

UNIVERSIDADE DE COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

COMPILADORES 2013/2014

Compilador para linguagem iJava

Autor:

David CARDOSO

Número: 2011164039

Autor:

Bruno CACEIRO

Número: 2008107991

2 de Junho de 2014

Índice

1	Introdução	2
2	Análise Lexical	3
2.1	Tokens	3
2.1.1	Comentários	5
2.1.2	Tratamento de Erros	5
3	Análise Sintática e Semântica	6
3.1	Gramática	7
3.1.1	Gramática Inicial	7
3.1.2	Gramática Final	8
3.2	Tratamento de Erros Sintáticos	11
3.3	Árvore de Sintaxe Abstrata	12
3.3.1	Estruturas	12
3.3.2	Criação da Árvore	14
3.3.3	Exemplo	15
3.3.4	Impressão da Árvore	16
3.4	Análise Semântica	17
3.5	Tabela de Símbolos	18
3.5.1	Estruturas	18
3.5.2	Criação da Tabela de Símbolos	19
3.5.3	Impressão da Tabela de Símbolos	19
3.6	Tratamento de Erros Semânticos	21
4	Geração de Código	23

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, recorrendo ao *yacc/bison*, 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, em *LLVM*.

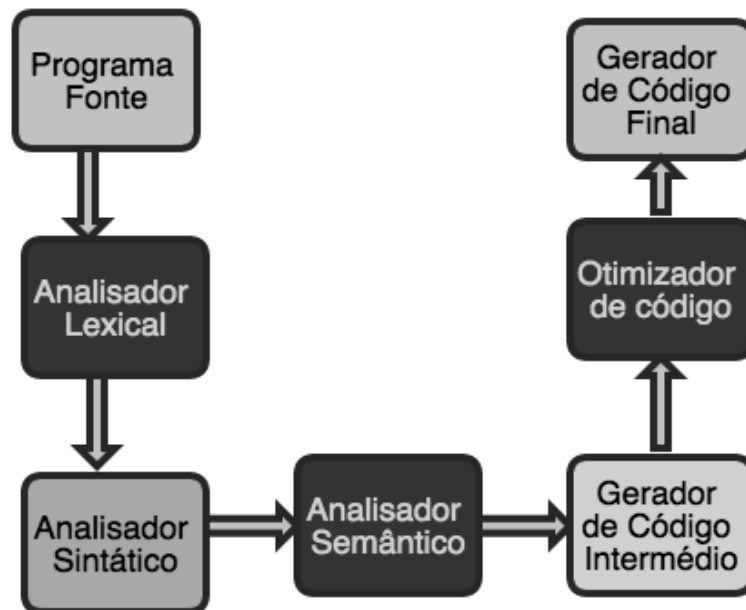


Figura 1: Fases de Compilação

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*. Assim, 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 ID's diferentes (*case sensitive*).
 - **Expressão Regular:** $[a - zA - Z_\$]([a - zA - Z_\$0 - 9])^*$
- **INTLIT:** Sequências de dígitos decimais e sequências de dígitos hexadecimais (incluindo a-f e A-F) precedidas de "0x"
 - **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:** "]"
- **ASSIGN:** "="
- **SEMIC:** ";"
- **COMMA:** ","

Para a Análise Sintática foi necessário separar alguns *tokens* devido às diferentes prioridades que cada operador tem. Assim, os *tokens* foram agrupados pela ordem de precedência de operações.

- **OP1:** "&&"
- **OP1OR:** "|"
- **OP2:** "<" | ">" | "<=" | ">="
- **OP2EQS:** "==" | "!="
- **OP3:** "+" | "-"
- **OP4:** "*" | "/" | "%" | "
- **NOT:** "!"

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.

- **RESERVED:**

- 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 Comentários

Existem duas maneiras de fazer comentários:

- Comentar apenas uma linha `//<código>`.
Caso seja detectado (`//`) todo o código que se segue é ignorado até encontrar uma mudança de linha.

