LINGPON 2.0 COMPILADOR PARA FRAMEWORK JAVA 1.0

Leonardo Trevisan Silio, (A. F. Ronszcka), [J. A. Fabro, J. M. Simão]

***Abstract*— Este artigo a respeito do trabalho realizado no tocante a extenso do LingPON para Framework Java 1.0, utilizando as tecnologias flex, bison e C++.**

I. INTRODUÇÃO

O LingPON é uma tecnologia desenvolvida por Adriano Francisco Ronszcka em sua tese de doutorado. Ele propõe um método de conversão de código padronizado em conjunto a um compliador para sua linguagem baseada no Paradigma Orientado a Notificações (PON). O PON, por sua vez é um paradigma que busca reduzir inconsistências e redundâncias encontradas em paradigmas como o imperativo. É comum as situações onde encontramos a seguinte estrutura apresentada na figura 1.

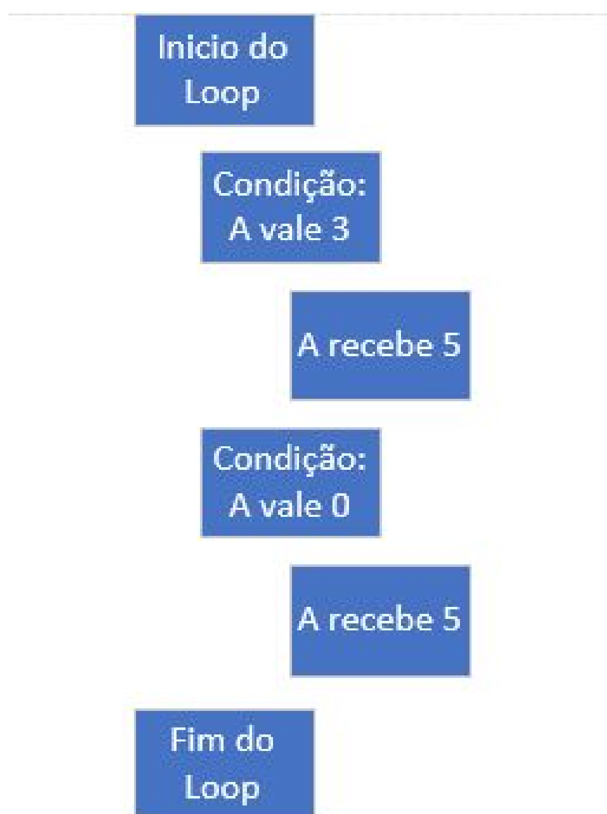

Fig. 1. Exemplo estrutural de um trecho de código com um lao aninhado com varias condições.

A figura 1 uma representação simplista de parte de um sistema hipotético. Nela, é fácil perceber que caso a variável A seja 3, seu novo valor ser 5. Porém, dado está estrutura, a segunda condição nunca será validada. Mesmo assim, sempre será testada. Apesar do exemplo simples, em aplicações reais isso acaba ocorrendo muito. O PON vêm com uma proposta de reduzir tais problemas relacionando as condições e suas premissas as ações que ocorrem quando as primeiras são válidadas. No caso, apenas quando o valor de A fosse alterado as premissas seriam testadas evitando que muitas condiões em um laço extenso fossem retestadas de forma redundate. O LingPON vêm como uma forma de facilitar a utilização do paradigma, uma vez que existem frameworks em várias linguagens, como Java, C# e C++, que suportam o PON. A ideia proposta um combilador para uma linguagem única, declarativa, que enalteça as qualidades do PON que possa ser convertida, atravéz do uso de flex, bison e C++, em várias linguagens alvo, cada uma utilizando seu respectivo framework PON. Permitindo-se utilizar o paradigma apartir de uma linguagem padronificada. Sendo assim, era vital a criação de um compilador de código que fosse uma extensão para a linguagem Java, visto que java é popular, para ampliar a tecnologia para mais plataformas.

II. METODOLOGIA

A. O método usado, as tecnologias, e o LingPON

A conversão de código para uma estrutura genérica é feita através de: Flex, que cuida da análise léxica do código fonte, ou seja, utiliza de expresses regulares para transformar o código LingPON em um conjunto te tokens que serão enviados ao bison; O Bison, por sua vez, é a tecnologia que ira receber os tokens e definir as regras e estrutuas sintáticas que existem no código. Por isso ele chamado de analisador sintático; Por fim, vem a estrtura genérica citada, o grafo PON. Uma estrutura orientada a objetos feita em C++. A figura 2 mostra como esse processo é realizado de forma simplificada.

