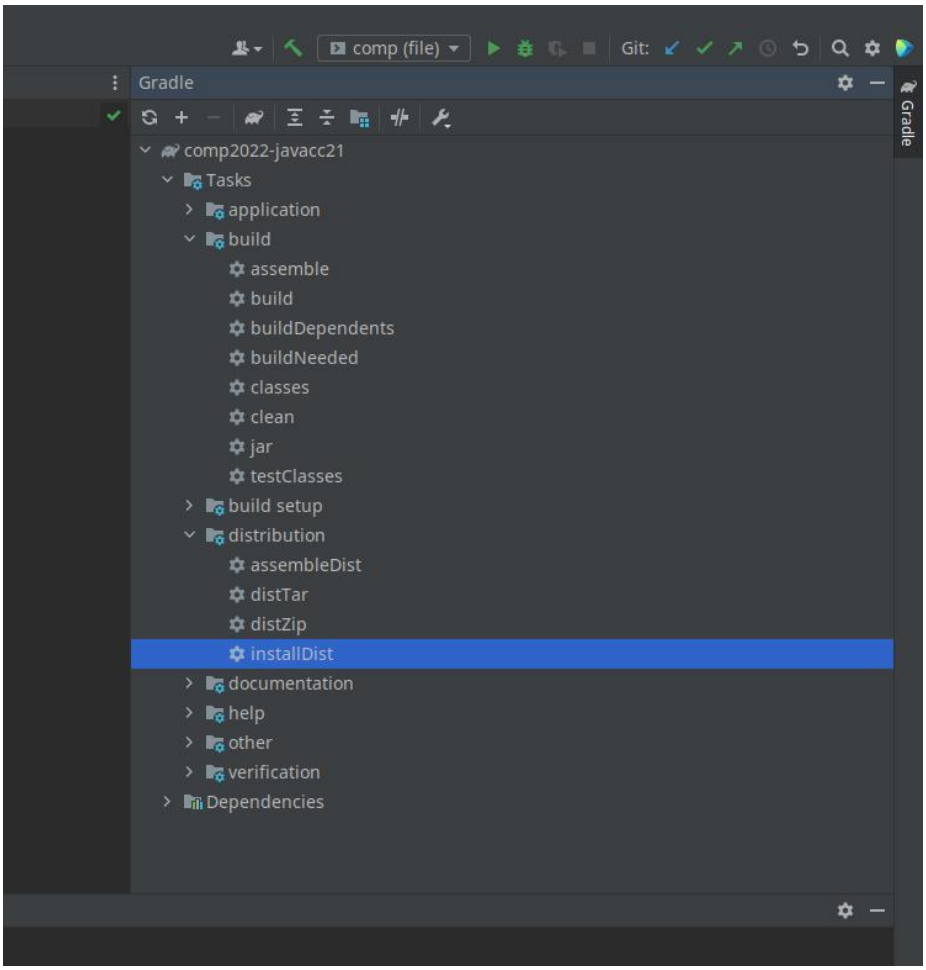
Apoio ao Projeto

Programação

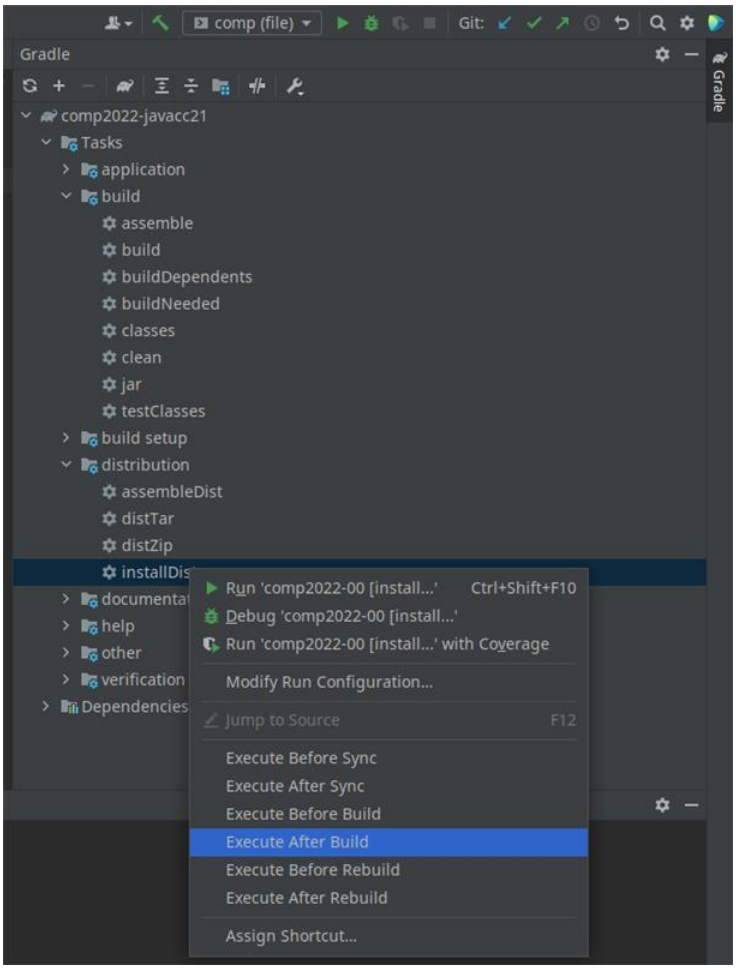
Dica para quem usa IntelliJ IDEA

Como já foi dito, para compilar o vosso código devem usar a task de gradle "installDist". Dentro o IDEA podem fazer isto de várias formas. Há quem use o terminal para chamar o comando diretamente.

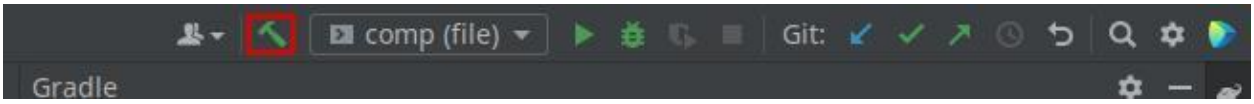
Também é possível usar a tab do gradle e escolher a opção "installDist", como mostra a imagem seguinte.



Uma outra forma, a que eu tenho usado, é forçar que a task "installDist" seja sempre executada depois da task "build" e usar o botão de build da GUI. Primeiro, devem forçar a sua execução, fazendo como indicado na imagem seguinte.



Depois, podem usar o botão de build, o martelo verde.



Comecem o Projeto pelos Testes

Ao passar a gramática que está no enunciado para JavaCC, aconselhamos que a testem logo que tenham algumas regras, muito antes de estar completa. Desta forma podem apanhar problemas muito mais cedo, e também ir entendendo como é que algumas das coisas funcionam.

Para testar, usem testes unitários. Podem ver um exemplo no ficheiro `test/ExampleTest.java`. Esse ficheiro tem algo como:

```
@Test
 public void testExpression() {
```

```
 TestUtils.parse("2+3\n10+20\n");
 }
```

Tal como esse código está, o parser é chamado, mas não está a testar nada de facto. Da forma como construímos o estágio de exemplo, para o parsing, a função .parse() nunca lança exceção, mesmo que o código que se dá ao parser tenha erros de parsing. A ideia é que o .parse() retorne sempre, e que o tratamento de erros seja feito após o estágio.

Para fazer o tratamento de erros, é preciso ver a lista de reports, que se obtém através do resultado do parser (i.e. r.getReports()).

O .getReports() retorna uma lista de objectos Report, que contém o tipo do report (debug, info, warning, error, etc), de onde é que o report vem (lexical, syntactical, sematic, etc), string com a mensagem, e exceção se for erro.

Portanto, quando fazem um teste em que o código que dão é suposto ser válido, a lista de reports não pode ter nenhum report do tipo Error. Nesse caso podem usar a função `TestUtils.noErrors(List<Report>)`, recebe uma lista de reports e verifica se algum deles é do tipo Error. Se não for, não faz nada e o teste passa. Se for, manda exceção e a execução pára ali (ou seja, o teste falha).

Também podemos querer testar casos que falham. Por exemplo, podemos considerar que o código `1+2;` não é válido, e termos um teste em que isto deve dar um erro. É para este caso que serve o `TestUtils.mustFail(List<Report>)`. Recebe uma lista de reports, e garante que há pelo menos um tipo de report do tipo Error. Se houver, não faz nada, e o teste passa (é suposto haver erro). Se não houver erros, lança excepção e o teste pára ali e falha.

No exemplo acima, esse teste deve falhar, uma vez de que o parser que vem com o vosso repositório base não suporta múltiplas expressões. Logo, para esse teste ficar completo, deveria ser algo como:

```
@Test
 public void testExpression() {
 var result = TestUtils.parse("2+3\n10+20\n");
 TestUtils.mustFail(result.getReports());
 }
```

Correr os testes

Se estiverem a usar um IDE como o IntelliJ, podem facilmente correr os testes usandos a interface gráfica (e.g., setinha verde na função do teste unitário). Sem IDE, pode correr "gradle test". Além de compilar o código, também vai correr os testes.

Se algum dos testes falhar, vai gerar uma página HTML com um relatório detalhado sobre o que falhou (o link para o relatório aparece na mensagem de execução do gradle).

Code Updates

Testar Regras em Isolado

*Resumo: novos métodos TestUtils.parse(), TestUtils.parse("1+2*3", "Expression")*

(isto não é obrigatório, é para facilitar os vossos testes, só usam se quiserem)

Foi adicionado aos vossos repositórios novas versões (*overloads*) do método `TestUtils.parse()`que recebe o nome da regra gramatical que querem testar, para poderem testar regras em isolado, sem passar um programa completo.

