

# **Compiladores - Projecto**

## **iJava**

João Ricardo Lourenço, N° 2011151194  
Joaquim Pedro Bento Gonçalves Pratas Leitão, N° 2011150072

29 de Maio de 2014

### **Relatório**

## Índice

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Análise Lexical</b>	<b>5</b>
<b>3</b>	<b>Análise Sintática</b>	<b>8</b>
<b>4</b>	<b>Construção da Árvore de Sintaxe Abstracta</b>	<b>9</b>
<b>5</b>	<b>Análise Semântica</b>	<b>10</b>
5.1	Tabelas de Símbolos . . . . .	10
5.2	Deteção de Erros Semânticos . . . . .	11
<b>6</b>	<b>Geração de Código</b>	<b>12</b>
<b>7</b>	<b>Apreciação do Trabalho</b>	<b>13</b>

# 1 Introdução

O presente trabalho pretende-se desenvolver um compilador para a linguagem *iJava*, um pequeno subconjunto da linguagem Java (versão 5.0). Por ser um subconjunto de uma outra linguagem, todos os programas que respeitem as regras impostas em *iJava* são também, garantidamente, programas válidos em *Java*.

Nesta linguagem todos os programas são constituídos por uma única classe, que possui métodos e atributos estáticos, e públicos. Para além disso, a classe necessita obrigatoriamente de ter um método *main*, onde a execução do programa se inicia.

Podemos utilizar literais dos tipos inteiro e booleano e variáveis inteiras, booleanas e arrays uni-dimensionais de inteiros e booleanos.

A linguagem implementa também expressões aritméticas e lógicas, operações relacionais simples, instruções de atribuição e controlo (*if – else* e *while*).

Os métodos definidos, e os respetivos valores de retorno, podem ser de qualquer tipo acima mencionado, com exceção do método *main*, que tal como em *Java* possui como tipo de retorno o tipo *void*.

É também possível passar parâmetros (literais inteiros) ao nosso programa através da linha de comandos. É o método *main* que vai receber esses parâmetros, armazenando-os num array de objetos do tipo *String*. Embora este tipo de dados não esteja incluído na lista de tipos permitidos em *iJava*, a sua utilização apenas é permitida no método *main*, com a mera finalidade de obter os parâmetros passados ao programa aquando da sua invocação.

O desenvolvimento do compilador foi dividido em três fases distintas.

Numa primeira fase foi realizada a *Análise Lexical* do programa fonte, onde são identificados *tokens*, isto é, cadeias pertencentes à linguagem e que têm significado e relevância para o programa.

Seguiu-se a *Análise Semântica*, composta por quatro etapas principais: **FIXME**, **MUDAR A MANEIRA COMO ESTAMOS A DIZER O QUE SE SEGUE**

- **Tradução da gramática-fonte** (fornecida em notação *EBNF*) para o yacc 3 e realização da **Análise Sintática** do programa, permitindo assim reconhecer se as sequências de *tokens* que o constituem pertencem à linguagem, permitindo-nos assim detetar eventuais erros de sintaxe.
- **Construção da árvore de sintaxe abstrata**, etapa que é realizada em simultâneo com a *Análise Sintática*. A árvore de sintaxe abstrata irá representar o nosso programa a compilar, recorrendo a uma estrutura em árvore para representar as estruturas sintáticas das cadeias que constituem o programa a compilar.

- **Construção da tabela de símbolos**, utilizadas para armazenar informações relevantes sobre a classe (seus atributos e métodos), bem como sobre cada método definido pelo programador (como, por exemplo, o tipo de retorno e os argumentos).
- **Verificação de erros semânticos**, etapa principal da *Análise Semântica*, onde são realizadas verificações de tipos, garantindo que para cada operação a realizar não existem incompatibilidades de tipos entre os operandos nela envolvidos.

A última fase do trabalho consistiu na *Geração de Código Intermédio*, da qual resulta, na representação intermédia de *LLVM*, um programa equivalente ao que pretendemos compilar.

METER IMAGEM BONITINHA, TIPO CACEIRO???

