Especificação para Compilador de Featherweight Java+-

Parte IV - Semântico

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 semântico para o compilador de FJ+- que está sendo implementado no decorrer do semestre da disciplina de Tradutores (116459) ministrado pela professora Cláudia Nalon. Note que, neste trabalho foi adicionado a seção 5 em relação à entrega anterior. Todo o código referente a este trabalho pode ser encontrado em [2].

Nesta entrega foi implementada a análise estática de código, em especial operações de *typechecking* do código, pois FJ+- é estaticamente tipada. Para isto foi implementada a tabela de classes, a tabela de funções e a tabela de símbolo, definindo desta forma a noção de escopo de forma mais precisa. Não foi implementada nenhuma otimização de código nem alguma forma de *prunning* da árvore sintática (AST). Uma explicação detalhada do semântico pode ser encontrada na seção 5.

Este trabalho apresenta uma breve explicação da sintaxe da linguagem na seção 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 na seção 3, apresentação da Gramática livre de contexto para FJ+- na seção 4, breve descrição do sistemas de tipos, apresentação da tabela de símbolos e discussão sobre os escopos e enumeração dos dos erros semânticos da linguagem na seção 5, explicação de como rodar exemplos de programas FJ+- na seção 6, e finalmente as considerações finais na sessão 6.

2 FJ+-

FJ+- será construido a partir de Featherweight Java [3], 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. E pussuir apenas um construtor, pois FJ não possúi `overload` de funções.

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

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 estas 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 ($\sim$). 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 referência.

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 símbolos.

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.

As relações de precedência especificados na seção 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 Id { classMembers }
      | class Id extends Id { }

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 | _)*
```

Árvore Sintática Anotada

Durante o processo de *parsing* é construída a AST anotada que reflete de forma tão precisa quanto possível esta GLC definida em 1. As principais anotações introduzidas foi a posição em linhas e colunas que aquela estrutura se encontra. Para isto foi utilizado Bison Locations definido na documentação do bison [4].

5 Semântico

Após o parse, a AST é checada estaticamente para criação da tabela de classes, tabela de funções, tabela de variáveis, definição dos escopos necessários, e identificações de erros de tipos.

As mudanças de código da última entrega para esta se resumem em simplificação das estruturas da AST relacionadas à listas, como por exemplo, o nó `Stmt`, que foi mesclado ao nó `StmtList`. Esta decisão simplifica o acesso o acesso à estrutura de `Stmt_u`.

Política de Conversão de tipos

A política de conversão de tipos utilizada é a mesma descrita em [3], para explicá-la é preciso antes introduzir o conceito de Subtipo formalmente descrita em 1, e implementada pela função `int isSubType(char *lhs, char *rhs)` declarada em `/include/symbol_table.h`.

Table 1. Subtyping

$$C <: C \qquad \frac{C <: D \qquad C <: E}{C <: E} \qquad \frac{class\ C\ extends\ D\ \{\ \ldots\ \}}{C\ <:\ D}$$

Para invocar um método é necessário que os argumentos sejam subtipos dos argumentos formais daquele método.

A expressão de retorno de um método também deve ser subtipo do retorno declarado daquele método.

O construtor é tratado como um método qualquer que não pode pussui retorno. E uma classe deve ter exatamente um construtor. Apenas construtores e a main não possuem retorno. Ao se instanciar um novo objeto utilizando o operador `new`, o construtor daquela classe é invocado.

Tabela de Símbolos

A Tabela de símbolos é formada pela tabela de classes (CT), a qual contém uma tabela de funções e uma tabela de variáveis. A tabela de funções também contém uma tabela de variáveis. Definindo assim os escopos de atributos da classe, e variáveis internas à função, respectivamente. Todas estas tabelas foram implementadas sendo tabelas hash, utilizando a biblioteca livre disponibilizada e documentada em [5].

Internamente o primeiro passo na criação da CT é criar a classe `Object`. A classe Object não possui nenhum atributo e somente uma função: A main. A main não deve ser jamais acessada como um método por nenhuma outra classe. Esta decisão foi tomada para não ter necessidade de criar uma nova estrutura para checar e guardar o código da main na tabela de símbolos, pois assim podemos trivialmente tratá-la como uma função qualquer.

A definição da tabela de classes pode ser encontrada em `./include/symbol_table.h`, em Listing 2 reproduzimos sua definição para maior clareza.

Listing 2. Tabela de Classes

