Compilador
iJava

Projecto da Cadeira de Compiladores

João Oliveira - 2010129856

João Simões - 2011150045

FCTUC - Departamento de Engenharia Informática

1 Introdução

Este projecto consistiu no desenvolvimento de um compilador para a linguagem *iJava*, de imperative *Java*. Esta linguagem é uma restrição da linguagem *Java*, para facilitar a implementação do compilador.

Uma das características da linguagem é o facto de cada ficheiro conter uma única classe, sendo que cada programa consiste apenas num ficheiro. Uma classe pode ter variáveis globais e métodos, que por sua vez podem ter variáveis locais. Para além disto, é obrigatória a existência de um método *main*, método este que será a função de entrada do programa. Este mesmo método tem como parâmetro, por defeito, um array de *Strings*, contendo os parâmetos de entrada, tal como em *Java*.

É ainda possível implementar expressões aritméticas, lógicas e relacionais, assim como statements de atribuição, indexação e controlo (*if-else* e *while*).

Relativamente aos tipos de variáveis existentes na linguagem, é possivel usar varíaveis do tipo *int* e *boolean*. É ainda possível criar arrays unidimensionais dos tipo apresentados anteriormente.

Por fim, o desenvolvimento do compilador fez-se em várias fases bem definidas, fases essas que apresentamos de seguida.

Fase 1 Análise Lexical

- Identificação dos tokens aceites pela linguagem, recorrendo ao *LEX*. Nesta ferramenta definimos os tokens através de expressões regulares.

Fase 2 Análise Sintática

- Tradução da gramática dada para a linguagem *YACC* e resolução de conflitos e ambiguidades;
- Criação das estruturas de dados que representam os nós da Árvore de Sintaxe Abstracta;
- Implementação das funções responsáveis pela construção da AST;
- Deteção de erros sintáticos.

Fase 3 Análise Semântica

- Criação das estruturas de dados a utilizar para a construção da tabela de símbolos;
- Implementação dos procedimentos responsáveis pelas inserções na tabela de símbolos;
- Detecção de erros semânticos.

Fase 4 Geração de código intermédio

- Implemententação das funções responsáveis pela geração de código *LLVM*.

2 Análise Léxical

A primeira fase da construção do compilador consistiu na definição dos tokens aceites pela linguagem, através de expressões regulares. Para este efeito usou-se o *LEX*, que constrói internamente um autómato determinístico capaz de reconhecer esses tokens.

2.1 Tokens

Após terminada a fase de análise lexical, obtivemos os seguintes *tokens*:

- **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** - ,
- **BOOLLIT** - *true, false*
- **ID** - Corresponde a todas as sequências alfanuméricas inciadas por uma letra (maiúscula ou minúscula), que podem conter os caracteres "$"e "_".

2.2 Detecção de Erros Lexicais

A análise lexical é realizada da esquerda para a direita, caracter a caracter, até ao momento em que o autómato atinge um estado morto. Quando este mesmo estado é atingido, o analisador regressa ao último estado final, caso em que é encontrado um *token* válido. Caso o analisador não tenha atingido previamente um estado final, é então gerado um erro lexical. Para melhor identificação do erro lexical, é impressa a linha e a coluna da posição onde começa o *token*. Para este efeito, necessitamos de mecanismos para obter informação da posição.

Para tal, existe uma variável com o nome *colNo*, responsável por armazenar a coluna actual, sendo esta variável incrementada de cada vez que se acaba de processar um *token* válido. Relativamente à linha, obtemos o seu valor actual através da variável *yylineno*, já implementada pelo *YACC*.

2.3 Tratamento dos Comentários

A linguagem *iJava* permite a existência de comentários de linha (iniciados por "*//*") e comentários de bloco (colocados entre "*/* */*").

No caso dos comentários de linha, uma simples expressão regular permite manter actualizada a informação de linhas e colunas. No caso dos comentários de bloco, usou-se um estado *COMMENT*, no qual se entra quando se detecta o *token* "*/**"e do qual se sai quando se detecta o *token* "**/*". Dentro deste estado, apesar de se ignorar o texto que não representa o final do comentário, actualiza-se a informação de posição, mantendo assim a sua correcção.