Expressão Regular: `//".*`

- Comentar um bloco de código `/*<código>*/`

```
1  < COMMENT > <<EOF>> {BEGIN 0}
2  < COMMENT > "*/"      {BEGIN 0}
3  < COMMENT > "\n"      {}
4  < COMMENT > .          {}
5  "/*"                  {BEGIN COMMENT}
6
```

Listing 1: Detecção de Comentários

Caso seja detectado (`/*`) todo o código é ignorado até que seja encontrado o seu correspondente (`*/`). Caso seja detectado o EOF então é apresentada uma mensagem de erro: `"Line %d, col %d: unterminated comment"`. Foi criado um estado adicional no *lex* para poder tratar esta situação. Foi a criação deste estado que permitiu ignorar todos os *tokens* presentes dentro do comentário.

2.1.2 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"`

Para podermos imprimir as mensagens de erro com o número da linha e coluna criámos uma variável *column* para poder contar as colunas e utilizamos a variável *yylineno*, disponibilizada pelo *yacc*, para sabermos o número das linhas. Para a contagem de colunas aumentamos a variável referente às colunas consoante o tamanho de cada *token*. Quando há uma mudança de linha, inicializa-se o contador das colunas a 1.

Para tratar o caso dos comentários criámos duas variáveis adicionais (*Commentline*, *Commentcolumn*) para poder guardar a coluna e linha onde o comentário é inicializado.

3 Análise Sintática e Semântica

De forma a realizar a análise sintática foi utilizada a ferramenta *lex*, para reconhecer e isolar os *tokens*, sendo que de seguida serão enviados para o *yacc* que irá ser responsável por verificar se estes pertencem à gramática da linguagem.

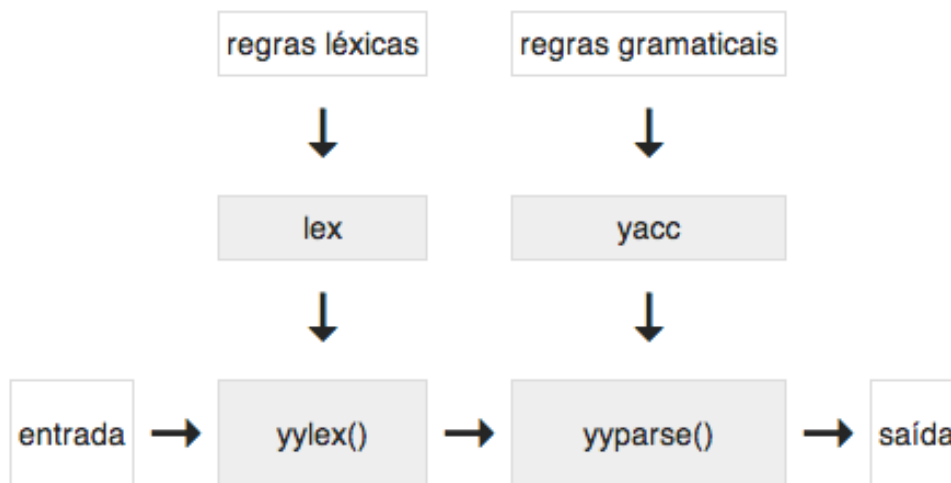


Figura 2: Relacionamento entre *lex* e *yacc*, retirado de <http://pt.wikipedia.org/wiki/Yacc>

Para realizar a ligação entre o *yacc* e o *lex* foram definidos *tokens* no *yacc*, posteriormente importados pelo *lex*. Sempre que o *lex* detecta uma sequência de caracteres correspondente a um *token* válido retorna um código acordado entre o *yacc* e *lex* de forma a identificar de forma única esse *token* (*enums*), sendo este valor enviado para o *yacc*. Caso o *token* corresponda a um tipo passível de ser processado (*INTLIT*, *BOOLIT*, *ID* e *operadores*), é guardado na variável *yyvar* o valor introduzido pelo utilizador.

Sendo a variável *yyval* também responsável por definir o tipo de retorno de cada regra da gramática, foi necessário acrescentar outros tipos de dados, de forma a permitir a criação da árvore de sintaxe abstrata. Assim, foi criada uma *union*, definida no *yacc*, que permite partilhar, no mesmo espaço de memória, vários tipos diferentes.

```
1 %union{
2     struct _Node* node;
3     char* token;
4     struct _idList* listId;
5     int type;
6 }
```

Listing 2: Union

Para permitir a percepção, pelo *yacc*, da linha e coluna que estão atualmente a serem utilizadas, foram também declaradas várias variáveis externas, partilhadas pelo *lex*.

```
1 extern int column;
2 extern int yylineno;
3 extern char* yytext;
4 extern int yyleng;
```

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 de um compilador. Para a gramática da linguagem *iJava* usamos a notação **BNF** (*Backus Naur Form*), visto que a mesma é utilizada pelo *yaac* e permite remover as ambiguidades.