```
struct Variable_s{
    char *name, *type;
    Class *tref;
    int line, ch_begin, ch_end;
    UT_hash_handle hh;
};
```

```
struct Function_s{
    char *name, *type;
    Class *tref;
    int line, name_begin, name_end;
    Variable *vars;
    StmtList *stmts;
    FormalArgs *fargs;
    Class *_this;
    UT_hash_handle hh;
};

struct Class_s{
    char *selfName;
    char *superName;
    struct Class_s *_super;
    int line;
    Function *functions;
    Variable *fields;
    UT_hash_handle hh;
};
```

Note que `UT_hash_handle` é a estrutura necessária para que esta estrutura seja uma tabela Hash.

Uma classe possui um nome, o nome da super classe, a referencia para esta super classe, a linha a que ela pertence no código fonte, a tabela com suas funções e a tabela com seus atributos.

As funções possuem um nome, um tipo, uma referência para a classe que representa este tipo, a linha a que ela pertence no código fonte, o local correspondente à coluna no código fonte ao nome desta função. A tabela com as variáveis locais àquela função, referencia na AST ao código desta função, referencia na AST aos argumentos formais e a classe a que esta função pertence, respectivamente.

As variáveis possuem um nome, um tipo, referência para classe do tipo, linha no código fonte a coluna correspondente ao nome.

Note que no caso da função retornar ou variáveis possuírem tipos primitivos `int` ou `bool`, a variável `_tref` deverá ser nulo.

`_tref` e `_this` são utilizados para facilitar a resolução de tipos e subtipos.

`line` e as demais variáveis de demarcação do código fonte são importantes para demarcar o local em que o erro de tipo ocorre. É importante notar que isto é possível pois o arquivo foi bufferizado na variável global `char **source`, onde cada linha do arquivo é um dimensão da variável.

É importante observar que as regras de escopo estão bem definidas com esta decisão de implementação da CT. Pois as variáveis pertencente ao escopo das classes está em nível diferente das variáveis pertencentes às funções. O que nos permite definir uma variável de função com o mesmo nome de um atributo da classe a que pertence, um exemplo claro disto são as variáveis `fst` e `snd` do construtor Pair em 3.

Type Checking

Nesta subseção listmaos todos os erros e os warns gerados pelo analisador semântico.

Erros que serão encontrados:

- Construtores redeclarados;

- Construtor que não possui o mesmo nome que a classe a que pertence;
- Método sem tipo de retorno;
- Tipo do retorno real do método não compatível com o tipo do retorno declarado;
- Atribuição de tipos incompatíveis;
- Chamada de método com argumentos de tipos incompatíveis;
- Instanciação de novo objeto (new) com tipos incompatíveis;
- Chamada de método com número incorreto de argumentos;
- Instanciação de novo objeto (new) número incorreto de argumentos;
- Referencia a tipo não declarado;
- Main e construtores não podem possuir a instrução return;
- Atribuição não pode ser feita para invocação de método;
- Atribuição não pode ser feita para criação de novo objeto;
- Chamada de método em uma variável de tipo primitivo;
- Referencia uma variável não declarada;
- Referencia a um método não declarado;
- Referencia a um atributo não declarado.

Warns que serão encontrados:

- Funções com mesmo nome;
- Variáveis redeclaradas;
- Classes redeclaradas;
- Código inalcançável (após return).

Os warns basicamente significam que aquele trecho de código foi ignorado. No caso de funções, variáveis e classes, aquele trecho de código sequer é colocado na tabela de símbolos.

6 Rodando exemplos

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

Os arquivos estão organizados da seguinte forma: dentro da pasta src encontra-se os arquivos principais do léxico, do sintático e as implementações do módulo AST e symbol_table. Na pasta include encontra-se os headers dos módulos da tabela de símbolos e da AST.

Junto do projeto foi fornecido um makefile para facilitar a compilação. Este makefile irá compilar os arquivos da tabela de símbolos 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 ter o bison 3.0.5 ou superior instalado.

Após compilar devidamente deve-se executar o programa gerado, utilize o comando **./fj [-ast] [-ct] arquivo_fonte**. Por exemplo **./fj -ct -ast exemplos/class_example**.

A flag -ast irá mostrar a ast parseada e a flag -ct irá mostrar a class table, Caso não haja erros léxicos, sintáticos ou semânticos. Isto é, caso seja encontrado algum erro estas flags opcionais não serão consideradas. No entanto as flags ainda serão consideradas em caso de warn.

Os casos de teste se encontram em `/exemplos/semantico`, os código com erros são os mais interessantes e possuem sufixo `_err.fj`. Os testes corretos não capazes de demonstrar quanta checagem foi realizada pelo *typechecker*, no entanto apenas com código correto com a flag -ct irá printar a classtable e demonstrar os respectivos escopos das classes e das funções.

Testes Corretos

Primeiramente, o exemplo canônico de programa FJ apresentado em [3] é apresentado em Listing 3. Com algumas pequenas modificações que tiram proveito dos blocos de instrução de FJ+-;

Listing 3. Exemplo canônico de programa FJ modificado