3 Análise Sintática

Após terminada a fase da análise lexical, segue-se a análise sintática. Uma vez que nesta fase é necessário ter em conta as prioridades dos operadores, foram feitos alguns ajustes nos tokens detectados. Estas alterações são explicadas na próxima secção.

A ferramenta utilizada, o *YACC*, é um analizador sintático *LALR(1)*. Assim, foi preferida na nossa implementação da gramática recursividade à esquerda, por resultar numa menor utilização da pilha.

3.1 Alterações na Análise Lexical

Como dito anteriormente, houve necessidade de alterar a organização dos *tokens* relativos aos operadores. Estas alterações devem-se à necessidade de se ter em conta as prioridades dos operadores. Assim, após alterações, obtivemos os seguintes tokens relativos aos operadores.

- **AND** - &&
- **OR** - ||
- **RELCOMPAR** - $<, >, <=, >=$
- **EQUALITY** - $==, !=$
- **ADDITIVE** - $+, -$
- **MULTIPLIC** - $*, /, \%$

Para além das alterações acima referidas, foram ainda feitas outras em relação à contagem da linha e da coluna. Uma vez que aquando da mostra da mensagem de erro sintático, o número da linha e da coluna apresentadas têm de corresponder ao início do *token* inválido, esta informação foi guardada de forma a poder ser acedida no analisador sintático.

3.2 Gramática da Linguagem

A gramática da linguagem *iJava*, apresenta-se de seguida segundo a notação *EBNF*:

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

Após a interpretação da gramática dada, definimos a gramática no *YACC*. A gramática obtida após definição de prioridades e após alterações relativamente à gramática acima, foi a seguinte (todas as alterações e definições de precedências serão abordadas com detalhe nas proximas secções).

Listing 1: Gramática obtida segundo a representação do YACC

```
1
2 start: CLASS ID '{' decls '}'
3       | CLASS ID '{' '}'
```

```
4
5  decls: decls fielddecl
6       | decls methoddecl
7
8  fielddecl: STATIC type ID idlist ';'
9
10 methoddecl: PUBLIC STATIC methodtype ID '(' formalparams ')' '
     {' vardecl stmtlist '}'
11
12 methodtype: type
13           | VOID
14
15 formalparams: type ID formalparamslist
16             | STRING '[' ']' ID
17             |
18
19 formalparamslist: formalparamslist ',' type ID
20                 |
21
22 stmtlist: stmtlist statement
23         |
24
25 vardecl: vardecl type ID idlist ';'
26        |
27
28 idlist: idlist ',' ID
29       |
30
31 type: INT '[' ']'
32     | BOOL '[' ']'
33     | INT
34     | BOOL
35
36 statement: '{' stmtlist '}'
37           | IF '(' expr ')' statement ELSE statement  %prec
               ELSE
38           | IF '(' expr ')' statement    %prec IFX
39           | WHILE '(' expr ')' statement
40           | PRINT '(' expr ')' ';'
41           | ID '[' expr ']' '=' expr ';'
42           | ID '=' expr ';'
43           | RETURN expr ';'
44           | RETURN ';'
45
46 expr: exprindex
47     | exprnotindex
48
49 exprindex: ID
```

```
50            | INTLIT
51            | BOOLLIT
52            | '(' expr ')'
53            | expr DOTLENGTH
54            | PARSEINT '(' ID '[' expr ']' ')'
55            | ID '(' args ')'
56            | ID '(' ')'
57            | exprindex '[' expr ']'
58
59 exprnotindex: NEW INT '[' expr ']'
60              | NEW BOOL '[' expr ']'
61              | '!' expr                %prec UNARY
62              | ADDITIVE expr           %prec UNARY
63              | expr AND expr
64              | expr OR expr
65              | expr RELCOMPAR expr
66              | expr EQUALITY expr
67              | expr ADDITIVE expr
68              | expr MULTIPLIC expr
69
70 args: expr argslist
71     | expr
72
73 argslist: ',' args
```

3.3 Tradução da Gramática dada para o YACC

Como referido anteriormente, foram necessárias diversas alterações à gramática dada na notação *EBNF*. Nesta sub-secção vamos comentar essas alterações explicando a sua razão de ser.