Por exemplo, se tiveram uma regra *Expression*, podem escrever:

```
TestUtils.parse("1+2*3", "Expression");
```

No entanto, para isto funcionar, precisam de implementar um novo método *parse()* na interface JmmParser, que tem a seguinte assinatura:

```
JmmParserResult parse(String jmmCode, String startingRule, Map<String, String> config)
```

Tip: este método pode passar a ser o vosso método principal, e o método *parse* anterior chamar este, com o nome da vossa regra inicial (e.g. `return parse(jmmCode, "Start", config);`)

Para facilmente adaptarem o código que já têm a este novo método, podem usar a função `SpecsSystem.invoke()`. Por exemplo, onde antes tinham:

```
parser.Start();
```

Podem passar a ter:

```
SpecsSystem.invoke(parser, "Start");
```

Na realidade, vão querer colocar algo como:

```
SpecsSystem.invoke(parser, startingRule);
```

Quando testam uma regra que não a de topo (e.g., "Start") e o parser falha, este não manda excepção. Além disso, as regras costumam ter como retorno *void*. Para aceder ao *root node* de forma genérica usem o método .rootNode(), e se for *null*, isso quer dizer que que houve problemas:

```
SpecsSystem.invoke(parser, startingNode);
 var root = parser.rootNode();
 if (root == null) {
```

```
 throw new ParseException(parser, "Parsing problems, root is null");
}
```

Testes para parser

Resumo: classe ParserTest com unit tests

Programming Project

Foi adicionada a classe `pt.up.fe.comp.ParserTest` à pasta `test` do vosso repositório, contém vários testes unitários usando os ficheiros que já estavam na pasta `test/fixtures/public`.

Cada teste corresponde a um ficheiro, alguns devem passar no vosso parser, outros devem falhar.

Para carregar os ficheiros está-se a usar um conceito de Java que é o *resource*. Representa ficheiros que estão no classpath, e podem ser acedidos usando o classpath como raiz do caminho.

No ficheiro `build.gradle` indicou-se que a pasta `test` é uma pasta de resources, então os ficheiros dentro dessa pasta podem ser acedidos como *Java resources*.

```
resources {
  srcDir 'test'
}
```

Para ler um *Java resource* podem usar o método `SpecsIo.getResource(String resource)`, irá devolver uma String com o conteúdo da resource:

```
SpecsIo.getResource("fixtures/public/HelloWorld.jmm")
```

Testes para Regras do Parser

Resumo: class GrammerTest com unit tests para regras específicas, opcionais

Programming Project

Foi adicionada a classe `pt.up.fe.comp.GrammarTest` à pasta `test` do vosso repositório, contém vários testes unitários para regras específicas da vossa gramática.

Estes testes são opcionais, o objetivo é ajudar ao desenvolvimento da vossa gramática. Estes testes usam o método para testar regras em isolado, que é facultativo (mais detalhes aqui).

Inicialmente estes testes vão estar desabilitados, porque cada trabalho irá dar nomes diferentes à suas regras. Para habilitar os testes, coloquem o nome correspondente à vossa regra nos campos estáticos no início do ficheiro `GrammarTest`:

```
public class GrammarTest {

    private static final String IMPORT = "";
    private static final String MAIN_METHOD = "";
    private static final String INSTANCE_METHOD = "";
    private static final String STATEMENT = "";
    private static final String EXPRESSION = "";
```

OLLIR Library and Examples

The OLLIR library (file `/libs/ollir.jar`) has been updated with some fixes that where due. The fixes include:

- `BinaryOpInstruction` no longer extends `UnaryOpInstruction`, they both now extend a new class `OpInstruction`
- `if` statements now accept single variables (i.e., `SingleOpInstruction`) and any operation (i.e., `OpInstruction`) that returns a boolean. Before `if` statements only accepted binary operations, and unary operations such as `!` had to be written as binary operations
- Semantic analysis is now performed at the `if` statements. Code such as the following is invalid:
  - `if(i.i32)` - condition is not a boolean
  - `if(i.i32 +.i32 $2.N.i32)` - operation does not return a boolean
  - `if(invokespecial(this, "<init>").V)` - is not an operation or a single variable
- Ollir tests in `test/fixtures/public/ollir` have been updated to conform with the semantic analysis

Support for converting the AST directly to Jasmin (groups of two elements)

A new patch has been deployed, which adds a new stage, `AstToJasmin`, for groups that will convert the AST directly to Jasmin.

Patch notes:

- New interface `AstToJasmin`, that should be registered in `config.properties` with the key `AstToJasminClass`
- Overload for `TestUtils.backend()` that receives a `JmmSemanticsResult`
- `TestUtils.backend()` overloads that receive a String of Java-- code now call `AstToJasmin` if property has a value in file `config.properties`
- New protected method `AJmmVisitor.visitAllChildren()`, which can be used as a default visit method
- Improves error message when the class name of a stage is empty in the `config.properties` file

OLLIR Fixes

Update to the OLLIR library. Patch notes:

- GetFieldInstruction no longer extends PutFieldInstruction, now both classes extend new class FieldInstruction
- Adds FieldInstruction.getFieldType()
- Adds ArrayType.getElementClass(), which returns the name of the class of the elements of the array, when the type is an array of objects
- Verifies if boolean literals are either 0.bool or 1.bool (e.g. 2.bool throws an exception)

CP2 Tests and several fixes

You have received an update with several tests that will be used during Checkpoint 2, which will occur during classes this week. Don't worry if you have tests failing right now, you will have until 16/05/2022 to fix the tests. During this week we will only assess how the project currently is, and it will only be graded based on the code you have on Monday next week.

Patch notes:

- Adds battery of tests for Checkpoint 2
- Updates OLLIR, adding `ArrayType.getArrayType()`. This method replaces `ArrayType.getTypeOfElements()`, which becomes deprecated. Previous method was very easy to confuse with `Type.getTypeOfElement()`
- Improves `SymbolTable.print()`, now prints local variables
- Fixes OLLIR example `Fac.ollir`, fixes typo and adds import

Gramática

Porquê usar JmmNode?

No exercício 4.1 modificam o código de forma a que os nós gerados pelo JavaCC implementem a interface `JmmNode`. É na ficha que esta interface permite desacoplar o parser do resto do código, para clarificar o que isto significa, ficam aqui mais alguns detalhes.

O JavaCC gera várias classe que implementam a sua própria árvore (e.g., `Node`, `Token`, `BaseNode`). Se não utilizarmos a interfa mencionado ce `JmmNode`, todo o código que vem a seguir que precise usar a árvore (e.g., estágios de análise, de optimização) ficaria dependente dessas classes do JavaCC.

O JavaCC é apenas um de vários parser generators que podem usar para fazer parsing e gerar a árvore, há vários outros que poderiam usar (e.g., Antlr). Ao fazerem o `Node` do JavaCC implementar a interface `JmmNode`, por um lado, o código que vem a seguir só tem que conhecer JmmNodes, e trabalhar sobre eles; por outro, permite que possam substituir o parser de JavaCC por outro, desde que esse outro parser gere uma árvore cujos nós sejam JmmNodes também.

Por último, se inspecionarem o código do `JmmParserResult`, vão reparar que o construtor chama um método `sanitize()` sobre o vosso nó:

```
public JmmParserResult(JmmNode rootNode, List<Report> reports, Map<String, String> config) {
    this.rootNode = rootNode != null ? rootNode.sanitize() : null;
    this.reports = reports;
    this.config = config;
}
```

Esse `sanitize()` não é mais que converter a vossa árvore para JSON, e de JSON de volta para a árvore:

```
default JmmNode sanitize() {
    return fromJson(this.toJson());
}
```

O método `fromJson` usa uma outra implementação do `JmmNode`, o `JmmNodeImpl`. Isto quer dizer que a partir do `JmmParserResult`, não vão estar a usar nenhuma das classes geradas pelo JavaCC.

```
static JmmNode fromJson(String json) {
    return JmmNodeImpl.fromJson(json);
}
```

jjtThis, #BinOp(2)

A partir do exercício 5 do tutorial, vão ter código na gramática (ficheiro .jj) deste género:

```
AdditiveExpression #void :
 MultiplicativeExpression
 (
 (< PLUS > MultiplicativeExpression { jjtThis.put("op", "ADD"); }) #BinOp(2) |
 (< MINUS > MultiplicativeExpression { jjtThis.put("op", "SUB"); }) #BinOp(2)
 )?
 ;
```

Dando mais alguns detalhes acerca do que está a acontecer, dentro de uma regra JavaCC, por cada regra chamada (e.g., `MultiplicativeExpression`), à partida vão estar a gerar um nó com esse nome.

Se na vossa árvore não estiver a aparecer esses nós, é porque por defeito, o JavaCC21 ativa uma opção que remove automaticamente nós que tenham apenas um filho. Para desativar esta opção, coloquem no início da gramática `SMART_NODE_CREATION = false;`. Os tokens à partida não estão a gerar nós, porque no início da gramática vão ter a opção `TOKENS_ARE_NODES=false;`.

O # é uma diretiva que cria um nó com o nome que está em frente, por exemplo `#BinOp` cria um nó `BinOp` naquele momento. Cada vez que um nó é criado (e.g., `MultiplicativeExpression` , `#BinOp`), esse nó vai para uma stack. Ao colocarem parentesis e um número entre os parêntesis à frente da diretiva, o parser vão remover tantos nós da stack quanto o número que colocarem, e pô-los como filhos do nó que criaram. Ou seja, `#BinOp(2)` vai buscar os dois nós anteriormente criados, neste caso pelas duas chamadas ao `MultiplicativeExpression`, e colocá-los como filhos do `BinOp`. Depois, coloca o `BinOp` na stack.

O `jjtThis` é uma referência para o nó que está a ser construído nessa altura, que no exemplo em cima é o `BinOp`. Portanto, o código `{ jjtThis.put("op", "ADD"); }) #BinOp(2)` está a colocar o valor "ADD" associado à chave "op" no nó `BinOp` que está a ser criado.