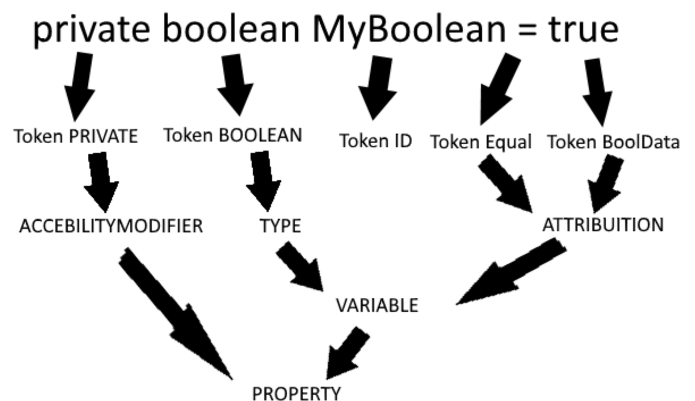

Fig. 2. Exemplo de interpretação de código através de flex e bison.

Na figura 2 é notavél que na verdade, o conjunto Flex e Bison montam uma árvore de expressões. As expressões terminais executam código C++ e salvam a estrutura criada em um grafo. Existem muitos elementos do PON que são representados no LingPON, são eles:

- FBE: Estrutura que representa entidade em um programa PON, em analogia a OO temos as classes.
- Atributos: Referência de uma FBE a outras e adição de metadados como números inteiros e textos a estrutura.
- Métodos: Comandos que podem ser executados por uma FBE, conceito semelhante ao empregado em Orientação a Objetos.
- Regra: Sendo um dos elementos mais importantes do PON a regra é composta por uma condição e uma ação. A condição tem suas premissas. O interessante que vale salientar é que: Ao alterar um Atributo, as premissas associadas serão notificadas. Caso seus valores mudem, eles notificarão uma condição, que por sua vez, caso torne-se verdadeira, ira notificar métodos, criando assim uma cadeia de notificações, evitando redundâncias e testes desnecessários.

Assim carrega-se um grafo orientado a objeto, onde temos classes que representam os elementos acima citados entre outros, como blocos de código em diferentes linguagens, por exemplo. O que foi feito na extensão do LingPON foi criar um programa em C++, mais especificamente uma classe de compilador que herda da classe padrão de compiladores que se associará com o código de carregamento do grafo. Esta classe deverá implementar um método para leitura do grafo e geração do código em Java, utilizando o Framework PON 1.0.

B. Alterações básicas iniciais

É inevitável que a estrutura original possa necessitar de pequenas adaptações, estas, é claro, não comprometem o funcionamento de outras extensões produzidas nem outro dispositivo do projeto original. Porém, o LingPON funciona com base em blocos de código para seus métodos, tais trechos pertencem a diversas linguagens, ou como são chamadas *Targets* (alvos). Na figura 3, a primeira alteração é no tocante ao código flex, nota-se que foi adiciona o termo *FRAMEWORK_JAVA_1_0*, seguindo o padrão utilizado por outros suportes a linguagens já feitos. Da mesma forma, a figura 4 mostra a adio de um *token* ao código bison.

```
FRAMEWORK_JAVA_1_0          {
                                if (code_block == 0) {
                                    return FRAMEWORK_JAVA_1_0;
                                } else if (code_block == 1) {
                                    code_block = 2;
                                    return FRAMEWORK_JAVA_1_0;
                                } else {
                                    code += strdup(yytext);
                                }
                            }
```

Fig. 3. Regra adicionada ao arquivo lex.l com expressão regular simples, contendo apenas um texto direto *FRAMEWORK_JAVA_1_0*.

O conjunto de classes que compreende um *Framework* que da suporte aos objetos referentes ao PON usado pelo grafo contém também uma classe que representa um *Target*. Ele foi alterado para compreender o Java. Isso pode ser visto nas figuras 5 e 6.

```
%token FRAMEWORK_CPP_2_0
%token NAMESPACES_CPP_2_0
//Adicionado TOKEN realcionado ao framework Java
%token FRAMEWORK_JAVA_1_0
```

Fig. 4. Adição do *Token FRAMEWORK_JAVA_1_0* ao arquivo bison.y.

```
public:

    static const int FRAMEWORK_CPP_2_0_TARGET = 1;
    static const int NAMESPACES_CPP_2_0_TARGET = 2;
    static const int FRAMEWORK_JAVA_1_0_TARGET = 4;
```