## 2 Análise Lexical

Tal como referimos anteriormente, na *Análise Lexical* procedemos à identificação dos *tokens* da nossa linguagem. Sempre que é detectada a presença de um comentário no programa a compilar, seja do tipo `//...` (comentários de apenas uma linha) ou do tipo `/*...*/` (comentários multi-linha), os caracteres incluídos nesse comentário são ignorados.

Sempre que é detectado um caracter, ou uma sequência de caracteres, que não constitui nenhum *token* é detectado um erro lexical, sendo impressa uma mensagem de erro, indicando a existência de um caracter ilegal, juntamente com a sua posição no programa.

Adicionalmente, caso se verifique a ocorrência de um comentário multi-linha que não foi devidamente terminado, o erro lexical é também detectado, sendo impressa uma mensagem de erro que indica a posição no programa onde o comentário foi iniciado.

Em seguida, apresentamos a lista dos *tokens* válidos na linguagem *iJava* e a lista dos *tokens* reservados que, por essa razão, não estão disponíveis na nossa linguagem:

- **ID**: Sequências alfanuméricas (maiúsculas e minúsculas) começadas por uma letra, podendo conter também símbolos como `_` e `$`
- **INTLIT**: Sequências de dígitos decimais e hexadecimais (incluindo a-f e A-F) precedidas de `0x`. Este *token* pode ser descrito também na forma da seguinte expressão regular: `[0-9]+—0x[0-9a-fA-F]+`
- **BOOLLIT**: `true` | `false`
- **INT**: `int`
- **BOOL**: `boolean`
- **NEW**: `new`
- **IF**: `if`
- **ELSE**: `else`
- **WHILE**: `while`
- **PRINT**: `System.out.println`
- **PARSEINT**: `Integer.parseInt`
- **CLASS**: `class`
- **PUBLIC**: `public`
- **STATIC**: `static`

- **VOID:** *void*
- **STRING:** *String*
- **DOTLENGTH:** *.length*
- **RETURN:** *return*
- **OCURV:** (
- **CCURV:** )
- **OBRACE:** {
- **CBRACE:** }
- **OSQUARE:** [
- **CSQUARE:** ]
- **OP1:** && |||
- **OP2:** <|>|==|!|=|<=|>=
- **OP3:** " + " | −"
- **OP4:** " \* " | "/" | "%"
- **NOT:** "!"
- **ASSIGN:** " = "
- **SEMIC:** ";"
- **COMMA:** ", "
- **RESERVED:** *abstract | continue | for | switch | assert | default | goto | package | synchronized | do | private | this | break | double | implements | protected | throw | byte | import | throws | case | enum | instanceof | transient | catch | extends | short | try | char | final | interface | finally | long | strictfp | volatile | const | float | native | super | null | ++ | --*

Quando implementámos a *Análise Sintática*, para resolver problemas de ambiguidade da gramática foi necessário, entre outras ações que iremos abordar na próxima secção, separar os *tokens* **OP1**, **OP2**, **OP3** e **OP4** nas diferentes sequências alfanuméricas que os constituíam. Assim, temos ainda os seguintes *tokens*:

- **AND** ("&&") e **OR** ("||"), originados a partir do *token* **OP1**

- **LE** ( $\ll$ ), **GE** ( $\gg$ ), **EQ** ( $=$ ), **NEQ** ( $\neq$ ), **LEQ** ( $\leq$ ) e **GEQ** ( $\geq$ ), originados a partir do *token* **OP2**
- **PLUS** ( $+$ ) e **MINUS** ( $-$ ), originados a partir do *token* **OP3**
- **MULT** ( $*$ ), **DIV** ( $/$ ) e **MOD** ( $\%$ ), originados a partir do *token* **OP4**

### 3 Análise Sintática



## 4 Construção da Árvore de Sintaxe Abstracta

## 5 Análise Semântica

### 5.1 Tabelas de Símbolos

## 5.2 Detecção de Erros Semânticos

## 6 Geração de Código

## 7 Apreciação do Trabalho