3.1.1 Gramática Inicial

```
Start → Program
Program → CLASS ID OBRACE FieldDecl | MethodDecl CBRACE
FieldDecl → STATIC VarDecl
MethodDecl → PUBLIC STATIC ( Type | VOID ) ID OCURV
           [ FormalParams ] CCURV OBRACE VarDecl Statement CBRACE
FormalParams → Type ID COMMA Type ID
FormalParams → STRING OSQUARE CSQUARE ID
VarDecl → Type ID COMMA ID SEMIC
Type → ( INT | BOOL ) [ OSQUARE CSQUARE ]
Statement → OBRACE Statement CBRACE
Statement → IF OCURV Expr CCURV Statement [ ELSE Statement ]
Statement → WHILE OCURV Expr CCURV Statement
Statement → PRINT OCURV Expr CCURV SEMIC
Statement → ID [ OSQUARE Expr CSQUARE ] ASSIGN Expr SEMIC
Statement → RETURN [ Expr ] SEMIC
Expr → Expr ( OP1 | OP2 | OP3 | OP4 ) Expr
Expr → Expr OSQUARE Expr CSQUARE
Expr → ID | INTLIT | BOOLLIT
Expr → NEW ( INT | BOOL ) OSQUARE Expr CSQUARE
Expr → OCURV Expr CCURV
Expr → Expr DOTLENGTH | ( OP3 | NOT ) Expr
Expr → PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV
Expr → ID OCURV [ Args ] CCURV
Args → Expr COMMA Expr
```

Lembramos que, em notação **ENBF**, os símbolos [...] englobam *tokens* opcionais e {...} implicam a repetição dos *tokens* 0 ou mais vezes.

3.1.2 Gramática Final

A gramática que nos foi dada era ambígua e por isso tivemos de efetuar diversas alterações para permitir a análise sintática ascendente com o *yacc*.

Algumas das alterações que efetuámos foram:

- Criação de estados adicionais para as regras que implicam a repetição de *tokens*.
- Criação de estados adicionais para as regras que continham *tokens* opcionais
- Estabelecimento de regras de prioridade de forma a gerir regras de precedência entre operadores
- Definição das regras de associação dos operadores (à esquerda/direita)

```
1 %nonassoc IFX
2 %nonassoc ELSE
3
4 %left OP1OR
5 %left OP1
6 %left OP2EQS
7 %left OP2
8 %left OP3
9 %left OP4
10 %right NOT
11 %left OSQUARE DOTLENGTH
```

Listing 4: Associação de Operadores

Apesar de termos usado recursividade à direita, visto ser mais intuitiva, uma solução mais eficiente seria utilizar recursividade à esquerda, pois assim teríamos em memória apenas os elementos que estaríamos a analisar visto que estamos a efetuar reduções à medida que estamos a ler o *input*.

Foram ainda necessárias realizar alterações na gramática, de forma a impedir a indexação de *arrays* (*a*[1][2]). Para tal as expressões foram divididas em dois tipos, expressões indexáveis e não indexáveis.

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

```
Start :
      CLASS ID OBRACE field_or_method_declaration CBRACE;

field_or_method_declaration :
      FieldDecl field_or_method_declaration
      | MethodDecl field_or_method_declaration
      ;
```

```

FieldDecl :
    STATIC VarDecl VarDecl_REPETITION;

MethodDecl :
    PUBLIC STATIC method_type_declaration ID OCURV FormalParams OCURV
    OBRACE VarDecl_REPETITION statement_declaration_REPETITION CBRACE;

method_type_declaration :
    Type
    | VOID
    ;

FormalParams :
    Type ID several_FormalParams
    | STRING OSQUARE CSQUARE ID
    ;

several_FormalParams :
    COMMA Type ID several_FormalParams
    ;

VarDecl_REPETITION :
    VarDecl VarDecl_REPETITION
    ;

VarDecl :
    Type ID several_var_decl_in_same_instructionOPTIONAL SEMIC;

several_var_decl_in_same_instructionOPTIONAL :
    COMMA ID several_var_decl_in_same_instructionOPTIONAL
    ;

Type :
    INT OSQUARE CSQUARE
    | BOOL OSQUARE CSQUARE
    | INT
    | BOOL
    ;

statement_declaration_REPETITION :
    Statement statement_declaration_REPETITION
    ;

Statement :
    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_indexOPTIONAL ASSIGN Expr SEMIC
| RETURN return_expression SEMIC
;

several_statement :
    Statement several_statement
|
;

array_indexOPTIONAL :
    OSQUARE Expr CSQUARE
|
;

return_expression :
    Expr
|
;

IndexableExpr :
    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 :
    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
;

Args_OPTIONAL :
    Args
|
;

Args :
    Expr comma_expr;

comma_expr :
```

```

COMMA Expr comma_expr
|
;

