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

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*.

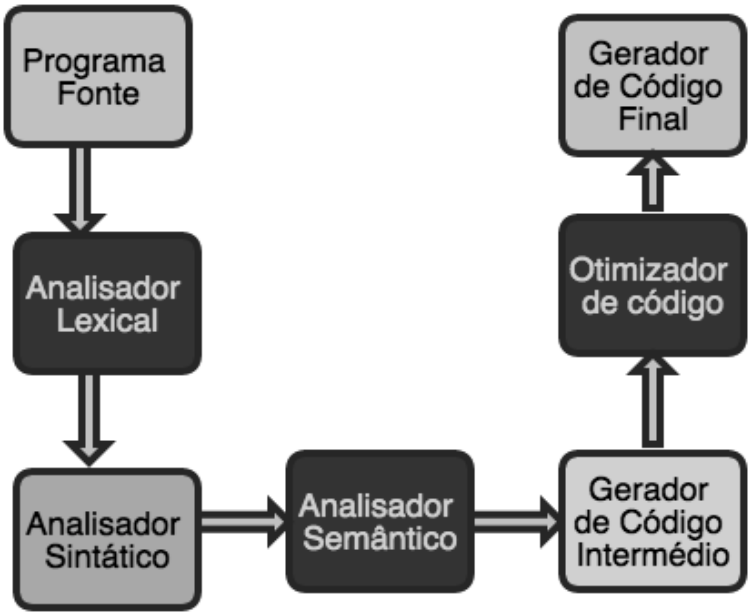

Figura 1: Fases de Compilação

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:** $(([0-9])+|("0x"[0-9a-\text{f}A-F]+))$
- **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.

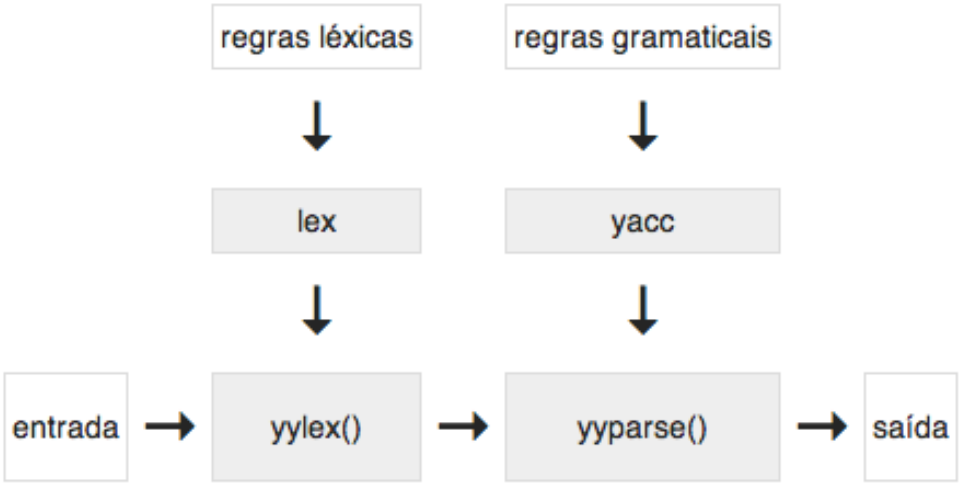

Figura 2: Relacionamento entre *lex* e *yacc*, retirado de *http://pt.wikipedia.org/wiki/Yacc*

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 *yylval* 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.

```
%union{
    struct _Node* node;
    char* token;
    struct _idList* listId;
    int type;
}
```

Listing 2: Union

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

```
extern int column;
extern int yylineno;
extern char* yytext;
extern int yyleng;
```

Listing 3: Variáveis externas

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* usámos 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)

```
%nonassoc IFX
%nonassoc ELSE

%left OP1OR
%left OP1
%left OP2EQS
%left OP2
%left OP3
%left OP4
%right NOT
%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 CCURV
   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:

```
/* General Node */
typedef struct _Node
{
    //Type of the Node (to identify the type of the node)
    NodeType n_type;

    //Type of the Struct (Int, Void, String,...)
    Type type;

    //Id or list of id's
    listID* id;

