# Especificação para Compilador de Featherweight Java+-

## Parte III - Sintático

**Pedro Abreu**[1,*]

[1]UnB, CIC, Brasilia, Brasil
[*]abreu223@hotmail.com

## 1 Introdução

> *You'll finally really learn whatever programming language you're writing a compiler for. There's no other way. Sorry!*
>
> – Steve Yegge[1]

Neste trabalho apresentaremos a especificação do analisador sintático para o compilador de FJ+- que está sendo implementado no decorrer do semestre da disciplina de Tradutores ministrado pela professora Cláudia Nalon.

Para isto apresentaremos uma breve explicação da semântica da linguagem na sessão 2, discutiremos a implementação do analisador léxico, contendo uma lista da nomenclatura de todos tokens, seus mecanismos de tratamento de erro, a forma de comunicação entre o léxico e o sintático e listagem dos ajustes necessários para o analisador sintático na sessão 3, apresentação da Gramática livre de contexto para FJ+-, destacando os ajustes necessários na sessão 4, exemplos de programas na sessão 4, e finalmente as considerações finais na sessão 5.

## 2 FJ+-

FJ+- será construido a partir de Featherweight Java[2], proposto por Igarashi *et. al.*, com a supressão de casts, e adição de condicionais (*if* e *if then else*), loop (*while*), tipos primitivos `int` e `boolean`, bloco de instruções e declaração e atribuição de variáveis

Portanto todos os elementos da linguagem de FJ+- são: declaração de variável, atribuição, bloco de instruções, *if*, *if then else*, *while*, os tipos primitivos `int` e `boolean`, declaração de classes, construtores de objetos, invocação de método, acesso a atributos e variáveis. Seguindo, sempre que aplicável, a semântica de FJ.

Classes em FJ são compostos por zero ou mais declarações de atributos, um único construtor que recebe como argumento a inicialização de cada um dos atributos da classe (casados por nome) e métodos. FJ+- possuirá o mesmo comportamento de construtor de FJ, o que significa dizer que o construtor deverá estar sempre presente, inicializando cada um dos atributos da classe e suas superclasses.

A diferença maior entre FJ e FJ+- será no corpo dos métodos e na "main", pois será possível realizar blocos de instruções.

Exemplos de programas e como serão analisados pelo analisador léxico disponibilizado junto deste trabalho serão fornecidos na sessão 3.

### Operações e Expressões

Para Inteiros serão implementadas operações primitivos binárias: soma (+), subtração (-), multiplicação (*) e divisão (/), BEQ (==), BLE (<=), BGT (>=), BLT (<) e BGT(>). A multiplicação e a divisão

têm maior precedência que a soma e a subtração. Note que, apesar de destas últimas operações serem entre inteiros, a comparação em si é um booleano.

Para Booleanos serão implementadas operações primitivas binárias: OR (||), AND (&&), E a operação unária: NOT (∼). Not tem maior precedência, seguida pelas comparações entre inteiros com a mesma precedência entre si, seguido por AND e finalmente OR.

As operações para inteiros e booleanos em FJ+- possuem comportamento iguais ao do Java ™.

Os blocos de instrução serão delimitados por chaves ('{' e '}'). O *while*, *if* e o *if then else* também terão comportamento igual ao Java ™, isto é, receberão como parâmetro um *booleano* e executarão um bloco de instrução cada. No caso do *while*, será executará o bloco de instrução zero ou mais vezes, de acordo com a avaliação da expressão parâmetro.

O `this` é uma variável que deve aparecer apenas dentro de métodos e construtores.

Passagens de parâmetro de `ints` e `booleans` sempre serão realizadas por valor, e passagens de objetos serão sempre realizadas por referencia.

Todas declarações terão escopo estático, as declarações de método terão escopo público, isto é, visibilidade global. Uma variável declarada dentro de uma função terá escopo delimitado por aquela função. Mas variáveis declaradas dentro de if ou while não terão escopo delimitado por aquele bloco.

## 3 Léxico

No Arquivo ./src/fj.l fornecido em conjunto deste trabalho está a descrito detalhadamente as regras implementadas pelo analisador léxico.

Todos os erros encontrados pelo analisador léxico dirá exatamente em qual linha e qual coluna aquele erro foi encontrado. Para o caso de tentar declarar uma variável que começa com um número na décima linha por exemplo, exibirá uma mensagem parecida com *Line 10,1: Malformed arithmetic expression*.

As keywords são `int, boolean, this, Object, class, new, true, false, return, super, extends, if, else, while`.