Uma das alterações efectuadas, é relativa à abordagem do que é "opcional"([...]), que tem "zero ou mais repetições"({...}) e situações em que temos várias opções.

- Nas situações em que temos símbolos "opcionais", dividimos a regra em duas regras distintas para abrager os dois casos posssíveis, o caso em que tem o símbolo e o caso em que não. Noutros casos, simplesmente considerámos a possiblidade de ter um síbolo não terminal a derivar a cadeia vazia.

- Quando existem "zero ou mais repetições", considerámos a possibilidade de ser derivada a cadeia vazia, sendo ainda adicionada recursividade de forma a permitir várias repetições do mesmo símbolo.

- No caso de termos várias opções relativamente ao símbolo, é criada uma nova regra que contempla todos os símbolos possíveis.

3.3.1 Prioridade de Operadores

Outras alterações efectuadas têm a ver com a definição de prioridade de operadores. Apresentamos abaixo as prioridades que definimos no *YACC*.

Listing 2: Prioridades

```
1  %left OR
2  %left AND
3  %left EQUALITY
4  %left RELCOMPAR
5  %left ADDITIVE
6  %left MULTIPLIC
7  %right UNARY
8  %left '[' DOTLENGTH
```

Segundo o *YACC*, as prioridades definidas mais abaixo, têm maior prioridade do que as definidas acima. Por exemplo, *AND* tem maior prioridade do que *OR* e menor do que *EQUALITY*.

Estas prioridades representam as prioridades da linguagem *Java*, que são as mesmas que se aplicam na linguagem *iJava*.

3.3.2 Ambiguidade *if-else*

Uma ambiguidade muito comum na definição da gramática de uma linguagem está relacionada com o *if-else*. Veja-se o seguinte exemplo:

```
IF '(' expr ')' IF '(' expr ')' statement ELSE statement
```

Como podemos verficar acima, o "*ELSE statement* "pode estar associado ao *IF* exterior ou ao interior. São então possíveis duas árvores de derivação, existindo portanto uma ambiguidade. Para resolver esta ambiguidade, foram definidas as seguintes prioridades:

```
%nonassoc IFX
%nonassoc ELSE
```

Desta forma, dando maior prioridade à redução do *ELSE*, o *ELSE* é sempre associado ao *IF* mais recente.

3.3.3 Indexação

Relativamente à indexação, foram feitas algumas alterações relativamente à gramática inicial, de forma separar as expressões indexáveis das não-indexáveis. Como podemos verificar (ver linhas 46-68), a *expr* pode derivar em *exprindex* e *exprnotindex*.

- Uma vez que a linguagem não permite a existência de arrays **não unidimensionais**, as regras "*NEW INT [expr]* "e "*NEW BOOL [expr]* "estão incluídas nas não indexáveis, não sendo assim possível a inicialização de arrays não unidimensionais.
- O operador de indexação tem maior prioridade do que qualquer operador (excepto *DOTLENGTH*, que tem a mesma). Desta forma, nunca será possível indexar uma expressão unária (excepto *DOTLENGTH*) ou binária, pelo que estas são não-indexáveis.

- Todos os outros tipo de expressões, estão incluídas nas indexáveis.

3.4 Árvore de Sintaxe Abstracta

Concluída a definição da grámatica, segue-se então a construção da AST.

3.4.1 Estruturas de Dados

Para a construção da AST, foram definidos diversos nós, sendo estes apresentados e analisados detalhadamente de seguida. Optámos por definir um nó para cada regra da gramática, por ser para nós conceptualmente mais fácil visualizar a árvore desta forma.

