

UNIVERSIDADE DE COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

COMPILADORES 2013/2014

---

# Compilador para linguagem iJava

---

*Autor:*

David CARDOSO

Número: 2011164039

*Autor:*

Bruno CACEIRO

Número: 2008107991

28 de Maio de 2014

# Índice

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Análise Lexical</b>	<b>3</b>
2.1	Tokens . . . . .	3
2.1.1	Tratamento de Erros . . . . .	5
<b>3</b>	<b>Análise Sintática e Semântica</b>	<b>6</b>
3.1	Gramática . . . . .	6
3.1.1	Estado Final da Gramática . . . . .	6
3.2	Árvore de Sintaxe Abstrata . . . . .	8
3.2.1	Exemplo . . . . .	9
3.3	Análise Semântica . . . . .	9
3.4	Tabela de Símbolos . . . . .	10
3.5	Tratamento de Erros Semânticos . . . . .	11
<b>4</b>	<b>Geração de Código</b>	<b>11</b>

# 1 Introdução

Este projecto consiste no desenvolvimento de um compilador para a linguagem *iJava* (imperative Java), que consiste num pequeno subconjunto da linguagem Java (versão 5.0). Os programas da linguagem *iJava* são constituídos por uma única classe (a principal), contendo necessariamente um método *main*, e podendo conter outros métodos e atributos, todos eles estáticos e (possivelmente) públicos.

O projecto foi estruturado em 3 fases, primeiramente foi feita a Análise Lexical, implementada na linguagem *C* e utilizando a ferramenta *lex*. A segunda fase consistiu na análise sintática, com a construção da árvore de sintaxe abstrata e análise semântica (tabelas de símbolos, deteção de erros semânticos). No final foi feita a geração de código.

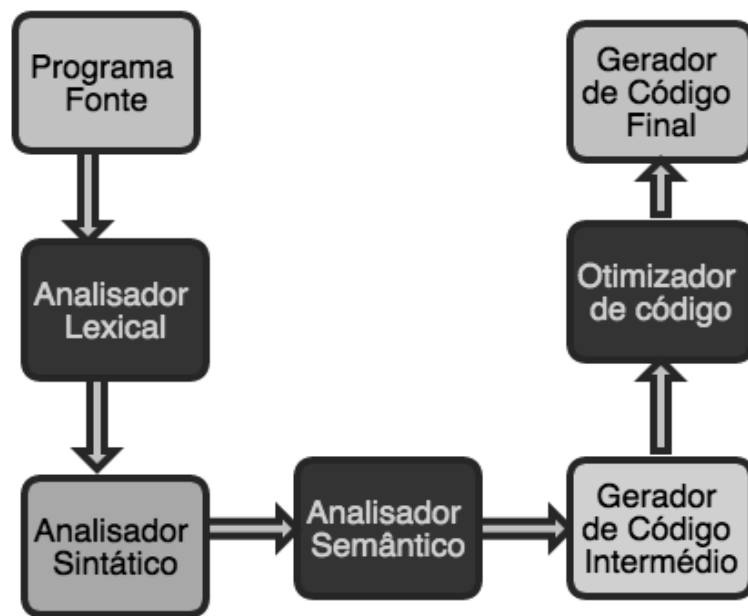


Figura 1: Fases de Compilação

## 2 Análise Lexical

A Análise Lexical consiste em analisar a entrada de linhas de caracteres e produzir uma sequência de símbolos (*tokens*) que podem ser manipulados mais facilmente por um *parser*. É uma forma de verificar um determinado alfabeto, neste caso o alfabeto da linguagem *iJava*. Esta análise pode ser dividida em três fases:

- Extração e classificação de *tokens*;
- Eliminação de delimitadores e comentários;
- Tratamento de erros;

### 2.1 Tokens

- **ID:** Sequências alfanuméricas começadas por uma letra, onde os símbolos "\_" e "\$" contam como letras. Maiúsculas e minúsculas são consideradas letras diferentes
- **INTLIT:** Sequências de dígitos decimais e sequências de dígitos hexadecimais (incluindo a-f e A-F) precedidas de "0x"
- **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: ”]”

Foi necessário separar alguns *tokens* devido às diferentes prioridades que cada operador tem.