Token final na gramática: <LF> para <EOF>

A gramática originalmente fornecida tem, na regra inicial, o token <LF> como final. Isto significa que tem de haver um mudança de linha para o parsing terminar corretamente e gerar a árvore.

```
SKIP : " " | "\t" | "\r" ;

TOKEN :
    < PLUS : "+" > |
    < MINUS : "-" > |
    < TIMES : "*" > |
    < DIVIDE : "/" > |
    < OPEN_PAREN : "(" > |
    < CLOSE_PAREN : ")" > |
    < INTEGER : (["0" - "9"])+ > |
  < LF : "\n" >
;

Start : AdditiveExpression <LF> ;
```

Uma forma mais robusta de especificar a gramática é considerar que essas mudanças de linha, tal como o resto do white space, são para ignorar, e que o token <EOF>, de ***E**nd **O**f **F**ile*, marca o final do texto que vai ser analisado. Este token não precisa de ser declarado pois é automaticamente definido pelo JavaCC21.

```
SKIP : " " | "\t" | "\r" | "\n";
```

```
TOKEN :
    < PLUS : "+" > |
    < MINUS : "-" > |
    < TIMES : "*" > |
    < DIVIDE : "/" > |
    < OPEN_PAREN : "(" > |
    < CLOSE_PAREN : ")" > |
    < INTEGER : (["0" - "9"])+ >
;

Start : AdditiveExpression <EOF> ;
```

Construção e Anotação da AST

Resumo: como poderiam implementar suporte para expressões como a=2 do tutorial. Inclui #void, #<nome>[(<filhos>)], jjtThis, lastConsumedToken

Uma forma inicial de implementarem suporte para assignments no tutorial poderia ser:

Start : (AdditiveExpression | Assignment) <EOF> ;

Assignment :
 < IDENTIFITER > < EQUAL > AdditiveExpression
;

Assumindo como string de entrada `a=2`, como estamos a usar a opção `TOKENS_ARE_NODES = false;`, a AST gerada vai ter nós apenas para o Assignment, e a AdditiveExpression:

Start
 Assignment
 AdditiveExpression

x\xPara podermos ter um nó para o lado esquerdo do assignment (o identifier), podemos usar o cardinal (#). Quando colocamos #<NOME>, criamos um nó com esse nome, e a classe Java equivalente (tenham cuidado e não usem nomes de classes que já existem, como Integer).

Portanto, modificando código:

< IDENTIFITER > **#Id** < EQUAL > AdditiveExpression

Obtemos esta AST:

Start
 Assignment

Id
AdditiveExpression

No entanto, queremos guardar na árvore o valor da variável, que neste caso é a. Para guardar valores nos nós, certifiquem-se primeiro que estão a injetar o código que implementa a interface `JmmNode` (vejam o ficheiro .jj do assignment 4 do tutorial).

Sendo um JmmNode, podem fazer o seguinte:

< IDENTIFITER > **({jjtThis.put("foo", "bar"); })** #Id < EQUAL > AdditiveExpression

(Detalhe: têm que por o código **{...}** dentro de parêntesis, ou o JavaCC dá erro)

Isto permite anotar a informação que quiserem nos nós:

Start
Assignment
Id (foo: bar)
AdditiveExpression

(Detalhe: para obter este dump que tem informação sobre o conteúdo dos nós, usem o nó que está no `JmmParserResult`, e façam `result.getRootNode().toTree()`)

Para aceder ao nome da variável, que está no token, podem usar a variável `lastConsumedToken`:

< IDENTIFITER > ({jjtThis.put(**"name", lastConsumedToken.getImage()**); }) #Id < EQUAL > AdditiveExpres sion

AST resultante:

Start
Assignment
Id (name: a)
AdditiveExpression

Se quisermos transformar o assignment numa operação binária do tipo `assign`, podemos usar de novo o cardinal (#), e indicar o número de nós que queremos que passem a filhos desse nó:

< IDENTIFITER > ({jjtThis.put("name", lastConsumedToken.getImage()); }) #Id < EQUAL > AdditiveExpression **#BinOp(2)**

Isto cria um novo nó, e captura os dois nós anteriores como filhos do BinOp:

Start
Assignment
BinOp
Id (name: a)
AdditiveExpression

É preciso guardar o tipo da operação no nó:

(< IDENTIFITER > ({jjtThis.put("name", lastConsumedToken.getImage()); }) #Id< EQUAL > AdditiveExpression**) ({jjtThis.put("op", "assign");})** #BinOp(2)

(Detalhe: por causa do jjtThis do BinOp, é importante ter os parêntesis à volta dos nós que vão ser capturados, de outra forma o JavaCC queixa-se)

AST:

Start
 Assignment
 BinOp (op: assign)
 Id (name: a)
 AdditiveExpression

Finalmente, na AST que temos agora há alguma redundância. Com o nó BinOp anotado, já sabemos que essa parte da AST representa um assignment, e não precisamos do nó `Assignment`. Para o remover da AST, usamos a diretiva `#void`:

Assignment **#void** :

Com isto, obtemos esta AST:

Start
 BinOp (op: assign)
 Id (name: a)
 AdditiveExpression

A Gramática não é a Linguagem (Java--)

Resumo: trade-off parsing vs análise sobre AST, gramática aceita mais coisas do que o suposto

Já poderão ter reparado, a gramática no enunciado não representa estritamente a linguagem Java--, no sentido em que aceita mais coisas do que o suposto. Por exemplo, a regra `Expression: Expression '[' Expression ']'` aceita o código `(1+2)[0]`, mas este código não é Java-- válido.

Estes casos serão detectados no estágio seguinte, na análise semântica, sobre a AST. Seria possível fazer algumas destas detecções a nível da gramática, no entanto isto implica complicar a gramática. Não há propriamente uma resposta certa acerca do que deve ser feito a nível da gramática no parser, ou sobre a AST na análise.

Podem pensar nisto como um *trade-off*, em que estamos a complicar de um lado para simplificar do outro. Neste caso, escolhemos simplificar na parte da gramática e do parsing, e complicar na parte da AST e da análise.

No exemplo `(1+2)[0]` não chega a haver *trade-off*. Como a linguagem aceita código como `(foo())[0]`, e *foo* pode retornar qualquer coisa, durante a análise semântica vão ter que verificar se a expressão que está associada ao array retorna um tipo que é array. Ou seja, esse trabalho vai ter que ser feito de qualquer das formas, estar a adicionar verificações extra durante o parsing não permite poupar trabalho mais tarde, logo a gramática pode simplesmente aceitar qualquer expressão.

Exemplo de SCAN (ou Lookahead)

Resumo: utilização de SCAN, LL, regra para method

O JavaCC é um parser LL com lookahead global de 1, ou seja, em condições normais apenas consegue ver um token de cada vez. Então se houver regras ao mesmo nível a começarem pelo mesmo token, o JavaCC não vai conseguir distinguir entre as duas regras, e escolhe sempre a primeira.

Por exemplo, considerem uma regra para detetar métodos que tem duas regras, uma para métodos de instancia e outro só para o main method:

method: mainMethod | instanceMethod

Se ambas as regras começarem com o token < PUBLIC >, o JavaCC vai escolher sempre o *mainMethod*, pois é q primeira regra, e não consegue ver para além desse token.

No entanto, se o segundo token da regra *mainMethod* for < STATIC >, esse token não aparece na regra *instanceMethod*, e pode ser usado para distinguir as duas regras. Nesse caso pode-se usar o SCAN com valor 2:

method: mainMethod | instanceMethod

Neste caso, a regra *mainMethod* só é aceite se os dois primeiros tokens da regra (< PUBLIC > < STATIC >) fizerem match.

Encadeamento de Operações e Associatividade

Pegando no exercício 1 (Secção 6.1) do tutorial de introdução ao JavaCC21, vamos tentar perceber o conceito de associatividade.

A associatividade de um operador diz a que operador (à esquerda ou à direita) um operando pertence no caso em que está rodeado de dois operadores **iguais**. Por exemplo, no código abaixo o operando 6 pertence ao operador da esquerda.

```
48/6/2
```

O resultado esperado é 4. Primeiro dividimos 48 por 6 e depois dividimos esse resultado, 8, por 2. Caso a divisão fosse associativa à direita então o resultado esperado seria 16, pois primeiro iríamos dividir 6 por 2 e só depois dividiríamos 48 por 3. Logo:

```
48/6/2 == (48/6)/2 != 48/(6/2)
```

É importante que este conceito seja respeitado para garantir uma correta semântica das expressões aritméticas na vossa linguagem.