Listing 3: Expressão

```
typedef struct _expr
{
    ExprType type;
    OpType op;
       struct _expr *expr1;
       struct _expr *expr2;
       char *idOrLit;
       ArgsList *argsList;
} Expr;
```

- **type** - permite identificar o tipo da expressão (Binária, Unária, *ID*, *boolean*, *int*, Chamada de Função, *parseInt*, indexação, *new BOOL[]*, *new INT[]*);
- **op** - permite identificar o operador aplicado na expressão. Estes operadores são aplicados nas expressões binárias, unárias e no caso particular do *.length*;
- **expr1** - esta variável corresponde à expressão à esquerda do operador nas operações binárias, assim como a expressão utilizada nas operações unárias. No caso da indexação, este campo corresponde à expressão a ser indexada. Por fim, no caso das operações do tipo *new BOOL[]*, *new INT[]*, este campo corresponde ao tamanho do array a ser inicializado.
- **expr2** - no caso das operações binárias, a *expr2* corresponde à expressão à direita do operador. Este campo é utilizada nas expressões do tipo *Indexação*, correspondendo ao índice a ser usado.
- **idOrLit** - esta variável tem como objectivo armazenar um *ID* da expressão ou um literal;
- **argsList** - é através desta variável que são guardados os argumentos "passados"quando é feita a chamada de uma função.

Listing 4: Lista de Argumentos

```
typedef struct _argsList ArgsList;

struct _argsList
{
        Expr *expr;
    struct _argsList *next;
};
```

Representa a lista de argumentos a serem passados aquando da chamada de uma função.

Listing 5: Statement

```
typedef struct _stmt
{
    StmtType type;
    char *id;
    Expr *expr1;
    Expr *expr2;
    struct _stmt *stmt1;
    struct _stmt *stmt2;
    struct _stmtList *stmtList;
} Stmt;
```

- **type** - permite distinguir os vários tipos de statements (*compound statement*, *if-else*, *return*, *while*, *print*, *store*, *store array*);
- **id** - esta variável permite guardar o *id* da variável a que é atribuida uma expressão, no *store* e no *storearray*;
- **expr1** - é utilizada no caso de existir apenas uma expressão na regra. No caso de existirem duas regras, a regra mais à exquerda é armazenada neste campo.
- **expr2** - apenas é utilizada no caso de existirem duas expressões na regra, correspondendo à expressão mais à direita;
- **stmtList** - caso o *statement* seja do tipo *compound statement*, então todas as *statements* são adicionadas a esta lista de *statments*.

Listing 6: Lista de Statements

```
typedef struct _stmtList
{
    Stmt *stmt;
    struct _stmtList *next;
} StmtList;
```

Esta lista é utilizada como estrutura auxiliar de *compound statements*.

Listing 7: Lista de IDs

```
typedef struct _idList
{
        char *id;
        struct _idList *next;
} IDList;
```

Permite armazenar os id's das variáveis quando estas são declaradas em formato de lista definindo o tipo delas uma única vez.

Listing 8: Declaração de Variável

```
typedef struct _varDecl
{
    Type type;
        int isStatic;
        IDList *idList;
} VarDecl;
```

- **type** - corresponde ao tipo da variável a ser declarada(*int, bool, int[], bool[]*);
- **isStatic** - identifica a variável como sendo estática ou não;
- **idList** - uma vez que podem ser declaradas várias variáveis do mesmo tipo definindo o tipo apenas uma vez, é usada uma estrutura auxiliar para manter a lista dos *ID*s das variáveis.

Listing 9: Lista de Declarações

```
typedef struct _varDeclList
{
        VarDecl *varDecl;
        struct _varDeclList *next;
} VarDeclList;
```

Estrutura auxiliar usada para manter todas as declarações de variáveis locais de um método.

Listing 10: Lista de Parâmetros

```
typedef struct _paramList
{
    Type type;
       char *id;
       struct _paramList *next;
} ParamList;
```

- **type** - representa o tipo do parâmetro;

- **id** - armazena o *id* do parâmetro;
- **next** - ponteiro para o parâmetro seguinte.

Listing 11: Declaração de Método

```
typedef struct _methodDecl
{
    Type type;
       char *id;
       ParamList *paramList;
       VarDeclList *varDeclList;
       StmtList *stmtList;
} MethodDecl;
```

- **type** - representa o tipo do valor de retorno da função. Para além dos tipos de variáveis é ainda possível retornar *void*;
- **id** - armazena o nome do método;
- **paramList** - contém a lista de parâmetros recebidos pela função;
- **varDeclList** - lista de declarações de variáveis locais à função;
- **stmtList** - lista de *statments da função*.