```

Listing 5: Gramática Final

3.2 Tratamento de Erros Sintáticos

Para o tratamento de erros sintáticos utilizamos a função *yyerror* que imprime a linha, a coluna e o input onde ocorreu o erro através da utilização das variáveis *yylineno*, *column*, *yytext*. Assim, sempre que é detetado um erro um sintático é impressa uma mensagem de erro e terminada a execução do programa.

```

1 void yyerror (char *var) {
2     printf ("Line %d, col %d: %s: %s\n",yylineno ,
3             (int)(column-strlen(yytext)), var , yytext);
4     Error = 1;
5 }

```

Listing 6: Função para tratamento erros sintáticos

3.3 Árvore de Sintaxe Abstrata

3.3.1 Estruturas

Para a construção da 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 Node's children (Method's/ Statements, Operators)
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     //Not used lololol
25     char isStatic;
26 }Node;
27
28
29 /* Linked list of ID's (for multiple declaration of variables) */
30 typedef struct _idList
31 {
32     //ID
33     char* id;
34
35     //Next ID
36     struct _idList* next;
37 }listID;
```

Listing 7: Estruturas para representação de Nós da Árvore

Optámos por usar apenas uma única estrutura para representar toda a Árvore de Sintaxe Abstrata uma vez que qualquer nó poderia ser reduzido a um caso mais abstrato. Esta solução permitiu também que as operações realizadas sobre a árvore tenham sido simplificadas, visto não ser necessário realizar qualquer *cast* ou conversão de tipo. Uma vez que a linguagem *iJava* apresenta menos funcionalidades do que a linguagem *Java*, foi possível optar por esta representação, mas tal poderia não se adaptar a um projeto mais complexo.

```

1  /* Type of the Node (to identify the type of the node)
2   - This enum allows us to uniquely identify each node and manipulate the data
   accordingly
3  */
4  typedef enum {NODE_PROGRAM,
5                NODE_VARDECL,
6                NODE_METHODDECL,
7                NODE_METHODPARAMS,
8                NODE_METHODBODY,
9                NODE_PARAMDECL,
10               NODE_COMPOUNDSTAT,
11               NODE_IFELSE,
12               NODE_PRINT,
13               NODE_RETURN,
14               NODE_STORE,
15               NODE_MUL,
16               NODE_DIV,
17               NODE_MOD,
18               NODE_NOT,
19               NODE_MINUS,
20               NODE_PLUS,
21               NODE_LENGTH,
22               NODE_LOADARRAY,
23               NODE_CALL,
24               NODE_NEWINT,
25               NODE_NEWBOOL,
26               NODE_PARSEARGS,
27               NODE_WHILE,
28               NODE_STOREARRAY,
29               NODE_INTLIT,
30               NODE_BOOLLIT,
31               NODE_ID,
32               NODE_AND,
33               NODE_OR,
34               NODE_LESS,
35               NODE_GREATER,
36               NODE_LESSEQUAL,
37               NODE_GREATEREQUAL,
38               NODE_DIFFERENT,
39               NODE_EQUAL,
40               NODE_NULL,
41               NODE_UNARYPLUS,
42               NODE_UNARYMINUS,
43               NODE_DONTPRINT
44             } NodeType;
45
46  /*Type of Node/Token (Int , Void, String,...) */
47  typedef enum {TYPE_VOID,
48                TYPE_INT,
49                TYPE_BOOL,
50                TYPE_INT_ARRAY,
51                TYPE_BOOL_ARRAY,
52                TYPE_STRING_ARRAY

```

53 } Type;

Listing 8: Identificadores únicos de cada *Node*

3.3.2 Criação da Árvore