A soma, a subtração, a multiplicação, e a divisão têm associatividade à esquerda. Como exemplo, a exponenciação, que irei representar com o operador ^, tem associatividade à direita.

```
A^B^C
```

Na expressão acima, o operando B pertence ao operador da direita, portanto:

```
A^B^C == A^(B^C) != (A^B)^C
```

Associatividade e Recursividade

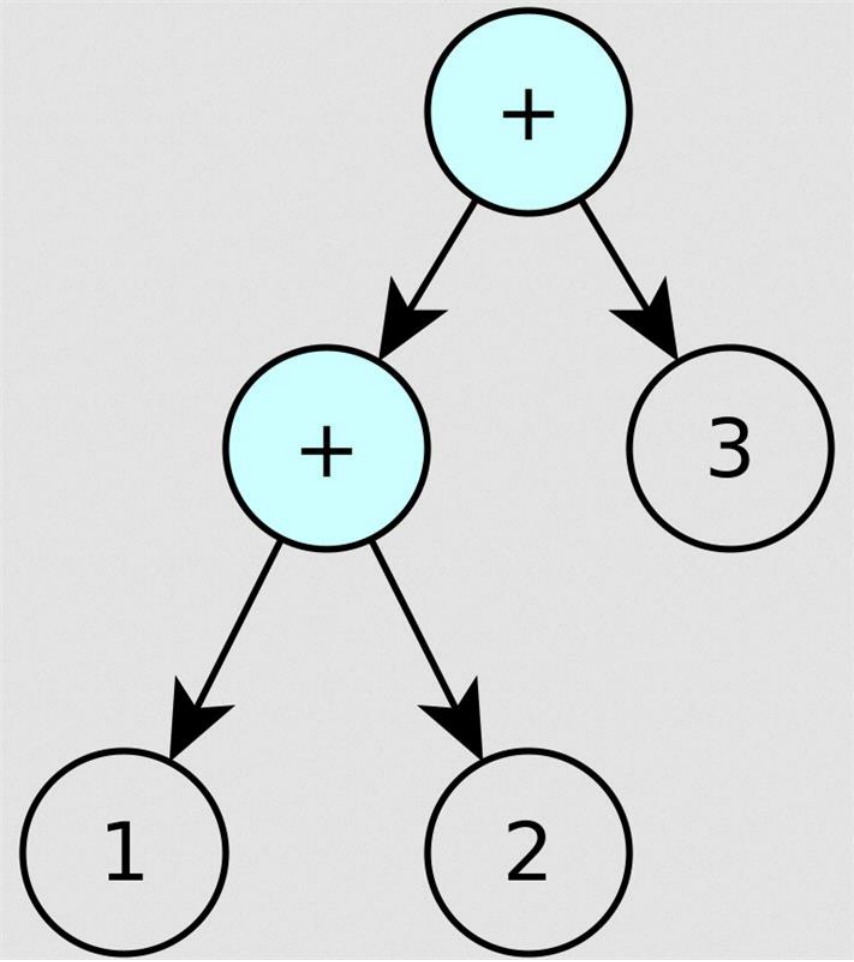
Para a explicação destes conceitos, vamos usar uma gramática muito simples (notação eBNF), onde temos apenas uma regra que define uma expressão que pode ser um número ou uma soma de dois números.

```
A -> N "+" N | N
```

Voltando ao exercício 1 apresentado no tutorial, queremos extender esta gramática de forma a permitir o encadeamento de somas. A soma, como explicado antes, deve ter associatividade à esquerda, e a forma de termos associatividade à esquerda especificada na gramática é usando recursividade à esquerda. Algo como:

```
A -> A "+" N | N
```

Para expressões como "1+2+3", isto gera uma árvore que faz com que as operações mais à esquerda sejam avaliadas primeiro:



O problema é que o **JavaCC21** gera parsers para gramáticas do tipo **LL**, que **não permitem recursividade à esquerda**. Se tiverem uma regra desta forma, o parser é gerado mas, ao fazer parsing, vai entrar num loop infinito até parar com uma exceção de stack overflow.

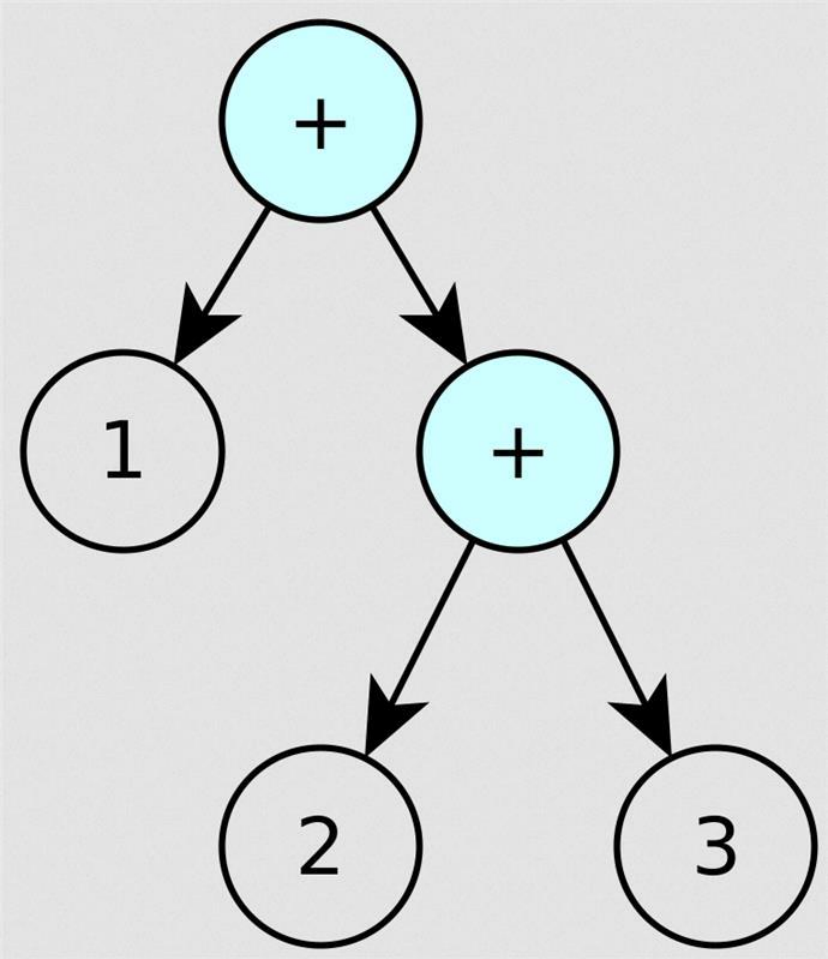
Podemos remover este problema reescrevendo a gramática, transformando-a em:

```
A -> N A'
A' -> "+" N A' | ε
```

Ou mesmo em:

```
A -> N "+" A | N
```

Mas agora temos outro problema. Como temos recursividade à direita, o parsing de expressões como "1+2+3" gera uma árvore que faz com que as operações mais à esquerda sejam avaliadas em último lugar:



Desta forma deixamos de ter associatividade à esquerda, o que poderá levar a resultados errados.

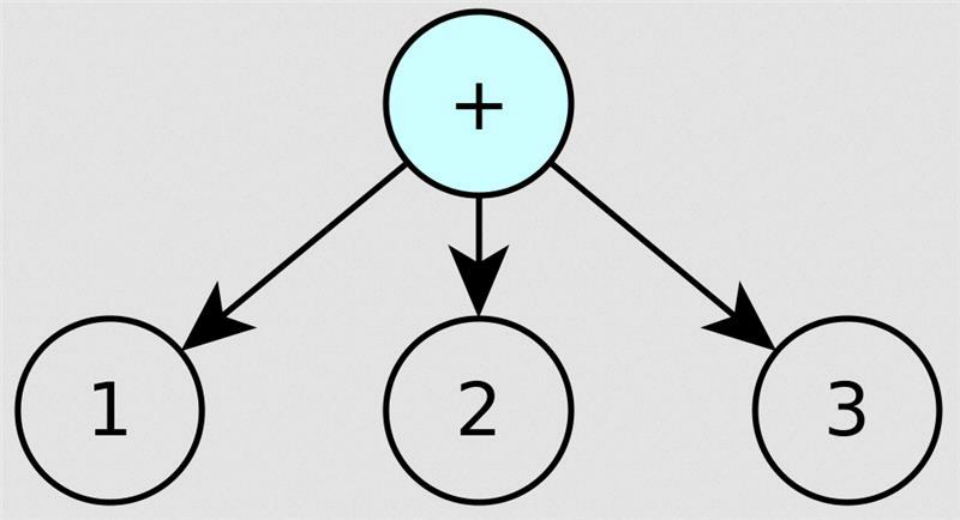
Podemos optar por uma versão com repetição, iterativa e sem recursividade, dada, por exemplo, pela gramática:

```
A -> N {"+" N}
```

A tradução para JavaCC21 seria muito direta, usando * para indicar repetição:

```
A : N ("+" N)*;
```

Com o JavaCC21, isto vai gerar uma árvore como a seguinte, em que os filhos estão ao mesmo nível:



Enquanto esta árvore é sintaticamente correta, como são todas as anteriores, está desprovida de significado, de semântica. Não só não existe o conceito de associatividade, como a soma deixa de ser uma operação binária.

A forma de lhe dar essa semântica é um processamento pós-parsing, onde os nós seriam agrupados como na primeira árvore. Iterativamente percorreríamos a lista de operandos, da esquerda para direita, e contruiríamos nós binários corretamente. Esta é a solução. Felizmente, o JavaCC21 tem um "atalho". As diretivas do tipo #NodeName(Number), que já têm sido usadas, são um atalho para a manipulação e reestruturação da árvore, evitando o pós-processamento.

A definição correta em JavaCC21 é:

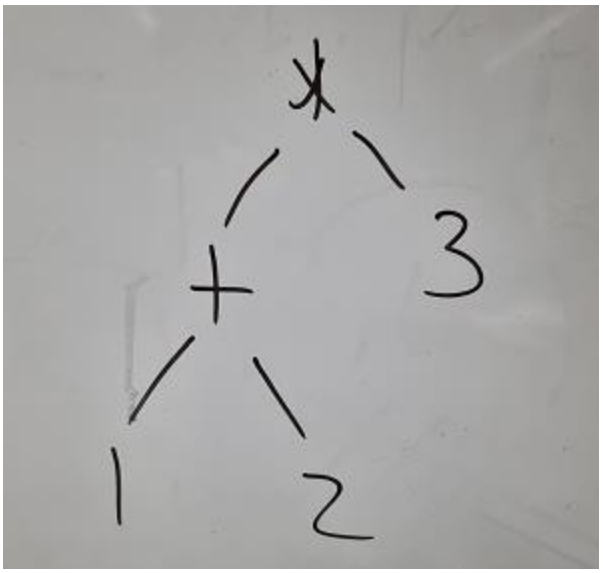
```
A : N ("+" N)* #Add(2);
```

Estamos a dizer ao parser para, iterativamente e da esquerda para a direita, ir formando nós do tipo Add, cada um deles com os 2 nós mais recentemente gerados. Estes nós serão do tipo N ou Add, sendo garantido, pela ordem de iteração, que se houver um filho do tipo Add, é sempre o primeiro filho. A associatividade é garantida e conseguimos exatamente a árvore que desejamos! 🙂

