = name;
7      }
8
9      public void setAge(int age){
10         this.age = age;
11     }
12
13     public String getName(){
14         return this.name;
```

```
15      }
16
17      public int getAge(){
18          return this.age;
19      }
20  }
```

Código 1.9: Classe Employee em Java

Para cada uma das variáveis de instância que desejamos ler e / ou gravar informações nela, é necessário criarmos métodos `get` e `set` parecidos com os criados no código acima. Se criarmos outras classes com variáveis de instância privada teremos que repetir todo o processo de criação dos métodos para acessar estas variáveis. Podemos chamar estes códigos de `Boilerplate Code`.[6] As linguagens em geral não oferecem suporte a criação destes códigos.

Dmitriev e Watson (DMITRIEV; WATSON, 2004) argumentam que fazer a definição de um programa deveríamos poder passar ao compilador as informações que queremos sem entrar nos detalhes da linguagem de programação. Por exemplo, poderíamos selecionar através de uma interface gráfica um arquivo para carregar no código-fonte sem precisar conhecer a sintaxe da linguagem.

Ou ao especificar uma interface gráfica poderíamos determinar as ações que que um determinado botão deverá executar, sem precisar escrever o código-fonte relacionado as ações. Mas com os editores de texto usados em programação isto não é possível, uma vez que eles não nos fornecem estas facilidades. A única verdadeira razão pela qual usamos o texto hoje é porque não temos editores melhores do que editores de texto.

Para ajudar a resolver estes problemas, o uso de ferramentas visuais de geração de código, como por exemplo, os Ambientes de Desenvolvimento Integrado (IDE) auxiliam na geração de parte destes códigos necessários para o programa. Cada vez mais os ambientes de integrados de desenvolvimento, como Eclipse, Netbeans e Codeblocks, oferecem ao programador recursos e funcionalidades em uma ou mais linguagens de programação, como: `Java`, `C`, `C++`, `C#` e `.NET`. Isto possibilita ao programador o uso de interfaces gráficas para criação de classes, criação de métodos e variáveis, recurso de autocompletar código, verificação do código-fonte e alguns tipos de erros em tempo de edição.

Na Figura 3 mostramos o uso do IDE Eclipse[7] para gerar parte do código de uma classe. O usuário informa a definição de uma nova classe Java através de uma interface gráfica, na qual o usuário informa o pacote, nome da classe, superclasse, interface e também

[6] Boilerplate Code é a denominação dada a códigos repetitivos que são necessários para a linguagem de programação. Geralmente estes códigos possuem pouca ou nenhuma alteração e poderiam facilmente ser gerados automaticamente.

[7] (ECLIPSE FOUNDATION, 2017)

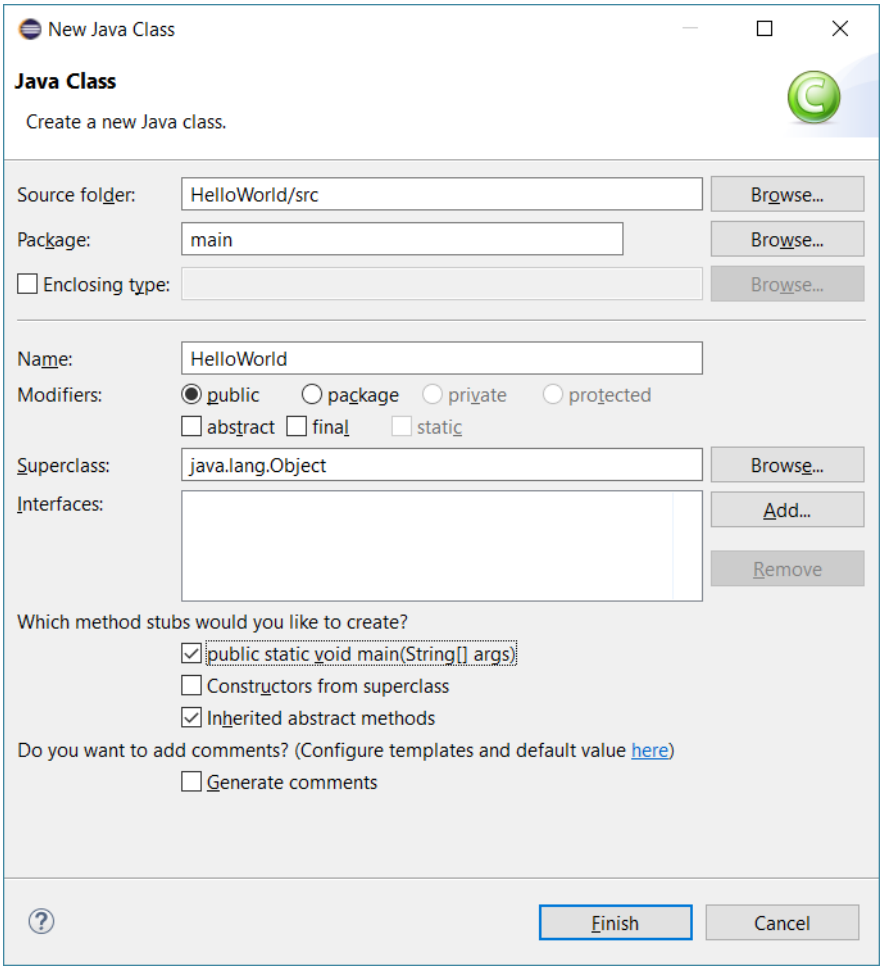


Figura 3: Nova Classe Java

se deseja a geração do método `main` e o código da classe é gerado pelo IDE.

Segue abaixo o código para esta classe gerado pelo IDE Eclipse:

```
1 package main
2
3 public class HelloWorld {
4     public static void main(String[] args) {
5         // TODO Auto-generated method stub
6
7     }
8 }
```

Código 1.10: Código Java gerado pelo IDE Eclipse

No entanto os IDE's não são integrados a linguagem de programação e não podem agir como metaobjetos da linguagem, não fornecendo assim os recursos e vantagens de metaprogramação em tempo de compilação.

Os metaobjetos de Cyan podem agir apenas em tempo de compilação e permitem ao

programador participar do processo de compilação. Cyan possui mais fases de compilação que linguagens tradicionais e permite que os metaobjetos interajam na maioria delas. Esta interação é feita através de um protocolo de metaobjeto denominado MOP. Maiores detalhes em relação a esta interação serão fornecidos no próximo capítulo.

1.12 Proposta

Nesta dissertação será apresentado um plugin para o IDE Eclipse e uma biblioteca de Codegs. Codegs são um tipo especial de metaobjetos existente em Cyan que serão integrados através deste plugin ao IDE Eclipse.

Métodos de um metaobjeto de tempo de compilação são chamados em pontos específicas do compilador e controlam a compilação de código fonte em Cyan.

Codegs combinam o poder dos metaobjetos de tempo de compilação de Cyan com as facilidades do uso de interfaces gráficas para a geração de código. Através de uma interface gráfica acoplada ao IDE um Codeg pode receber informações que serão utilizadas para alterar o comportamento do programa, adicionar informações a ele ou inspecionar o código-fonte, o que é tarefa de um metaobjeto.

Codegs resolvem parte dos problemas elencados anteriormente por Dmitriev e Watson (DMITRIEV; WATSON, 2004) ao fornecer ao programador uma interface gráfica, na qual ele poderá definir informações pré-determinadas e que são necessárias para a ação desejada. Em tempo de compilação, o Codeg poderá usar estas informações para fazer conferências e/ou modificações no código gerado pelo compilador. Deste modo, ao usar os Codegs o programador não necessitará conhecer detalhes específicos da linguagem inerentes a ação do Codeg.

Para dar suporte à execução dos Codegs de Cyan desenvolvemos um plugin para o IDE Eclipse e uma biblioteca de Codegs para avaliar o plugin e o funcionamento desta construção. Mostraremos o funcionamento de um Codeg chamado `color`. A Figura 4 mostra a anotação @color(red), que se refere ao Codeg `Color`, a qual recebe um identificador `red` e que após o programador parar o mouse em cima desta anotação durante a edição do programa no IDE Eclipse com o plugin que foi construído para a dissertação, o "Codeg" chama a interface gráfica na qual o programador poderá escolher a cor desejada. Os Codegs serão melhor explicados na Seção 4.

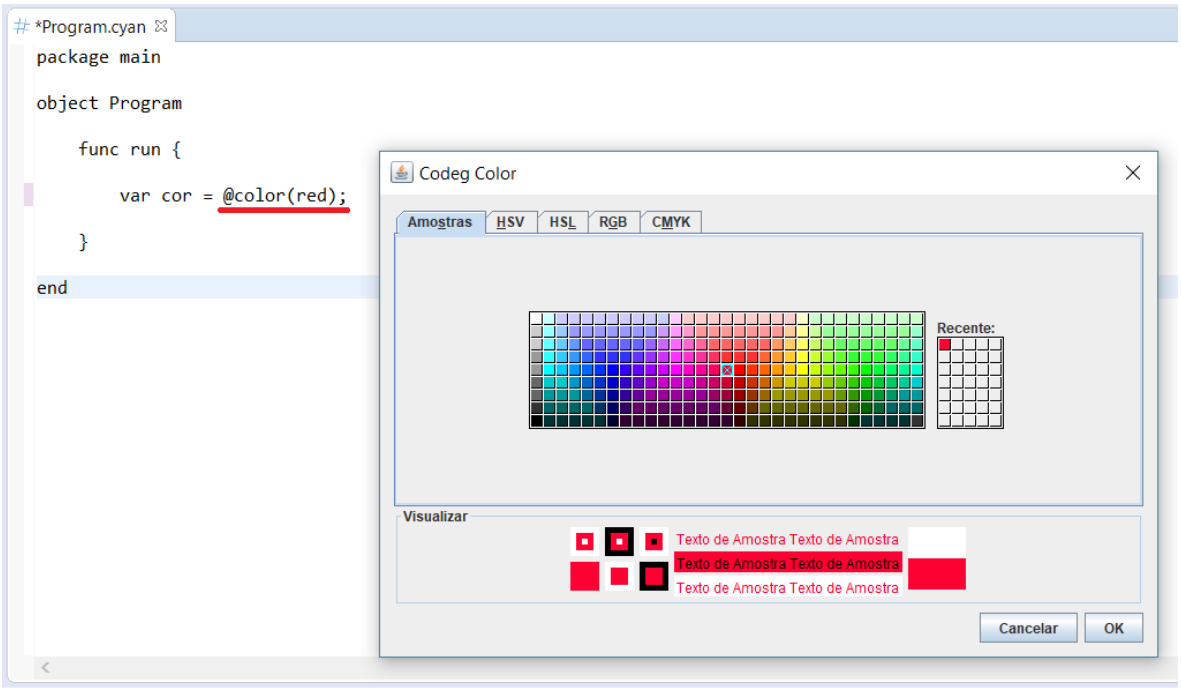


Figura 4: Escolha de cor usando o Codeg

1.13 Organização do Trabalho

No Capítulo 2 é descrita metaprogramação em diferentes linguagens, permitindo ao leitor compreender melhor o seu conceito.

As vantagens e desvantagens do uso de interfaces gráficas para geração de código são exploradas no Capítulo 3.

O Capítulo 4 descreve o procedimento de implementação do plugin, a biblioteca de Codegs, bem como as facilidades oferecidas ao programador com o uso dos Codegs.

Os resultados da dissertação e uma síntese dos resultados obtidos no Capítulo 5.

2 Metaprogramação em Tempo de Compilação

Metaprogramação em tempo de compilação permite a geração de código em tempo de compilação (GUIMARÃES, 2018). Essas transformações podem alterar a Árvore de sintaxe abstrata (ASA) de um programa (GROOVY, 2016) ou diretamente no código-fonte gerado. Em algumas linguagens como Cyan, a metaprogramação se dá através de objetos que denominamos de metaobjetos. Já em outras linguagens como LISP, metaprogramação em tempo de compilação é feita em parte através de macros.

Guimarães (GUIMARÃES, 2014) diz que um metaobjeto de tempo de compilação é um objeto cujos métodos são executados em compilação. O compilador inclui o metaobjeto ao seu próprio código. Métodos do metaobjeto podem modificar como a compilação é feita. Eles podem fazer conferências adicionais, inserir métodos e variáveis de instância em classes e prototipos, modicar métodos existentes entre outros.

Tratt (TRATT, 2004) diz que a metaprogramação em tempo de compilação permite ao usuário de uma linguagem de programação um mecanismo para interagir com o compilador para permitir a construção de arbitrários fragmentos de programa por código de usuário. Steele (STEELE, 1998) argumenta que "um objetivo principal na concepção de uma linguagem de programação deve ser o de planejar o crescimento", sendo assim a metaprogramação em tempo de compilação é uma mecanismo poderoso para permitir que uma linguagem seja expandida. A metaprogramação de tempo de compilação permite aos usuários, por exemplo, adicionar novos recursos para uma linguagem de programação (SHEARD; BENAISSA; PASALIC, 1999) ou aplicar otimizações específicas de aplicações (SEEFRIED; CHAKRAVARTY; KELLER, 2004).

A metaprogramação de tempo de compilação foi reconhecida como uma ferramenta valiosa para habilitar técnicas de programação como: virtualização de linguagem (sobrecarga/substituição da semântica da linguagem de programação original para permitir a incorporação de linguagens específicas de domínio) (MCCOOL; QIN; POPA, 2002), reificação[1] do programa (fornecendo programas com meios para inspecionar o seu próprio código) (SKALSKI, 2005) (ATTARDI; CISTERNINO, 2001), auto-otimização (aplicação automática de otimizações específicas de domínio com base na reificação do programa) (CROSS; SCHMIDT, 2002), construção de programa algorítmico (geração de código tedioso para escrever com as abstrações suportadas por uma linguagem de programação) (SHEARD; JONES, 2002) (SKALSKI, 2005).

[1] Significado de Reificação: Transformação em coisa. Disponível em: <https://www.dicio.com.br/reificacao/>. Acesso em: 25/10/2017.

Neste capítulo veremos o uso de metaprogramação em diferentes linguagens de programação.

2.1 Nemerle

Nemerle é uma linguagem de programação de propósito geral de alto nível com digitação estática para plataformas baseadas na infra-estrutura de linguagem comum (Mono e .NET) que incorpora recursos dos paradigmas de programação funcional, orientada a objetos e imperativo (YURYEVICH, 2012) (VILINSKI, 2012). Nemerle foi influenciada por `C#` e `LISP`.

Em Nemerle a metaprogramação se dá através de macros. Brabrand e Schwartzbach (BRABRAND; SCHWARTZBACH, 2002) distinguem macros em duas categorias principais: macros que agem no nível sintático e macros que agem no nível léxico. Macros de nível sintático operam diretamente na Árvore de Sintaxe Abstrata (ASA), que estrutura uma representação do programa e as macros que agem no nível léxico são menos poderosas uma vez que elas operam essencialmente em uma `String` de texto.

As macros de Nemerle são semelhantes aos metaobjetos de tempo de compilação ou anotações de outras linguagens (`Active Annotations` de Xtend ou anotações de Java). As macros de Nemerle não são diretamente relacionadas às macros de LISP.

No exemplo abaixo, temos uma definição de uma macro `m` (VILINSKI, 2012):

```
1 macro m() {
2     Nemerle.IO.printf ("compile-time\n");
3     <[ Nemerle.IO.printf ("run-time\n") ]>;
4 }
```

Código 2.1: Macro em Nemerle

E um método o qual chama esta macro:

```
1 module M {
2     public Main() : void {
3         m();
4     }
5 }
```

Código 2.2: Código-fonte em Nemerle

Após compilado o código-fonte do programa seria:

```
module M {
  public Main() : void {
    Nemerle.IO.printf ("run-time\n");
  }
}
```

Código 2.3: Código-fonte resultante em Nemerle

e durante a compilação será impresso na saída a string "`compile-time\n`".

O uso do operador `<[...]>` é usado na construção da Árvore de Sintaxe Abstrata (ASA), ou seja, o operador converte código Nemerle em objeto da ASA.

Podemos fazer a substituição de parâmetros através do uso dos operadores `$(...)` ou `$ID` dentro do operador `<[...]>`. Por exemplo, podemos definir um laço de repetição em Nemerle usando macro:

```
macro for (init, cond, change, body)
{
  <[
    $init;
    def loop () : void {
      if ($cond) { $body; $change; loop() }
      else ()
    };
    loop ()
  ]>
}
```

Código 2.4: Macro for em Nemerle

E podemos usar a macro da seguinte forma:

```
for (mutable i = 0, i < 10, i++, printf ("%d", i))
```

Código 2.5: Chamada da macro em Nemerle

Em compilação este código seria alterado para:

```
mutable i = 0;

def loop() : void {
  if (i < 10) { printf ("%d", i); i++; loop() }
  else()
};
loop()
```

Código 2.6: Código Nemerle resultante

Nemerle permite que criemos nova sintaxe para as macros definidas, por exemplo, poderíamos modificar a sintaxe da `macro for` e deixá-la mais parecida com a sintaxe de `C`:

```
for (mutable i = 0; i < n; --i) {
  sum += i;
  Nemerle.IO.printf ("%d\n", sum);
}
```

Código 2.7: Definição de sintaxe em Nemerle

Para isto devemos definir quais `tokens` e elementos formarão a assinatura da chamada da macro `for`:

```
macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")", body)
```

Código 2.8: Chamada de macro usando sintaxe em Nemerle

O `syntax` permite adicionar a macro `for` uma assinatura. No exemplo acima, adicionamos alguns elementos de forma à tornar a assinatura bem parecida com a sintaxe em C.

Um exemplo com `if` seria:

```
macro @if (cond, e1, e2)
syntax ("if", "(", cond, ")", e1, Optional (";"), "else", e2)
{
  <[
    match ($cond)
    {
      | true => $e1
      | _    => $e2
    }
  ]>
}
```

Código 2.9: Macro para a instruçã if

A chamada da macro seria:

```
if ( a < b ) print("aa"); else print("bb")
```

Código 2.10: Chamada de macro em Nemerle

E em compilação seria substituída por:

```
match a < b {
  | true => print("aa")
  | _ => print("bb")
```

Código 2.11: Código resultante em Nemerle

Vilinski (VILINSKI, 2012) distinguiu três estágios na qual as macros podem operar em elementos de hierarquia de classes. Cada macro pode ser anotada com uma fase, na qual ela deve ser executado:

- a fase `BeforeInheritance` (anterior à herança) é realizado depois de analisar todo o programa e verificar os tipos declarados, mas antes de criar uma relação de subtipagem entre eles. Dá à macro uma liberdade para mudar a hierarquia de herança e operar na construção da Árvore de Sintaxe de classes e membros;
- `BeforeTypedMembers` (anterior à tipagem) é quando a herança de tipos já está definida. As macros ainda podem operar na construção da Árvore de Sintaxe, mas podem utilizar informações relacionadas a subtipagem;
- a fase `WithTypedMembers` (com tipagem) é após a definição dos tipos dos métodos. Neste estágio os métodos já podem ser analisados com o tipo definidos. As macros podem percorrer toda a ASA e fazer conferências, por exemplo em classes, métodos e variáveis de instância. Nesta fase não é possível mais fazer alterações na ASA, consequentemente não é possível fazer alterações no tipo dos elementos nas classes.

No Código abaixo (VILINSKI, 2012), antes da definição da macro é usado a palavra-chave `Nemerle.MacroUsage` e na sequência enviado como parâmetro a fase e em qual tipo de elemento que a macro irá agir.