    //The Node's children (Method's/ Statements, Operators)
    struct _Node* n1;
    struct _Node* n2;
    struct _Node* n3;

    //Next node
    struct _Node* next;

    //Literals (to store the values)
    char* value;

  //Not used lololol
    char isStatic;
}Node;


/* Linked list of ID's (for multiple declaration of variables) */
typedef struct _idList
{
  //ID
  char* id;

  //Next ID
  struct _idList* next;
}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 operação 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.

```
/* Type of the Node (to identify the type of the node)
  - This enum allows us to uniquely identify each node and manipulate the data
     accordingly
*/
typedef enum {NODE_PROGRAM,
              NODE_VARDECL,
              NODE_METHODDECL,
              NODE_METHODPARAMS,
              NODE_METHODBODY,
              NODE_PARAMDECL,
              NODE_COMPOUNDSTAT,
              NODE_IFELSE,
              NODE_PRINT,
              NODE_RETURN,
              NODE_STORE,
              NODE_MUL,
              NODE_DIV,
              NODE_MOD,
              NODE_NOT,
              NODE_MINUS,
              NODE_PLUS,
              NODE_LENGTH,
              NODE_LOADARRAY,
              NODE_CALL,
              NODE_NEWINT,
              NODE_NEWBOOL,
              NODE_PARSEARGS,
              NODE_WHILE,
              NODE_STOREARRAY,
              NODE_INTLIT,
              NODE_BOOLLIT,
              NODE_ID,
              NODE_AND,
              NODE_OR,
              NODE_LESS,
              NODE_GREATER,
              NODE_LESSEQUAL,
              NODE_GREATEREQUAL,
              NODE_DIFFERENT,
              NODE_EQUAL,
              NODE_NULL,
              NODE_UNARYPLUS,
              NODE_UNARYMINUS,
              NODE_DONTPRINT
             } NodeType;

 /*Type of Node/Token (Int, Void, String,...) */
typedef enum {TYPE_VOID,
              TYPE_INT,
              TYPE_BOOL,
              TYPE_INT_ARRAY,
              TYPE_BOOL_ARRAY,
              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:

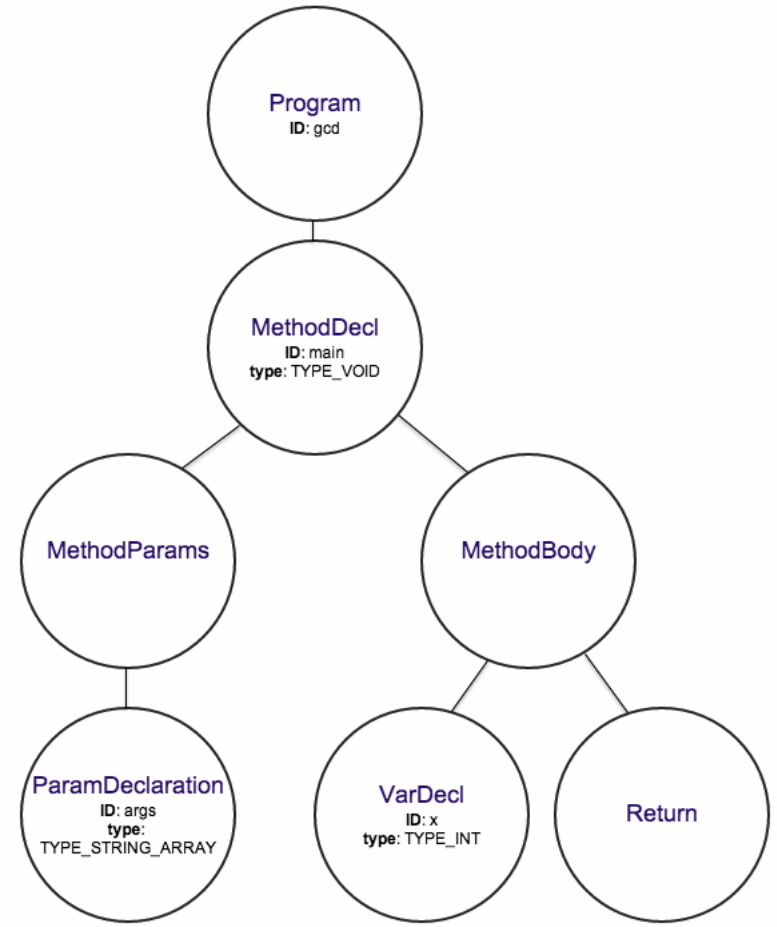

Figura 3: Árvore de Sintaxe Abstrata para o Programa Exemplo

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 $NODE\_ID$.