```
1  class A extends Object {
2      A() { super(); }
3  }
4
5  class B extends Object {
6      B() { super(); }
7  }
8
9  class Pair extends Object {
10     Object fst;
11     Object snd;
12     Pair(Object fst, Object snd) {
13         super();
14         this.fst = fst;
15         this.snd = snd;
16     }
17     Pair setfst(Object newfst) {
18         return new Pair(newfst, this.snd);
19     }
20 }
21 /* main */
```

```
22 A myA;
23 B myB;
24 Pair myPair;
25 myA = new A();
26 myB = new B();
27 myPair = new Pair(myA, myB);
28
29 myPair.setfst(new B());
```

Rodando o exemplo pair 3 com a flag -ct teremos a saída descrita em 4:

Listing 4. Representação da CT do exemplo Pair

```
---------------------- # Class Table # ----------------------
Main:{
}
8: A extd Object{
        9: A(){
                0: A this[0-0];
        }
}
14: B extd Object{
        15: B(){
                0: B this[0-0];
        }
}
19: Pair extd Object{
        20: Object fst[12-14];
        21: Object snd[12-14];
        22: Pair(Object fst, Object snd){
                0: Pair this[0-0];
                22: Object fst[17-19];
                22: Object snd[29-31];
        }
        27: Pair setfst(Object newfst){
                0: Pair this[0-0];
                27: Object newfst[24-29];
        }
}
```

Os números que seguem as variáveis são as colunas de início e fim, respectivamente, em que a declaração da variável aparece no código fonte. Note que o this está sempre presente na linha 0 e coluna 0 de uma função. A variável this aponta, naturalmente, para o próprio objeto o qual o método foi invocado. e possui tipo da própria classe.

Em Listing 5 apresentamos um exemplo da implementação dos naturais em FJ.

Listing 5. Implementação dos Naturais em FJ

```
1 class Int extends Object{
```

```
2       Int(){ super(); }
3
4       Int add(Int rhs){
5           return rhs.add(this);
6       }
7   }
8
9   class O extends Int{
10      O(){ super(); }
11
12      Int add(Int rhs){
13          return rhs;
14      }
15  }
16
17  class S extends Int{
18      Int num;
19
20      S(Int num){
21          this.num=num;
22      }
23
24      Int add(Int rhs){
25          return this.num.add(new S(rhs));
26      }
27
28  }
29  new S(new S(new O())).add(new S(new O()))
```

Rodando o exemplo Int 5 com a flag -ct teremos a saída descrita em 6:

Listing 6. Representação da CT do exemplo Int

```
------------------------ # Class Table # ------------------------
Main:{
}
8: Int extd Object{
        0: super(){
        }
        9: Int(){
                0: Int this[0-0];
        }
        11: Int add(Int rhs){
                0: Int this[0-0];
                11: Int rhs[17-19];
        }
}
```