- **OP1:** "&&"
- **OP1OR:** "|"
- **OP2:** "<" | ">" | "<=" | ">="
- **OP2EQS:** "==" | "!="
- **OP3:** "+" | "-"
- **OP4:** "\*" | "/" | "%"
- **NOT:** "!"
- **ASSIGN:** "="
- **SEMIC:** ";"
- **COMMA:** ","
- **RESERVED:** O *iJava* é um subconjunto da linguagem *Java*, como tal, existe um conjunto de funcionalidades que embora não sejam suportadas, têm de ser consideradas. Assim, foi necessário tratar todo um conjunto de palavras reservadas de forma a permitir que sejam lexicalmente válidas mas não sintaticamente.
  - abstract | assert | break | byte | case | catch | char | const | continue | default | do | double | enum | extends | final | finally | float | for | goto | implements | import | instanceof | interface | long | native | package | private | protected | short | strictfp | super | switch | synchronized | this | throw | throws | transient | try | volatile | null | ++ | -

### 2.1.1 Tratamento de Erros

Se forem detectados erros lexicais no ficheiro de entrada então é impressa uma mensagem de erro no *stdout*:

- "Line<num linha>,col<num coluna>:illegal character('<c >'\\n)"
- "Line<num linha>,col<num coluna>:unterminated comment\\n"

## 3 Análise Sintática e Semântica

### 3.1 Gramática

A gramática é a maneira formal de especificar a sintaxe de uma linguagem. Desenvolver uma gramática não ambígua é um dos passos mais importantes para o sucesso do compilador. Para a gramática da linguagem *iJava* usámos a notação **BNF** (*Backus Naur Form*). A gramática que nos foi dada era ambígua e por isso tivémos de efectuar diversas alterações para permitir a análise sintática ascendente com o *yacc*.

Algumas das alterações que efectuámos foram:

- Usar recursividade à direita, nas regras onde era possível existirem uma ou mais repetições
- Criação de estados adicionais para as regras que continham *tokens* opcionais

#### 3.1.1 Estado Final da Gramática

A gramática final, utilizada no *yacc* é a seguinte apresentada.

$$\text{START} \rightarrow \text{CLASS ID OBRACE field\_or\_method\_declaration CBRACE}$$
$$\begin{aligned} \text{field\_or\_method\_declaration} &\rightarrow \text{FieldDecl field\_or\_method\_declaration} \\ &| \text{MethodDecl field\_or\_method\_declaration} \end{aligned}$$
$$\text{FieldDecl} \rightarrow \text{STATIC VarDecl VarDecl\_REPETITION}$$
$$\begin{aligned} \text{MethodDecl} &\rightarrow \text{PUBLIC STATIC method\_type\_declaration ID OCURV Formal-} \\ &\text{Params CCURV OBRACE VarDecl\_REPETITION statement\_declaration\_REPETITION} \\ &\text{CBRACE} \end{aligned}$$
$$\begin{aligned} \text{method\_type\_declaration} &\rightarrow \text{Type} \\ &| \text{VOID} \end{aligned}$$
$$\text{FormalParams} \rightarrow \text{Type ID several\_FormalParams}$$
$$| \text{STRING OSQUARE CSQUARE ID}$$
$$\text{several\_FormalParams} \rightarrow \text{COMMA Type ID several\_FormalParams}$$
$$\text{VarDecl\_REPETITION} \rightarrow \text{VarDecl VarDecl\_REPETITION}$$

VarDecl  $\rightarrow$  Type *ID* several\_var\_decl\_in\_same\_instruction OPTIONAL *SEMIC*

several\_var\_decl\_in\_same\_instruction OPTIONAL  $\rightarrow$  *COMMA ID* several\_var\_decl\_in\_same\_instruct

Type  $\rightarrow$  *INT OSQUARE CSQUARE*

| *BOOL OSQUARE CSQUARE*  
| *INT*  
| *BOOL*

statement\_declaration\_REPETITION  $\rightarrow$  Statement statement\_declaration\_REPETITION