```
1 [Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers,
2                       Nemerle.MacroTargets.Method)]
3 macro MethodMacro ( ... ) { }
```

Código 2.12: Definição de estágio de macro em Nemerle

2.2 Scala

Scala, como o próprio nome sugere (que significa “linguagem escalável” (ODERSKY; SPOON; VENNERS, 2008)), foi construída desde o início para dar suporte a extensibilidade (BURMAKO; ODERSKY, 2012).

Uma das formas de extensibilidade de Scala é através de metaprogramação, sendo que uma das formas de metaprogramação em Scala ocorre através de Macros (Scala possui diversas outras formas de metaprogramação). As macros à primeira vista são semelhantes a funções, exceto pelo fato de possuírem o prefixo `macro`. Em Scala as macros são chamadas em tempo de compilação, e elas podem manipular diretamente objetos da Árvore de Sintaxe Abstrata (ASA) e não operam diretamente sobre os dados (tempo de execução)(BURMAKO, 2017) (BURMAKO; ODERSKY, 2012).

Segue abaixo uma definição prototípica de macro:

```
1 def m(x: T): R = macro implRef
```

Código 2.13: Macro em Scala

A definição da macro é indicada a palavra chave `macro` seguido por um identificador que se refere a um método de implementação de macro estático.

Se durante a verificação de tipos o compilador encontrar uma aplicação da macro `m(args)`, ele expandirá essa aplicação invocando o método de implementação de macro correspondente com as árvores de sintaxe abstrata das expressões de argumento como argumentos. O resultado da implementação da macro é outra árvore de sintaxe abstrata (BURMAKO, 2017).

Segue exemplo do funcionamento de macro em Scala[2].

O código abaixo faz a definição de uma macro e faz referência a uma definição de macro `Asserts.assertImpl`:

```
1 def assert(cond: Boolean, msg: Any) = macro Asserts.assertImpl
```

Código 2.14: Macro em Scala

A expressão `<[ expr ]>` é a anotação da árvore de sintaxe abstrata que representa a expressão `expr`. Na realidade, as árvores de sintaxe seriam construídas a partir dos tipos em `scala.reflect.api.Trees` e as duas expressões acima seriam assim:

```
1 assertImpl(c)(<[ x < 10 ]>, <[ "limit exceeded" ]>)
```

Código 2.15: Chamada da macro em Scala

onde `c` é um argumento de contexto que contém informações coletadas pelo compilador na chamada e os outros dois argumentos são árvores de sintaxe abstratas que representam as duas expressões `x < 10` e `limit exceeded`.

Uma chamada `assert(x < 10, "limit exceeded")` levaria em tempo de compilação para uma invocação:

```
1 Literal(Constant("limit exceeded"))
2
3 Apply(
4   Select(Ident(TermName("x")), TermName("$less"),
5   List(Literal(Constant(10)))))
```

Código 2.16: Código em Scala

Abaixo segue uma possível implementação da macro `assert`:

[2] Exemplo retirado de: <https://docs.scala-lang.org/overviews/macros/overview.html>

```
1  import scala.reflect.macros.Context
2  import scala.language.experimental.macros
3
4  object Asserts {
5    def raise(msg: Any) = throw new AssertionError(msg)
6    def assertImpl(c: Context)
7      (cond: c.Expr[Boolean], msg: c.Expr[Any]) : c.Expr[Unit] =
8     if (assertionsEnabled)
9        <[ if (!cond) raise(msg) ]>
10       else
11       <[ () ]>
12 }
```

Código 2.17: Macro assert em Scala

Como mostra o exemplo, uma implementação de macro leva várias listas de parâmetros. Primeiro vem um único parâmetro, de tipo `scala.reflect.macros.Context`. Isto é seguido por uma lista de parâmetros que têm os mesmos nomes que os parâmetros de definição de macro. Mas onde o parâmetro de macro original tem tipo `T`, um parâmetro de implementação de macro tem tipo `c.Expr[T]`. `Expr[T]` é um tipo definido em `Context` que gera uma árvore de sintaxe abstrata do tipo `T`. O tipo de resultado da implementação de macro `assertImpl` é novamente um convertida em objeto da árvore de sintaxe, de tipo `c.Expr[Unit]` (BURMAKO, 2017).

2.3 Xtend

O `Xtend` é uma linguagem de programação estaticamente tipada que é traduzida em código-fonte Java compreensível. Sintaticamente e semanticamente, a Xtend tem raízes na linguagem de programação Java, mas de acordo com os criadores a melhora em vários aspectos. Esta linguagem é mais concisa e possui algumas funcionalidades adicionais como sobrecarga de operadores, inferência de tipos[3] e métodos de extensão[4].

Em `Xtend` a metaprogramação se dá através das `Active annotations`, que permitem a participação do desenvolvedor no processo de tradução de código fonte Xtend para código-fonte Java. Isto mostra-se bastante útil quando o programa em Xtend, se não forem usadas as active annotations, contém muitos `Boilerplate Code` (XTEND, 2016).

`Active Annotations` permitem fazer alterações e conferências em 4 (quatro) fases de compilação, descritas a seguir (XTEND, 2016):

[3] Inferência de tipos é a capacidade de detectar o tipo de dados em compilação.
[4] Os métodos de extensão permitem adicionar novos métodos a tipos existentes sem modificá-los.

- fase 1 - `Register Globals`: a primeira fase do compilador cria a Árvore de Sintaxe Abstrata (ASA), permitindo assim a uma `Active Annotation` (ou código associado a uma) criar e registrar novos tipos Xtend nesta fase;

- fase 2 - `Transformation`: na segunda fase, pode-se fazer alterações nas classes Xtend compiladas, assim como ter acesso à ASA da fase anterior. Neste ponto já não é possível mais registrar novos tipos Xtend. Esta fase de compilação acaba sendo a mais útil, já que aqui pode se fazer alteração em todo código Xtend a ser gerado;

- fase 3 - `Validation`: a terceira fase permite participar da análise semântica do código. `Active Annotations` podem fazer a validação durante as duas primeiras fases, mas somente agora tem todas as transformações concluídas, ou seja, a ASA já esta completa e pronta para geração de código;

- fase 4 - `Code Generation`: na quarta e última fase do ciclo de vida do compilador é gerado o código fonte Java e também permite gerar novos arquivos associados, como por exemplo exportar dados para arquivos em XML.

Por exemplo, para a criação de `get` e `set` de variáveis podemos usar a anotação `@Accessors` (XTEND, 2016):

```
1 @Accessors String name
```

Código 2.18: Anotação em Xtend

Será compilado para o código Xtend:

```
1 private String name;
2
3 public String getName() {
4   return this.name;
5 }
6
7 public void setName(final String name) {
8   this.name = name;
9 }
```

Código 2.19: Código Java gerado

Então, por padrão, um getter público e um método de setter público são criados. O `@Accessors` pode ser configurado para dizer que você só quer um ou outro e para alterar a visibilidade. Você também pode usar a anotação no nível de classe para fazer o mesmo em todos os campos.

Aqui está um exemplo (XTEND, 2016) mais complexo, que mostra como funciona:

```
@Accessors class Person {
  String name
  String firstName
  @Accessors(PUBLIC_GETTER, PROTECTED_SETTER) int age
  @Accessors(NONE) String internalField
}
```

Código 2.20: Anotação em Xtend

Que será compilado para código fonte Java:

```
@Accessors public class Person {
  private String name
  private String firstName
  @Accessors(PUBLIC_GETTER, PROTECTED_SETTER) private int age
  @Accessors(NONE) private String internalField

  public String getName() {
    return this.name;
  }

  public void setName(final String name) {
    this.name = name;
  }

  public String getFirstName() {
    return this.firstName;
  }

  public void setFirstName(final String firstName) {
    this.firstName = firstName;
  }

  public int getAge() {
    return this.age;
  }

  protected void setAge(final int age) {
    this.age = age;
  }
}
```

Código 2.21: Código Java gerado

Podemos criar também outras anotações além das `Actives Annotations` nativas de Xtend. Para exemplificar, vamos criar uma anotação por nome `Extract`(XTEND, 2016), a qual irá colocar todos os métodos públicos de uma classe em uma interface que será criada pela anotação. Para isto, colocamos a anotação antes da classe que queremos extrair os seus métodos públicos (XTEND, 2016):

```
1 @Extract
2 class ExtractExample {
3
4     override void myPublicMethod() { }
5
6     protected def void myPrivateMethod() {  }
7 }
```

Código 2.22: Anotação Extract em Xtend

Note que na linha 4 o método público é considerado como já redefinido, uma vez que a anotação o colocará em uma interface podemos considerar como se já existisse esta interface.

Após a compilação de `Xtend`, o código desta classe gerada em Java fica conforme abaixo:

```
1 @Extract
2 @SuppressWarnings("all")
3 public class ExtractExample implements ExtractExampleInterface {
4   @Override
5   public void myPublicMethod() {   }
6
7   protected void myPrivateMethod() {   }
8 }
```

Código 2.23: Código Extract Java gerado

O código da interface gerada em Java fica conforme abaixo:

```
1 @SuppressWarnings("all")
2 public interface ExtractExampleInterface {
3   public abstract void myPublicMethod();
4 }
```

Código 2.24: Interface Java

A seguir vamos detalhar a criação da anotação `Extract` utilizada no exemplo acima (XTEND, 2016):

```
@Target(ElementType.TYPE)
@Active(ExtractProcessor)
annotation Extract {}

class ExtractProcessor extends AbstractClassProcessor {

    override doRegisterGlobals(ClassDeclaration annotatedClass,
       RegisterGlobalsContext context) {
        context.registerInterface(annotatedClass.interfaceName)
    }
    def getInterfaceName(ClassDeclaration annotatedClass) {
        annotatedClass.qualifiedName+"Interface"
    }
    override doTransform(MutableClassDeclaration annotatedClass,
       extension TransformationContext context) {
        val interfaceType = findInterface(annotatedClass.
           interfaceName)
        interfaceType.primarySourceElement = annotatedClass
        // add the interface to the list of implemented
           interfaces
        annotatedClass.implementedInterfaces = annotatedClass.
           implementedInterfaces + #[interfaceType.
           newTypeReference]

        // add the public methods to the interface
        for (method : annotatedClass.declaredMethods) {
            if (method.visibility == Visibility.PUBLIC) {
                interfaceType.addMethod(method.simpleName) [
                    docComment = method.docComment
                    returnType = method.returnType
                    for (p : method.parameters) {
                        addParameter(p.simpleName, p.type)
                    }
                    exceptions = method.exceptions
                    primarySourceElement = method
                ]
            }
        }
    }
}
```

Código 2.25: Classe da anotação

Na linha 1 determinamos em qual contexto que um tipo de anotação é aplicável. Os contextos de declaração em que um tipo de anotação podem ser aplicáveis estão denotados no código-fonte de constantes `enum` de `java.lang.annotation.ElementType`.

Logo em seguida na linha 2 passamos o nome da classe da anotação e na linha 3 passamos o nome da anotação.

Após fazermos a definição da anotação, na linha 5 passamos a construção da classe. Para este caso, a classe deve estender `AbstractClassProcessor`, pois contém as assinaturas dos métodos que deverão ser redefinidos para execução da anotação.

A criação da interface será na fase 1 (`Register Globals`) e na fase 2 (`Transform`). Na primeira fase ocorre a criação da ASA e onde podemos registrar novos tipos XTend, neste caso uma interface. Já na segunda fase, com a interface já registrada, podemos adicionar métodos à interface.

Na fase 1, linha 7, recebemos dois parâmetros, `annotatedClass` do tipo `Class Declaration` e `context` do tipo `RegisterGlobalsContext`. O `annotatedClass` é a classe que está usando a anotação (classe anotada), que no exemplo descrito acima, seria a classe `ExtractExample` e o `context` é a ASA da própria classe anotada que contém métodos para modificação, onde faremos o registro da nossa interface. Na linha 8 fazemos o registro desta interface com o nome retornado pelo método `getInterfaceName` descrito a seguir.

Na linha 11, criamos um método que retorna o nome da classe enviada como parâmetro adicionada de um sufixo "interface". No exemplo acima, este método retorna "`ExtractExampleInterface`". Note que devido a inferência de tipos em `Xtend` não precisamos colocar o tipo de retorno do método e nem explicitar que a linha 12 é um retorno. Caso o programador queira deixar explícito, seria o equivalente ao código abaixo:

```
1 def String getInterfaceName(ClassDeclaration annotatedClass) {
2     return annotatedClass.qualifiedName+"Interface"
3 }
```

Código 2.26: Método getInterfaceName

Na segunda fase de compilação o método da linha 15 é chamado. Este método recebe dois parâmetros, sendo o primeiro `annotatedClass` do tipo `ClassDeclaration` e o segundo `context` do tipo `TransformationContext`. O `annotatedClass` é a classe anotada e o `context` é a representação da ASA e contém métodos para modificação dos tipos contido nela. Por exemplo, podemos adicionar ou modificar método das classes, porém nesta etapa não é possível mais registrar um novo tipo de Xtend na ASA.

Na linha 16, o método `findInterface`, procura na ASA as interfaces que já estão implementadas na classe anotada (`annotatedClass`). Note que, como foi declarado na linha 11 o método `getInterfaceName`, podemos utilizá-lo na classe anotada. A classe

é `ExtractExample` portanto `annotatedClass.interfaceName` retorna `ExtractExample Interface`. Logo o `interfaceType` é a interface gerada por esta anotação.

Em seguida, na linha 19, adicionamos a interface encontrada na ASA a lista atual de interfaces da classe. O #[ ] é utilizado em `Xtend` para representar listas ou `Arrays`.

Na linha 22, percorremos todos os métodos declarados na classe anotada, e para cada método encontrado, verificados na linha 23 se é um método público.

Caso seja um método público, na linha 24 é chamado o método `addMethod` que irá adicionar uma assinatura de método à interface. Este método toma como parâmetro o nome do método e uma função anônima, delimitada por [e]. Quando a função anônima é colocada após uma chamada de método, ela é considerada como o último parâmetro, o que acontece neste exemplo. Nesta função anônima são inicializados os dados da assinatura de método como tipo de retorno, parâmetros, exceções etc. Esta função anônima é chamada dentro do método `addMethod` tendo como contexto um objeto do novo objeto da AST que representa um método.

As variáveis do novo objeto que representa uma assinatura de método que são inicializadas são:

- docComent: comentários existentes no método;
- returnType: tipo de retorno do método;
- parameters: parâmetros existentes (pode haver mais que um por isso percorremos uma lista de parâmetros);
- exceptions: exceções utilizadas no método;

Após o `addMethod` inicializar as variáveis, adicionamos este método à nossa interface (`interfaceType`).

No exemplo acima, a classe `ExtractProcessor` da anotação `Extract`, herdou a classe `AbstractClassProcessor` a qual contém os métodos que são necessários redefinir para processar a anotação.

Para a criar uma anotação deve-se herdar uma destas classes "base", que contém os métodos necessários para cada tipo de anotação, que são listadas abaixo:

- `AbstractClassProcessor`: transformações de classes;
- `AbstractInterfaceProcessor`: transformações de interfaces;
- `AbstractAnnotationTypeProcessor`: transformações de tipos de anotações;
- `AbstractEnumerationTypeProcessor`: transformações de tipos de enumerações;

- `AbstractMethodProcessor`: transformações de métodos;
- `AbstractFieldProcessor`: transformações de variáveis.

Para usar a notação é necessário herdar de uma destas classes abstratas descritas acima. Em seguida a anotação já pode ser colocada antes do objeto a ser modificado.

2.4 Groovy

`Groovy` é uma linguagem dinâmica opcionalmente tipada desenvolvida para a máquina virtual Java (JVM) com muitas características inspiradas em linguagens como `Ruby`, `Python` e `Smalltalk`. Como muitas outras linguagens modernas `Groovy` tem uma abordagem diferente de `XTend` para a geração de código, não gerando código fonte em Java e sim gerando `Java bytecodes`. (KÖNIG; GLOVER, 2007).

A compilação em `Groovy` (GROOVY, 2016) consiste em 9 (nove) fases, sendo elas:

- fase 1 - Inicialização (`Initialization`): leitura do arquivo fonte;
- fase 2 - Análise (`Parsing`): análise léxica e sintática do arquivo fonte gerando os `tokens`[5];
- fase 3 - Conversão (`Conversion`). utilização dos `tokens` gerados na fase anterior para a construção da Árvore de Sintaxe Abstrata (ASA);
- fase 4 - Análise Semântica (`Semantic Analysis`): faz a análise semântica e permite o uso de anotações que façam alterações diretamente na ASA;
- fase 5 - Canonização (`Canonicalization`):[6] permite o uso de anotações que façam alterações diretamente na ASA;
- fase 6 - Seleção de Instrução (`Instruction Selection`). Antigamente era usado para determinar o conjunto de instruções Java a a ser utilizado. Nesta fase pode-se fazer também a conferência de tipos;
- fase 7 - Geração de Classes (`Class Generation`): gera na memória o código fonte em Java bytecode;
- fase 8 - Saída (`Output`): grava a saída para o sistema de arquivos;

[5] `Token` é um caractere ou segmento de texto significativo que pode ser manípulado pelo analisador sintático, por exemplo, o tipo "Int" seria um `token`.

[6] A documentação da linguagem não deixa clara a real diferença entre a quarta e a quinta fase, deduzindo que na quarta fase, o usuário pode fazer alteração na ASA durante a sua criação, ou na quinta fase, alterar a ASA após a sua criação por completa.

- fase 9 - Finalização (`Finalization`): limpa os arquivos que já foram utilizados do código e da memória.

As transformações globais de Groovy podem ser aplicadas em qualquer fase, mas as transformações locais só podem ser agir na fase Análise Semântica (fase 4) ou posterior. Ao chegar nestas fases o compilador ou está gerando ou já possui a ASA, permitindo assim ao programador interagir diretamente, adicionando, removendo ou modificando elementos da ASA.

Segue um exemplo do código fonte onde é feito o uso de uma anotação chamada de `Shout`.

```
1 @Shout
2 def greet() {
3     println "Hello World"
4 }
5
6 greet()
```

Código 2.27: Groovy: Anotação `Shout`

Neste exemplo, a anotação `Shout` (gritar), será responsável em converter para maiúscula as saídas. Portanto, a saída deverá ser "HELLO WORLD".

Na linha 1 é usada a anotação, seguida, na linha 2, pelo método que será transformado por esta anotação.

A definição é dada através de uma anotação (`@interface`) `.java` ou `.groovy`, na qual indicará qual a classe que será responsável por fazer as modificações na ASA:

```
1 @GroovyASTTransformationClass("org.ufscar.transformation.
    ShoutASTTransformation")
2 public @interface Shout { }
```

Código 2.28: Groovy: Interface `Shout`

A linha 1 indica qual classe será responsável pela manipulação da anotação e a linha 2 define o nome da anotação.

Segue abaixo a classe da anotação:

```
1 @GroovyASTTransformation(phase=CompilePhase.SEMANTIC_ANALYSIS)
2 class ShoutASTTransformation implements ASTTransformation {
3
4     @Override
5     void visit(ASTNode[] nodes, SourceUnit sourceUnit) {
6         ClassCodeExpressionTransformer trn = new
              ClassCodeExpressionTransformer() {
```

```
7               private boolean inArgList = false
8               @Override
9               protected SourceUnit getSourceUnit() {
10                  sourceUnit
11              }
12
13              @Override
14              Expression transform(final Expression exp) {
15                  if (exp instanceof ArgumentListExpression) {
16                      inArgList = true
17                  } else if (inArgList &&
18                      exp instanceof ConstantExpression && exp.
                           value instanceof String) {
19                      return new ConstantExpression(exp.value.
                           toUpperCase())
20                  }
21                  def trn = super.transform(exp)
22                  inArgList = false
23                  trn
24              }
25          }
26          trn.visitMethod((MethodNode)nodes[1])
27      }
28  }
```

Código 2.29: Groovy: Classe `ShoutASTTransformation`

Na linha 1 é definida em qual fase de compilação será manipulada esta anotação. Neste exemplo, será na fase 4, durante a Análise Semântica.

Na linha 2, a classe implementa a interface `ASTTransformation` que possui os métodos que deverão ser redefinidos para a modificação da ASA. A ASA será construída na fase 3, mas modificada na fase 4.

Na linha 5 o método `visit` deve ser redefinido, sendo que o mesmo recebe recebe um parâmetro `nodes` que deve ter tamanho 2 e o parâmetro `sourceUnit` do tipo `SourceUnit`. O `sourceUnit` é a ASA completa somente do arquivo que usa a anotação (objeto anotado) e possui alguns métodos para manipulação da ASA, já o `nodes`, recebe um array de `ASTNode`, sendo o primeiro elemento a anotação e o segundo elemento o nó anotado, que no nosso exemplo seriam `Shout` e o objeto da ASA que representa `greet()`.

Na linha 6 criamos um objeto `trn` do tipo `ClassCodeExpressionTransformer`, que será responsável por modificar as expressões contidas no objeto anotado. Criamos uma classe anônima que herda de `ClassCodeExpressionTransformer` com os método

`getSourceUnit` e `transform` que fará a manipulação necessária da ASA.

Na linha 9 redefinimos o método que retorna o `sourceUnit`, que contém a ASA do objeto anotado.

Na linha 14, redefinimos o método `transform` e passamos o parâmetro `exp` que é uma expressão (`Expression`), que será convertido para maiúsculo.

Na linha 15, verificamos se a expressão é do tipo `ArgumentList Expression`, caso seja modificamos uma variável de controle e na linha 21 chamamos o método `transform` de `ClassCodeExpressionTransformer` ou superclasse desta.

Caso o valor esteja em uma lista de expressões e seja uma constante `String`, retornamos um novo objeto do tipo `ConstantExpression` com o valor da constante passada convertida em maiúscula na linha 19.

Na linha 21 chamamos o método `transform` da super classe, passando como parâmetro a `exp` que será convertida para maiúscula.

Na linha 23 retornamos da função anônima o objeto `trn`, da linha 6.

Na linha 26, o `trn` visita o segundo elemento da lista `nodes`, que é objeto anotado, modificando o valor das Strings literais que encontrar. No nosso exemplo, o segundo elemento é o método `greet()`, e o valor do `println` é modificado.

2.5 Lisp

Lisp é uma linguagem funcional dinamicamente tipada, sendo que o código fonte (programa) e os dados em Lisp são representados por `S-Expressions`[7] (que vamos chamar de listas), o que permite que Lisp manipule o seu código fonte assim como manipula outros tipos de dados.

`S-Expressions` são geralmente expressas como notação de prefixo entre parênteses (também conhecido como notação polonesa de Cambridge) desde uma expressão matemática (LEITÃO, 2008) até mesmo qualquer outra estrutura mais elaborada, como por exemplo, macros ou funções, garantindo assim que as expressões não serão ambíguas.

A expressão “7 * 2 + 16 / 4 * 2” com as precedências usuais em Lisp ficaria:

```
1 (+ (* 7 2)
2    (* (/ 16 4) 2))
```

Código 2.30: Programa em Lisp

[7] `S-Expressions` é a representação no código-fonte Lisp de uma lista, mas não é tecnicamente uma lista em si. Essa distinção é a mesma que entre uma seqüência de dígitos decimais e seu valor numérico (REED, 2017).

A definição de uma função chamada fatorial em Lisp é dada abaixo:

```
1 (defun fatorial (n)
2   (do ((i n (- i 1))
3        (resultado 1 (* resultado i)))
4       ((= i 0) resultado)))
```

Código 2.31: Programa em Lisp: Definição de função

A função é definida na linha 1 e recebe um argumento `n` o qual o tipo não é definido e esta função é encerrada na linha 4 com o fechamento dos parênteses.

Os argumentos da instrução `do` são três `(do (condição inicial) (código a ser repetido) (condição final))` e são passados respectivamente nas linhas 2, 3 e 4. Na condição inicial, a variável `i` é criada e inicializada com o valor de `n`, neste caso 5. Ela vai até 0 e recebe o valor `n - 1` ao final de cada passo. Esta função é usada a seguir no Código 2.32.