Um id é dado pela seguinte expressão regular $[A-z_][A-z\_0-9]*$. Ids serão utilizados para nomes de variáveis, métodos e classes, e keywords em geral tais como /textttint, boolean,this, Object, etc. Note que elas não podem começar com números.

Existem dois lexemas para comentários. Comentário multilinhas é tudo aquilo que começar com $/*$ e terminar com $*/$. Toda linha que começar com $//$ será considerado comentário unilinha. Ou seja, tudo que for encontrado dentro dos marcadores $/**/$ tudo após $//$ até a quebra de linha será ignorado.

Os tokens gerados pelas operações são: `PLUS, MINUS, MULT, DIV, NOT, BAND, BOR, BEQ, BGE, BLE, BGT, BLT`.

Espaços em branco e quebras de linhas são ignorados. No entanto é importante observar que as quebras de linhas são importante para o sintático saber em qual linha tal token se entra. Esta informação fará parte da tabela de simbolos.

### Ajustes feitos no analisador léxico

Foram retiradas o token de keyword e agora cada keyword tem seu próprio token.

Os tokens de um único caracteres que não sejam operações agora são passados como caracter diretamente. Estes tokens são `COMMA, SEMICOLON, L_PAREN, R_PAREN, L_CURL_BRACKETS, R_CURL_BRACKETS, DOT`

Foram retirados os tratamentos de erro de tokens, devido feedback negativo.

## 4 Sintático

Em Listing 1 apresentamos a Gramática de FJ+-, observe que os terminais analisados pelo analisador léxico encontram-se coloridos de vermelhos e em negrito.

Note que foi necessário retirar todas operações $*$ e $+$ pois esta é incompatível uma gramática com operações semânticas.

As operações de "ou" foram mantidas nesta especificação da gramática por facilidade de notação. Na implementação, cada um é uma regra própria.

As relações de precedência especificados na sessão anterior foram omitidas da gramática por facilidade de implementação fornecida pelo bison que podem ser encontradas no cabeçalho do arquivo fj.y fornecido. A solução do problema de "*dangling if*" foi resolvida da mesma forma.

**Listing 1.** Gramática livre de Contexto do FJ+-

```
program → class stmtList

class → ε
      | class ID extends { classMembers }
      | class ID extends { }

classMembers → classMember
      | classMembers classMember

classMember → varDecl ;
      | constrDecl
      | functionDecl

constrDecl → Id ( formalArgs ) { stmtList }

functionDecl → type Id ( FormalArgs ) { stmtList }

stmtList → ε
      | stmtList stmt

argList → ε
      | exp
      | argList , exp

varDecl → type Id idList

idList → ε
      | idList , Id

type → Id | int | bool

exp → var | binOp | ( exp ) | primary

assignment → var '=' exp
```

```
var → ID | object

object → fieldAcess | methodInvoc | New

fieldAcess → var . Id

methodInvoc → Exp . Id ( ArgList )
        | Id ( ArgList )

New → new Id ( argList )

binOp → exp (+ | -) exp
        | exp (* | /) exp
        | ~ exp
        | exp comp exp

comp → >= | <= | > | < | == | && | ||

primary → TRUE | FALSE | NUM

stmt→ if exp Suite
        | if exp Suite else Suite
        | while exp Suite
        | varDecl ;
        | assignment ;
        | return exp ;
        | var ;

suite → { stmtList )
        | stmt

Num → (digit)+

Id → (letter | _) (letter | digit | _)*
```

### Tabela de Símbolos

A tabela de símbolos apresentada nesta etapa foi armazenadas as informações de variáveis, métodos e classes.

Para variáveis foram armazenadas as informações de nome, tipo, linha e coluna.

Para métodos foram armazenadas as informações de nome, tipo, linha, coluna e formal arguments.

Para classes foram armezenadas as informações de nome da classe, nome da sua superclasse, linha e coluna.

Informações de escopo foram omitidas nesta etapa pois seria necessário varrer a árvore novamente, por ser um atributo herdado e não possuimos uma L-gramática, nem queremos introduzir operações semânticas

no meio das regras.

## 5 Rodando exemplos

Nesta sessão analisaremos alguns exemplos de programas fornecidos junto deste trabalho para serem analisados pelo analisador sintático.

Os arquivos estão organizados da seguinte forma: dentro da pasta src encontra-se os arquivos principais do léxico e do sintático. Na pasta bib encontra-se os módulos da tabela de simbolos e da ast.