Statement  $\rightarrow$  *OBRACE* several\_statement *CBRACE*

| *IF OCURV Expr CCURV Statement %prec IFX*  
| *IF OCURV Expr CCURV Statement ELSE Statement*  
| *WHILE OCURV Expr CCURV Statement*  
| *PRINT OCURV Expr CCURV SEMIC*  
| *ID array\_index OPTIONAL ASSIGN Expr SEMIC*  
| *RETURN return\_expression SEMIC*

several\_statement  $\rightarrow$  Statement several\_statement

array\_index OPTIONAL  $\rightarrow$  *OSQUARE Expr CSQUARE*

return\_expression  $\rightarrow$  Expr

IndexableExpr  $\rightarrow$  *ID*

| *INTLIT BOOLLIT ID OCURV Args\_ OPTIONAL CCURV*  
| *OCURV Expr CCURV*  
| Expr *DOTLENGTH*  
| IndexableExpr *OSQUARE Expr CSQUARE*  
| *PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV*

Expr  $\rightarrow$  Expr *OP1* Expr *%prec OP1*

| Expr *OP1OR Expr %prec OP1OR*  
| Expr *OP4 Expr %prec OP4*  
| Expr *OP3 Expr %prec OP3*  
| Expr *OP2 Expr %prec OP2*  
| Expr *OP2EQS Expr %prec OP2EQS*  
| *OP3 Expr %prec NOT*



```

| NOT Expr %prec NOT
| NEW INT OSQUARE Expr CSQUARE
| NEW BOOL OSQUARE Expr CSQUARE
| IndexableExpr

```

$\text{Args\_OPTIONAL} \rightarrow \text{Args}$

$\text{Args} \rightarrow \text{Expr comma\_expr}$

$\text{comma\_expr} \rightarrow \text{COMMA Expr comma\_expr}$

## 3.2 Árvore de Sintaxe Abstrata

Para a árvore de sintaxe abstrata optámos por utilizar um nó genérico (*Node*) com a seguinte estrutura:

```

1  /* General Node */
2  typedef struct _Node
3  {
4      //Type of the Node (to identify the type of the node)
5      NodeType n_type;
6
7      //Type of the Struct (Int, Void, String,...)
8      Type type;
9
10     //Id or list of id's
11     listID* id;
12
13     //The tree next nodes (the case of if)
14     struct _Node* n1;
15     struct _Node* n2;
16     struct _Node* n3;
17
18     //Next node
19     struct _Node* next;
20
21     //Literals (to store the values)
22     char* value;
23
24     char isStatic;
25 }Node;
26
27
28 /* Linked list of ID's (for multiple declaration of variables) */
29 typedef struct _idList
30 {
31     char* id;
32     struct _idList* next;
33 }listID;

```

Listing 1: Estruturas para representação de Nós da Árvore

### 3.2.1 Exemplo

Considerando o seguinte programa:

```
1 class gcd {  
2   public static void main(String [] args) {  
3     int x;  
4   }  
5 }
```

Listing 2: Programa Exemplo

A árvore gerada é a seguinte:

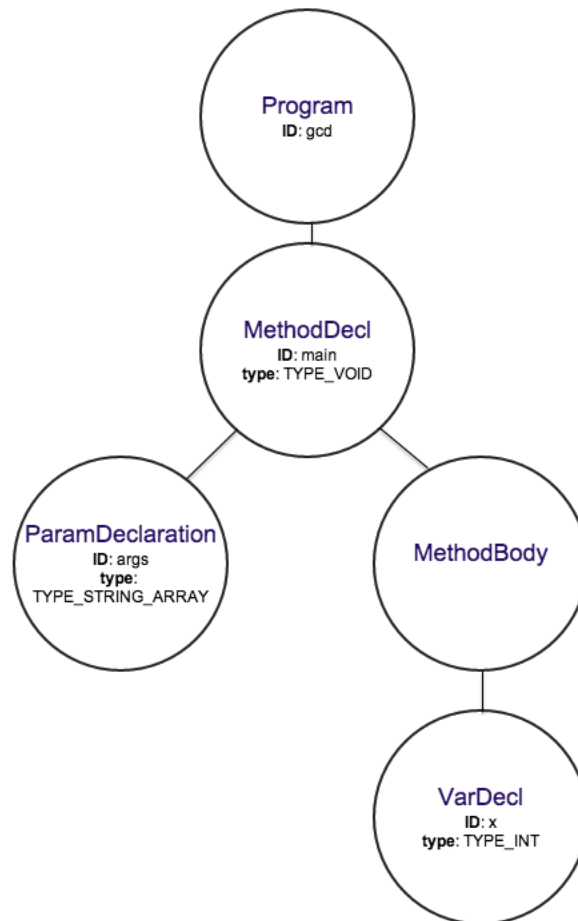


Figura 2: Árvore de Sintaxe Abstrata para o Programa Exemplo

## 3.3 Análise Semântica

A Análise Semântica tem como principais objectivos a ligação das definições de variáveis com a sua utilização, a verificação da correcção de tipos, declarações e chamadas de funções. Esta análise é dividida em duas fases:

1. Para cada *scope* no programa:

- Processar as declarações:
    - Adicionar novas entradas na tabela de símbolos
    - Apresentar mensagem de erro caso haja variáveis repetidas
  - Processar os *statements*
    - Procurar variáveis que não foram declaradas neste *scope* ou no *scope* global, e caso não tenham sido, apresentar mensagem de erro
2. Processar todos os *statements* do programa outra vez
- Usar a tabela de símbolos para determinar o tipo de cada expressão e procurar erros de tipo (atribuições, cálculos, contagem e tipos de argumentos correctos,...)

### 3.4 Tabela de Símbolos

Para representação da tabela de símbolos, foram utilizadas as seguintes estruturas:

```

1  /* Table Node */
2  typedef struct _TableNode
3  {
4      //Type of the Node (to identify the type of the node)
5      TableType n_type;
6
7      //Type of the Struct (Int, Void, String,...)
8      Type type;
9
10     //ID or list of id's
11     listID* id;
12
13     //Next node
14     struct _TableNode* next;
15
16     //If is a param
17     char isParam;
18 }TableNode;
19
20 /* Table */
21 typedef struct _Table{
22     TableNode* table;
23     struct _Table* next;
24 }Table;

```

Listing 3: Estruturas para representação da Tabela de Símbolos

### 3.5 Tratamento de Erros Semânticos

Para o tratamento de erros semânticos utilizamos as seguintes funções:

```
1  /* Check if global variables have the same name as the Methods:
2     - If they have print error message: "Symbol %s already defined" and exit
3  */
4  void    checkIfExists(char* id, Table* local);
5  void    checkSemanticErrors(Node* ast, Table* local, Table* main);
6  void    checkErrors(Node* ast, Table* symbols, Table* main);
7
8  /* Check if ID exists:
9     - If exists return the Type
10    - If doesn't exist, print error message: "Cannot find symbol %s\n" and exit
11  */
12  int     checkifIDExists(char* id, TableType type, Table* table, Table* main);
13  Table*  getMethodTable(Table* main, char* methodID);
14
15  /* Check if the Literal is valid (decimal / hexadecimal / octal)
16     - If it is invalid print error message: "Invalid literal %s\n" and exit
17  */
18  void    validIntLit(char* lit);
19  void    checkTypes(Node* ast, Table* main);
20  void    operatorError2Types(int op, int n1, int n2);
21  void    operatorError1Types(int op, int n1);
22  void    assignmentError(char* var, int n1, int n2);
23  void    assignmentErrorArray(char* var, int n1, int n2);
24  void    setTable(Table* oi);
25  int     getFunctionType();
26  void    statementError(int op, int n1, int n2);
27  void    statementError1oranother(int op, int n1, int n2, int n3);
28  char*   getFunctionName();
29  void    getErrorCall(int i, char* name, int n1, int n2);
```

Listing 4: Funções para tratamento de erros semânticos

## 4 Geração de Código