```
1 (print (fatorial 5))
```

Código 2.32: Programa em Lisp: Chamada da função fatorial

Agora que já vimos um pouco sobre a sintaxe de Lisp, vamos à metaprogramação em tempo de compilação. Uma das formas de metaprogramação em Lisp se dá através de Macros, as quais permitem a modificação do código fonte a ser compilado, podendo personalizar e transformar o Lisp em sua própria linguagem de programação (BARSKI, 2010).

Brabrand e Schwartzbach (BRABRAND; SCHWARTZBACH, 2002) distinguem macros em duas categorias principais: macros que agem no nível sintático e macros que agem no nível léxico. Macros de nível sintático operam diretamente na Árvore de Sintaxe Abstrata (ASA), que estrutura uma representação do programa e as macros que agem no nível léxico são menos poderosas uma vez que elas operam essencialmente em uma `String` de texto.

Por exemplo, se definirmos uma macro por nome `four`,

```
1 (defmacro four () (+ 2 2))
```

Código 2.33: Programa em Lisp: Definição de macro

e usar esta macro em alguma expressão:

```
1 (print (four))
```

Código 2.34: Programa em Lisp: Chamada de macro

equivale a colocarmos `(+ 2 2)` no lugar de `four`, uma vez que a macro é calculada em tempo de compilação.

O ciclo um programa em Lisp pode ser resumido em 4 etapas:

1. Leitura do código fonte Lisp;
2. caso o código tenha uma chamada a uma macro, expande-se em uma nova expressão, e repete-se até que não haja mais macros a serem expandidas;
3. compilação do código Lisp;
4. execução do código compilado.

A seguir um exemplo da criação de uma macro denominada `while` que se assemelha a `while` de C:

```
1 (defmacro while (test &rest body)
2     `(tagbody start
3         (if ,test
4             (progn
5                 ,@body
6                 (go start)))))
```

Código 2.35: Programa em Lisp: Definição de macro `while`

No código acima, na linha 1, é criada a macro `while` e definido que receberá dois argumentos: `test` e `body`. O primeiro argumento a ser passado para a função é o `test` e o `&rest` faz com que os demais argumentos sejam agrupados em uma única lista que é atribuída à variável `body`. O `&rest body` seria equivalente a um "`Object ... body`" em Java.

Na linha 2, o `backquote (‘)` trata aquela linha como dado, não sendo calculado pela macro. Ao manter a linha exatamente como está, após a expansão da macro, será criado um rótulo `start` para ser utilizado pelo comando "`go`" na linha 6.

Na linha 3, o `,test` é substituído pela expressão passada como argumento na primeira linha, sendo que a , (vírgula) indica que ali será utilizado o valor de uma variável. Logo em seguida, na linha 4, `progn` agrupa os comandos que são executados se a expressão do `if` da linha 3 é verdadeira. O `progn` equivale ao `{` e `}` das linguagens baseadas em C.

Na linha 5, o `,@body` é substituído pelo código passado como argumento, sendo que o `@` indica que a lista criada por `&rest` na primeira linha foi agrupada e será expandida em uma ou mais instruções sem parênteses. Se `body` é `(I1 I2 I3)`, `,@body` é `I1 I2 I3`.

Finalmente na linha 6, é colocado o desvio para o comando `start` criado anteriormente.

O código abaixo mostra o uso da macro `while`:

```
1 (setf x 0)
2 (while (< x 10)
3     (print x)
4     (incf x))
```

Código 2.36: Programa em Lisp: Uso da macro `while`

No código acima, na linha 1, inicializamos a variável `x` em 0 e, na linha 2, fazemos a chamada para a macro `while` passando três argumentos, sendo o primeiro a condição para execução do "comando" `while`.

Com isto, em compilação, o código será expandido para:

```
1 (setf x 0)
2 (tagbody start
3     (if (< x 10)
4         (progn
5             (print x)
6             (incf x)
7             (go start))))
```

Código 2.37: Programa em Lisp: Expansão do código da macro `while`

Com a criação da macro `while`, criamos uma estrutura de repetição existente em outras linguagens como, por exemplo, C e Java.. Com isto personalizamos Lisp de forma a atender as necessidades do programador.

2.6 Cyan

Cyan é uma linguagem orientada a objetos baseada em protótipos estaticamente tipada. Protótipos desempenham o papel similar a de classes (GUIMARÃES, 2017, p. 6). Cyan suporta herança única, interfaces, protótipos genéricos, funções anônimas estaticamente tipadas além de um Protocolo de Metaobjetos (MOP). O MOP de Cyan permite controlar o processo de compilação de diferentes formas assim como produzir e alterar código durante a compilação (GUIMARÃES, 2017, p. 6).

Segue abaixo um exemplo de um programa em Cyan:

```
1 package main
2 object Program
3     public func run {
4         Out println: "Hello World";
5     }
6 end
```

Código 2.38: Programa em Cyan: `Hello World`

O programa acima imprime o texto “Hello World” como resultado do programa. A execução de um programa em Cyan inicia sempre por um método `run`.

As subseções a seguir apresentam uma visão geral sobre a linguagem Cyan:

2.6.1 Protótipos, objetos, métodos e variáveis de instância

Cyan declara objetos que desempenham função parecida a de classes em Java. Estes objetos são chamados de protótipos para diferenciar dos objetos criados em tempo de execução (GUIMARÃES, 2017).

Segue abaixo exemplo de criação e uso de protótipo em Cyan:

```
1  package main
2  object Book
3
4      private var String title;
5      private var String author;
6
7      public func init {
8          title = "";
9          author = "";
10     }
11     public func getTitle -> String {
12         ^ self.title
13     }
14     public func setTitle: String title {
15         self.title = title
16     }
17     public func getAuthor -> String {
18         ^ self.author
19     }
20     public func setAuthor: String author {
21         self.author = author
22     }
23     public func getBook -> String {
24         ^"Title: " ++ self.title ++ " - Author: " ++ self.author;
25     }
26
27 end
```

Código 2.39: Protótipo em Cyan: `Book`

O código acima inicia com a declaração do pacote. Pacotes são coleções de protótipos. Variáveis de instância devem ser privadas e devem ser inicializadas ou na declaração ou

em cada método `init` ou `init:`, caso contrário, dará erro em tempo de compilação.

Declaração variáveis de instâncias e métodos são feitos com as palavras-chave `func` e `var` ou `let` respectivamente. Os tipos básicos suportados em Cyan são: `Boolean`, `Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Nil` e `String`. É necessário a declaração do método `init:` que ficará responsável pela inicialização das variáveis de instância. Este método `init` poderá receber parâmetros, neste caso a declaração seria:

```
1     public func init: String title, String author {
2         self.title = title;
3         self.author = author;
4     }
```

Código 2.40: Método em Cyan com parâmetros

Cyan possui um metaobjeto `init` que faz a inicialização de variáveis de instância, sendo assim o código acima poderá ser substituído por:

```
1     @init(title, author)
```

Código 2.41: Metaobjeto `init`

Os parâmetros são passados ao método após os "`:`". O tipo do retorno do método é informado após o "`->`" e o valor de retorno é dado após o "`^`" neste caso. Pode-se usar a palavra-chave `return` também..

Segue abaixo um código em Cyan, com o uso deste protótipo:

```
1  package main
2  object Program
3
4      public func run {
5
6          var book = Book clone;
7          var String title;
8          var String author;
9
10         title = "A culpa é das estrelas";
11         book setTitle: title;
12         author = "John Green";
13         book setAuthor: author;
14         Out println: (book getBook);
15     }
16
17 end
```

Código 2.42: Programa em Cyan: Uso de protótipo

No caso do construtor do protótipo receber argumentos, podemos usar a palavra-chave "`new:`" para instanciar o protótipo:

```
1  package main
2
3  object Program
4
5      public func run {
6          var book = Book new: "A culpa é das estrelas", "John
               Green";
7          Out println: (book getBook);
8      }
9
10 end
```

Código 2.43: Programa em Cyan: Uso da palavra-chave `new`

Todos objetos possuem um método `clone` que retorna um clone do objeto que recebeu a mensagem. Este é o método de criação de objetos mais comum em linguagens baseadas em protótipos. (GUIMARÃES, 2017).

2.6.2 Estruturas de decisão e repetição

A instrução `if` tem sintaxe similar a de outras linguagens como Java e C. No entanto em Cyan não é necessário o uso dos parênteses na expressão a ser avaliada. Espera-se como retorno da expressão um valor lógico `true` ou `false`.

No protótipo `book` abaixo é exemplificado o uso desta instrução:

```
1  package main
2
3  object Book
4
5      private var String title;
6      private var String author;
7      private Int pages;
8
9      @init(title, author, pages)
10
11     public func getTitle -> String {
12         ^ self.title
13     }
14
15     public func getBookInfo -> String {
16         if pages < 100 {
```

```
17              ^ "Title: " ++ self.title ++ " is small";
18          }
19          else { // opcional
20              ^ "Title: " ++ self.title ++ " not is small";
21          }
22      }
23
24      public func getBookRating -> String {
25          if pages < 100 {
26              ^ "The book is small";
27          }
28          else if pages < 300 {
29              ^ "The book is medium";
30          }
31          else {
32              ^ "The book is big";
33          }
34      }
35
36  end
```

Código 2.44: Programa em Cyan: Instrução `if`

O uso da instrução `else` é opcional e pode ser aninhado conforme exemplo no método `getBookRating`.

Podemos usar também a instrução de repetição `while`:

```
1   package main
2
3   object Program
4
5       public func run {
6
7           var Int number = 10;
8           while number >= 1 {
9               Out println: number;
10              --number
11          }
12
13      }
14
15  end
```

Código 2.45: Programa em Cyan: Instrução `while`

Também não é necessário colocar a expressão booleana do `while` entre parênteses.

2.6.3 Protótipos e Métodos abstratos

Protótipos e métodos em Cyan podem ser abstratos. O conceito e funcionamento é similar ao de classes abstratas em Java, C# e C++. Para isto é necessário usar a palavra-chave `abstract` antes do protótipo e dos métodos que serão abstratos. Não é necessário que todos os métodos serem abstratos.

Segue abaixo um exemplo de um protótipo abstrato com método abstrato:

```
1  package main
2
3  public abstract object BookRate
4
5      func init {
6          rate = 1.5;
7      }
8
9      abstract func getPrice -> Double
10
11     func rateValue -> Double {
12         ^ rate;
13     }
14
15     var Double rate;
16
17 end
```

Código 2.46: Programa em Cyan: Protótipo e Método abstrato

2.6.4 Interface e Herança

Cyan permite o uso de interface e herança única. A definição de uma interface é dada abaixo:

```
1 package main
2
3 interface BookData
4     func getTitle -> String
5 end
```

Código 2.47: Programa em Cyan: Interface

A classe abstrata deverá ser herdada e, se houver, seus métodos abstratos deverão ser redefinidos. Abaixo segue o código de da classe `Book` que herda de `BookRate` e implementa a interface `BookData`. O método da interface e os abstratos são redefinidos:

```
1  package main
2  object Book extends BookRate implements BookData
3
4      private var String title
5      private var Double price
6      @init(title, price)
7
8      override
9      func getPrice -> Double {
10         ^ price * rateValue
11     }
12
13     override
14     func getTitle -> String {
15         ^title
16     }
17 end
```

Código 2.48: Programa em Cyan: Herança

É necessário usar `override` antes dos métodos que serão redefinidos. Neste caso o `getPrice` e o `getTitle`.

2.6.5 Protótipos e Métodos `final`

Protótipos declarados como `final` não podem ser herdados. Já os métodos declarados como `final` não poderão ser redefinidos.

```
1  public final object Book
2      ...
3  end
```

Código 2.49: Programa em Cyan: Protótipo Final

```
1  object Book
2      Double rate = 1.5;
3      public final func getRate -> Double {
4          ^ rate
5      }
6  end
```

Código 2.50: Programa em Cyan: Método Final

2.6.6 Union e Sobrecarga de método

Em Cyan temos um tipo chamado de `Union`. Sua sintaxe é `Union<X, Y>` e é considerado supertipo de `X` e `Y`. Cyan permite também a sobrecarga de métodos.

Um exemplo é dado no código abaixo:

```
1  package main
2  object Overloading
3
4      private var Union<String, Int, Double> value;
5      @init(value)
6
7      overload
8      func setData: String s {
9          value = s;
10     }
11
12     func setData: Int x {
13         value = x;
14     }
15
16     func setData: Double y {
17         value = y;
18     }
19
20     func getData -> String {
21         type value
22             case String s {
23                 ^ s ++ " is a String";
24             }
25             case Int x {
26                 ^ x ++ " is a Int";
27             }
28             case Double y {
29                 ^ y ++ " is a Double";
30             }
31     }
32 end
```

Código 2.51: Programa em Cyan: `union` e sobrecarga de métodos

No exemplo acima declaramos uma variável de instância `union<String, Int, Double>` chamada `value`. Esta variável pode receber valores de um destes três tipos:

`String`, `Int` ou `Double`. Para recuperar o valor salvo nesta variável é necessário usar o comando `type`, que está exemplificado no método `getData`.

Neste protótipo também exemplificamos o uso de sobrecarga de operadores do método `setData`. A sobrecarga só ocorre quando a quantidade de parâmetros é igual, caso contrário são considerados como métodos distintos (GUIMARÃES, 2017). É necessário usar a palavra-chave `overload` apenas antes do primeiro método com sobrecarga de operadores.

O uso deste protótipo é dada abaixo:

```
1  package main
2
3  object Program
4
5      public func run {
6          var example = Overloading clone;
7
8          example setData: "Example";
9          Out println: (example getData);
10
11         example setData: 50;
12         Out println: (example getData);
13
14         example setData: 25.4;
15         Out println: (example getData);
16     }
17
18 end
```

Código 2.52: Programa em Cyan: Chamada a método com sobrecarga de operadores

As saídas respectivas serão:

```
Example is a String
50 is a Int
25.4 is a Double
```

2.6.7 Metaobjetos em tempo de compilação

Cyan suporta metaprogramação em tempo de compilação, através de objetos denominados de metaobjetos. Os metaobjetos de Cyan interagem diretamente com o compilador atráves de uma interface denominada de Protocolo de Metaobjetos (*Metaobject Protocol* - MOP). Esta interface MOP é composta de classes e interfaces do compilador Cyan que fornecem o suporte necessário para a implementação dos metaobjetos (GUIMARÃES, 2017).

Ao programar, o programador escreve anotações de metaobjetos. Em compilação, quando compilador encontra estas anotações de metaobjetos, ele carrega a classe Java que corresponde a esta anotação e cria um metaobjeto que representa a anotação na ASA. Para cada anotação de metaobjeto é gerado um metaobjeto na ASA, relação de um para um. Este metaobjeto tem um classe Java correspondente denominada de classe de metaobjeto. Pode ser gerado diversos metaobjetos que correspondem a mesma classe de metaobjeto, relação de um para muitos.

Cyan possuem diferentes tipos de metaobjetos:

1. precedidos da anotação "@" (at), como o `@init` exemplificado anteriormente na Seção 2.6.1;

2. números literais terminados com letras, como em:

```
object Program
    public func run {
        var Int color = 00FFFF_hex;
        Out println: "Cyan color is " ++ color;
    }
end
```

Código 2.53: Programa em Cyan: Números literais

o resultado produzido seria: "Cyan color is 65536";

3. strings literais, como em:

```
object Program
    public func run {
        var String path = n"D:\Cyan\CyanIcon.png";
        var RegExpr expr = r"[A-Z0-9]+";
        Out println: "Path of image is " ++ path;
    }
end
```

Código 2.54: Programa em Cyan: Strings literais

o resultado produzido seria: "Path of image is `D:\Cyan\CyanIcon.png`". O pacote `cyan.lang` define duas sequências `n` e `r`. O primeiro `n` para `Strings` com o caractere escape que não é levado em consideração, como por exemplo no endereço de um arquivo. O segundo para expressões regulares do protótipo `RegExpr` (GUIMARÃES, 2017).

4. macros, como em:

```
object Program
    public func run {
        var Int age = 21;
        assert age > 18;
        Out println: "Is major";
    }
end
```

Código 2.55: Programa em Cyan: Macro `assert`

que em compilação `assert` é traduzido para `if` e será expandido para:

```
object Program
    public func run {
        var Int age = 21;
        if (!(age > 18)) {System exit: 1}
        Out println: "Is major";
    }
end
```

Código 2.56: Programa em Cyan: Macro `assert` expandida

5. Codegs: são metaobjetos de tempo de compilação precedidos de "@" e que são suportados pelo plugin (desenvolvido para esta dissertação). Os Codegs devem implementar a interface `ICodeg`. Os detalhes serão fornecidos na Seção 4.

Para os metaobjetos são necessárias classes Java denominadas classes de metaobjetos que implementam uma das interfaces definidos pelo MOP. Esta classes deverão ser compiladas em `".class"` para JVM[8] e colocados em um diretório específico na biblioteca do compilador de Cyan.

Ao compilar um programa em Cyan é necessário passar ao compilador a localização da biblioteca de Cyan, que contém os arquivos `.class` que serão carregados pelo compilador. Com isto o compilador carrega os métodos dos metaobjetos que serão responsáveis em tratar a anotação e retornar para o compilador o código referente a anotação do metaobjeto encontrado. Ou seja, ao colocar nesta biblioteca o arquivo `.class` desta classe de metaobjeto, a anotação do metaobjeto já estará disponível para ser usado em um arquivo `".cyan"`.

[8] JVM - Java Virtual Machine (Máquina Virtual Java) é uma máquina de computação abstrata. A JVM não conhece a linguagem de programação Java, apenas um formato binário particular, o formato de arquivo `.class`. O arquivo `.class` contém instruções da máquina virtual Java (ou *bytecodes*) e outras informações auxiliares.

2.6.8 Protocolo de metaobjetos – MOP

Tradicionalmente um processo de compilação possui ao menos 4 (quatro) fases: análise léxica, análise sintática, análise semântica e geração de código. Para facilitar o entendimento vamos inicialmente usar este conceito de 4 (quatro) fases. Podemos dizer que o MOP permite a ação dos metaobjetos em praticamente todas as fases, com exceção da análise léxica. A ação na fase de geração de código é restrito às classes de metaobjetos de `cyan.lang`.

O MOP além de fornecer as classes e interfaces, também descreve as interações entre o código-fonte que está sendo compilado, o compilador, o metaprograma e as anotações de metaobjeto. Um programa que herda deste conjunto de classes e interfaces do MOP podemos chamar de metaprograma. O MOP informa ao compilador quais classes do metaprograma devem ser utilizados e em quais pontos do código (GUIMARÃES, 2018).

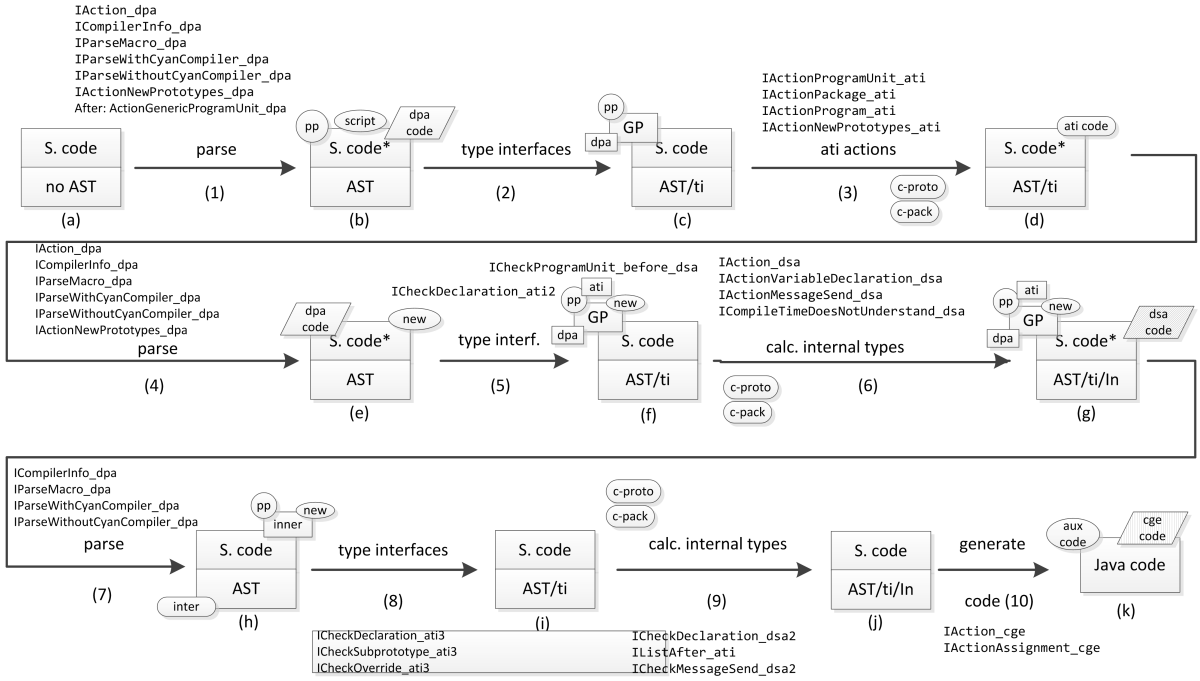


Figura 5: Fases de compilação Cyan (GUIMARÃES, 2017, p. 109).

A Figura 5 mostra as fases utilizadas pelo compilador de Cyan iniciando no canto superior esquerdo e as demais fases são conectadas por uma seta. A compilação em Cyan é dividida em 10 (dez) fases.

Podemos dividir estas 10 (dez) fases em 4 (quatro) etapas. Estas etapas de compilação seriam:

- dpa (During PArsing) - durante a análise sintática (fases 1 e 4);
- ati (After Typing the Interfaces) - nesta etapa (fases 3, 5 e 8) são definidos os tipos das variáveis de instância, parâmetros de métodos, tipo de retorno dos métodos,

protótipos herdados e variáveis implementadas. Os tipos associados a expressões que estão dentro de métodos só são calculados na fase dsa;

- dsa (During Semantic Analysis) - durante a análise semântica (fases 6 e 9);
- cge (Code GEneration) - durante a geração de código Java (fase 10).

O início da compilação é representado pelo retângulo com o conteúdo "`S. code/AST`". Isso significa que no início da compilação temos o código-fonte (`S. code`) do programa e não temos a Árvore de Sintaxe Abstrata (AST na Figura). Cada seta com um número abaixo representa uma fase de compilação e acima desta seta tem uma lista com nome de interfaces que poderão/deverão ser implementadas, caso o metaojeto necessite agir naquela fase.

Durante a primeira fase de compilação são criados objetos da ASA para representar as anotações de metaobjetos que forem sendo encontradas no código-fonte. Cada anotação de metaobjeto tem um objeto da ASA correspondente, numa relação de 1-1 (GUIMARÃES, 2018). Caso algum metaobjeto implemente a interface `IAction_dpa`, o método `dpa_codeToAdd` do metaobjeto é chamado e deverá gerar um código Cyan válido, que será colocado após a anotação do metaobjeto.

É nesta primeira fase que são processados alguns dos metaobjetos citados acima como números literais, Strings literais e macros. O código `var Int color = 00FFFF_hex`, exemplificado anteriormente na Seção 2.6.7, é processado nesta fase. O código da classe `CyanMetaobjectNumberHex` é dado abaixo.

```
1  public class CyanMetaobjectNumberHex extends CyanMetaobjectNumber
       {
2      // code
3      @Override
4      public void dpa_parse(ICompilerAction_dpa compilerAction,
         String code) {
5          int n = 0;
6          String numberStr = code.substring(0,  code.length() - 3);
7          try {
8              n = Integer.valueOf(numberStr, 16);
9          }
10         catch (  NumberFormatException e ) {
11             addError("Number is not in hexadecimal");
12             return ;
13         }
14         this.setInfo(new StringBuffer("" + n));
15     }
16
```

```
17      @Override
18      public StringBuffer dsa_codeToAdd(ICompiler_dsa compiler_dsa)
            {
19          return (StringBuffer ) getInfo();
20      }
21      // code
22  }
```

Código 2.57: Metaobjeto `CyanMetaobjectNumberHex`

O método `dpa_parse` confere se o número está realmente em hexadecimal. No entanto a geração de código no formato da ASA deste metaobjeto, é dada somente durante a etapa `dsa` pelo método `dsa_codeToAdd` do metaobjeto.

Na segunda fase já temos a ASA construída porém faltam informações dos objetos, como o tipo. Na segunda fase é definido os tipo de campo das variáveis de instância, parâmetros do método, tipos do retorno do método e interfaces implementadas de um protótipo (GUIMARÃES, 2018). Nesta fase não há interação dos metaobjetos, conforme mostrado na Tabela 2.

A terceira fase permite aos metaobjetos adicionar variáveis de instância, variáveis compartilhadas e métodos para os protótipos (GUIMARÃES, 2018). O metaobjeto `init` gera um método construtor para o protótipo e inicializa as variáveis de instância. Nesta etapa ele pode verificar se os parâmetros informados para o metaobjeto são variáveis de instância e conferir o tipo. Isto é possível pois já foram atribuídas aos objetos da ASA na fase anterior.

Por exemplo, vamos dizer que um metaobjeto precise agir durante a terceira fase de compilação, neste caso é necessário implementar um daqueles métodos listados na fase 3 da Tabela 2 e na Figura 5, por exemplo podemos usar o `IActionProgramUnit_ati`. Na terceira fase de compilação já temos a ASA com o tipo das interfaces, que é o tipo das variáveis de instância, tipo dos parâmetros dos métodos e retorno. Neste caso o metaobjeto já pode fazer conferências adicionais no código gerado e/ou adicionar métodos e variáveis de instâncias ao protótipos existentes.

Este metaobjeto implementa a interface `IActionProgramUnit_ati` e redefine o método `ati_methodCodeList`, que é responsável pela expansão da anotação do metaobjeto.

O código abaixo do metaobjeto `init`:

```
1   @init(title, author)
```

Código 2.58: Programa em Cyan: Metaobjeto `init`

será “expandido” para:

```
1     public func init: String title, String author {
2         self.title = title;
3         self.author = author;
4     }
```

Código 2.59: Programa em Cyan: Código gerado em compilação

A segunda etapa da análise sintática ocorre na quarta fase de compilação. Nesta fase todo o programa é analisado novamente. As interfaces desta fase são apenas para os novos protótipos que eventualmente tenham sido criados nas fases anteriores (GUIMARÃES, 2018).

A quinta fase de compilação define o tipo dos objetos, assim como a segunda fase, e conforme mostrado na Tabela 2 e na Figura 5, possui duas interfaces com métodos que fazem verificação mas não podem adicionar códigos.

A sexta fase do compilador de Cyan, também faz a análise semântica iniciada na fase anterior e é responsável por atribuir tipos para variáveis e expressões que estão dentro dos métodos (GUIMARÃES, 2018). A maioria dos Codegs implementados nesta dissertação agem durante esta fase, pois nesta fase já temos elementos o suficiente para fazer conferências e adicionar código após a anotação do metaobjeto. Podemos também gerar novos protótipos. Conforme mostrado na Tabela 2 e na Figura 5, a fase 6 é a última fase que permite a criação e modificação de objetos da ASA e geração de código.

Por exemplo, vamos dizer que um metaobjeto precise agir durante a sexta fase de compilação, neste caso é necessário implementar um daqueles métodos listados na fase 6 da Tabela 2 e na Figura 5, por exemplo podemos usar o `IAction_dsa`. Na sexta fase de compilação já temos a ASA praticamente pronta, com o tipo das interfaces. Neste caso o metaobjeto já pode fazer conferências adicionais no código gerado e/ou adicionar métodos e variáveis de instâncias ao protótipos existentes.

Na sétima fase de compilação o MOP disponibiliza apenas interfaces que podem fazer análise sintática de DSL’s acopladas aos metaobjetos, um tópico que está fora do escopo desta dissertação. A oitava e nona fase possuem apenas métodos para verificação da ASA, as interfaces disponíveis dão suporte semelhante às verificações na terceira e na sexta fase respectivamente. A diferença aqui é que já temos a ASA do programa finalizada.

E na décima e última fase de compilação é gerado o código para JVM.

Em relação as interfaces, quando um metaobjeto implementa uma interface ele deve implementar os métodos desta interface que não possuem “corpo”. Esses métodos são chamados pelo compilador sempre que o metaobjeto implementa esta interface. Nas Tabelas 2 e 3 estão relacionados as interfaces que os metaobjetos devem implementar para

Fase	Interface	Método de Compilação
1	IAction_dpa	dpa_codeToAdd
	IParseMacro_dpa	dpa_parseMacro
	IParseWithCyanCompiler_dpa	dpa_parse
	IParseWithoutCyanCompiler_dpa	dpa_parse
	IActionNewPrototypes_dpa	dpa_NewPrototypeList
2	-	-
3	IActionProgramUnit_ati	ati_codeToAdd
		ati_codeToAddToPrototypes
		ati_methodCodeList
		ati_methodCodeListThisPrototype
		ati_beforeMethodCodeList
		ati_instanceVariableList
		ati_instanceVariableListThisPrototype
		ati_NewPrototypeList
		ati_renameMethod
	IActionPackage_ati	ati_CodeToAdd
		ati_methodCodeList
		ati_beforeMethodCodeList
		ati_instanceVariableList
		ati_NewPrototypeList
	IActionProgram_ati	ati_CodeToAdd
		ati_methodCodeList
		ati_beforeMethodCodeList
		ati_instanceVariableList
4	IAction_dpa	dpa_codeToAdd
	IParseMacro_dpa	dpa_parseMacro
	IParseWithCyanCompiler_dpa	dpa_parse
	IParseWithoutCyanCompiler_dpa	dpa_parse
	IActionNewPrototypes_dpa	dpa_NewPrototypeList
5	ICheckProgramUnit_before_dsa	before_dsa_checkProgramUnit
	ICheckDeclaration_ati2	ati2_checkDeclaration
6	IAction_dsa	dsa_codeToAdd
		dsa_NewPrototypeList
	IActionVariableDeclaration_dsa	dsa_codeToAddAfter
	IActionMessageSend_dsa	dsa_checkUnaryMessageSend
		dsa_checkUnaryMessageSendMostSpecific
		dsa_checkSelectorMessageSend
		dsa_checkSelectorMessageSendMostSpecific
	ICompileTimeDoesNotUndestand_dsa	dsa_analyzeReplaceMessageWithSelectors
		dsa_analyzeReplaceUnaryMessage
7	-	-

Tabela 2: Interfaces e Métodos de Compilação – Fases 1 à 7

Fase	Interface	Método de Compilação
8	ICheckProgramUnit_ati3	ati3_checkProgramUnit
	ICheckSubprototype_ati3	ati3_checkSubprototype
	ICheckPackage_ati3	ati3_checkPackage
	ICheckProgram_ati3	ati3_checkProgram
	ICheckDeclaration_ati3	ati3_checkDeclaration
9	ICheckProgramUnit_dsa2	dsa2_checkProgramUnit
	ICheckPackage_dsa2	dsa2_checkPackage
	ICheckProgram_dsa2	dsa2_checkProgram
	ICheckSubprototype_dsa2	dsa2_checkSubprototype
10	IAction_cge	cge_javaCodeClassBody
		cge_javaCodeStaticSection
		cge_javaCodeBeforeClass
		cge_codeToAdd
	IActionAssignment_cge	cge_changeRightHandSideTo

Tabela 3: Interfaces e Métodos de Compilação – Fases 8 à 10

agir em cada uma das 8 fases de compilação e seus respectivos métodos. Os métodos que geram códigos começam com `etapa_codeToAdd`, por exemplo, `dsa_codeToAdd`.

Para os metaobjetos são necessárias classes Java denominadas classes de metaobjetos que implementam uma das interfaces, devem ser compiladas em `".class"`, para JVM[9] e colocados em um diretório específico na biblioteca do compilador de Cyan. Os detalhes de como isto é feito podem ser encontrados em Guimarães (GUIMARÃES, 2017) .

Ao compilar um código em Cyan é necessário passar ao compilador a localização da biblioteca de Cyan, que contém os arquivos `.class` do pacote `cyan.lang` que serão carregados pelo compilador. Quando um programa usa uma anotação de um metaobjeto do pacote `cyan.lang`, o compilador cria um metaobjeto baseado no arquivo `.class`. Este metaobjeto será responsável por fazer conferências e gerar código. Isto pode ser feito com qualquer pacote de Cyan: ao colocar um arquivo `.class` de uma classe de metaobjeto em um diretório específico do pacote, o metaobjeto correspondente estará disponível para todos os que importarem o pacote.

[9] JVM - Java Virtual Machine (Máquina Virtual Java) é uma máquina de computação abstrata. A JVM não conhece a linguagem de programação Java, apenas um formato binário particular, o formato de arquivo `.class`. O arquivo `.class` contém instruções da máquina virtual Java (ou *bytecodes*) e outras informações auxiliares.

3 Programação por Interfaces Gráficas

MYERS (MYERS, 1990) afirma que linguagens de programação são difíceis de aprender e usar exigindo habilidades que muitas pessoas não possuem e portanto devemos encontrar maneiras de tornar a tarefa de programação mais acessível aos usuários. EDWARDS (EDWARDS, 2005) complementa que a programação é extremamente difícil e estende nossas habilidades mentais além de seus limites naturais.

Na área de usabilidade temos dois problemas básicos relacionados a esta dificuldade em programar que são chamados de precipício da execução e precipício da avaliação. O precipício da execução é a dificuldade de traduzir o modelo conceitual em uma ação a ser executada e o precipício da avaliação é a dificuldade de determinar se um estado observável atende aos objetivos desejados. Estes dois precipícios ocorrem quando há uma incompatibilidade entre a representação física e o significado conceitual (EDWARDS, 2005).

EDWARDS (EDWARDS, 2005) diz que um dos principais motivos destes precipícios é que a programação de modo textual é difícil e ao mesmo tempo pobre para representar um programa. A programação pode ser representada em um modelo abstrato de dados e o programador poderia usar uma interface gráfica para manipular diretamente esse modelo: Programação WYSIWYG - "What You See Is What You Get" - que pode ser traduzido como "O que você vê é o que você obtém".

MYERS (MYERS, 1990) propõe que uma abordagem para este problema é o uso de interfaces gráficas o que pode ser denominado de "Programação Visual" ou "Programação Gráfica". Outra abordagem é o uso de gráficos para ilustrar programas previamente escritos de modo a facilitar a compreensão. Isto é denominado "Visualização do Programa", geralmente são usados durante a depuração ou no ensino de programação.

Programação Visual refere-se a qualquer sistema que permite que o usuário especifique um programa através de duas ou mais formas dimensionais. Embora esta seja uma definição muito ampla, linguagens textuais convencionais não são considerados bidimensionais porque os compiladores ou interpretadores processam o código fonte como uma sequência de caracteres que é unidimensional.

Visualização do Programa é um conceito diferente de Programação Visual. Na Programação Visual os gráficos são usados para criar o próprio programa, já na Visualização do Programa, o programa é especificado na forma convencional (textual) e os gráficos são usados apenas para ilustrar algum aspecto do programa ou sua a execução.

O sistema visual humano e o processamento de informações visuais são claramente

otimizados para dados multidimensionais. No entanto programas de computador são convencionalmente apresentados em uma forma textual unidimensional não utilizando este poder. A apresentação bidimensional de um programa como fluxogramas e até mesmo programas estruturados em blocos são úteis para melhor entendimento do programa (MYERS, 1990).

Outra motivação para usar gráficos é que eles tendem a ser uma descrição de alto nível das ações desejadas e, portanto, podem tornar a tarefa de programação mais fácil inclusive para programadores profissionais.

As seções seguintes apresentam algumas ferramentas visuais que geram código.

3.1 Protocolo Metaobjeto de Tempo de Edição - ETMOP

EISENBERG e KICZALES (EISENBERG; KICZALES, 2006) dizem que durante muito tempo, linguagens como Pascal, dialetos de Lisp e `High Performance` Fortran, usaram anotações em metadados para criar programas mais expressivos. Assim como ocorre com protocolos de metaobjetos de tempo de compilação e de tempo de execução, o Protocolo Metaobjeto de Tempo de Edição (ETMOP) permite que as anotações estendam a forma como o código é editado e processado.

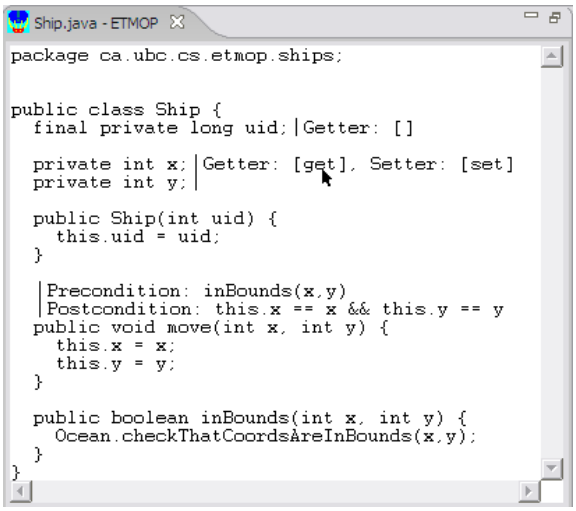


Figura 6: Editor para ETMOP

A Figura 6 apresenta o editor para o ETMOP que foi implementada como um plugin para o Eclipse. O editor do ETMOP possui uma arquitetura de texto simples. Os programas são armazenados como texto simples da maneira tradicional e as anotações são escritas usando a sintaxe Java 5. Ao carregar um arquivo no editor ele é analisado e gerado uma árvore de sintaxe abstrata (ASA) que então pode ser processada de diversas maneiras. Para salvar o arquivo, a ASA é serializada de volta para texto (EISENBERG; KICZALES, 2006).

O código apresentado de forma gráfica pelo editor do ETMOP é mostrado abaixo na forma textual, que é a forma a ser salva em disco.