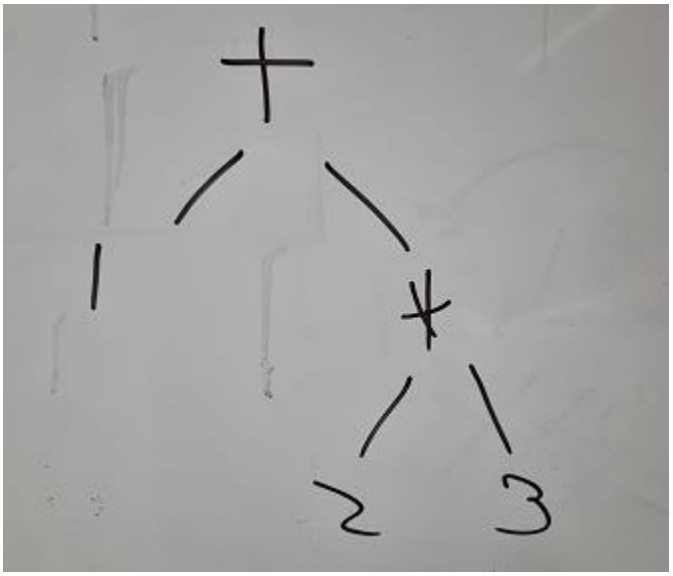
Operator Precendence in JavaCC

Summary: how to implement operator precedence in JavaCC

When you have several chained operations, such as `1 + 2 * 3`, there is an order of operations that must be respected. If operator precedence is not respected, you will obtain the following AST:



This AST first performs the addition, and then the multiplication. However, multiplication has precedence over addition, and you should obtain the following AST instead:



*Tip: notice that if * has higher precedence that +, + will never appear as a child of *, unless inside parenthesis. This will be important later.*

Now the multiplication is done before the addition. To implement operator precedence in JavaCC, we will use the same technique as in the calculator from the tutorial. If you do not support operator precedence yet, probably you will have to completely rewrite your expression rule.

The main idea is to define a grammar rule for each priority level (Java operators precedence), start at the lowest priority level, and at each level we can only call the rule of the next priority level.

Tip: All elements in the expression grammar rule are considered operators, including new, dot and [].

Simple Precedence Example

Consider a grammar with the operators `+,-,*,[],()`, with the same priority level as in Java. Since `+` has the lowest priority, we could start with the following rule:

```
Expr:
 AdditiveExpr
 ;
```

Additive level has two operators in this example, `+` and `-`:

```
AdditiveExpr:
 MultiplicativeExpr ( ("+" | "-" ) MultiplicativeExpr)*
 ;
```

Since the next priority level is Multiplicative, and Additive expression can only have operands of Multiplicative priority or higher. Next is MultiplicativeExpr:

```
MultiplicativeExpr:
 ArrayAccess ("*" ArrayAccess)*
;
```

The same idea. Notice that as we go deeper in the priority level, the more restricted the operands can be. When we arrive at the last priority level and there are no more operators, the operands have to be expression terminals, also solving the problem of left recursion:

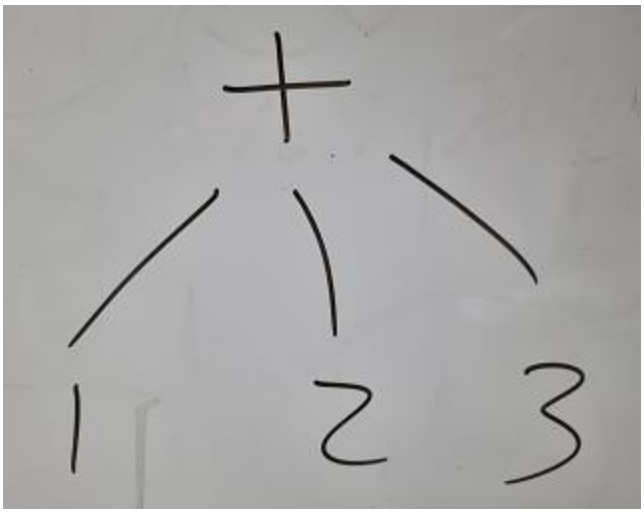
```
ArrayAccess:
 ExprTerminal ("[" Expr "]")*
 ;

ExprTerminal:
 IntLiteral |
 "(" Expr ")"
;

IntLiteral:
 < INTEGER >
 ;
```

Tip: consider 1 + a[0]. Because of operator precedence, even if the initial grammar allows `Expression[Expression]`, with precedence we will never capture `1+a` as the left operand of the array access, only the `a`.

This way it is possible to implement operator precedence, however we still have a few problems with the AST.

Restructuring the AST

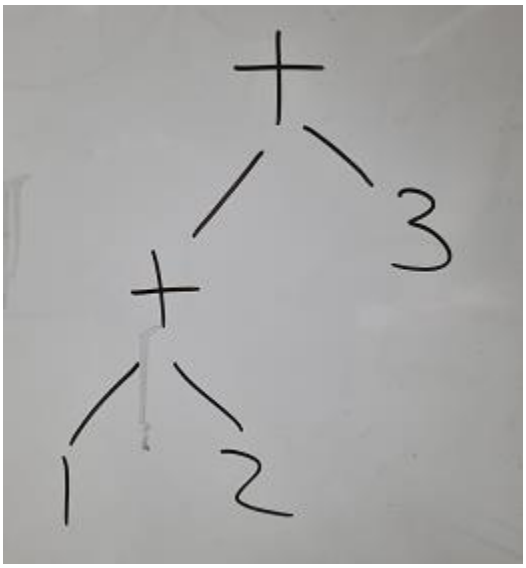
With the current grammar, if we parse `1+2+3` we will obtain an AST similar to:



We want operators such as + to be binary operators, since this will helps us during AST processing. To do this, when there is an operation, we will create a new node and capture the two already created nodes as children of the new node:

AdditiveExpr:
 MultiplicativeExpr (("+" | "-") MultiplicativeExpr **#BinOp(2)**)*
 ;

This way we will obtain an AST similar to this:



Cleaning the AST

The operations are now in the correct order, but the AST obtained by this grammar should actually be looking like this:

Expr
 AdditiveExpr

```
BinOp
 BinOp
  MultiplicativeExpr
   ArrayAccess
    ExprTerminal
     IntLiteral
   MultiplicativeExpr
    ArrayAccess
     ExprTerminal
      IntLiteral
 MultiplicativeExpr
  ArrayAccess
   ExprTerminal
    IntLiteral
```

There are a lot more nodes than in the previous example. Most of these nodes are not necessary, since with only the IntLiteral and the BinOp nodes we will have all the necessary nodes to reconstruct the expression `1+2+3`. To remove the unnecessary nodes, with use the **#void** directive:

```
Expr #void:
 AdditiveExpr
 ;
```

By adding **#void** to all nodes except IntLiteral (and relying on our created BinOp), we obtain the following AST:

```
BinOp
 BinOp
   IntLiteral
   IntLiteral
 IntLiteral
```

AST Annotations

The structure now looks good, however we are not storing information about what operations are done, nor the values of the operands. We will need to use the variable `jjtThis` to access the node being built to store the information we want. With some restructuring, we can write `AdditiveExpr` like this:

```
AdditiveExpr #void:
 MultiplicativeExpr ( ("+" MultiplicativeExpr) ({ jjtThis.put("op", "add"); }) #BinOp(2)
 | ("-" MultiplicativeExpr) ({ jjtThis.put("op", "sub"); }) #BinOp(2)
 )*
 ;
```

And `IntLiteral` like this:

```
IntLiteral:
 < INTEGER > { jjtThis.put("value", lastConsumedToken.getImage()); }
;
```

Tip: If you create a node using #, `jjtThis` refers to the first node that appears on its right. If the rule does not create a node with #, `jjtThis` refers to the node created by the rule itself, e.g. `IntLiteral`

If you apply all these steps, you will obtain an AST that is clean, annotated and well-structured, and respects the precedence of operators.

```
BinOp (op: add)
 BinOp (op: add)
   IntLiteral (op: 1)
   IntLiteral (op: 2)
 IntLiteral (op: 3)
```

Final Remarks for Checkpoint 1

Summary: error handling, JmmNode, operator precedence, AST clean-up, AST annotation, AST Dump with Annotated Values

Error Handling

One of the requests for the first checkpoint is to capture exceptions thrown by the JavaCC21 parser. We have been requesting you to differentiate the exception between a Lexical and a Syntactical problem. However, since it might not be trivial to analyze this difference, we now only want you to simply process the error to report the line and column of the error, and just report the problem as "Syntactical". In a nutshell, what we want is a try-catch specifically for the "ParseException" class, the class that is automatically generated by JavaCC. This is simply a new catch in the "SimpleParser" class that customizes a new Error report. The idea is as follows:

- Catch a ParseException.
- Retrieve line and column from the token inside the Parse exception (e.getToken()).
- Get error message from the ParseException.
- Create a new error report using the previous information, with Stage type "SYNTACTIC".
- If you added support for testing specific rules, where there is an error the program will throw a generic RuntimeException instead of a ParseException. You have to obtain the ParseException by traveling the chain of causes, using the method .getCause(). Alternatively, you can use the method TestUtils.getException(e, ParseException.class) to directly obtain the ParseException instance (or null if no ParseException is found).