Listing 12: Lista de Declarações

```
typedef struct _declList
{
    DeclType type;
       union
       {
              VarDecl *varDecl;
              MethodDecl *methodDecl;
       };
    struct _declList *next;
} DeclList;
```

- **type** - permite identificar se é a declaração de um método ou de uma variável global;
- **varDecl** - caso se trate de uma variável global, então é criado um novo nó do tipo *Variable Declaration*;
- **methodDecl** - caso se trate da declaração de um método, então é criado um novo nó do tipo *Method Declaration*

Listing 13: Classe

```
typedef struct _class
{
        char *id;
        DeclList *declList;
} Class;
```

Esta estrutura armazena o nome da classe e a lista de declarações.

3.4.2 Construção da Árvore

Após termos definido as estrutura de dados acima, seguem-se os procedimentos utilizados para efectuar novas inserções na AST.

Listing 14: Funções utilizadas na construção da AST

```
Class* insertClass(char*, DeclList*);

DeclList* insertDecl(DeclType, void*, DeclList*);

VarDecl* insertFieldDecl(Type, char*, IDList*);

VarDeclList* insertVarDecl(VarDeclList*, Type, char*, IDList
    *);

IDList* insertID(char*, IDList*);

StmtList* insertStmtList(Stmt*, StmtList*);

Stmt* insertStmt(StmtType, char*, Expr*, Expr*, Stmt*, Stmt*,
       StmtList*);

ParamList* insertFormalParam(Type, char*, ParamList*, int);

MethodDecl* insertMethodDecl(Type, char*, ParamList*,
    VarDeclList*, StmtList*);

Expr* insertExpr(ExprType, char*, Expr*, Expr*, char*,
    ArgsList*);

ArgsList* insertArg(Expr*, ArgsList*);
```

4 Análise Semântica

Concluída a construção da *AST* e a identificação de erros sintáticos, segue-se a análise semântica. Para tal, foi construída uma tabela de símbolos global e uma tabela local para cada método declarado.

4.1 Arquitetura da Tabela de Símbolos

Segue-se abaixo um esquema da estrutura da tabela de simbolos.

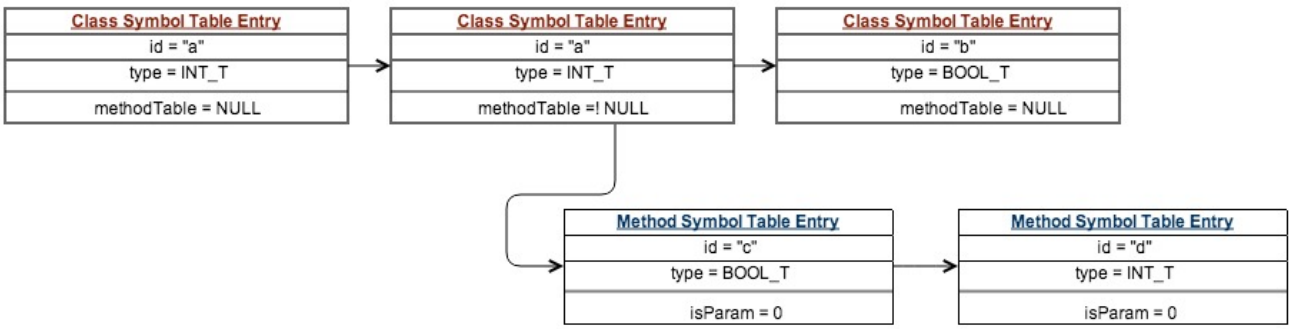

Figura 1: Esquema tabela de símbolos.

Como se pode verificar, a tabela é representada através de uma lista ligada em que os nós representam variáveis globais e métodos. Cada método contém um ponteiro para uma nova lista, a sua tabela local, que contém todos os símbolos locais a esse método. Os argumentos dos métodos aparecem no ínicio desta lista (com a *flag* *isParam* a 1).

De seguida, apresentam-se as estruturas de dados utilizadas para representar a tabela de símbolos.