```
class Ship {
  @Getter ("")
  final private long uid ;

  @CombineRight @Getter("get") @Setter("set")
  private int x;
  @Previous
  private int y;

  public Ship (long uid) {
    this.uid = uid;
  }

  @CombineTop
  @Precondition("inBounds (x, y)")
  @Postcondition("this.x == x && this.y == y")
  public void move(int toX , int toY) {
    x = toX ;
    y = toY ;
  }

  private boolean inBounds ( int x, int x) {
    //.. check that coords are within bounds ..
  }
}
```

Código 3.1: ETMOP - Código serializado para texto

A análise semântica se dá em duas fases. Na primeira fase, o texto é analisado para produzir um ASA para o programa. Nesta ASA, as anotações existentes ficam anexadas às declarações. Comentários são armazenados e podem ser recuperados como `Strings`.

A segunda fase de análise é analisar a ASA para produzir metaobjetos de edição de renderização (REMO's) para cada nó da ASA. Os nós sem anotações recebem uma instância de DefaultREMO. Quando as anotações são encontradas, um registro é consultado para determinar se uma metaclasse REMO foi registrada para essa anotação. Se assim for, uma instância dessa metaclasse é criada para o nó e a própria anotação é sinalizada para indicar que um REMO foi criado para isso.

O ETMOP suporta uma forma restrita de compartilhamento de REMO entre nós da ASA. Múltiplos nós da ASA podem compartilhar um REMO único, desde que

sejam imediatamente vizinhos irmãos. Isso permite que um REMO agrupe a renderização de irmãos e entenda os nós AST quando isso for apropriado. Na Figura 6 é isso que permite uma única linha vertical e a anotação `Getter/Setter` aparecer ao lado de várias declarações de variáveis (EISENBERG; KICZALES, 2006).

A renderização da ASA ocorre em duas etapas. Na primeira etapa é gerado um leiaute lógico, que define como os metaobjetos de renderização (REMO's) serão agrupados e renderizados. A próxima etapa é o leiaute físico que cria os elementos gráficos e exibe na tela.

No exemplo mostrado na Figura 6, o código é apresentado no editor do ETMOP no formato renderizado, permitindo ao programador clicar no método `getter` para editá-lo ou clicar nas outras partes do código para editá-los.

3.2 *Metaprogramming System* - MPS

O `Metaprogramming System` (MPS) desenvolvido pela JETBRAINS (JETBRAINS, 2015) é um editor focado no uso de Linguagens Específicas de Domínio (DSL). A linguagem de programação nativa do MPS é denominada de *BaseLanguage*, que inicialmente era uma cópia de Java, mas atualmente MPS dá suporte à geração de código-fonte nas principais linguagens como Java, C, XML, FHTML, Latex e JavaScript.

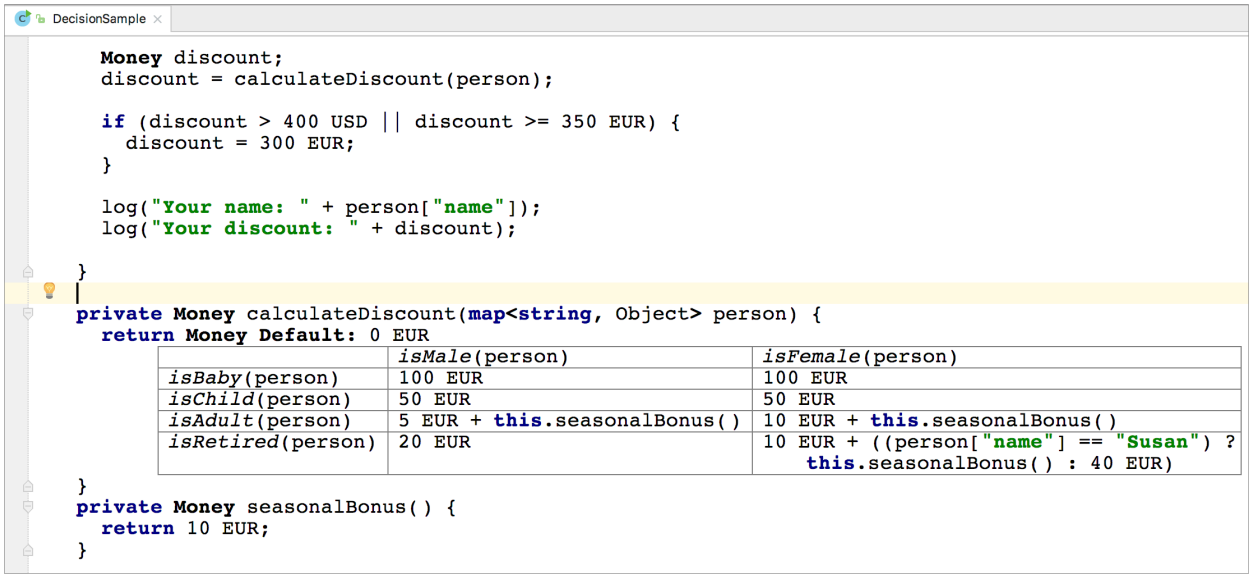
A idéia do MPS é uma apresentação não textual do código-fonte do programa, para isto é mantido o código atualizado na Árvore de Sintaxe Abstrata (ASA), que descreve completamente o código do programa.

A tarefa do editor MPS é então visualizar o ASA de maneira fácil de usar e fornecer meios para uma edição efetiva. Para linguagens textuais clássicas, o editor deve fornecer ao usuário uma interface de forma semelhante a um editor de texto, já para as notações gráficas, o editor deve assumir os hábitos de um editor de diagramação (JETBRAINS, 2015).

O MPS permite a criação de DSL's e, ao cria-lás, podem ser definidas as regras para edição e renderização de código, assim como também pode especificar o tipo de idioma do sistema e as restrições. Isso permite que o MPS controle o código do programa e, assim, torne a programação com a DSL menos propensa a erros (JETBRAINS, 2015).

A figura 7 mostra o uso de uma DSL em MPS, na qual o editor permite que o uso de uma instrução condicional `if` seja definida em um formato de tabela.

A Figura 8 mostra o uso de um "editor de projeção" que permite que o usuário edite a representação do código da ASA em uma terminologia que pessoas em sua área possam entender. Ele pode imitar o comportamento de um editor textual para anotações textuais ou de diagramas para linguagens gráficas, tabular para editar tabelas e assim por



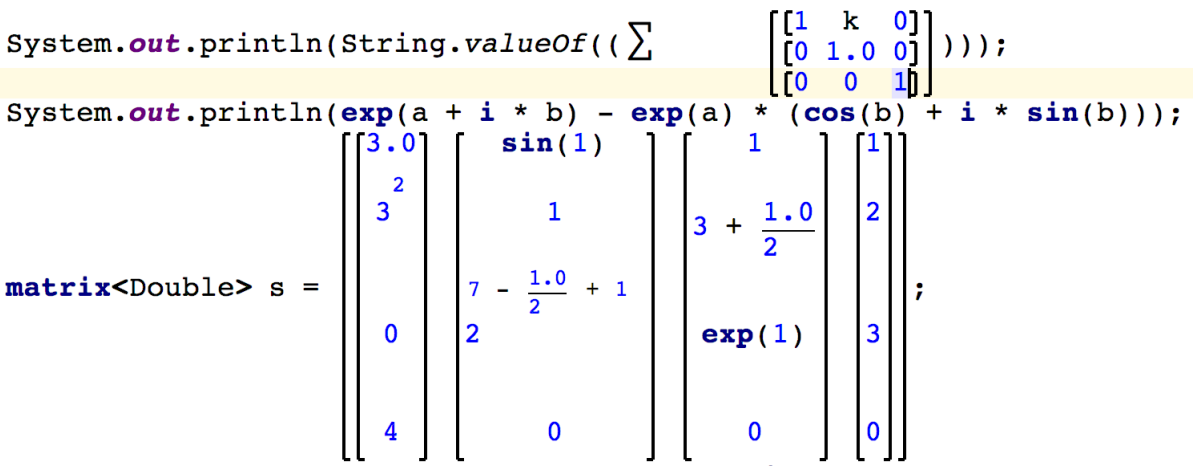
DecisionSample

```
    Money discount;
    discount = calculateDiscount(person);

    if (discount > 400 USD || discount >= 350 EUR) {
      discount = 300 EUR;
    }

    log("Your name: " + person["name"]);
    log("Your discount: " + discount);

  }
  private Money calculateDiscount(map<string, Object> person) {
    return Money Default: 0 EUR
```

	isMale(person)	*isFemale*(person)
isBaby(person)	100 EUR	100 EUR
isChild(person)	50 EUR	50 EUR
isAdult(person)	5 EUR + this.seasonalBonus( )	10 EUR + this.seasonalBonus( )
isRetired(person)	20 EUR	10 EUR + ((person["name"] == "Susan") ? this.seasonalBonus( ) : 40 EUR)

```
  }
  private Money seasonalBonus( ) {
    return 10 EUR;
  }
```

Figura 7: Interface do MPS: Uso de DSL