The following code is pretty much the code you need for the error report, assuming 'ex' is any Throwable (e.g. ParseException, RuntimeException) and 'e' is a ParseException:

```
var e = TestUtils.getException(ex, ParseException.class)
```

```
// ... test if ‘e’ is null and handle ‘ex’ in a more generic way
Token t = e.getToken();
int line = t.getBeginLine();
int column = t.getBeginColumn();
String message = e.getMessage();
Report report = Report.newError(stage, line, column, message, e);
return JmmParserResult.newError(report);
```

JmmNode Implementation

For the students that still did not implement the JmmNode interface:

- The code should be identical to the one provided in the tutorial, so it should be a very straight forward copy-paste.
- The tutorial provides 2 INJECTS: one for the Token and another for the Node. For the project you will only need the INJECT code for the Node.
- Ensure you have the option `TOKENS_ARE_NODES = false;` at the beginning of your .jj file.

Operator Precedence

- Ensure the generated AST respects Java operator precedence (e.g. 1 + 2 * 3 will first multiply 2 by 3, and only add 1 after).
- This post which explains how you can implement the rule “expression” in order to have operator precedence, as well as other AST-related transformations.

AST Cleanup

This step is basically to have the AST as clean as possible, without redundant nodes. For instance the following tree has nodes that do not add meaningful information to the tree, for the code `if(2<3)`:

IfStatement
 Expression
 Expr1
 Expr2
 Lessthan
 2
 3

You can see that the nodes `Expression`, `Expr1` and `Expr2` do not add any information here, as we know that the first child of the if statement must be an expression. A clean version of the tree would be:

IfStatement
 Lessthan

2
3

AST annotation

Add all the relevant information provided by Tokens inside the nodes. Examples:

- Name of a variable
- Value of an integer or boolean
- Is the declared variable an array?
- If the class extends another class and which

AST Dump with Annotated Values

The AST returned by the parser uses JavaCC nodes (e.g. BaseNode) that do not print the values that are annotated in the nodes. Instead of using the method .dump() , use the JmmNode method .toTree(), after converting the AST to JmmNodeImpl nodes.

To convert the AST, use the method .sanitize(), it will return an AST with nodes that print the annotated values. This is only needed if you want to print the AST inside SimpleParser.java, when you pass a JmmNode to the constructor JmmParserResult, .sanitize() is automatically called.

In summary, to print the AST with annotated values you can use code like this:

```
var root = ((JmmNode) parser.rootNode()).sanitize();
System.out.println(root.toTree());
```

Checkpoint 2

Implementing the Remaining Stages

By the end of Checkpoint 2, you will be generating Jasmin code, which means that you will be implementing the three remaining stages of the pipeline. Each stage has an interface to implement: JmmAnalysis, JmmOptimization and JasminBackend. You must create a class for each of these stages, each one implementing one of the interfaces.

For instance, consider you will implement the JmmAnalysis interface using a new class JmmAnalyser. Initially you might have code similar to the following:

```
package pt.up.fe.comp;

import java.util.Collections;
```

```
import pt.up.fe.comp.jmm.analysis.JmmAnalysis;

import pt.up.fe.comp.jmm.analysis.JmmSemanticsResult;

import pt.up.fe.comp.jmm.analysis.table.SymbolTable;

import pt.up.fe.comp.jmm.parser.JmmParserResult;


public class JmmAnalyser implements JmmAnalysis {

    @Override

    public JmmSemanticsResult semanticAnalysis(JmmParserResult parserResult) {
SymbolTable symbolTable = null;


        return new JmmSemanticsResult(parserResult, symbolTable,
Collections.emptyList());

    }

}
```

The interface contains a single method, semanticAnalysis, that receives the JmmParserResult from the previous stage, JmmParser, and returns a JmmSemanticsResult. The JmmSemanticsResult receives the JmmParserResult, an instance of SymbolTable that you will have to create, and a list of new reports originated in this stage.

To add the new stage to your compiler, go to your `Launcher` class and add the new analysis stage:

```
// ... previous code


// Parse stage

JmmParserResult parserResult = parser.parse(input, config);


// Check if there are parsing errors

TestUtils.noErrors(parserResult.getReports());


// Instantiate JmmAnalysis

JmmAnalyser analyser = new JmmAnalyser();


// Analysis stage
```

```
JmmSemanticsResult analysisResult = analyser.semanticAnalysis(parserResult);

// Check if there are parsing errors
TestUtils.noErrors(analysisResult.getReports());

// ... add remaining stages
```

You will need to do similar steps for the remaining two stages, JmmOptimization and JasminBackend.

Enabling Stages for Tests

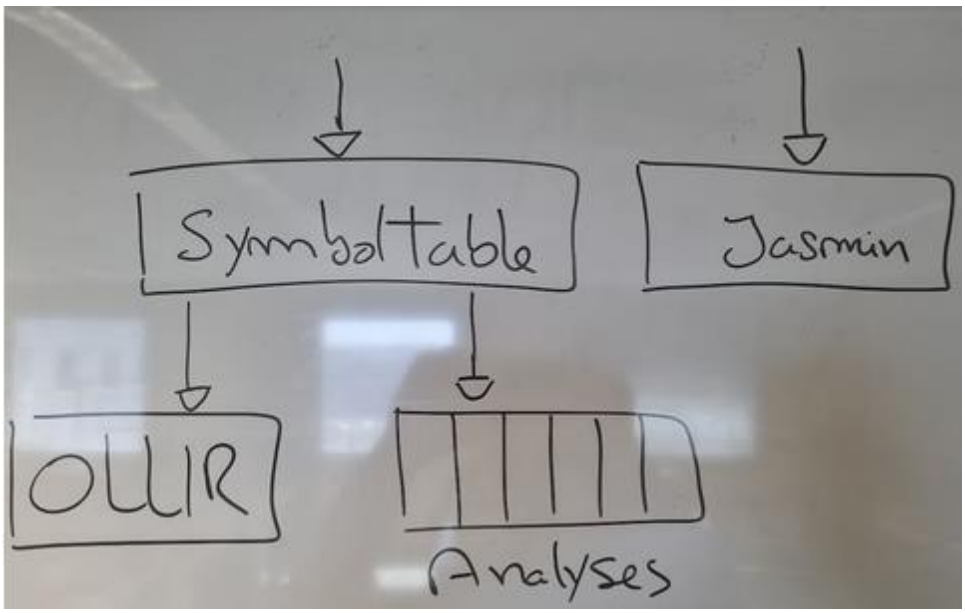
There are methods in TestUtils for each of new stages, they are `TestUtils.analyse(String jmmCode)` for `JmmAnalysis`, `TestUtils.optimize(String jmmCode)` for `JmmOptimization` and `TestUtils.backend(String jmmCode)` for `JasminBackend`. For these functions to work, you have to update the file `config.properties` that is in the root of your repository. In each of these test functions for a specific stages all of the preceding stages are called. For instance, when you call optimize, first the parser is tested, then the analyser is tested, and finally, the optimizer is tested.For each stage, you must provide the fully qualified name (i.e., package name + class name) of your class that implements it. For instance, considering the class `JmmAnalyser` introduced in the previous section, we would change the file `config.properties` like this:

```
# The fully qualified name of your class that implements the interface JmmParser
ParserClass = pt.up.fe.comp.SimpleParser
# The fully qualified name of your class that implements the interface JmmAnalysis
AnalysisClass = pt.up.fe.comp.JmmAnalyser
# The fully qualified name of your class that implements the interface
JmmOptimization
OptimizationClass =
# The fully qualified name of your class that implements the interface JasminBackend
BackendClass =
```

NOTE: When running these methods if you have an error with a message such as `Could not instantiate JmmAnalysis from class ''`, this means the stage has not been declared in the file `config.properties`.

Work as a Team

At this stage of the project, it is possible (and highly recommended) that you split tasks between the team members. Right from the start, it is possible to work on the SymbolTable and the Jasmin generation at the same time. After the SymbolTable is done, it is possible to start work on the semantic analysis and OLLIR generation, and several of the analyses can also be developed independently of each other. The diagram below illustrates these dependencies:



NOTE: When generating OLLIR code, it will be very useful to have a function/visitor that receives an Expression node and returns the Type of the expression. This function/visitor will be important both for OLLIR generation and the analyses you will perform over the AST. Take into account that such a function will need the symbol table, as well as knowing in which function the expression is.

If you want to implement the stage `JasminBackend` and generate Jasmin code before implementing the stage `JmmOptimization`, to test the `JasminBackend` stage you cannot use the method `TestUtils.backend(String jmmCode)`, since it will call the `JmmOptimization` stage. Instead, use the method overload `TestUtils.backend(OllirResult ollirResult)`, that only calls the `JasminBackend` stage. Use the `OllirResult` constructor that receives a String with OLLIR code and the configuration map:

```
public OllirResult(String ollirCode, Map<String, String> config)
```

Your repository already contains several OLLIR examples, in the folder `test/fixtures/public/ollir`. To use them, read the files as a resource, as in `ParserTest`, to create an `OllirResult`:

```
new OllirResult(SpecsIo.getResource("fixtures/public/ollir/Fac.ollir"),
Collections.emptyMap())
```

Symbol Table

The symbol table is a data structure that stores information related to the variables and other symbols in the source code. For instance, which variables are declared inside a given method, or if the class extends another class.

You will have to implement the interface `SymbolTable`, which contains methods that retrieve that kind of information (e.g., `List<String> getImports()`, `String getClassName()`). All methods of the interface are documented in the code, please refer to code to see what each method should return.

You are free to implement the SymbolTable as you see fit. One possible way to implement it is to use Map instances internally in your implementation of the SymbolTable, to map information between a method signature, and information related to that method. Another way is to create a class that stores information related to just a single method and use a single Map that maps signatures to instances of that class.

You do not need to support method overloading (I.e. when two or more methods have the same name but different parameters), so using the name of the method as its signature is sufficient in this case. If you want to support method overloading as an extra, consider that your method signature must have information about the name of the method and its parameters.

To build the symbol table you will need to get the information from the AST. You can either visit the AST manually (e.g., code that checks the type of the node and visits the children) or use the Visitor pattern (e.g. extend AJmmVisitor). Since the language is relatively simple, the analysis does not need to go very deep in the AST, and both approaches are viable.