```
1  /* Method to create an Id (a) */
2  listID* insertID(Node* currentNode, char* id);
3  /* Method to create a new VarId (int a,b,c (to link a to b to c)) */
4  listID* newVarID(char* id, listID* next);
5  /* Method to get an operator type (NODE_PLUS; NODE_NOT,...) */
6  NodeType getOperatorType(char* op);
7  /* Method to create a Null Node, used for mandatory child Nodes */
8  Node* createNull();
9  /* Method to create the Program itself (class gcd) */
10 Node* insertClass(char* id, Node* statements);
11 /* Method to create a new VarDecl (int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r;) */
12 Node* newVarDecl(int type, char* id, listID* moreIds, Node* next);
13 /* Method to link nodes */
14 Node* setNext(Node* current, Node* next);
15 /* Method to set a Node static */
16 Node* setStatic(Node* currentNode);
17 /* Creates a New Method Declaration Node:
18    public static void cenas(){statement1; statement2;return;} */
19 Node* newMethod(int type, char* id, Node* params, Node* varDecl, Node*
    statements);
20 /* Creates a Compound Node ({ expression1;expression2}) */
21 Node* insertCompound(Node* expression);
22 /* Creates an If node (if(true)) */
23 Node* insertIf(Node* expression, Node* statement1, Node* statement2);
24 /* Creates a Print Node (System.out.println(" blica"))*/
25 Node* insertPrint(Node* expression);
26 /* Creates a While Node (while(true)) */
27 Node* insertWhile(Node* expression, Node* statements);
28 /* Creates a Return Node (return; return true;)/
29 Node* insertReturn(Node* expression);
30 /* Creates a Store Node (a = 5, b[5] = true) */
31 Node* insertStore(char* id, Node* arrayIndex, Node* expression);
32 /* Creates a Terminal Node (a, 5, true, false) */
33 Node* createTerminalNode(int n_type, char* token);
34 /* Creates a New Param Declaration Node (String[] argv, int a) */
35 Node* newParamDecl(int type, char* id, listID* moreIds, Node* next);
36 /* Creates a .length Node (argv.length) */
37 Node* insertDotLength(Node* expression);
38 /* Creates a Load Array Node (c[1]) */
39 Node* insertLoadArray(Node* expression, Node* indexExpression);
40 /* Creates a parse int Node (Integer.parseInt(argv[1])) */
41 Node* insertParseInt(char* id, Node* indexExpression);
42 /* Creates a new array Node (new int[1]) */
43 Node* insertNewArray(int type, Node* expression);
44 /* Creates a Method Call (teste(a,b)) */
45 Node* createCall(char* id, Node* args);
46 /* Creates an unary expression ( !true; -1;...)/
```

```

47 Node* insertExpression(char* op, Node* exp);
48 /* Creates an binary expression ( 1 + 1; true != false; a < b) */
49 Node* insertDoubleExpression(Node* exp1, char* op, Node* exp2);
50

```

Listing 9: Funções para a criação da árvore

Para a criação da árvore é realizada pelo *yacc* uma análise ascendente do *input*, isto é, os nós vão sendo criados e agrupados de forma a representar a estrutura de cada regra gramatical. Assim, num primeira instância, são criados os nós terminais (declaração de variáveis, *BOOLIT*, *INTLIT*) sendo posteriormente agrupados a operadores e expressões de forma a criar a estrutura do programa.

3.3.3 Exemplo

Considerando o seguinte programa:

```

1  class gcd {
2      public static void main(String[] args) {
3          int x;
4          return;
5      }
6  }

```

Listing 10: Programa Exemplo

A árvore gerada é a seguinte:

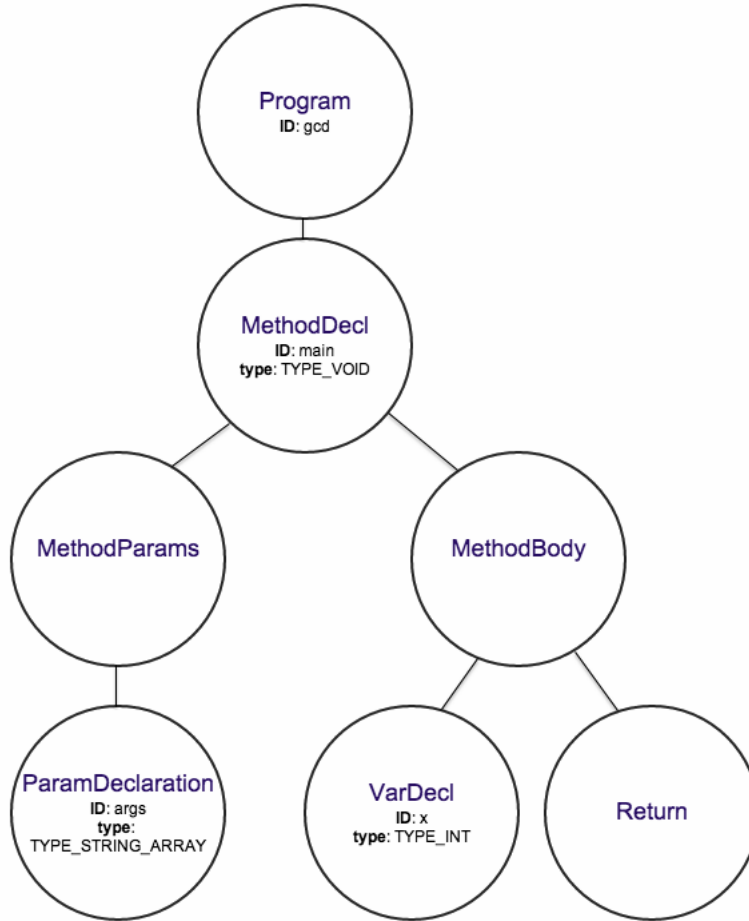


Figura 3: Árvore de Sintaxe Abstrata para o Programa Exemplo

3.3.4 Impressão da Árvore

Para imprimir a Árvore de Sintaxe Abstrata percorremos todos os nós

```

1  /* Method to print tabs */
2  void printTabs(int i);
3  /* Method to print the Syntax Abstract Tree */
4  void printAST(Node* AST);
5  /* Method to print the ID/ID's of the Node */
6  void printIDs(listID* ids, int tabs, int n_type, int type);
7  /* Recursive method to print the tree */
8  void printSubTree(Node* currentNode, int tabs);
9

```

Listing 11: Funções para impressão da Árvore

Para impressão da árvore de Sintaxe Abstrata tivémos em conta a indentação correta para cada Nó e utilizamos a estrutura auxiliar *NODE_STRING*[] onde guardamos a *Node* que identifica cada tipo de *Node*. Para tal, aproveitamos o facto do *NODE_ID* de cada nó ser parte uma enumeração, possibilitando assim que haja uma correspondência direta entre cada índice do array *NODE_STRING*[] e do *NODE_ID*.

```

1 static const char *NODESTRING[] = {"Program",
2     "VarDecl",
3     "MethodDecl",
4     "MethodParams",
5     "MethodBody",
6     "ParamDeclaration",
7     "CompoundStat",
8     "IfElse",
9     "Print",
10    "Return",
11    "Store",
12    "Mul",
13    "Div",
14    "Mod",
15    "Not",
16    "Sub",
17    "Add",
18    "Length",
19    "LoadArray",
20    "Call",
21    "NewInt",
22    "NewBool",
23    "ParseArgs",
24    "While",
25    "StoreArray",
26    "IntLit",
27    "BoolLit",
28    "Id",
29    "And",
30    "Or",
31    "Lt",
32    "Gt",
33    "Leq",
34    "Geq",
35    "Neq",
36    "Eq",
37    "Null",
38    "Plus",
39    "Minus",
40    "DON'T PRINT THIS!"
41 };
42

```

Listing 12: Estrutura auxiliar para impressão da árvore

3.4 Análise Semântica

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

1. Processar as declarações de variáveis e métodos:

- Adicionar novas entradas na tabela de símbolos
- Apresentar mensagem de erro caso haja *Ids* repetidos

2. Processar os *statements*

- Verificar se as variáveis utilizadas foram declaradas no *scope* local (do método) ou no *scope* global, e em caso de variável não declarada, apresentar mensagem de erro
- Usar a tabela de símbolos para determinar o tipo de cada expressão e procurar erros de concordância de tipo (atribuições, cálculos, contagem e tipos de argumentos corretos,...)

3.5 Tabela de Símbolos

A Tabela de símbolos é criada a partir da Árvore de Sintaxe Abstrata, onde se percorrem todos os nós relativos às declarações de métodos e atributos. Assim é criada uma tabela para cada *scope*, ou seja uma tabela global do programa e uma para cada método.

3.5.1 Estruturas

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 13: Estruturas para representação da Tabela de Símbolos

3.5.2 Criação da Tabela de Símbolos

Durante a criação da tabela de símbolos é realizada a verificação da existência de IDs duplicados dentro do mesmo *scope*.