```
System.out.println(String.valueOf((∑ [[1 k 0] [0 1.0 0] [0 0 1]])));
System.out.println(exp(a + i * b) - exp(a) * (cos(b) + i * sin(b)));
matrix<Double> s = [[3.0] [sin(1)] [1] [1]
                    [3^2] [1] [3 + 1.0/2] [2]
                    [0] [7 - 1.0/2 + 1] [exp(1)] [3]
                    [4] [2] [0] [0]];
```

Figura 8: Interface do MPS: Uso de Editor de Projeção.

diante, tornando a programação mais intuitiva (JETBRAINS, 2015).

MPS usa uma abordagem generativa. O programador pode pode definir geradores para uma linguagem de programação para transformar o código do usuário em código compilável escrito em uma linguagem mais convencional, geralmente de propósito geral (JETBRAINS, 2015).

3.3 Codea

Como exemplo de ferramentas visuais que geram códigos podemos citar o Codea (TWOLIVESLEFT, 2015) que é um Ambiente de Desenvolvimento Integrado (IDE) desenvolvido para o IOs que permite que o programador interaja visualmente com o código, podendo assim escolher visualmente uma cor, conforme demonstrado na Figura 9, e até

mesmo adicionar visualmente uma imagem ou um som ao programa. Após o programador escolher uma cor o Codea gerará o código-fonte equivalente.

Codea foi desenvolvido para a linguagem de programação Lua.[1] O Codea através de uma interface gráfica auxilia a geração de código em tempo de edição, não tendo o poder de um metaobjeto.

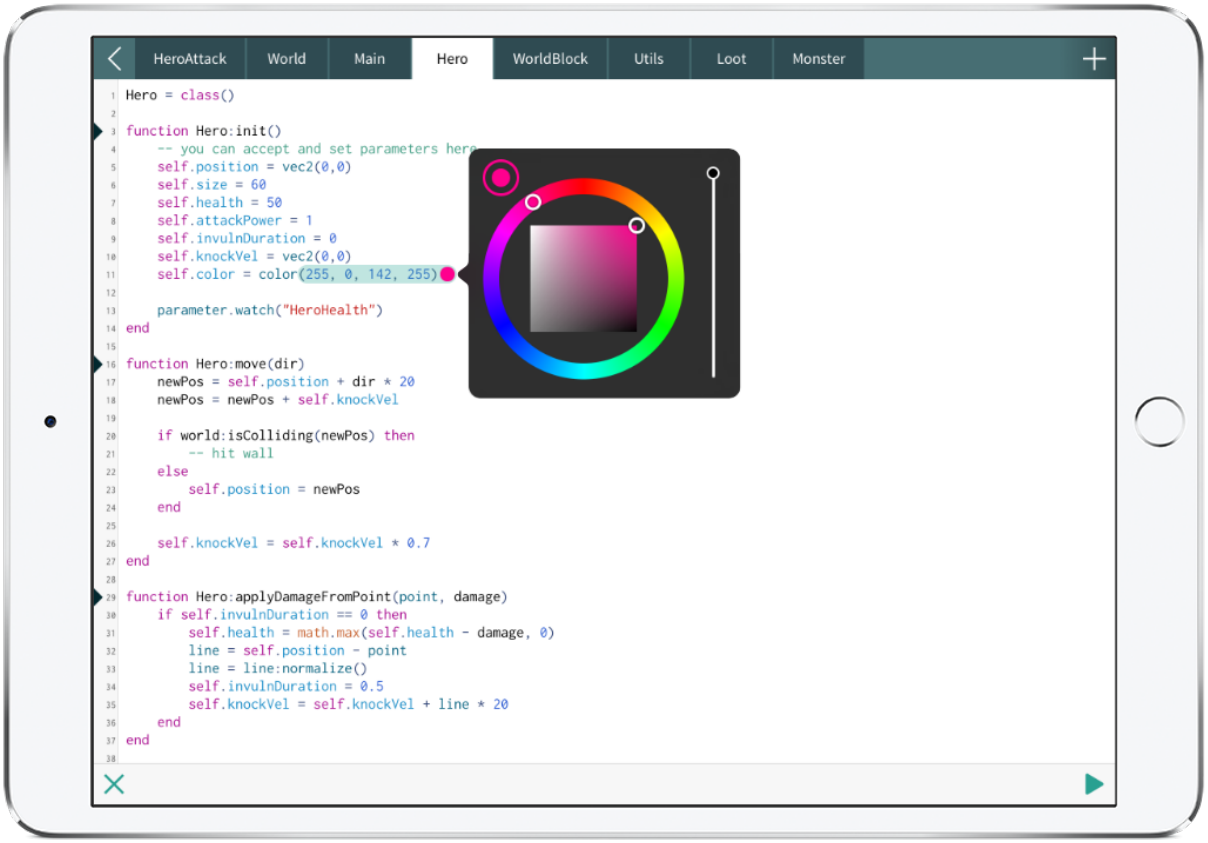


Figura 9: Interface do Codea: Escolha de cor usando interface gráfica.

3.4 Swift

Swift é uma linguagem de programação criada pela Apple para iOS, Mac, Apple TV e Apple Watch.[2] Swift foi criado com a intenção de substituir a programação em `Objective-C` nos produtos da Mac e iOS (GOODWILL; MATLOCK, 2015).

A programação em muitas partes do Swift é parecida com o desenvolvimento em `C` e `Objective-C`. No entanto, Swift fornece suas próprias versões para os tipos fundamentais em `C` e `Objective-C` incluindo `Int` para inteiros, `Double` e `Float` para valores de ponto flutuante, `Bool` para valores booleanos e `String` para dados textuais. Fornece também versões dos três tipos de coleção primários: `Array`, `Set` e `Dictionary` (Apple Inc., 2017b).

[1] Disponível em: <http://www.lua.org/>
[2] Disponível em: <https://www.apple.com/swift/playgrounds/>

Além de tipos familiares, Swift apresenta tipos avançados não encontrados em `Objective-C`, como tuplas. Tuplas permitem que você crie e transmita agrupamentos de valores. Você pode usar uma tupla para retornar vários valores de uma função como um único valor composto (Apple Inc., 2017b).

Apple Inc. (Apple Inc., 2017a) disponibiliza para Swift os ambientes de programação Xcode para MacOs e Swift Playgrounds para o iPad. Tanto o Xcode quanto o Swift Playgrounds, mostrados nas Figuras 10 e 11, possuem uma versão interativa do idioma Swift incorporado diretamente.

Esta versão interativa é denominada de REPL. REPL é acrônimo de *Read-Eval-Print-Loop* que permite que o programador digite uma linha de código e o resultado aparece imediatamente semelhante a um `script`. REPL possibilita a execução do código em tempo de edição, ou seja, o programador não necessita compilar e executar o código para visualizar o resultado (Apple Inc., 2017a).

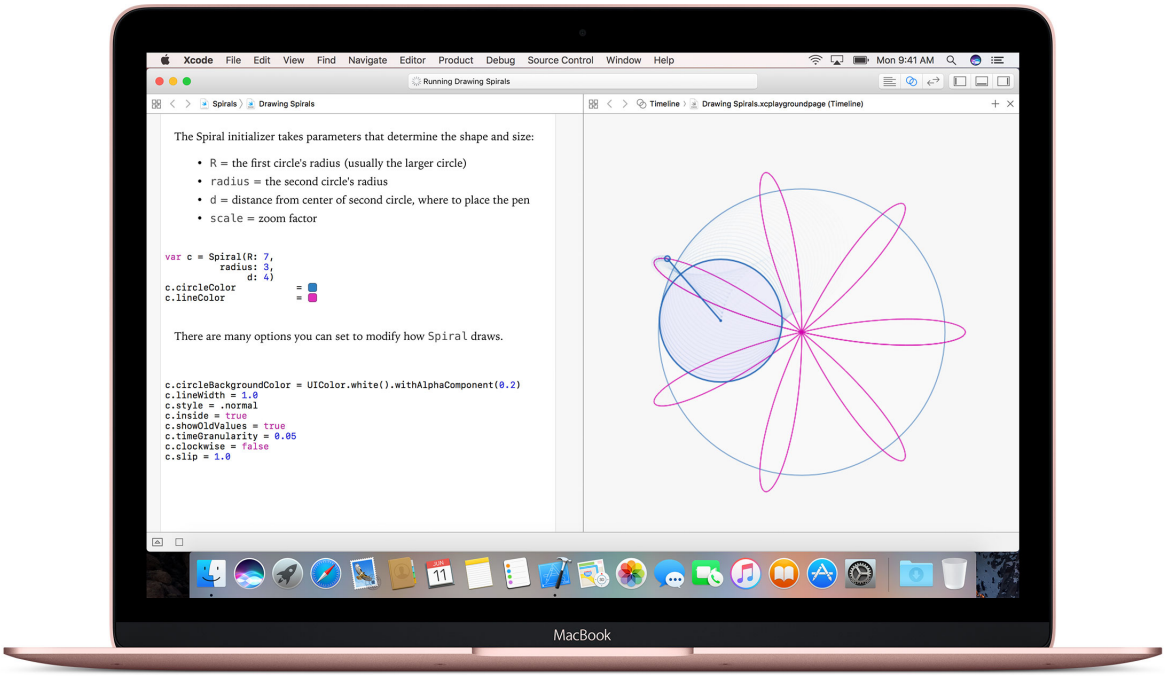


Figura 10: Xcode

3.5 Controladores Lógicos Programáveis - CLP

Os Controladores Lógicos Programáveis (CLP) são equipamentos eletrônicos de última geração utilizados em sistemas de automação flexível. Permitem desenvolver e alterar facilmente a lógica para acionamento das saídas em função das entradas.

Em 1978 a *National Electrical Manufactures Association* (NEMA) determinou a seguinte definição para CLP, denominada NEMA Standard ICS3-1978: “Um equipamento de

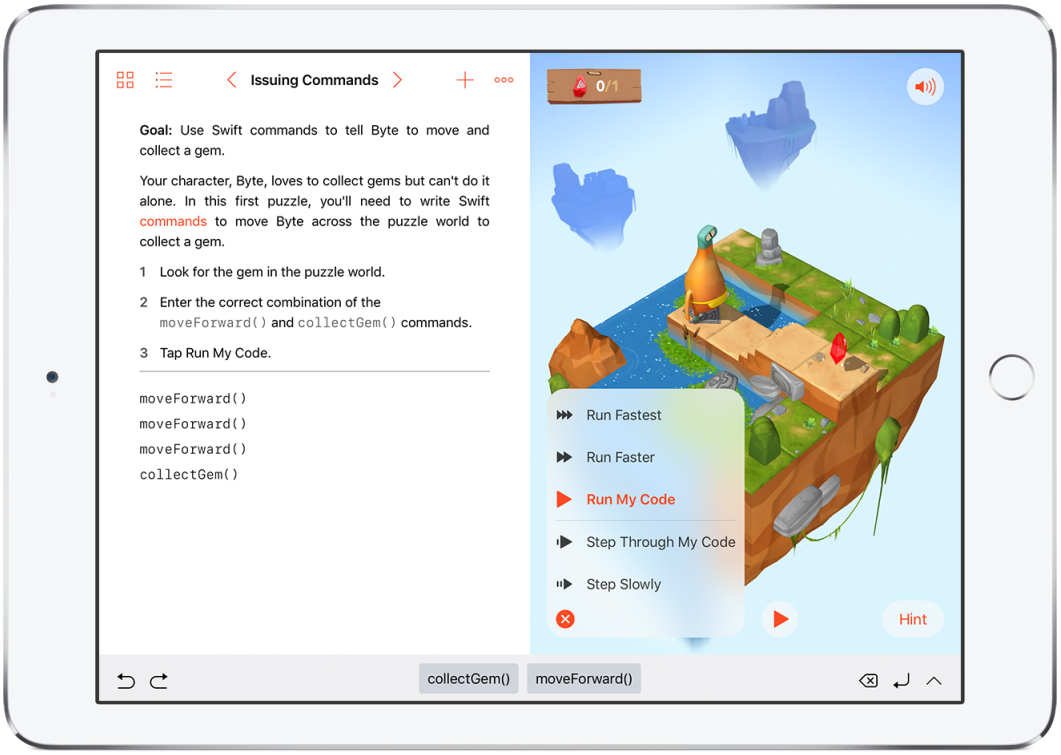


Figura 11: Swift Playgrounds

lógica digital, operando eletronicamente que usa memória programável para armazenamento interno das instruções de implementação específica como lógica, sequencial, temporização, contagem e operações aritméticas, para controle de máquinas e processos industriais com vários modelos de módulos de entradas e saídas digitais e analógicas." (SENAI, 2015).

Conforme mostrado na Figura 12, o CLP é um computador para aplicações específicas que possui uma unidade central de processamento (UCP), memória e interface de entrada e saída.

Os CLP's são usados para automação de sistemas de controle e sistemas de manufatura. Por exemplo, em uma linha de produção que utiliza esteiras metálicas para movimentação de produtos da linha de produção para o setor de expedição, o sistema de controle deve:

- reconhecer que o produto foi colocado na esteira;
- iniciar o transporte;
- verificar se as dimensões estão dentro das especificadas;
- acionar o sistema de alarme caso o produto não esteja em conformidade;
- enviar para expedição caso o produto esteja em conformidade.

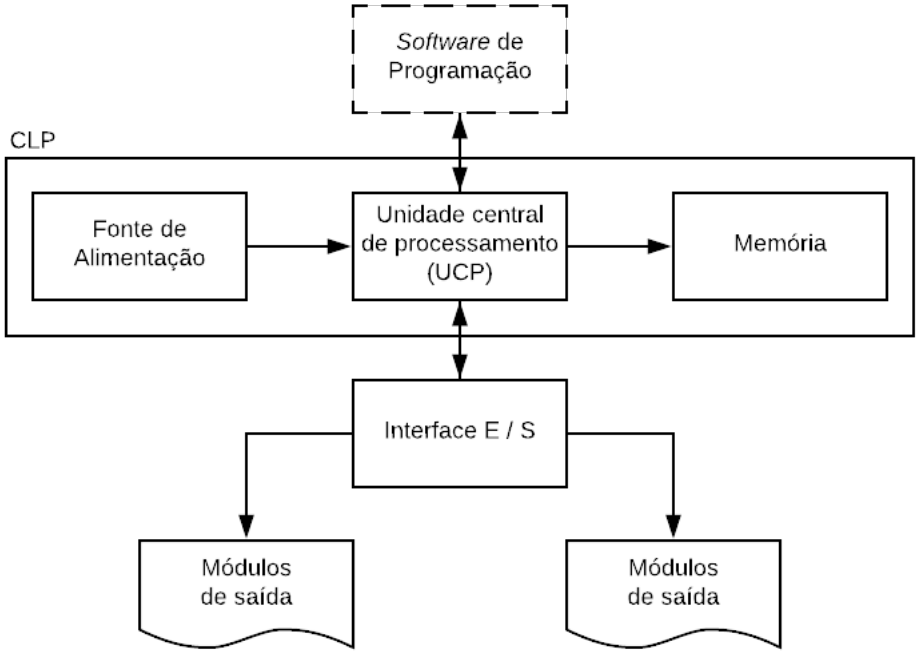


Figura 12: Diagrama de blocos da estrutura básica do CLP (SENAI, 2015)

Para isto, o CLP irá controlar o fluxo descrito acima através de informações recebidas de sensores e do acionamento de equipamentos como: motores, temporizadores e sinalizadores.

O CLP possue processador e memória para armazenamento e execução de programas. Os programas podem ser feitos através de:

- linguagens de programação (C, Pascal, Basic e outras);
- ST - *Structured Text* (texto estruturado);
- IL - *Instruction List* (lista de instruções);
- LAD - *Ladder Diagram* (diagrama de contatos);
- FBD - *Function Block Diagram* (diagrama de blocos de função).

A Figura 13 mostra a comparação entre texto estruturado, lista de instruções, diagrama de blocos de função e diagrama de contatos.

Diagrama de contatos (LAD) e diagrama de blocos de função (FBD) são linguagens de programação que usam símbolos para representar suas funções e são detalhados nas próximas subseções.

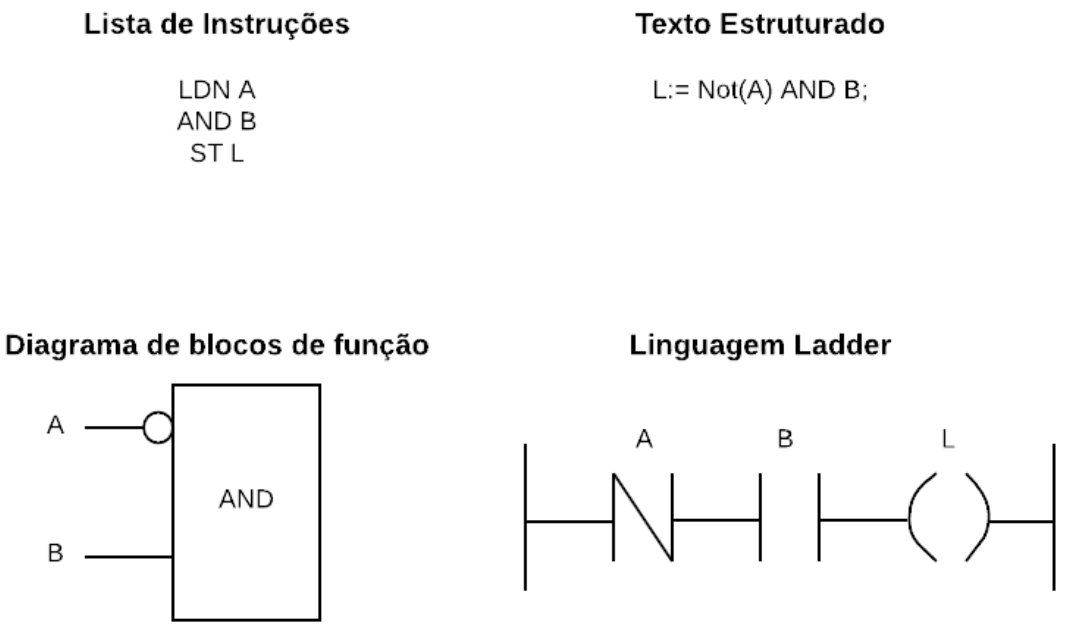


Figura 13: Comparativo de linguagens para programação de CLP (SENAI, 2015).

3.5.1 LAD - Ladder Logic

Ladder logic é uma linguagem de programação gráfica onde a programação é feita combinando diferentes elementos gráficos. Esses elementos gráficos, conforme exemplificado na Figura 13, são chamados de símbolos.

Os símbolos usados em Ladder são parecidos com símbolos elétricos definidos em normas técnicas.[3] A lógica Ladder foi originalmente criada para técnicos, eletricistas e pessoas com conhecimentos na área elétrica que costumam trabalhar com diagramas elétricos (SENAI, 2015).

3.5.2 FBD

Diagrama de Blocos de Função (FBD) é uma linguagem gráfica para programação de CLP baseado na interpretação do comportamento do sistema eletroeletrônico. Este comportamento pode ser descrito em termos do fluxo de sinais entre os elementos do diagrama, caracterizando-se por uma grande semelhança com circuitos eletrônicos digitais (SENAI, 2015).

Conforme mostrado na Figura 13, os elementos da linguagem FBD devem ser interconectados por linhas que representa o fluxo, seguindo a convenção utilizada em

[3] Norma IEC (*International Electrotechnical Commission*) 60617. Disponível em: <http://std.iec.ch/iec60617>

geral para linguagens gráficas na área eletroeletrônica. Nesta convenção as entradas são representadas no lado esquerdo do bloco e as saídas no lado direito.

4 Codegs - Metaobjetos visuais

Um metaobjeto em Cyan é um objeto em tempo de compilação que pode adicionar códigos a um programa e fazer conferências no código-fonte, além das que já são feitas pelo compilador (GUIMARÃES, 2017). Em Cyan os metaobjetos agem somente em tempo de compilação e interagem diretamente com o compilador através de uma interface denominada Protocolo de metaobjetos (*MetaObject Protocol* - MOP) (GUIMARÃES, 2017).

Esta interface MOP é composto de classes e interfaces do compilador Cyan que fornecem o suporte necessário para a implementação dos metaobjetos (GUIMARÃES, 2017). Quando os métodos de um metaobjeto são chamados, eles podem chamar métodos do compilador. O MOP é responsável por definir quais métodos podem ser chamados.

Codeg (code + egg) é um tipo especial de metaobjeto em tempo de compilação de Cyan. O conceito de Codegs surgiu em 1997 intitulado de micro-cases, o qual era a implementação de pequenos programas, que geram trechos de códigos que são incluídos automaticamente em outros programas durante a compilação (GUIMARÃES; BIFFI, 1997).

A classe dos metaobjetos que são Codegs definem métodos que podem ser chamados pelo plugin do IDE em tempo de edição. Ao contrário dos demais metaobjetos de Cyan, os Codegs podem agir tanto em tempo de compilação quanto em tempo de edição. Codegs possuem ações distintas e bem definidas para cada um destes dois tempos.

Em tempo de edição os Codegs fornecem uma interface gráfica ao programador contendo informações pertinentes a função específica daquele metaobjeto. Estas informações são armazenadas em um arquivo dentro do diretório do protótipo que contém a anotação do Codeg. O Codeg pode utilizar a informação atualmente gravada e mostrar uma informação diferente ao usuário ou pode mostrar ao programador esta informação e oferecer recursos onde o programador pode fazer a alteração destas informações. Por exemplo, o Codeg pode guardar um texto informado pelo usuário, ou pode armazenar este texto em um outro arquivo e guardar o caminho deste arquivo.

Em tempo de compilação os Codegs interagem com o compilador podendo criar e/ou modificar protótipos, métodos, variáveis de instância, ou fazer conferências no código-fonte.

Codegs foram implementados em Java e para dar suporte ao seu uso foi criado um plugin, que aqui vamos chamar de `Pludeg`, que fará a integração dos recursos do IDE com o compilador de Cyan. Maiores detalhes da elaboração do `Pludeg` serão dados na senção 4.1.

O `Pludeg` pode ser baixado da página de Cyan, disponível em: <http://cyan-lang.org/>. Para instalar no eclipse basta colocar o arquivo `jar` dentro do diretório `dropins` dentro do caminho de instalação do Eclipse, por exemplo: `C:\Program Files\Elipse Neon\dropins`.

Com o plugin instalado no Eclipse, conforme mostrado na Figura 14, o programador terá disponível um menu com opções para uso com Cyan:

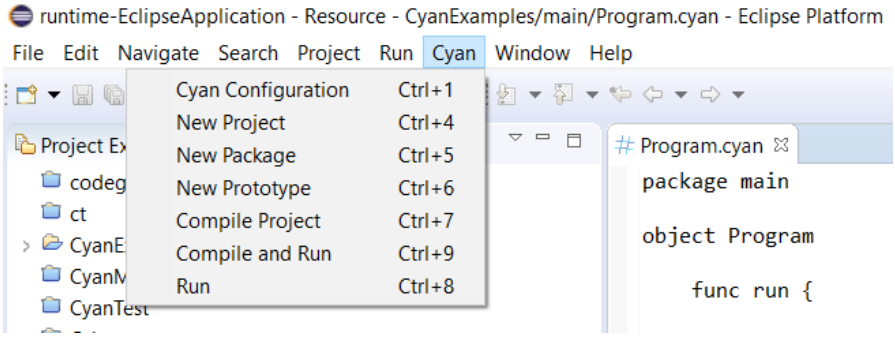


Figura 14: Menu Cyan no Eclipse

A Figura 15 mostra a tela de configuração dos diretórios das bibliotecas de Cyan. Isto é necessário para que o programador possa compilar e executar códigos em Cyan e utilizar os Codegs.

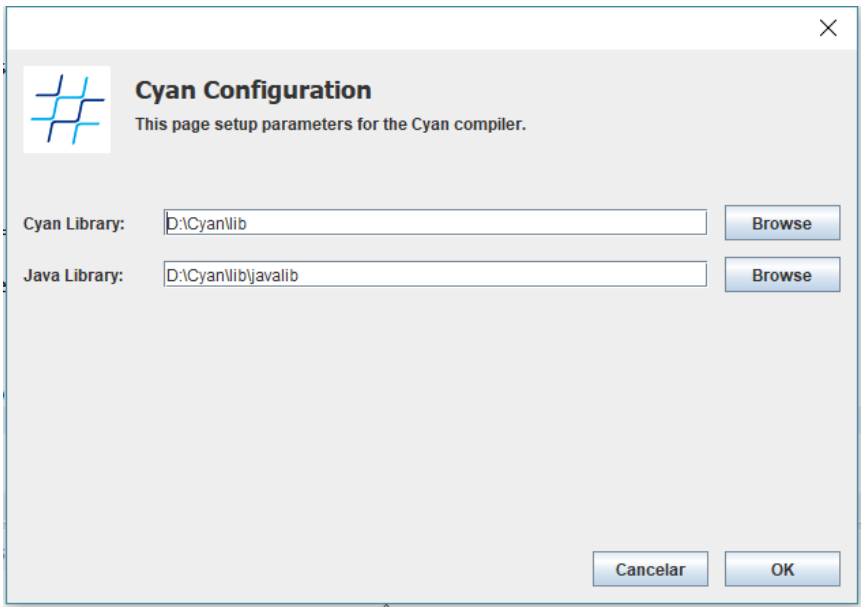


Figura 15: Tela de configuração de Cyan

Com o plugin instalado e configurado, ao criar ou modificar arquivos de Cyan o compilador mantém na memória uma versão atualizada da ASA do programa. Para isto quando o usuário faz a edição do arquivo, sempre quando houver um intervalo superior a 0,5 segundos entre uma tecla e outra pressionada no teclado, o plugin chama o compilador para fazer a análise sintática do arquivo em edição.

Quando o programador passar o mouse em cima de qualquer caracter no arquivo fonte, o plugin comunica com o compilador informando a posição do mouse e o compilador verifica se naquela posição há uma anotação de Codeg. Em caso de ser um Codeg, o plugin chama um dos métodos do compilador (a ser detalhado posteriormente) e este método chama o método do Codeg que abre a interface gráfica para configuração pelo usuário.

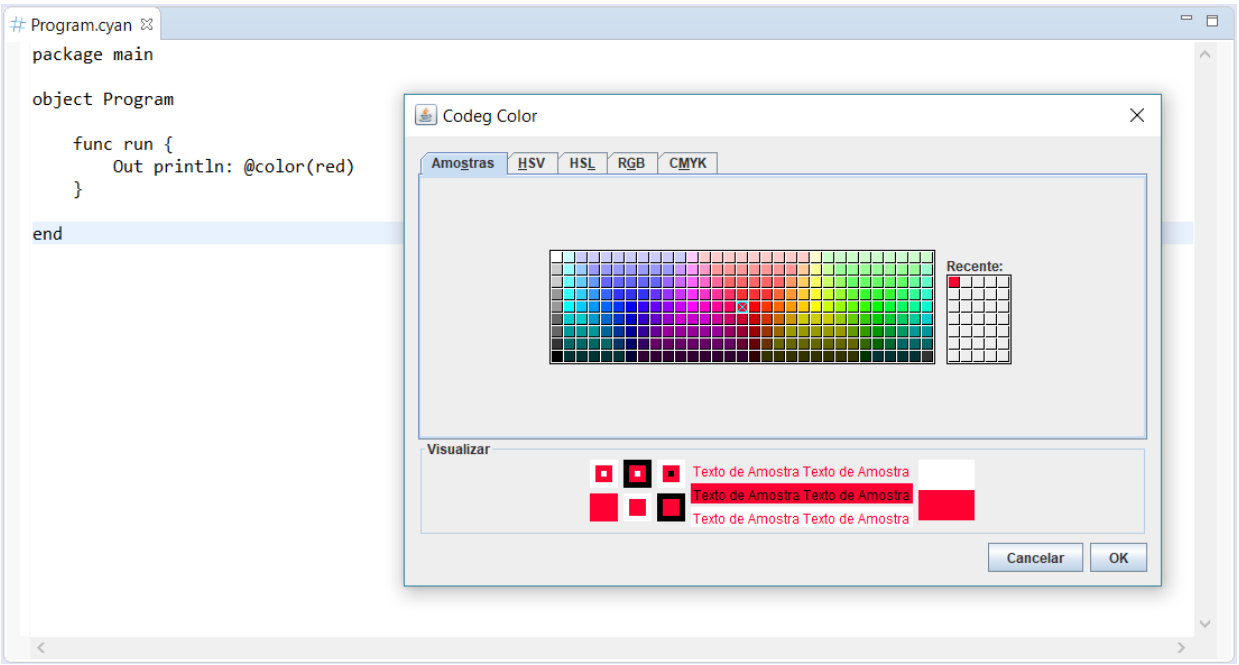


Figura 16: Interface Gráfica Codeg Color

Vamos exemplificar o uso do Codeg `color`. A figura 16 mostra o IDE EClipse com uma janela com um protótipo em Cyan que faz uso da anotação de metaobjeto `@color(red)`. O plugin chama o compilador para fazer a análise sintática do arquivo (e apenas deste arquivo) que estiver sendo editado no momento.

A Figura 17 mostra o fluxo de interação entre o plugin, o compilador e os Codegs. Nesta figura está contida apenas as informações necessárias para o entendimento da integração entre eles. O plugin é controlado pela classe `Activator` que é uma classe `Singleton` a qual contém uma lista de instâncias do compilador. Para cada projeto no `workspace` do Eclipse é necessário uma instância do compilador.

Sempre que o usuário modificar o código no editor do eclipse, o editor irá comunicar com o `Activator` (seta 1) para que seja feita a compilação do código-fonte modificado. Após compilado o `Activator` comunica-se com o compilador (seta 2) e recebe uma lista de erros. Caso tenham, eles serão apontados no editor para o programador.

Este fluxo acima faz com que o compilador mantenha a ASA sempre atualizada. Quando o usuário passa o mouse em cima de um código, o editor chama o método `searchCodeAnnotation` do compilador e, caso o ponteiro do mouse esteja sobre uma

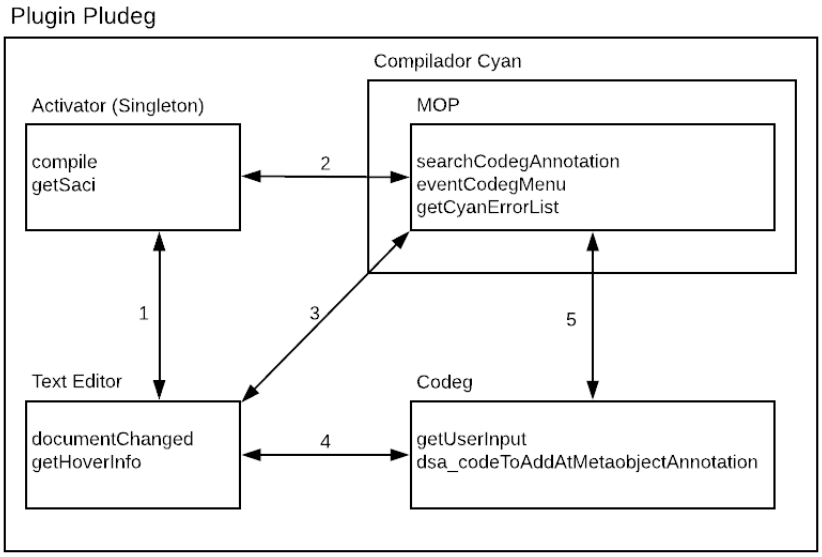


Figura 17: Fluxo de Dados entre o Codeg, Compilador e o Plugin

anotação de Codeg, este método retorna um objeto da ASA que representa o Codeg (seta 3).

Se `searchCodegAnnotation` retornar um objeto (não retornar `null`), o editor (Text Editor na Figura 17) chama o método "static"eventCodegMenu do compilador passando o compilador do projeto corrente (do texto sendo editado) e o objeto da ASA que representa a anotação do Codeg.

O método `eventCodegMenu` chama o método `getUserInput` do metaobjeto (Codeg) associado com o objeto da ASA que representa a anotação de Codeg (seta 5). O método `getUserInput` do Codeg deve usar uma interface gráfica para coletar informações dadas pelo usuário. Os dados coletados nesta iteração com o usuário devem ser retornados pelo método `getUserInput`.

Nos Codegs implementados, o compilador chama, em determinada fase, o método `dsa_codeToAdd MetaobjectAnnotation`, que passará ao compilador o código a ser inserido no código-fonte compilado. O nome do método e a fase podem ser diferentes de acordo com a necessidade do Codeg e consequentemente a interface que ele implementa.

4.1 O plugin Pludeg

O plugin Pludeg é necessário para integrar o compilador ao IDE Eclipse e, assim, acessar os recursos em tempo de edição dos Codegs.

Este plugin oferece suporte a:

- assistente para criação de projetos, pacotes e protótipos em Cyan (Figura 18);
- criação, compilação e execução de projetos em Cyan (Figura 14);
- editor de texto para arquivos de Cyan;
- captura da posição do mouse no editor;
- marcação de linhas com erro apontadas pelo compilador no editor;
- console para avisos e mensagens de erro geradas pelo compilador.

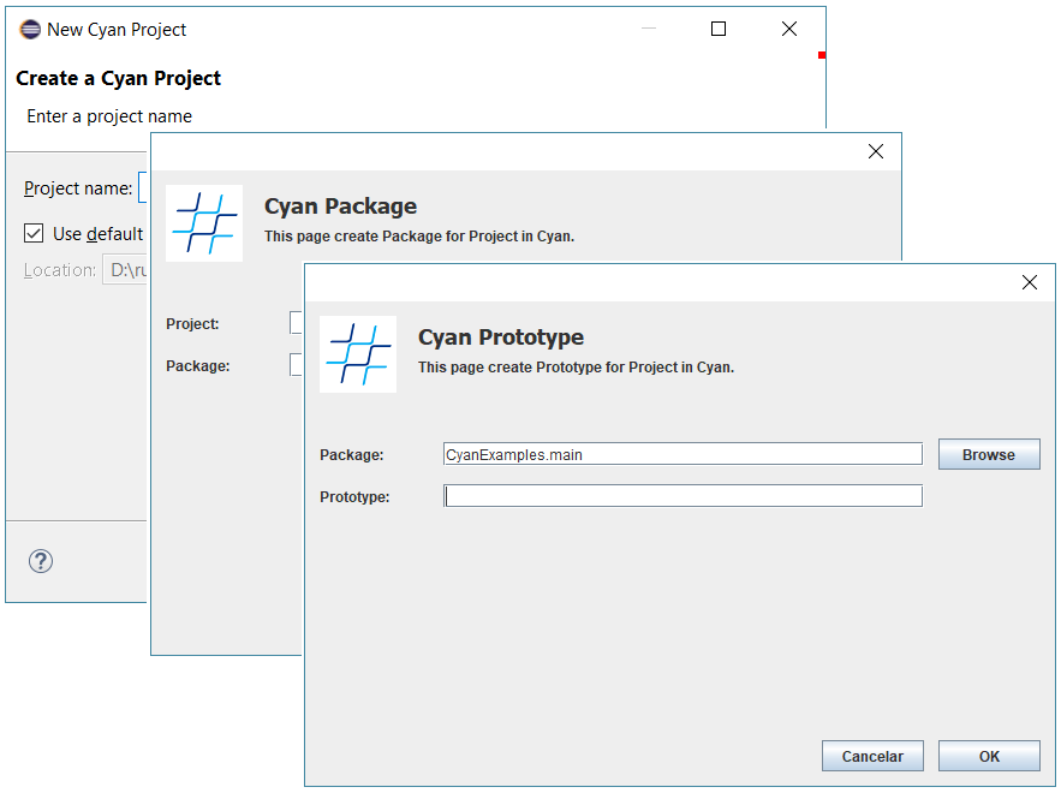


Figura 18: Wizards plugin

O funcionamento e o de fluxo de informações do Pludeg é dado como na Figura 19 abaixo:

1. ao abrir o IDE Eclipse com o *plugin* devidamente instalado, o *plugin* instância um objeto chamado de `Activator`;
2. o `Activator` contém as funcionalidades principais que serão resposnáveis por verificar o intervalo de digitação e chamar o compilador para fazer a análise sintática e consequentemente manter atualizada a ASA do código-fonte do arquivo em edição.;

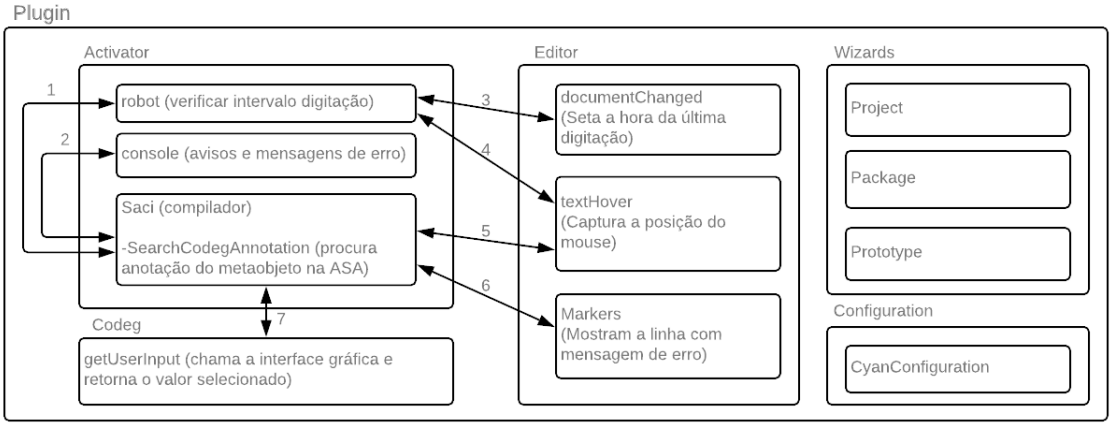


Figura 19: Fluxo de informações do plugin

3. o `Activator` instância os recursos a serem utilizados pela linguagem: o `Console` (Figura 19 seta 2 e Figura 20) para saída de textos e mensagens de erros e o `Markers` (Figura 19 seta 6 e Figura 21) que é responsável por assinalar em qual linha ocorre o erro sinalizado pelo compilador;

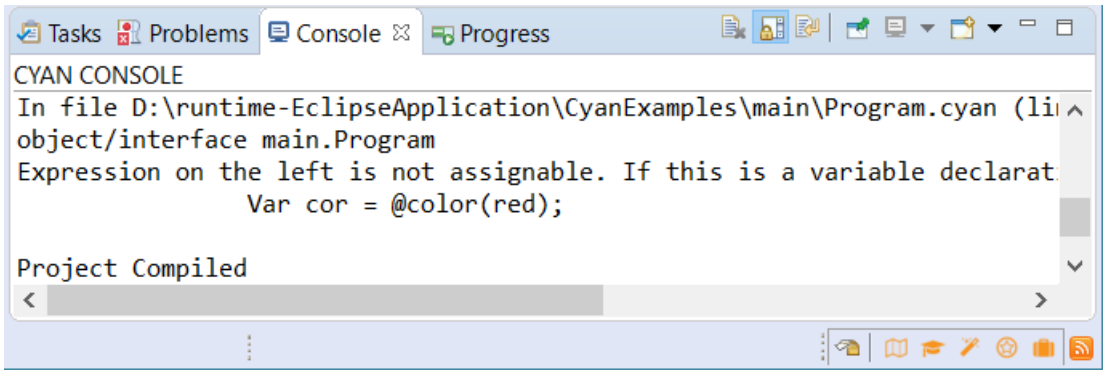


Figura 20: Console plugin

4. nesta classe também mantemos uma lista de instâncias do compilador para cada projeto sendo a relação de um para um, ou seja cada projeto aberto no IDE possui sua própria instância de compilação, permitindo assim editar vários projetos Cyan simultaneamente. Isto é necessário para que a ASA referente ao projeto esteja sempre atualizada.

5. durante a edição de um código fonte em Cyan o editor de texto específico de Cyan (também acoplado ao plugin) informa ao `Activator` as interações (digitação e posição do mouse — Figura 19 setas 3 e 4). Na classe `Activator` é instanciado uma `thread` que fica responsável por checar se o intervalo entre a última digitação e a última compilação for maior do que 0,5 segundos. Se for, a thread chama o método do compilador responsável por fazer a análise sintática do código fonte sendo editado.

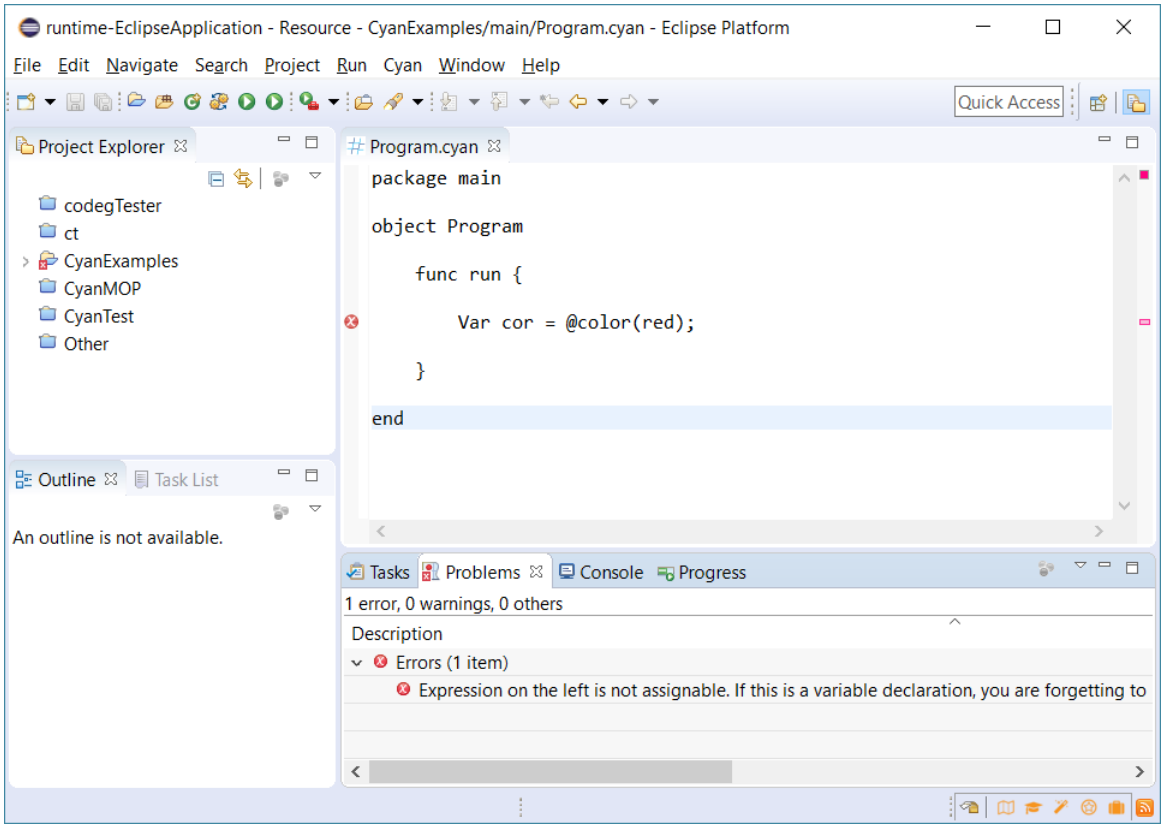


Figura 21: Editor de Cyan