Semantic Analysis Checklist

The following are the analyses that we will test and that must report an error.

Symbol Table

- Has information about imports and the declared class

- Has information about extends, fields and methods
- Has information about the parameters and local variables of each method

Type Verification

- *(new)* Verify if variable names used in the code have a corresponding declaration, either as a local variable, a method parameter or a field of the class (if applicable).
- Operands of an operation must types compatible with the operation (e.g. `int + boolean` is an error because + expects two integers.)
- Array cannot be used in arithmetic operations (e.g. `array1 + array2` is an error)
- Array access is done over an array
- Array access index is an expression of type integer
- Type of the assignee must be compatible with the assigned (`an_int = a_bool` is an error)
- Expressions in conditions must return a boolean (`if(2+3)` is an error)

Function Verification

- When calling methods of the class declared in the code, verify if the types of arguments of the call are compatible with the types in the method declaration
- In case the method does not exist, verify if the class extends another class and report an error if it does not. Assume the method exists in one of the super classes, and that is being correctly called
- When calling methods that belong to other classes other than the class declared in the code, verify if the classes are being imported

OLLIR and Jasmin

We have made available the OLLIR documentation provided to students in the past year, you can find it in Programming Project -> Files -> OLLIR. Regarding Jasmin, please refer to the tool's website, as well as this initial example and a reference of Java bytecode instructions.

We will be providing additional information regarding OLLIR and Jasmin, that will be collected in the FAQ.

By the end of Checkpoint 2, it is expected that you can generate OLLIR and Jasmin for:

- Basic class structure (including constructor <init>)
- Class fields
- Method structure (you can ignore stack and local limits for now, use `limit_stack 99` and `limit_locals 99`)
- Assignments
- Arithmetic operations (with correct precedence)
- Method invocation

Regarding the implementation of the stage `JmmOptimization`, it contains three methods, one method called `toOllir` and two methods called `optimize`. For Checkpoint 2 you only need to implement the method `toOllir` (this is documented in the code of `JmmOptimization`).

When generating OLLIR code, you only need to generate the string representing the code. You should use the following constructor for `OllirResult`, that receives the `JmmSemanticsResult` from the previous stage, the string with the OLLIR code, and a list of new reports for threportis stage:

```
public OllirResult(JmmSemanticsResult semanticsResult, String ollirCode,
List<Report> reports)
```

When generating Jasmin code, a similar situation applies, use the constructor for `JasminResult` that receives the `OllirResult` from the previous stage, the string with Jasmin code, and a list of new reports for this stage:

```
public JasminResult(OllirResult ollirResult, String jasminCode, List<Report>
reports)
```

AST Visitor Example - Import Collector

In this example we will consider a simple task, collect all import strings of the AST to a list, to explain different ways of accomplishing this with the JmmVisitor interface that is available.

We will use the following JMM code for this example:

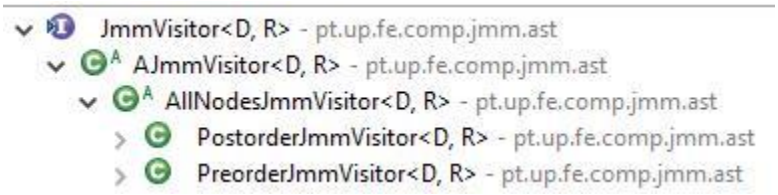
```
import foo.bar;
import a.b.c;
import d.e.f.g;

class ImportCollector {

// ...remaining code
```

And an AST with the following structure, for the code above:

```
Program
 ImportDecl
  Id (name: foo)
  Id (name: bar)
 ImportDecl
  Id (name: a)
  Id (name: b)
  Id (name: c)
 ImportDecl
  Id (name: d)
  Id (name: e)
  Id (name: f)
  Id (name: g)
 ClassDecl (name: ImportCollector)
 ... remaining AST
```

JmmVisitor Hierarchy

The image below shows the hierarchy class of `JmmVisitor` classes:



- `JmmVisitor`: Interface that represents all visitors of JmmNodes
- `AJmmVisitor`: Abstract class that implements several functions of `JmmVisitor`, and can be used to implement visitors that require control over how nodes are visited
- `AllNodesJmmVisitor`: Abstract class for visitors that always visit all nodes
- `PreorderJmmVisitor`: Implementation of `AllNodesJmmVisitor` that visits all nodes using a preorder traversal: first the parent, then the children
- `PostorderJmmVisitor`: Implementation of `AllNodesJmmVisitor` that visits all nodes using a postorder traversal: first the children, then the parent

When creating visitors, you will be usually extending either `AJmmVisitor` or `PreorderJmmVisitor`.

Note: `JmmVisitor` and its descendants are templated classes, which accept two parameters, D and R, which represent the type of the input object (D - data) and the type of the return object (R - return) of the `visit()` method, respectively. When implementing one of these classes, usually you will be setting these types. Often, you will only use one, or none, in those cases use a dummy type, such as Boolean or Integer.

Import Collector using PreorderJmmVisitor

First, we will see how to implement the import collector using a `PreorderJmmVisitor`. There are three main ways for collecting information with a visitor: 1) by using an object parameter, 2) by returning an object, or 3) by using an internal variable. We will see examples for the three cases.

Import Collector Using Parameter

The code below fully implements the import collector using a `PreorderJmmVisitor`. Let's go over the code and see what it does:

```
public class ImportCollectorWithParam extends PreorderJmmVisitor<List<String>,
Boolean> {

 public ImportCollectorWithParam() {
  addVisit("ImportDecl", this::visitImport);
 }

 private Boolean visitImport(JmmNode importDecl, List<String> imports) {

  var importString = importDecl.getChildren().stream()
   .map(id -> id.get("name"))
   .collect(Collectors.joining("."));

  imports.add(importString);

  return true;
 }
}
```

- Since we are collecting the imports by using a parameter, we have set the input data type D to `List<String>`, and since we are not returning anything, we have set the return value R to the dummy type `Boolean` (line 1, `extends PreorderJmmVisitor<List<String>, Boolean>`)
- We have added a single visit function, associated with the node `ImportDecl` (line 4)
- The visit function creates a stream with the children of the `ImportDecl` node, which are always of type `Id`, for each `Id` returns the corresponding name of the `Id`, and finally creates a string by joining all the names, inserting a dot between them (lines 9-11)
- The string representing the import is added to the parameter `imports`, which is a `List<String>` that is passed to all nodes during their visit (line 13)

To use this visitor, we can use code such as the following assuming the root node of the AST is in the variable `rootNode`:

```
var importCollector = new ImportCollectorWithParam();
var imports = new ArrayList<String>();
importCollector.visit(rootNode, imports);
```

If we print the variable `imports` after running this code, we obtain something like this:

```
[foo.bar, a.b.c, d.e.f.g]
```

Import Collector Using Return

If instead of passing a list of string to collect the imports, we want the visitor to build the list and return it, we can use the code below (changes regarding the first example are highlighted):

```
public class ImportCollectorWithReturn extends PreorderJmmVisitor<Boolean,
List<String>> {

 public ImportCollectorWithReturn() {
  addVisit("ImportDecl", this::visitImport);

  setDefaultValue(() -> Collections.emptyList());

  setReduceSimple(this::reduceSimple);
 }

 private List<String> reduceSimple(List<String> list1, List<String> list2) {
  return SpecsCollections.concatList(list1, list2);
 }

 private List<String> visitImport(JmmNode importDecl, Boolean dummy) {

  var importString = importDecl.getChildren().stream()
   .map(id -> id.get("name"))
   .collect(Collectors.joining("."));

  return Arrays.asList(importString);
 }
}
```

- Now we are returning a list of strings and not using the input parameter, so we have set the input data type D to the dummy type `Boolean`, and the return type R to `List<String>` (line 1)
- We now rely on the return value, so we need to set what default value should be returned by the nodes that do not have a visit function. In this case, they return an empty list (line 6)
- Each node will now return a list, we have to indicate how we should merge each pair of results (line 7). In this case, we will be concatenating the lists (lines 10-12)
- The visit for `ImportDecl` works in a similar way, but instead of adding the import string to an existing list, it returns a new list with one element, the import string (line 20)

To use this visitor, we can write the following and obtain the same output:

```
var importCollector = new ImportCollectorWithReturn();
var imports = importCollector.visit(rootNode, true);
```

Import Collector Using Field

Finally, you can use the internal state of the visitor to collect information, as the code below. When using this technique, be careful when reusing the same visitor instance, if you do not clear the internal state before the visit, you can obtain unexpected results.

```
public class ImportCollectorWithField extends PreorderJmmVisitor<Boolean,
Boolean> {

 private final List<String> imports;

 public ImportCollectorWithField() {
  imports = new ArrayList<>();

  addVisit("ImportDecl", this::visitImport);
 }

 public List<String> getImports() {
  return imports;
 }

 private Boolean visitImport(JmmNode importDecl, Boolean dummy) {

  var importString = importDecl.getChildren().stream()
   .map(id -> id.get("name"))
   .collect(Collectors.joining("."));

  imports.add(importString);

  return true;
 }
}
```

- Now we are not using the input parameter and the return value, so the data type D and the return type R are set to the dummy type `Boolean` (line 1)
- We declare an instance variable `imports`, which will collect the imports (line 3)
- The instance variable needs to be initialized (line 6) and we add a getter, to be able to access it (lines 11-13)
- The visit for `ImportDecl` works in a very similar way, but now we add the string to the internal variable of the visitor (line 21)

To use this visitor, we can write the following and obtain the same output:

```
var importCollector = new ImportCollectorWithField();
importCollector.visit(results.getRootNode(), true);
var imports = importCollector.getImports();
```

Import Collector using AJmmVisitor

Consider the import collector implemented with the PreorderJmmVisitor, which extends AllNodesJmmVisitor. Even if the imports are only at the beginning of the tree, it will always visit all nodes when collecting the imports.