Junto do projeto foi fornecido um makefile para facilitar a compilação. Este makefile irá compilar os arquivos da tabela de simbolos e da ast, o léxico, o sintático, em seguida linkar tudo isto e produz o executável chamado **fj**. Em seguida ele limpa todos códigos objetos para a pasta obj. Para compilar é necessário utilizar bison 3.0.5 ou superior.

Após compilar devidamente deve-se executar o programa **fj** gerado, oferecendo o arquivo de entrada com <, por exemplo **./fj < exemplos/class_example**.

Em caso de sucesso o programa irá printar os nós da árvore sintática de forma tão próxima da sintaxe quanto possível.

Em caso de erro, o programa irá printar uma lista dos erros com linha e coluna, e não ira printar os nós da árvore.

### Testes Corretos

Primeiramente, o exemplo canônico de programa FJ apresentado em[2] é apresentado em Listing 2. Com algumas pequenas modificações que tiram proveito dos blocos de instrução de FJ+-;

**Listing 2.** Exemplo canônico de programa FJ modificado

```
class A extends Object {
    A() { super(); }
}

class B extends Object {
    B() { super(); }
}

class Pair extends Object {
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) {
        super();
        this.fst = fst;
        this.snd = snd;
    }
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }
}
/* main */
A myA;
```

```
B myB;
Pair myPair;
myA = new A();
myB = new B();
myPair = new Pair(myA, myB);

myPair.setfst(new B());
```

Em Listing 3 apresentamos um exemplo da implementação dos naturais em FJ.

**Listing 3.** Implementação dos Naturais em FJ

```
class Int extends Object{
    Int(){ super(); }

    Int add(Int rhs){
        return rhs.add(this);
    }
}

class O extends Int{
    O(){ super(); }

    Int add(Int rhs){
        return rhs;
    }
}

class S extends Int{
    Int num;

    S(Int num){
        this.num=num;
    }

    Int add(Int rhs){
        return this.num.add(new S(rhs));
    }

}
new S(new S(new O())).add(new S(new O()))
```

### Testes Incorretos

Em Listing 4 apresentamos um caso de erro envolvendo operadores não implementados na linha 6 e expressão aritmética mal formada na linha 7. Ainda aproveitamos para demonstrar a utilização de `if then else`.

**Listing 4.** Exemplo erro de programa FJ+-, com if

```
1  int a, b, c, err;
2  boolean d;
3
4  a = 3;
5  b = 5;
6  d = a >> b;
7  err = 2a + c;
8
9  if(a >= b){
10     c = 4;
11 }
12 else{
13     c = 1;
14 }
```

## 6 Conclusão

FJ possui funções puras, no entanto FJ+- não serão puras pelo motivo de manter o estado do objeto, daí os métodos estarão acoplados a estes estados.

A definição formal das regras de computação encontra-se fora do escopo deste trabalho, pois estamos mais interessados nos aspectos práticos da linguagem que os teóricos.

A principal dificuldade associada a este trabalho foi a decisão de utilizar estruturas muito próximas da gramática. Assim, foi necessário fazer todos construtores dos nós da árvore na mão, consumindo muito tempo com retrabalho, principalmente alocando estruturas bem parecidas, que certamente teria um trabalho muito reduzido com alguma linguagem orientada a objetos utilizando a ide correta.

Ainda relacionado com esta mesma questão, a estrutura da gramática acabou muito mais inflexível, pois ao modificar uma regra, deve-se modificar diversas partes que lidam com a estrutura correspodente, entre elas *printar*, alocar, e atualizar a interface (i.e. o arquivo .h correspondente).

Certamente poderíamos ter adotado uma árvore mais polimórfica, com quantidade de filhos e atributos variáveis. No entanto não existe mágica, e apesar de facilitar esta etapa muito certamente teríamos que lidar de tratar estas estruturas nas etapas seguintes. O *trade-off* entre estas abordagens permanece nebulosa até esta etapa, no entanto certamente esta avaliação será clareada nas entregas seguintes.

Por falta de tempo ainda não foi implementado a destruição da árvore. Este problema deverá ser atacado antes de começar a implementação do analisador semântico, não se deve começar a implementar o semântico sem esta *feature*.

## References


1. Yegg, S. Sevey's blog rants. http://steve-yegge.blogspot.com.br/ (2012). [Acessado em 20 de Agosto de 2016].
2. Igarashi, A., Pierce, B. C. & Wadler, P. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, 132–146 (1999).