```
static const char *NODE_STRING[] = {"Program",
                    "VarDecl",
                    "MethodDecl",
                    "MethodParams",
                    "MethodBody",
                    "ParamDeclaration",
                    "CompoundStat",
                    "IfElse",
                    "Print",
                    "Return",
                    "Store",
                    "Mul",
                    "Div",
                    "Mod",
                    "Not",
                    "Sub",
                    "Add",
                    "Length",
                    "LoadArray",
                    "Call",
                    "NewInt",
                    "NewBool",
                    "ParseArgs",
                    "While",
                    "StoreArray",
                    "IntLit",
                    "BoolLit",
                    "Id",
                    "And",
                    "Or",
                    "Lt",
                    "Gt",
                    "Leq",
                    "Geq",
                    "Neq",
                    "Eq",
                    "Null",
                    "Plus",
                    "Minus",
                    "DON'T PRINT THIS!"
                    };

```

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:

```
/* Table Node */
typedef struct _TableNode
{
    //Type of the Node (to identify the type of the node)
    TableType n_type;

    //Type of the Struct (Int, Void, String,...)
    Type type;

    //ID or list of id's
    listID* id;

    //Next node
    struct _TableNode* next;

    //If is a param
    char isParam;
}TableNode;

/* Table */
typedef struct _Table{
    TableNode* table;
    struct _Table* next;
}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*.

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

```
/* Goes through the tables, prints the header and calls printSymbolsDecl
    method */
void printSymbols(Table* table);
/* Prints the declarations of each table */
void printSymbolsDecl(TableNode* tableNode);

static const char *OPERATOR_STRING[] = {"Program",
                  "VarDecl",
                  "MethodDecl",
                  "MethodParams",
                  "MethodBody",
                  "ParamDeclaration",
                  "CompoundStat",
                  "if",
                  "System.out.println",
                  "return",
                  "=",
                  "*",
                  "/",
                  "%",
                  "!",
                  "-",
                  "+",
                  ".length",
                  "[",
                  "call",
                  "new int",
                  "new boolean",
                  "Integer.parseInt",
                  "while",
                  "=",
                  "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:

```
/*Check if ID is already declared in the scope */
void    checkIfExists(char* id, Table* local);
/* Check if ID exists:
  - If exists return the Type
  - If doesn't exist, print error message: "Cannot find symbol %s\n" and exit
*/
int     checkifIDExists(char* id,TableType type, Table* table, Table* main);
/* Check if the Literal is valid (decimal / hexadecimal / octal)
  - If it is invalid print error message: "Invalid literal %s\n" and exit
*/
void    validIntLit(char* lit);
/*goes through the AST and check for errors in every node */
void    checkSemanticErrors(Node* ast, Table* local, Table* main);
/*check if terminal nodes are correct*/
void    checkErrors(Node* ast, Table* symbols, Table* main);
/*check if the types of the operators are correct, and propagates those types
    through the tree*/
void    checkTypes(Node* ast,Table* main);
/*Gets the local scope */
Table* getMethodTable(Table* main, char* methodID);
/* Get the type of the function */
int     getFunctionType();
/* Get the name of the function */
char*   getFunctionName();
/*Change to another scope */
void    setTable(Table * oi);

/* Prints the correspondent error */
void    operatorError2Types(int op,int n1, int n2);
void    operatorError1Types(int op,int n1);
void    assignmentError(char* var, int n1, int n2);
void    assignmentErrorArray(char* var, int n1, int n2);
void    statementError(int op, int n1,int n2);
void    statementError1oranother(int op, int n1, int n2, int n3);
void    getErrorCall(int i,char* name, int n1, int n2);
```

Listing 16: Funções para tratamento de erros semânticos

4 Geração de Código

5 Conclusão