Fig. 5. Adição de uma constante estática a classe para compreender a nossa linguagem alvo.

```
std::string Target::getTargetName() {

    switch(this->target) {

        case FRAMEWORK_CPP_2_0_TARGET: {
            return "FRAMEWORK_CPP_2_0";
        } break;

        case NAMESPACES_CPP_2_0_TARGET: {
            return "NAMESPACES_CPP_2_0";
        } break;

        case FRAMEWORK_JAVA_1_0_TARGET:
            return "FRAMEWORK_JAVA_1_0_TARGET";

        default: {
            return "undefined";
        } break;

    }

}
```

Fig. 6. Adição de um caso em um *switch* para receber o nome do alvo na forma de texto.

Por fim, como pode ser visto na figura 7, a regra no analisador sintático que relaciona o *target* com sua adição ao grafo PON.

```
target                      : FRAMEWORK_CPP_2_0 {
                                $$ = graph->createTarget(Target::FRAMEWORK_CPP_2_0_TARGET);
                            }
                            | NAMESPACES_CPP_2_0 {
                                $$ = graph->createTarget(Target::NAMESPACES_CPP_2_0_TARGET);
                            }
                            | FRAMEWORK_JAVA_1_0 {
                                $$ = graph->createTarget(Target::FRAMEWORK_JAVA_1_0_TARGET);
                            }
                            ;
```

Fig. 7. Regra no arquivo bison.y definindo o uso do *Target*.

A partir disto resta implementar a nossa classe *FrameworkJAVA10Compiler*.

C. Trabalhando no compilador para Java

Dado a estrutura até então apresentada e as modificações iniciais podemos iniciar o trabalho a respeito do compilador.

O que será feito, basicamente, é utilizar os métodos do grafo para possibilitar a retirada de dados que serão escritos em arquivos *.java*. A estrutura de códico mais comum é um laço de repetição onde são avaliados todos os itens de uma determinada caracteristíca. Por exemplo, toda FBE possui diversos métodos, então, dado um objeto que representa uma FBE, é retirado cada método de uma lista ou mapa (dicionário) e escrito no arquivo resultante um após o outro.

É importante salientar que um projeto em LingPON é um conjunto de FBEs, ou seja, muitos arquivos onde cada um termina por gerar um arquivo em java. Entre os arquivos fonte existe sempre um *Main* que possui um tratamento diferente dos outros FBEs. Isso acontece pois, no lugar de implementar a interface FBE, o *Main* herda da classe *NOPApplication*, que possui diversos métodos, entre eles o *codeAppliction* que usado no método de inicio do programa java, o *main* (note que não se deve confundir a FBE principal *Main* com a função de inicio de um programa java por padrão *main*).

Desta forma, o arquivo *Program.java* será sempre o mesmo, independente da aplicação, ao qual pode ser visto na figura 8.

```
public class Program
{
    public static void main(String args[])
    {
        Main m = new Main(SchedulerStrategy.NO_ONE);
        m.codeApplication();
    }
}
```

Fig. 8. Classe *Program* padrão.

O arquivo *Program.java* é escrito usando um método a parte, uma vez que ele não depende do estado do grafo, além, é claro, da estratégia definida na Fbe principal.

A figura 9 mostra o método de geração de código que chama todas as outras funções necessárias.

```
std::cout << "\nGenerating code..." << std::endl;
createMain();
for (std::map<std::string, Fbe*>::iterator it =
    graph->getFbes()->begin();
    it != graph->getFbes()->end(); it++)
{
    generateCodeFbe(it->second);
}
```

Fig. 9. Método de geração inicial.

O código é formado essencialmente por 4 ações: Um *print* para indicar que a geração começou; A segunda coisa é a chamada da função *createMain*, que cria o arquivo *Program.java*; O terceiro é o inicio de um laço que itera todas as Fbes declaradas ao longo do programa *LingPON*; Por fim, para cada iteração, temos o uso de um método *generateCodeFbe*, que cuida da geração de cada Fbe, que resultará e um arquivo novo para cada uma destas Fbes.