```
1 /* Method to add a variable declaration to a scope */
2 TableNode* addNewDeclTable(char isparam, TableNode* symbol, Node* ast,
3                             Table* table);
4 /* Goes through the AST and creates the scopes adding the Method's and
5    Variables */
6 Table* createSymbols(Node* ast);
```

Listing 14: Métodos para criação da Tabela de Símbolos

3.5.3 Impressão da Tabela de Símbolos

Tal como ocorreu com a Árvore de Sintaxe Abstrata foi utilizado um *array* de *Strings* para fazer a correspondência entre *NODE_ID* e o *output* esperado. Neste *array* no entanto foi necessário uma especial atenção ao fato do *output* esperado ser exatamente o *input* do utilizador e não o *token* de cada nó.

```
1 /* Goes through the tables, prints the header and calls printSymbolsDecl
   method */
2 void printSymbols(Table* table);
3 /* Prints the declarations of each table */
4 void printSymbolsDecl(TableNode* tableNode);
5
6 static const char *OPERATOR_STRING[] = {"Program",
7     "VarDecl",
8     "MethodDecl",
9     "MethodParams",
10    "MethodBody",
11    "ParamDeclaration",
12    "CompoundStat",
13    "if",
14    "System.out.println",
15    "return",
16    "=",
17    "*",
18    "/",
19    "%",
20    "!",
21    "-",
22    "+",
23    ".length",
24    "[",
25    "call",
26    "new int",
27    "new boolean",
28    "Integer.parseInt",
29    "while",
30    "=",
31    "IntLit",
```

```
32         " BoolLit" ,
33         " Id" ,
34         "&&" ,
35         " ||" ,
36         "<" ,
37         ">" ,
38         "<=" ,
39         ">=" ,
40         "!=" ,
41         "==" ,
42         " null" ,
43         "+" ,
44         "-"
45     };
```

Listing 15: Funções impressão da tabela de símbolos

3.6 Tratamento de Erros Semânticos

Para o tratamento de erros semânticos utilizamos as seguintes funções:

```
1 /*Check if ID is already declared in the scope */
2 void    checkIfExists(char* id, Table* local);
3 /* Check if ID exists:
4  - If exists return the Type
5  - If doesn't exist, print error message: "Cannot find symbol %s\n" and exit
6 */
7 int     checkifIDExists(char* id, TableType type, Table* table, Table* main);
8 /* Check if the Literal is valid (decimal / hexadecimal / octal)
9  - If it is invalid print error message: "Invalid literal %s\n" and exit
10 */
11 void    validIntLit(char* lit);
12 /*goes through the AST and check for errors in every node */
13 void    checkSemanticErrors(Node* ast, Table* local, Table* main);
14 /*check if terminal nodes are correct*/
15 void    checkErrors(Node* ast, Table* symbols, Table* main);
16 /*check if the types of the operators are correct, and propagates those types
17    through the tree*/
18 void    checkTypes(Node* ast, Table* main);
19 /*Gets the local scope */
20 Table*  getMethodTable(Table* main, char* methodID);
21 /* Get the type of the function */
22 int     getFunctionType();
23 /* Get the name of the function */
24 char*   getFunctionName();
25 /*Change to another scope */
26 void    setTable(Table* oi);
27
28 /* Prints the correspondent error */
29 void    operatorError2Types(int op, int n1, int n2);
30 void    operatorError1Types(int op, int n1);
31 void    assignmentError(char* var, int n1, int n2);
32 void    assignmentErrorArray(char* var, int n1, int n2);
33 void    statementError(int op, int n1, int n2);
34 void    statementError1oranother(int op, int n1, int n2, int n3);
35 void    getErrorCall(int i, char* name, int n1, int n2);
```

Listing 16: Funções para tratamento de erros semânticos

Na detecção de erros semânticos foi realizada a verificação da consistência entre os símbolos terminais e operadores.

Um dos problemas com que nos deparámos durante a verificação de erros foi a adequação da mensagem de erro ao nó onde ocorre o erro. Tal facto levou a que, apesar de todas as funcionalidades estarem implementadas, não fosse possível concluir a meta 2 atempadamente, embora os erros fossem apenas de *output* ("presentation error").

Foi necessária especial atenção com alguns tipos de nós como o *NODE_CALL* uma vez que era necessária realizar a verificação de todos os seus parâmetros, o *NODE_LENGTH* e *NODE_PARSEARGS* uma vez que poderiam utilizar *Strings*. Nos restantes nós foi necessário verificar os tipos dos nós filhos bem como o tipo resultante da aplicação do operador. Assim, durante a verificação semântica propagamos os tipos de forma a que todos os nós

guardem o tipo de retorno.

4 Geração de Código

Para a realização da geração de código foi necessário ter em conta certos aspetos, nomeadamente os parâmetros e tipo de retorno da função *main*, uma função obrigatória em *iJava* e em *LLVM*. Assim foi necessário permitir a recepção de parâmetros no formato definido pelo *ANSI-C* (*int main(int argc, char **argv)*).

Outro aspecto a ter em conta foi a necessidade de armazenar a dimensão dos *arrays* alocados em memória. Para tal, foi criada uma estrutura, de forma a permitir o armazenamento de dois valores, o espaço de memória alocado e o tamanho desse espaço.

Além disso, tivemos de ter em consideração o *Short Circuiting*, ou seja, o segundo argumento é apenas executado ou avaliado se o primeiro argumento não for suficiente para determinar o valor da expressão: quando o primeiro argumento de uma função *AND* é avaliado como falso, o valor global deve ser falso e quando o primeiro argumento da função *OR* for avaliado como verdadeiro, o valor global deve ser verdadeiro. Para realizar estes requisitos, recorreremos à utilização de saltos condicionais.

Outro aspecto a considerar, foi a necessidade de garantir que todas as funções apresentassem um chamada de retorno. Para não ser necessária a utilização de saltos condicionais, decidimos acrescentar sempre no final de cada função uma expressão de retorno, mesmo que a mesma não possa ser alcançada.

Por fim, foi necessário criar variáveis locais para armazenar os parâmetros, que são passados por valor e por isso não eram passíveis de ser acedidos da mesma forma que outras variáveis.

Devido ao facto de a árvore de sintaxe abstrata já conter os tipos de dados devolvidos pelos operadores, foi-nos possível gerar a maioria o código sem a necessidade de utilizar a tabela de símbolos, sendo que apenas a utilizamos para verificar a origem das variáveis (globais ou locais).

```
1 char* generateCode(Node* ast, Table* main);
2
3 typedef struct _callP{
4     Type type;
5     char name[100];
6     struct _callP* next;
7 }callParams;
```

Listing 17: Função e estrutura auxiliar

```
1 static const char* SYMBOLS.TYPE.SIZE[] = { "void", "i32", "i1",
2 "%.ArrayInt", "%.ArrayBool", "i8*", "Id", "i32",
3 "i1", "i1*", };
4
5 static const char *CODE_OPERATOR.STRING[] = {"Program",
6 "VarDecl",
7 "MethodDecl",
8 "MethodParams",
9 "MethodBody",
10 "ParamDeclaration",
11 "CompoundStat",
12 "if",
```



```
13      "System.out.println",
14      "return",
15      "=",
16      "mul",
17      "sdiv",
18      "srem",
19      "!",
20      "sub",
21      "add",
22      ".length",
23      "[",
24      "call",
25      "new int",
26      "new boolean",
27      "Integer.parseInt",
28      "while",
29      "=",
30      "IntLit",
31      "BoolLit",
32      "Id",
33      "and",
34      "or",
35      "slt",
36      "sgt",
37      "sle",
38      "sge",
39      "ne",
40      "eq",
41      "null",
42      "+",
43      "-",
44      };
```

Listing 18: Tamanho de dados e operadores