```
16: O extd Int{
        0: super(){
        }
        17: O(){
                0: O this[0-0];
        }
        19: Int add(Int rhs){
                0: O this[0-0];
                19: Int rhs[17-19];
        }
}
24: S extd Int{
        25: Int num[9-11];
        0: super(){
        }
        27: S(Int num){
                0: S this[0-0];
                27: Int num[11-13];
        }
        32: Int add(Int rhs){
                0: S this[0-0];
                32: Int rhs[17-19];
        }
}
```

Testes Incorretos

Em Listing 7 apresentamos um caso de erro envolvendo métodos. Ao rodar este programa com a flag -ct receberemos as mensagens de erro descrito em 8

Listing 7. Exemplo method_invk_err.fj

```
1  class bbar extends Object{
2      int i;
3      bool x(int q, bool p){
4          return this.i;
5      }
6  }
7
8  class bar extends bbar{
9
10 }
11 // this is a comment test
12 // Quis custodiet ipsos custodes?
13
14 class foo extends Object{
15     bar b;
16     int f() {
```

```
17          bar i;
18          bool uu;
19          i = b.x();
20          i = b.i;
21          b.i = b.y(1);
22          uu = b.x(true);
23          uu = b.x(true, 1);
24          uu = b.x(1, false);
25          uu = b.x();
26
27      }
28  }
```

Listing 8. Erros encontrados no exemplo method_invk_err.fj

```
Error 4: return must have type bool or some subtype, but was int
  ↪ at
        return this.i;
               ^

Error 19: method x expected 2 arguments but 0 was given
        i = b.x();

Error 19: lhs of assignment has different type then rhs in
        i = b.x();

Note: lhs of assignment has type bar and rhs has type bool

Error 20: lhs of assignment has different type then rhs in
        i = b.i;

Note: lhs of assignment has type bar and rhs has type int

Error 21:15: class bar does not have a method y
        b.i = b.y(1);
                ^

Error 21: lhs of assignment has different type then rhs in
        b.i = b.y(1);

Note: lhs of assignment has type int and rhs has type ??

Error 22:18: method x expected argument of type int but received
  ↪ has type bool at
        uu = b.x(true);
                 ^^^^

Error 22: method x expected 2 arguments but 1 was given
        uu = b.x(true);
```

```
Error 23:18: method x expected argument of type int but received
   ↪  has type bool at
         uu = b.x(true, 1);
                  ^^^^
Error 23:24: method x expected argument of type bool but
   ↪ received has type int at
         uu = b.x(true, 1);
                        ^
Error 25: method x expected 2 arguments but 0 was given
         uu = b.x();
```

7 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 esta etapa do trabalho foi a quantidade de erros semânticos associados às estruturas. No começo não havíamos conhecimentos das bibliotecas de tabela hash [5], portanto perdemos alguns dias nisso. Com a utilização desta biblioteca acreditamos que economizamos pelo menos uma semana de trabalho. Isto para não na eficiência adicionada ao trabalho, dizer pois estas tabelas são $O(1)$.

A utilização de nós não polimórficos e muito próximos da gramática facilitou a implementação desta etapa, pois muita informação de tipagem já se encontra diretamente na estrutura do nó em si. Isto facilita muito a navegação na AST.

A dificuldade maior é checar cada caso de erro, pois é fácil tentar acessar um elemento da estrutura inexistente. Como por exemplo acessar uma variável não declarada causaria um null pointer exception. Com isto o código acaba ficando cheio de casos e checagens para garantir que está tudo funcionando como esperado. No entanto não podemos garantir que todos erros foram encontrados nesta entrega.

Futuramente devemos discutir sobre a necessidade de adicionar o conceito de void para métodos terem a opção de não retornar nada.

Então tudo que faltará para ter um compilador FJ+- completo é gerar código.

References

1. Yegg, S. Sevey's blog rants. http://steve-yegge.blogspot.com.br/ (2012). [Acessado em 20 de Agosto de 2016].
2. Abreu, P. Git de tradutores. http://github.com/pedrotst/tradutores (2016). [Acessado em 07 de Novembro de 2016].
3. 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).
4. Donnelly, C. & Stallman, R. Bison documentation. *Free Software Foundation, Cambridge, USA* (1991).

5. O'Dwyer, A. ut hash - a hash table for c structures. http://troydhanson.github.io/uthash/index.html (2013). [Acessado em 07 de Novembro de 2016].