A forma com que o compilador trabalha é bem simples. É comum ver um código semelhante ao da figura 10. Podemos ver um laço de repetição para a analize de todas as instâncias, ou seja, todas as variaveis referênciadas pela Fbe em análise. Algo interessante e importante de salientar é o constante tratamento da Fbe *Main* que possui caracteristicas diferentes no tocante ao seu arquivo final, uma vez que outras Fbes, quando convertidas para classes implementam a interface *Fbe*, o *Main* herda da classe *NopApplication*. Em outras palavras, em muitos contextos existe a necessidade de tratar o código gerado pela Fbe principal do programa.

```
for (std::map<std::string, Instance*>::iterator it
    = instances->begin(); it != instances->end(); it++)
{
    localinstance = it->second;
    if (localinstance->getName() != "this")
    {
        newline(1);
        filestream << "public " << localinstance->getFbeName()
            << " " << localinstance->getName() << ";";
    }
}
```

Fig. 10. Exemplo de estrutura, no caso, adição de instâncias.

Na estrutura podemos ver que, caso a instância não é referente a Fbe *Main*, temos a criação de um atributo na classe java. Isso ocorrerá bastante ao longo do código do compilador. Logo, iremos salientar apenas duas coisas importantes: A primeira coisa são alguns tratamentos e formas como o código é adaptado; A segunda são a lista de elementos gerados, são eles:

- Adição de *imports*. (pode ser de forma estática, ou seja, não depende do estado da Fbe)
- Adição de atributos representando as instâncias da Fbe.
- Cria construtor para inicialização dos atributos.
- Adiciona atributos que são ponteiros para métodos. Isso acontece devido ao fato de que, ao notificar um método e necessário de um tipo de referência dele, para isso usa-se técnicas de *reflection*, necessitando assim de um objeto do tipo *MethodPointer*.
- Definição de regras e premissas e um método especifico.
- Cria métodos definidos na Fbe trasnferindo código para o arquivo java.
- Inicializa os ponteiros de métodos com as funções definidas até então.

A respeito dos detalhes que devem ser levantados além da estrutura básica de algoritimo de leitura do grafo e escrita de código apresentada, temos, primeiramente, a forma como os métodos são criados.

A questão interessante a respeito do *LingPON* é o fato de que, mesmo sendo possível criar uma linguagem genérica que fosse convertida para qualquer linguagem, ou seja, que os métodos associados a Fbe fossem criados apartir de um linguagem de algoritmos própria do *LingPON*, possivelmente, não seria possível especificar uma linguagem que atendesse a todos os paradigmas e peculiaridades dos idiomas algo. No lugar de adicionar uma camada de complexidade que pudesse trazer problemas e falta de suporte a frameworks especificos de arquiteturas/linguagens, temos a possibilidade de eleger o idioma alvo e escrever o algoritimo nele. Assim, podemos apenas procurar o código, no caso, do *Java* e escrevê-lo no

arquivo resultante.

```
//Procura código para Java Framework 1.0 (ID do target é 4, a principio)
for (std::map<std::string, CodeBlock*>::iterator
    itcode = method->getCodeBlocks()->begin();
    itcode != method->getCodeBlocks()->end(); itcode++)
{
    if (itcode->second->getTarget()->getTargetId() == 4)
    {
        filestream << itcode->second->getCode();
    }
}
```

Fig. 11. Busca de código para o framework especifico.

Na figura 11 podemos ver a busca pelo código definido como alvo para o *Framework Java 1.0*.

Outro ponto importante a levantar é a forma como são criados as regras e condições, que são elementos fundamentais do PON. Seja no método denominado *initRules*, que é reescrito da classe *NOPApplication* na classe *Main* ou no construtor das classes que representam as Fbes, as regras são definidas localmente e interagem através de referências. Isso é possível uma vez que o *Gabarage Collector* limpará elas da memória e não é necessário manter referência dos objetos. Na figura 12 temos um exemplo de como é definido as regras em um construtor.

```
//Define condição
Condition cond1 = new Condition(Condition.DISJUNCTION);
//Adiciona premissas
cond1.addPremise(new Premise(sectorA.atIntruderDetected),
    new NBoolean(true), premise.EQUAL, false);
cond1.addPremise(new Premise(sectorB.atIntruderDetected),
    new NBoolean(true), premise.EQUAL, false);
//Cria ação
Action action1 = new Action();
//Cria regra, adicionando condição, ação e estratégia
Rule rule1 = new Rule("rlInvasionDetection",
    scheduler, cond1, action1, false);
//Adiciona instigação a ação anteriormente criada
action1.addInstigation(new Instigation(this.mtSendSms));
```

Fig. 12. Exemplo de criação de regra.