```
1  package cyan_plugin.editors;
2
3  public class CyanTextHover implements ITextHover {
4
5      public String getHoverInfo(ITextViewer tv, IRegion r) {
6          try {
7              int offset = r.getOffset();
8
9              // hidden existing codes ...
10
11             Saci nSaci = Activator.getSaci(
                   projectDirectoryOrName);
12
13             CyanMetaobjectWithAtAnnotation moAnnotation =
                   nSaci.searchCodegAnnotation(offset);
14
15             if (moAnnotation != null) {
16                 nSaci.eventCodegMenu(nSaci, moAnnotation);
17                 Activator.setKeyPressed();
18             }
```

```
19              return null;
20          } catch (Exception e) {
21              //e.printStackTrace();
22              return "";
23          }
24      }
25  }
```

Código 4.1: Pludeg: Classe CyanTextHover

No código 4.2, o nSaci na linha 11 é o objeto que representa o compilador. Na linha 13, o método `searchCodegAnnotation` retorna o objeto da ASA que representa a anotação do metaobjeto.

6. a `thread` inicializada com o `robot`, Figura 19 seta 1 e Código 4.3, verifica se caso o tempo entre as digitações seja superior a 0,5 segundos é chamado o compilador para fazer a análise sintática do código-fonte do arquivo em edição, caso o mouse fique em cima de algum texto é passado a posição para o compilador verificar se se trata da anotação de um Codeg;

```
1  new Thread() {
2
3      @Override
4      public void run() {
5          try {
6              robot = new Robot();
7          } catch (AWTException ex) {
8              ex.printStackTrace();
9              return;
10         }
11         while (true) {
12             robot.delay(INTERVAL);
13             if (lastKeyPressed == null)
14                 lastKeyPressed = new Date();
15
16             long diff = new Date().getTime() - lastKeyPressed
                   .getTime();
17
18             // If the last typing is more than 0.5 sec
19             if (diff >= 500) {
20                 if (lastCompiled == null || lastCompiled.
                       getTime() < lastKeyPressed.getTime()) {
21                     compile(false, false, true);
22                     lastCompiled = new Date();
```

```
                    }
                }
            }
        }
}.start();
```

Código 4.2: Pludeg: Thread da classe Activator

7. caso seja um Codeg o plugin chama o método `eventCodegMenu` (Figura 19 seta 5) do compilador que é responsável por chamar o método do Codeg que abre a interface gráfica;

8. o Codeg é passado como parâmetro ao compilador (Figura 19 seta 7 e código 4.2) e o compilador passa ao Codeg as informações anteriores referentes à anotação deste Codeg que podem estar salvas em disco caso o Codeg já tenha sido configurado anteriormente. O compilador passa a um método `getUserInput` (Figura 19 seta 7) do Codeg as informações que estão em disco, que são referentes ao último uso do Codeg. Então o Codeg pode mostrar estas informações ao usuário;

9. após o usuário escolher as informações desejadas ele pressiona "OK" ou "Cancelar". O Codeg pode enviar as informações ao compilador através do objeto retornado por `getUserInput`. As informações coletadas nesta interação são colocadas em um vetor de bytes que chamaremos de UIB. Este UIB é passado ao compilador de Cyan através do retorno do método `getUserInput` (Figura 19 seta 7). O compilador grava este UIB em disco em um arquivo no diretório do protótipo em edição ou apenas fecha a interface em caso de ser a opção cancelar. Opcionalmente o método `getUserInput` pode optar por salvar o UIB diretamente em um arquivo, enviando este UIB para o método `saveBinaryDataFileToPackage` de um objeto passado como parâmetro a `getUserInput` e que representa uma visão restrita do compilador. Neste caso gravaria em arquivo o nome deste arquivo que foi salvo o UIB.

10. finalmente, ao compilar o programa, o compilador identifica as anotações de metaobjetos dos Codegs e chama o método do Codeg responsável por "transformar" o arquivo UIB. O nome do método muda de acordo com a interface do MOP implementada.

```
@Override
public StringBuffer dsa_codeToAddAtMetaobjectAnnotation(
   ICompiler_dsa compiler_dsa) {
  String codeAdd = new String(this.getMetaobjectAnnotation().
     getCodegInfo(), StandardCharsets.UTF_8);
  return new StringBuffer(codeAdd);
}
```

Código 4.3: Método do Codeg `color` chamado em compilação

No código acima, o método `dsa_codeToAddAtMetaobjectAnnotation` deve retornar código que irá ser acrescentado ao código fonte após a anotação do Codeg. O conteúdo do arquivo UIB produzido durante a edição é recuperado por
`this.getMetaobjectAnnotation().getCodegInfo()`
este valor é utilizado para produzir uma string com uma Cor (o Codeg color produz uma cor).

4.2 Biblioteca de Codegs

Foi criada uma biblioteca de Codegs a fim de testar o plugin e o uso dos Codegs. Futuramente a biblioteca de Codegs poderá ser ampliada. Nas próximas subseções são descritos os Codegs que foram implementados.

Os Codegs devem implementar a interface `ICodeg` que possui a assinatura dos métodos necessários e uma das interfaces existentes que permite ao Codeg agir em uma das fases de compilação. Estas interfaces seriam: `IAction_dpa`, `IActionProgramUnit_ati` ou `IAction_dsa`.

```
1 package meta;
2
3 public interface ICodeg {
4
5     default String getFileInfoExtension() { return "txt"; }
6
7     byte []getUserInput(ICompiler_ded compiler_ded, byte []
        previousCodegFileText);
8
9 }
```

Código 4.4: Interface `ICodeg`

O método `getFileInfoExtension` retorna a extensão do arquivo que mantém as informações armazenadas pelo compilador referentes ao Codeg. Já o método `getUserInput` é chamado sempre que o usuário passa o ponteiro sobre uma anotação de metaobjeto no código-fonte no IDE Eclipse, conforme descrito na Seção 4.1. O parâmetro `previousCodegFileText` é um vetor de bytes com as informações salvas da anotação do Codeg. Se esse método foi chamado anteriormente, o valor retornado por este método foi armazenado em um arquivo. Esse valor agora é passado como parâmetro para esse método, caso contrário `null` é passado como parâmetro.

4.2.1 Batch

O Codeg `batch` permite que o usuário selecione do disco um arquivo `.bat` e/ou digite os comandos que deseja executar no prompt de comandos do DOS. No próprio Codeg é salvo a saída gerada pela execução dos comandos e apresentado ao usuário. Este Codeg gera o nome de um arquivo de saída. Isto é, o código produzido durante a compilação é uma String com o nome do arquivo de saída, que é acrescentado ao código após a anotação do Codeg.

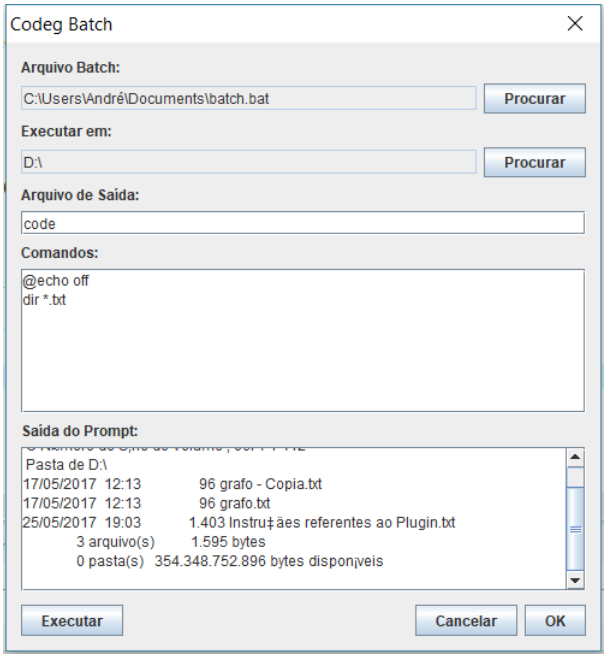


Figura 22: Codeg Batch.

A Figura 22 mostra a Interface Gráfica do Codeg `batch`, na qual podemos selecionar um arquivo `.bat` e ele carregaria o conteúdo deste arquivo no campo "Comandos". O usuário deve definir em qual diretório os comandos serão executados e um arquivo para guardar o resultado da execução dos comandos. Este arquivo é gravado no diretório do protótipo corrente. Embaixo é fornecido ao usuário o resultado da execução dos comandos (a mesma que será salva no arquivo escolhido anteriormente). Esta interface possui um botão adicional para o usuário executar os comandos inúmeras vezes sem precisar sair da interface gráfica.

Este Codeg permite gerar um arquivo para teste do programa que está sendo editado sem a necessidade de sair do IDE e ir até um prompt de comando para executar. O Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

4.2.2 Color

O Codeg `color` permite ao programador selecionar visualmente uma cor. Este Codeg em compilação irá acrescentar após a anotação do Codeg o código da cor selecionado pelo usuário em tempo de edição.

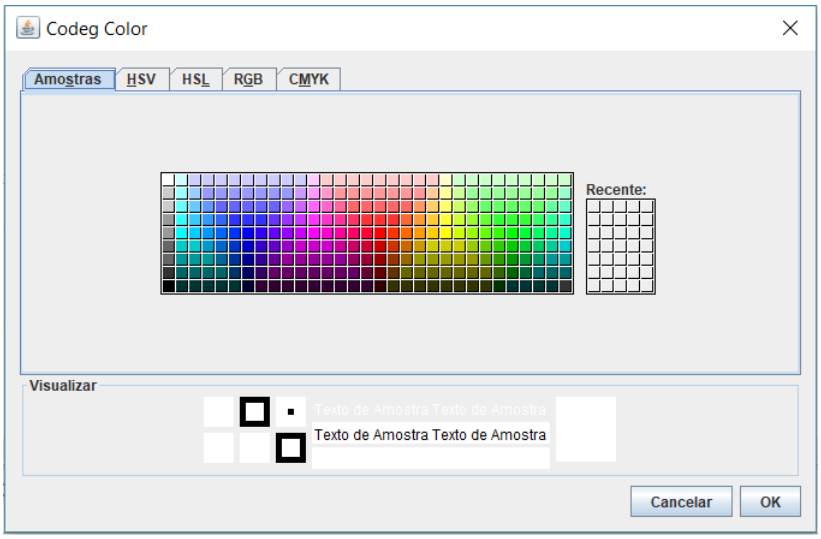


Figura 23: Codeg Color.

A Figura 23 mostra a Interface Gráfica do Codeg `color`, na qual o usuário visualiza a cor selecionada anteriormente, caso exista, e pode alterar a escolha da cor. Este Codeg permite que o programador escolha visualmente uma cor diretamente no editor do IDE. O Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

4.2.3 Command

O Codeg `cmd` permite ao programador executar códigos no prompt de comandos do DOS. Este Codeg não gera código-fonte.

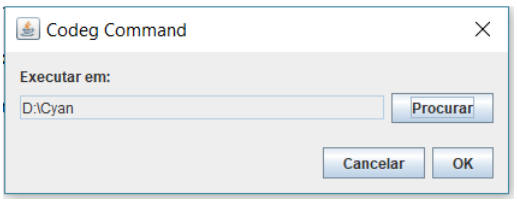


Figura 24: Codeg Command.

A Figura 24 mostra a Interface Gráfica do Codeg `cmd`, na qual o usuário escolhe o diretório onde deseja abrir a janela do prompt de comando. O Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

4.2.4 FoundFile

O Codeg `foundFile` permite ao programador selecionar um arquivo no disco. Em compilação o caminho do arquivo, como `String`, é acrescentado após a anotação do Codeg.

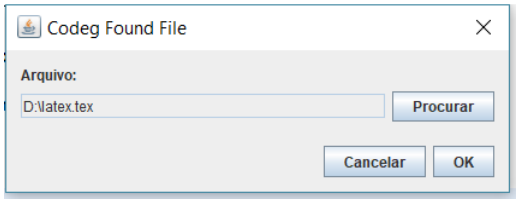


Figura 25: Found File.

A Figura 25 mostra a Interface Gráfica do Codeg `foundFile`, na qual o usuário escolhe o arquivo que deseja adicionar ao código-fonte. Em compilação é acrescentado o caminho do arquivo logo após a anotação do Codeg. O Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

4.2.5 Image

O Codeg `image` permite ao programador selecionar uma imagem no disco. Em compilação o caminho do arquivo da imagem selecionada é acrescentado após a anotação do Codeg.

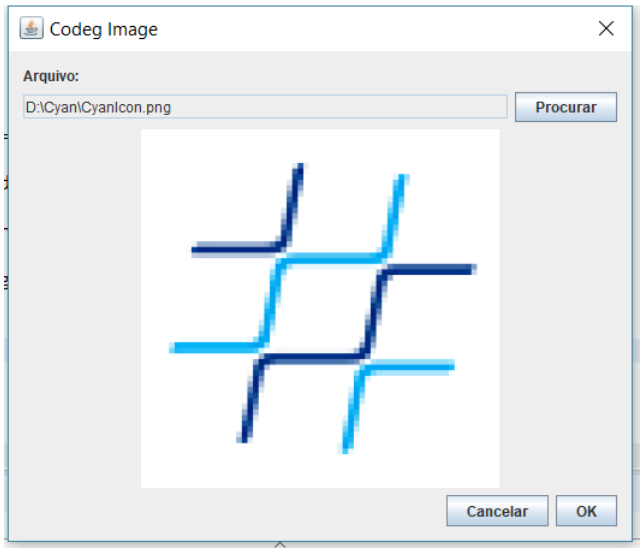


Figura 26: Codeg Image.

A Figura 26 mostra a Interface Gráfica do Codeg `image`, na qual o usuário visualiza a imagem selecionada anteriormente, caso exista, e pode escolher outra imagem para substituir a atual. Este Codeg permite que o programador escolha graficamente uma

imagem diretamente no editor do IDE. O Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

4.2.6 Latex

O Codeg `latex` na qual o programador selecionou um arquivo contendo as expressões em Latex, que são as mesmas exibidas logo abaixo. O programador pode fazer alteração nas expressões que o Codeg irá atualizar o código deste arquivo selecionado. Em seguida é mostrado a expressão compilada em Latex ao usuário. Em compilação a anotação do Codeg é substituído pelas expressões em Latex convertidas para instruções da linguagem Cyan.

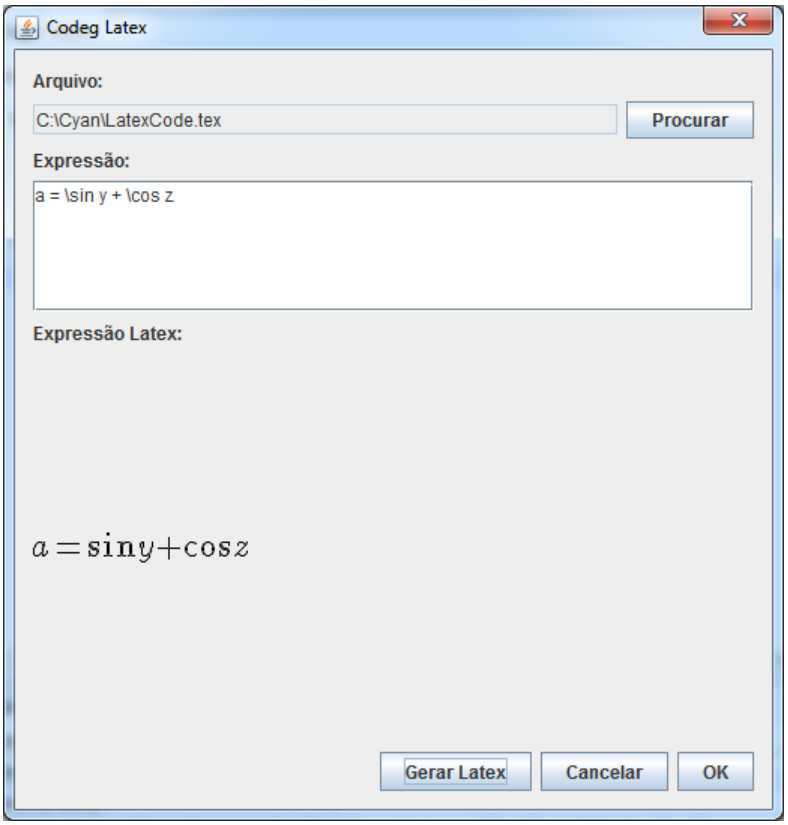


Figura 27: Codeg latex.

A Figura 27 mostra a Interface Gráfica do Codeg `latex`, com um arquivo escolhido pelo programador contendo expressões em Latex e a visualização do código compilado em Latex. Em compilação é adicionado o código gerado por esta Codeg imediatamente após a anotação do Codeg.

Com isto, o código abaixo:

```
1 a = \sin y + \cos z
```

Código 4.5: Instrução em Latex

ao usuário clicar em "Gerar Latex" é mostrado no campo "Expressão Latex": $a = \sin y + \cos z$, e ao compilar o código que será adicionado pelo compilador será:

```
1 a = Math sin: y + Math cos: z
```

Código 4.6: Instrução Latex em Cyan

Este Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

4.2.7 Sound

O Codeg `sound` permite ao usuário gravar uma mensagem de voz que, por exemplo, pode ser usada para explicar determinados trechos de códigos, que poderá ser reproduzida em qualquer momento em tempo de edição. Este Codeg não gera código-fonte.

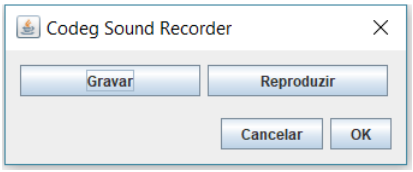


Figura 28: Codeg sound.

A Figura 28 mostra a Interface Gráfica do Codeg `sound`, que pode ser usado para gravar áudios de uma explicação mais detalhada do funcionamento do programa ou áudios de instruções ao programador. O Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

4.2.8 StringText

O Codeg `stringText` permite ao usuário armazenar textos. É bastante útil armazenar no Codeg textos grandes, deixando assim o código-fonte do programa mais "limpo". Em compilação é adicionado após a anotação do Codeg o texto armazenado pelo Codeg como `String`.

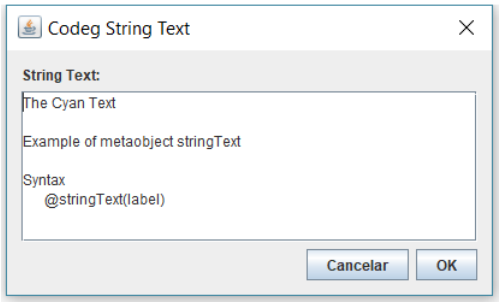


Figura 29: Codeg stringText.

A Figura 29 mostra a Interface Gráfica do Codeg `stringText`. Pode-se escrever códigos em linguagens específicas de domínio (DSL's) para Cyan. No entanto, para que isto seja possível, o código teria que ser interpretado em tempo de execução por Cyan, ou teria que ser feito outro Codeg baseado neste para agir em tempo de compilação. O Codeg implementa a interface `IAction_dsa` podendo agir na sexta fase do ciclo de vida do compilador de Cyan — Figura 5.

5 Conclusão

Nesta dissertação apresentamos o plugin Pludeg que fornece suporte à linguagem de programação Cyan no IDE Eclipse. Este plugin tornou possível o uso de Codegs em Cyan. Apresentamos também uma biblioteca de Codegs que comprovam o funcionamento do plugin de maneira adequada e satisfatória. Foi apresentado também uma análise bibliográfica que possibilitou compreender o uso atual de metaprogramação e o uso de ferramentas visuais para geração de código.

Diante disto foi possível observar que existe uma lacuna no que tange ao uso de metaprogramação junto com ferramentas visuais de geração de código. Atualmente não existe, ou não encontramos durante nossa pesquisa bibliográfica, linguagens que oferecem suporte a este tipo de metaobjeto e nem linguagens que possuam uma definição próxima do conceito de Codegs.

Os Codegs trazem conceitos inovadores tanto no ponto de vista de metaprogramação quanto no de ferramentas visuais, ao combinar o poder dos metaobjetos em tempo de compilação com as facilidades em fazer a definição de parâmetros de forma visual, traz um significativo auxílio ao programador, abstraindo algumas características e detalhes específicos da linguagem para o programador. Ao usar os Codegs, o programador não necessita escrever códigos de baixo nível, como para selecionar um arquivo, uma imagem, e até mesmo um código de cor. Ao mesmo tempo permite que o programador participe do processo de compilação do programa de uma forma simples e prática.

Foi possível perceber que existe uma lacuna em linguagens de programação as quais forçam os programadores a desenvolver programas somente em formas textuais, presos às particularidades das linguagens de programação.

Muitos trabalhos tratam do aperfeiçoamento e melhor uso de metaprogramação para fazer a abstração de dados mas não trazem ao programador facilidades para o uso (JETBRAINS, 2015).

Os Codegs atendem completamente as lacunas existentes na literatura relacionada à integração de programas textuais com ferramentas gráficas. Além disto, os Codegs têm todo o poder de metaobjetos convencionais. Eles podem acessar toda a informação do computador disponibilizada pelo protocolo de metaobjetos (MOP).

As contribuições destas dissertação foram:

1. revisão bibliográfica de metaprogramação em tempo de compilação em algumas linguagens e de ferramentas visuais para geração de código;

2. plugin para dar suporte à linguagem Cyan:
   a) suporte a criação de projetos e arquivos em Cyan;
   b) configuração das opções de compilação;
   c) editor de texto específico para Cyan, na qual é possível interagir com o compilador e identificar as anotações dos Codegs;
   d) `markers` para avisos e mensagens de erro do compilador;
   e) `console` para avisos e mensagens de erro do compilador.
3. suporte ao uso dos Codegs em Cyan e todas as interações entre o editor e a parte do compilador que gerencia Codegs;
4. biblioteca de Codegs descrita na seção 4.2.

Atualmente outros Codegs para tratar expressão regular estão sendo implementados por Eduardo Rocha, que é aluno do BCC do DComp da UFSCar.

Futuramente a biblioteca de Codegs poderá ser ampliada para atender a outras necessidades que não são atendidas pelos Codegs atuais. Por exemplo (GUIMARÃES, 2017, p. 149):

- console interativo para Cyan semelhante ao de linguagens de script. O usuário poderia apenas digitar `@console()` em qualquer local do código fonte de um arquivo de Cyan e executar o código do protótipo corrente;
- Codeg para teste. Este Codeg poderia mostrar uma planilha com valores de expressões capturadas em tempo de execução;
- Codegs que implementariam padrões de projeto. O programador fornece as informações e os Codegs geram o código;
- `perfectHashtable` que geraria uma tabela de hash perfeita, dada uma lista de chaves;
- `keepValues` que permitiria que o valor de uma variável seja salvo em arquivo durante o tempo de execução e depois em tempo de edição o programador visualize estes valores;
- `fsm` que permitiria que se configure de forma gráfica uma máquina de estado finito.
- `turingMachine`: como o nome diz, o usuário poderia definir graficamente uma máquina Turing.

Referências

ADOBE SYSTEMS INCORPORATED. *Adobe Dreamweaver CC*. 2017. Disponível em: <https://www.adobe.com/br/products/dreamweaver.html>. Citado na página 37.

Apple Inc. *Swift - Apple*. 2017. Disponível em: <https://www.apple.com/swift/>. Citado na página 87.

Apple Inc. *The Swift Programming Language (Swift 4.0.3)*. 2017. Disponível em: <https://swift.org/documentation/>. Citado 2 vezes nas páginas 86 e 87.

ATTARDI, G.; CISTERNINO, A. Reflection support by means of template metaprogramming. In: *GCSE*. [S.l.: s.n.], 2001. Citado 2 vezes nas páginas 32 e 45.

BARSKI, C. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* 1st. ed. San Francisco, CA, USA: No Starch Press, 2010. ISBN 1593272812, 9781593272814. Citado na página 62.

BARTLETT, J. *The art of metaprogramming, Part 1: Introduction to metaprogramming*. 2005. Disponível em: <https://www.ibm.com/developerworks/library/l-metaprog1/index.html>. Citado na página 29.

BRABRAND, C.; SCHWARTZBACH, M. I. Growing languages with metamorphic syntax macros. In: *PEPM*. [S.l.: s.n.], 2002. Citado 2 vezes nas páginas 46 e 62.

BURMAKO, E. *Def Macros*. 2017. Disponível em: <http://docs.scala-lang.org/overviews/macros/overview.html>. Citado 3 vezes nas páginas 49, 50 e 51.

BURMAKO, E.; ODERSKY, M. Scala macros, a technical report. In: . [S.l.: s.n.], 2012. Citado na página 49.

CHIBA, S. A metaobject protocol for c++. In: *OOPSLA*. [S.l.: s.n.], 1995. Citado na página 37.

CROSS, J. K.; SCHMIDT, D. C. Meta-programming techniques for distributed real-time and embedded systems. In: *WORDS*. [S.l.: s.n.], 2002. Citado na página 45.

CZARNECKI, K.; EISENECKER, U. W. Generative programming - methods, tools and applications. In: . [S.l.: s.n.], 2000. Citado na página 32.

DAMASEVICIUS, R. On the quantitative estimation of abstraction level increase in metaprograms. *Comput. Sci. Inf. Syst.*, v. 3, p. 53–64, 2006. Citado na página 29.

DAMASEVICIUS, R.; STUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. In: . [S.l.: s.n.], 2008. Citado 8 vezes nas páginas 19, 29, 30, 31, 32, 33, 34 e 36.

DIJKSTRA, E. W. Selected writings on computing: A personal perspective. In: *Texts and Monographs in Computer Science*. [S.l.: s.n.], 1982. Citado na página 34.

DMITRIEV, S.; WATSON, D. Language-oriented programming: The next programming paradigm. In: . [s.n.], 2004. Disponível em: <http://www.onboard.jetbrains.com/is1/articles/04/10/lop/>. Citado 3 vezes nas páginas 39, 40 e 42.

ECLIPSE FOUNDATION. *Eclipse Neon*. 2017. Disponível em: <http://eclipse.org/downloads/>. Citado na página 40.

EDWARDS, J. Subtext: uncovering the simplicity of programming. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005. (OOPSLA '05), p. 505–518. ISBN 1-59593-031-0. Disponível em: <http://dx.doi.org/10.1145/1094811.1094851>. Citado na página 81.

EISENBERG, A. D.; KICZALES, G. A simple edit-time metaobject protocol: Controlling the display of metadata in programs. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 696–697. ISBN 1-59593-491-X. Disponível em: <http://doi.acm.org/10.1145/1176617.1176679>. Citado 2 vezes nas páginas 82 e 84.

GOLDBERG, A.; ROBSON, D. Smalltalk-80: The language and its implementation. In: . [S.l.: s.n.], 1983. Citado na página 37.

GOODWILL, J.; MATLOCK, W. The swift programming language. In: *Beginning Swift Games Development for iOS*. [S.l.]: Springer, 2015. p. 219–244. Citado na página 86.

GROOVY. *Runtime and compile-time metaprogramming*. 2016. Groovy. Disponível em: <http://groovy-lang.org/metaprogramming.html#developing-ast-xforms>. Acesso em: 05 jun. 2016. Citado 2 vezes nas páginas 45 e 58.

GUIMARÃES, J. de O. Reflection for statically typed languages. In: *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*. [s.n.], 1998. p. 440–461. Disponível em: <https://doi.org/10.1007/BFb0054103>. Citado na página 37.

GUIMARÃES, J. de O. *Programming Languages Paradigms*. [S.l.], 2014. Citado 4 vezes nas páginas 29, 35, 37 e 45.

GUIMARÃES, J. de O. *The Cyan Language*. [S.l.], 2017. Acesso em 20/08/2017. Disponível em: <http://cyan-lang.org/learn/>. Citado 12 vezes nas páginas 17, 31, 37, 64, 65, 67, 72, 73, 75, 80, 93 e 110.

GUIMARÃES, J. de O. The cyan language metaobject protocol. In: *ECOOP (submetido)*. [s.n.], 2018. Disponível em: <http://cyan-lang.org/wp-content/uploads/2018/01/The-Cyan-MOP.pdf>. Citado 5 vezes nas páginas 45, 75, 76, 77 e 78.

GUIMARÃES, J. de O.; BIFFI, D. de O. *Implementação de micro-cases*. 1997. Disponível em: <http://bv.fapesp.br/pt/bolsas/51626/implementacao-de-micro-cases/>. Citado na página 93.

JACKSON, D. *Software Abstractions: Logic, Language, and Analysis*. [S.l.]: The MIT Press, 2006. ISBN 0262101149. Citado na página 32.

JETBRAINS. *JetBrains: Meta Programming System*. [S.l.], 2015. Acesso em: 11/11/2015. Disponível em: <https://www.jetbrains.com/mps/>. Citado 3 vezes nas páginas 84, 85 e 109.

KÖNIG, D.; GLOVER, A. *Groovy in Action*. Manning, 2007. (Manning Pubs Co Series). ISBN 9781932394849. Disponível em: <https://books.google.com.br/books?id=T6lQAAAAMAAJ>. Citado na página 58.

LEBLANG, D. B.; CHASE, R. P. Computer-aided software engineering in a distributed workstation environment. In: *Software Development Environments*. [S.l.: s.n.], 1984. Citado na página 38.

LEITÃO, A. M. From lisp s-expressions to java source code. *Comput. Sci. Inf. Syst.*, v. 5, p. 19–38, 2008. Citado na página 61.

MAES, P. Concepts and experiments in computational reflection. In: *OOPSLA*. [S.l.: s.n.], 1987. Citado na página 34.

MCCOOL, M. D.; QIN, Z.; POPA, T. Shader metaprogramming. In: *HWWS '02*. [S.l.: s.n.], 2002. Citado na página 45.

MYERS, B. A. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, Academic Press, Inc., Orlando, FL, USA, v. 1, n. 1, p. 97–123, mar 1990. ISSN 1045-926X. Disponível em: <http://dx.doi.org/10.1016/S1045-926X(05)80036-9>. Citado 2 vezes nas páginas 81 e 82.

NETBEANS. *NetBeans IDE*. 2017. Disponível em: <https://netbeans.org/downloads/>. Citado na página 37.

ODERSKY, M.; SPOON, L.; VENNERS, B. *Programming in scala*. [S.l.]: Artima Inc, 2008. Citado na página 49.

ORACLE. *Annotations*. 2017. Disponível em: <https://docs.oracle.com/javase/tutorial/java/annotations/>. Citado na página 36.

REED, M. *Lisp-list-vs-s-expression*. 2017. Disponível em: <https://stackoverflow.com/questions/10771107/lisp-list-vs-s-expression>. Citado na página 61.

SEEFRIED, S.; CHAKRAVARTY, M. M. T.; KELLER, G. Optimising embedded dsls using template haskell. In: *GPCE*. [S.l.: s.n.], 2004. Citado na página 45.

SENAI. *Controle Lógico Programável*. SENAI-SP Editora, 2015. (Automação). ISBN 9788583933601. Disponível em: <https://www.senaispeditora.com.br/catalogo/senai-sp-educacao-automacao/controle-logico-programavel/>. Citado 4 vezes nas páginas 17, 88, 89 e 90.

SHEARD, T.; BENAISSA, Z.-E.-A.; PASALIC, E. Dsl implementation using staging and monads. In: *DSL*. [S.l.: s.n.], 1999. Citado na página 45.

SHEARD, T.; JONES, S. P. Template meta-programming for haskell. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 37, n. 12, p. 60–75, dez. 2002. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/636517.636528>. Citado na página 45.

SKALSKI, K. Syntax-extending and type-reflecting macros in an object-oriented language. In: . [S.l.: s.n.], 2005. Citado na página 45.

STEELE, G. L. Common lisp: the language, 2nd edition. In: . [S.l.: s.n.], 1990. Citado na página 37.

STEELE, G. L. Growing a language. *Higher-Order and Symbolic Computation*, v. 12, p. 221–236, 1998. Citado na página 45.

STEPANOV, A. Future of abstraction. In: *A keynote ad-dress at Joint ACM Java Grande - Seattle, Washington, November 3-5*. [S.l.]: ISCOPE 2002 Conference, 2002. Citado na página 32.

TAHA, W.; SHEARD, T. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, v. 248, p. 211–242, 2000. Citado na página 29.

TRATT, L. Compile-time meta-programming in converge. In: . [S.l.: s.n.], 2004. Citado na página 45.

TWOLIVESLEFT. *Codea*. [S.l.], 2015. Acesso em: 05/11/2015. Disponível em: <http://twolivesleft.com/Codea/>. Citado na página 85.

VILINSKI, A. *Macros Tutorial*. 2012. Disponível em: <https://github.com/rsdn/nemerle/wiki/Macros-tutorial>. Citado 2 vezes nas páginas 46 e 49.

XTEND. *Active Annotations*. 2016. Xtend. Disponível em: <http://www.eclipse.org/xtend/documentation/204_activeannotations.html>. Acesso em: 05 jun. 2016. Citado 3 vezes nas páginas 51, 52 e 54.

YURYEVICH, C. V. *Nemerle Macros Intro*. 2012. Disponível em: <https://github.com/rsdn/nemerle/wiki/Nemerle-macros-intro>. Citado na página 46.