The code below is a modified version of the visitor that stores the imports in a parameter that it receives, that can now also returns how many nodes has visited:

```
public class ImportCollectorWithParamAndCounter extends
PostorderJmmVisitor<List<String>, Integer> {

 private int visits;

 public ImportCollectorWithParamAndCounter() {
  this.visits = 0;

  addVisit("ImportDecl", this::visitImport);

  setDefaultVisit((node, values) -> ++visits);
 }

 private Integer visitImport(JmmNode importDecl, List<String> imports) {

  var importString = importDecl.getChildren().stream()
   .map(id -> id.get("name"))
   .collect(Collectors.joining("."));

  imports.add(importString);

  return ++visits;
 }
}
```

- The return type changed from Boolean to Integer, to return the number of nodes visited. The visitor type was also changed to PostorderJmmVisitor, so that the last node visited is the root, returning the total number of visits (line 1)
- There is a new field, visits, to store the number of visits at that point (line 3), which is initialized to 0 (line 6)
- We set a default visit function that simply increments the visits counter, and returns the updated value (line 10)
- We change the visit method for ImportDecl to also return Integer (line 13) and the incremented value of visits (line 21)

We can call this new implementation of the import collector the following way:

```
var importCollector = new ImportCollectorWithParamAndCounter();
var imports = new ArrayList<String>();
var visits = importCollector.visit(rootNode, imports);
```

If we print the variables imports and visits, we can obtain the following output:

```
[foo.bar, a.b.c, d.e.f.g]
140
```

To collect the imports, that are at the beginning of the tree, the collector visited 140 nodes (all nodes of the tested AST). If we want to only visit the nodes related to the imports, we need to control what nodes are visited, and for that we need to implement AJmmVisitor instead.

Below is the code for a possible implementation of a import collector that visits only the necessary nodes:

```
public class ImportCollectorOptimized extends AJmmVisitor<List<String>, Integer> {

 private int visits;

 public ImportCollectorOptimized() {

  this.visits = 0;

  addVisit(AstNode.PROGRAM, this::visitProgram);
  addVisit(AstNode.IMPORT_DECL, this::visitImportDecl);

  setDefaultVisit((node, imports) -> ++visits);
 }

 private Integer visitProgram(JmmNode program, List<String> imports) {
  for (var child : program.getChildren()) {
   visit(child, imports);
  }

  return ++visits;
 }

 private Integer visitImportDecl(JmmNode importDecl, List<String> imports) {

  var importString = importDecl.getChildren().stream()
   .map(id -> id.get("name"))
   .collect(Collectors.joining("."));

  imports.add(importString);

  return ++visits;
 }
}
```

- Now the class implements AJmmVisitor instead of PostorderJmmVisitor (line 1)
- When implementing a AJmmVisitor, if a node is not explicitly visited, it will not be visited at all, so we need to add a visit function to the root node, Program (line 8), so that its children, which includes the

imports, are visited (node that if we set as default visit function a function that visits all the children of the visited node, we are reimplementing the behaviour of AllNodesJmmVisitor)
- We implement the visit function for the node Program, which simply visits its children (lines 14-20)

To run this implementation we can use the same code, just changing the instantiation of the importCollector, obtaining the following output:

```
[foo.bar, a.b.c, d.e.f.g]
5
```

Only 5 nodes where visited this time: the Program node and its 4 children (3 ImportDecl and 1 ClassDecl).

Conclusion

The import collector is very simple, and was used to illustrate the several kinds of visitors you might need to build. For this particular example, the first one, that accepts a list as a parameter, is arguably the simplest one, and might be the pattern you will be using more frequently when building visitors that visit all nodes. However, the other examples where presented to illustrate alternative ways of building a visitor.

In the case of the AJmmVisitor you have to be careful to explicitly visit all the nodes you intend to visit, otherwise they will not be visited.

Jasmin Generation

You can immediately start working on the OLLIR to Jasmin generation, by using and instance of `OllirResult`, which receives a string with OLLIR code, parses it, and returns a `ClassUnit`, which represents the root node from where you can start navigating the OLLIR Intermediate Representation (IR). You have several examples of OLLIR code in the folder `test/fixtures/public/ollir`.

When in doubt, reverse engineer it

Although we only provide one example of Jasmin code, you can generate as many examples of Java bytecodes as necessary. Jmm code is (almost) compatible with Java, so you can compile a Jmm file as if it was a Java file, and then decompile it.

For instance, consider HelloWorld.jmm in the fixtures examples. First, we rename HelloWorld.jmm to HelloWorld.java and use `javac` to compile it:

```
javac HelloWorld.java
```

However, if we try to compile it, we will have an error like this:

```
.\HelloWorld.java:1: error: '.' expected
 import ioPlus;
```

```
 ^
 1 error
```

HelloWorld.jmm uses an import that is in the default package (has no package), and that is not allowed in Java. We have to comment/remove that import and compile again. However, now we have another error:

```
HelloWorld.java:4: error: cannot find symbol
 ioPlus.printHelloWorld();
 ^
 symbol: variable ioPlus
 location: class HelloWorld
 1 error
```

We need to tell `javac` where `ioPlus` is. This is a library to assist your Jasmin code, and it is located in the folder `libs-jmm\compiled`. We have to provide that folder as part of the class path, with the flag `cp`:

```
javac -cp ./libs-jmm/compiled HelloWorld.java
```

Note: if you want to add more that one folder to the class path, you need to separate the folders with a character that depends on the OS you are. If you are in Windows, you use `;` (e.g. -cp lib1;lib2), in Linux you use `:` (e.g. -cp lib1:lib2)

Now we have compiled the Java code to Java bytecode, and we should have a file `HelloWorld.class` in the folder. To decompiled it and obtain the Java bytecodes, we can use the command `javap`:

```
javap -c HelloWorld.class
```

This should output the Java bytecodes for the class HelloWorld:

```
Compiled from "HelloWorld.java"
 class HelloWorld {
 HelloWorld();
 Code:
 0: aload_0
 1: invokespecial #1 // Method java/lang/Object."<init>":()V
 4: return

public static void main(java.lang.String[]);
 Code:
 0: invokestatic #7 // Method ioPlus.printHelloWorld:()V
 3: return
 }
```

Jasmin vs Java bytecodes

The code above is similar to the Jasmin string you have to generate from OLLIR, but is not quite the same. If you notice, calls to instructions `invokestatic` and `invokespecial` reference numbers. This is because the Java Virtual Machine (JVM) uses a Constant Pool, which is similar to a symbol table, and bytecodes directly reference this pool.

Storing information in the Constant Pool and keeping track of the references can be cumbersome when generating and verifying code, so we will use Jasmin, which abstracts tasks like this, and allows to write the same instructions in this manner:

```
invokestatic ioPlus/printHelloWorld()V
```

Notice that instead of referencing a number, we can just indicate the full qualified name of the class and the method to call. Jasmin contains several other differences like this, please use this site as a reference on how to write Jasmin instructions, and use this site as a reference for Java bytecfode instructions.

Example of Jasmin/Java bytecodes instructions

You will be converting OLLIR code, which is register based, to Java bytecodes, which is stack based. You still have "registers", in the form of local variables, but most Java bytecode instructions operate with a stack, reading and storing values there.

Here are some examples of Jasmin/Java bytecode instructions that you will need to use in this project:

- `ldc <constant>`: loads a constant to the top of the stack
- `ia dd`: pops the two top values in the stack, adds them as integers, and stores the result on the top of the stack
- `aload <local_var_number>`: reads the value that is in the local variable with the given number, interpreting it as a reference, and puts the value on the top of the stack
- `iaload`: takes two elements from the stack, the first is interpreted as a reference to an array, and the second as an index, stores on the top of the stack the value of the array in the corresponding index

Some notes

- The letters at the beginning of the instruction usually indicate the type handled by that instruction. `i` is integer, `a` is object, `ia` is integer array
- Local variables already contain data when you enter a method. If the method is an instance method (as opposed to a static method), local variable 0 contains a reference to `this`. If a method has parameters, each next local variable contains the corresponding parameter. For instance, if you have a Java instance method `foo(int a, int b)`, local variable 0 contains `this`, local variable 1 contains `a` and local variable 2 contains b

Testing Jasmin code

After you generate a string of Jasmin code, we recommend that you use the class `JasminResult` to test it. It has a constructor that simply receives a string with the Jasmin code, and has a method `compile()`, which will compile the Jasmin code and report if there was any compilation problem.

You can also run the Jasmin code! `JasminResult` has a `run()` method that it will execute your code, and has overloads that accept a list of arguments, and even simulated inputs, in case you want to create tests for code that requires user inputs.

Final Delivery

This is an overview of the work you are expected to have done by the due date. We will provide further details in the future.

For 18/20

In order to be eligible for a maximum final project grade of 18 out of 20, you are expected to have everything up to CP2 as well as the following:

- Generation of OLLIR and Jasmin for:
  - if and if/else
  - loops
  - arrays
- Ensure the expected flow of the compiler, going through every compilation step. You should accept a .jmm source code file and generate a .class file in the end;
- Ensure the generated .class file runs correctly within the JVM;
- Ensure the provided unit tests pass and provide your own test examples to highlight the features of your compiler;
- Computation of limits for the JVM:
  - stack
  - local variables

For 20/20

In order to be eligible for a maximum final project grade of 20 out of 20, you are expected to have everything described above as well as the following:

- Register allocation (60%) (-r <reg> flag)
  - this and method parameters are always allocated to the same registers
  - the main method does not have this since it is static
- Other predefined optimizations (20%) (-o flag)
  - constant propagation
  - elimination of unnecessary gotos (e.g., replacing whiles with do-whiles)
- Other optimization you may see fit (20%) (-o flag)
  - usage of more efficient JVM instructions
  - constant folding
  - common subexpression elimination
  - loop-invariant code motion
  - dead code elimination