III. RESULTADOS

A partir disso, podemos analisar o resultado obtido da geração de código. Na figura 13 vemos uma Fbe Alarme com uma variável booleana que representa o estado do alarme. Já na figura 14 temos o resultado após o processo de compilação.

```
fbe Alarm

    public boolean atStatus = false

end_fbe
```

Fig. 13. Código original em *LingPON*.

Na figura 14 foram retiradas os *imports*, devido ao fato de que é uma extensa lista, como foi optado por sempre importar tudo que possa ser necessário para o funcionamento do *framework*. Nela podemos ver um conjuntos de complexidades desnecessárias que foram evitadas, como

```
public class Alarm implements FBE
{

    public NBoolean atStatus = new NBoolean(false);

    public Alarm(){

    }

}
```

Fig. 14. Código obtido após aplicação do método.

conhecimentos de paradigmas como orientado a objetos e sintaxe especifica do Java.

Para um exemplo mais complexo, temos, nas figuras 15 e 16, vemos a implementação de um regra com subcondições entre outros detalhes.

```
rule rlFireAlarmA
    condition
        subcondition
            premise prSectorInPeaceA
                this.atIntruderDetected == false
            end_premise
            and
            premise prAlarmAOn
                alarmA.atStatus == true
            end_premise
        end_subcondition
        and
        subcondition
            premise prSensorA1State
                sensorA1.atState == true
            end_premise
            or
            premise prSensorA2State
                sensorA2.atState == true
            end_premise
        end_subcondition
    end_condition
    action sequential
        instigation parallel
            call sirenA1.mtFire(10)
            call sirenA2.mtFire(30)
            call this.mtNotifyInvasion()
        end_instigation
    end_action
end_rule
```

Fig. 15. Trecho de código *LingPON* com definição de uma regra.

```
Condition cond1 = new Condition(Condition.CONJUNCTION);

Subcondition subcond11 = new Subcondition(Condition.CONJUNCTION);
Subcondition subcond12 = new Subcondition(Condition.DISJUNCTION);

subcond11.addPremise(new Premise(alarmA.atStatus,
    new NBoolean(true), premise.EQUAL, false));
subcond11.addPremise(new Premise(this.atIntruderDetected,
    new NBoolean(false), premise.EQUAL, false));
cond1.addSubCondition(subcond11);

subcond12.addPremise(new Premise(sensorA1.atState,
    new NBoolean(true), premise.EQUAL, false));
subcond12.addPremise(new Premise(sensorA2.atState,
    new NBoolean(true), premise.EQUAL, false));
cond1.addSubCondition(subcond12);
```

Fig. 16. Resultado compilado da figura 15.

A figura 16 está reestruturada devido ao fato de que a ordem com que a geração de código ocorre torna, algumas vezes, a analise das regras um pouco mais dificil.

IV. DISCUSSÃO

Dado os exemplos de conversão de código apresentados podemos perceber as vantagens do *LingPON* sobre o uso

direto do *framework* java. A primeira coisa que temos é a simplicidade do idioma original próprio ao PON. Embora, como temos na figura 15, o número de linhas seja superior, a estruturação intuitiva pode ser muito mais fcil de compreender a estrutura da regra apresentada.

E ainda vale salientar que, com uma *IDE* apropriada, a facilidade que se obtem em utilizar do *framework* PON Java, uma vez que já se conhece os conceitos do paradigma é notável.

V. CONCLUSÕES

O *LingPON* é uma tecnologia importante no desenvolvimento do paradigma, logo, uma extesão para que este comece a suportar um idioma popular como Java é interessante para futuras aplicações. A facilidade obitida de se usar o *framework PON* para Java será essencial para possibilitar o surgimento de grandes projetos com melhor eficiência e velocidade de desenvolvimento.

REFERENCES

[1] LINHARES, Robson Ribeiro, Contribuio para o desenvolvimento de uma arquitetura de computao prpria ao paradigma orientado a notificaes. 2015. 354. Tese de Doutorado Universidade Tecnolgica Federal do Paran, Curitiba, 2015.

[2] SANTOS, Leonardo Araujo. Linguagem e compilador para o paradigma orientado a notificaes: avanos para facilitar a codificao e sua validao em uma aplicao de controle de futebol de robs. 2017. 294. Dissertao Universidade Tecnolgica Federal do Paran, Curitiba, 2017.

[3] PORDEUS, Leonardo Faix. Simulao de uma arquitetura de computao prpria ao paradigma orientada a notificaes. 2017. 365. Dissertao Universidade Tecnolgica Federal do Paran, Curitiba, 2017.