Listing 15: Estrutura da Tabela de Símbolos

```
typedef struct _methodTableEntry
{
    char* id;
    Type type;
    int isParam;
    struct _methodTableEntry* next;
} MethodTableEntry;

typedef struct _methodTable
{
    struct _classTable* broaderTable;
    MethodTableEntry* entries;

} MethodTable;

typedef struct _classTableEntry
{
    char* id;
    Type type;
    MethodTable* methodTable;
```

```
    struct _classTableEntry* next;
} ClassTableEntry;

typedef struct _classTable
{
    char* id;
    ClassTableEntry* entries;
} ClassTable;
```

Os métodos utilizados na construção da tabela de símbolos foram os que se apresentam na listagem seguinte.

Listing 16: Métodos para Construção da Tabela de Símbolos

```
ClassTable* buildSymbolsTables(Class*);
ClassTableEntry* newVarEntries(VarDecl*, ClassTableEntry*);
void newMethodEntry(MethodDecl*, ClassTableEntry*, ClassTable
    *);
void newMethodTable(MethodTable*, ParamList*, VarDeclList*);
```

- O método *buildSymbolsTables()* é o método que é chamado no *YACC* após a construção da *AST* para dar início à construção da tabela de símbolos.
- *newVarEntries* é o método que, a partir de uma lista de variáveis globais em que o tipo apenas é definido uma vez, as coloca na tabela de símbolos da classe.
- O método *newMethodEntry()* é responsável por introduzir na tabela de símbolos da classe toda a informação referente ao método, como o seu nome e tipo de retorno.
- *newMethodTable()* é o método que preenche a tabela de símbolos local a um método com a informação sobre os seus parâmetros e variáveis locais.

4.2 Detecção de Erros

Após a construção da tabela de símbolos, segue-se a deteção de erros semânticos. Apresentam-se de seguida os possíveis erros semânticos:

Listing 17: Estrutura da Tabela de Símbolos

```
1  Cannot find symbol %s
2  Incompatible type of argument %d in call to method %s (got %
       required %s)
3  Incompatible type in assignment to %s (got %s, required %s)
4  Incompatible type in assignment to %s[] (got %s, required %s)
5  Incompatible type in %s statement (got %s, required %s)
6  Incompatible type in %s statement (got %s, required %s or %s)
7  Invalid literal %s
8  Operator %s cannot be applied to type %s
9  Operator %s cannot be applied to types %s, %s
10 Symbol %s already defined
```

- **Erro 1** - A verificação da declaração de uma variável ou método já definidos é feita durante a construção da tabela de símbolos. Assim, antes da inserção de um novo símbolo na tabela, verifica-se se o símbolo já foi previamente definido. Note-se que, a linguagem permite a existência de um método e de uma várivel local com o mesmo nome, não sendo por isso gerado nenhum erro semântico;

- **Erro 2** - A comparação entre os tipos de parâmetros recebidos por um método e os argumentos passados na chamada deste mesmo método, é feita obtendo da tabela de símbolos local ao método os tipos recebidos pela função, comparando-os com os argumentos passados, obtidos da AST;

- **Erro 3, 4** - A verificação da correcção de tipos nas atribuições é feita através da comparação do tipo devolvido pela expressão correspondente ao valor a ser atribuído e o tipo da variável à qual estamos a fazer a atribuição;

- **Erro 5** - Este erro é gerado quando, por exemplo, o tipo da expressão no *return* é diferente do tipo de retorno do método. Para além disto, no caso de termos um *if* ou um *while* cuja expressão condicional não é do tipo boolean, é também gerado um erro semântico deste tipo;

- **Erro 6** - Nos casos em que é chamado o *System.out.println* com váriaveis cujo tipo não é *Integer* nem *Boolean*, então é gerado um erro indicando que o operador apenas aceita parâmetros deste tipo.

- **Erro 7** - Caso o literal não seja um octal (começado por "0"), hexadecimal (começado por "0x") ou decimal (os restantes casos), é então gerado um erro deste tipo;

- **Erro 8**

  - Quando um operador unário ou o *.length* é aplicado sobre um tipo sobre o qual essa operação não é valida. Por exemplo, o operador unário *not*, apenas pode ser aplicado sobre expressões do tipo *boolean*. Igualmente, o *.length* apenas pode ser usado em *arrays*;
  - Na inicialização de *arrays*, caso a expressão que indica o tamanho do *array* seja de um tipo diferente de *integer*, é gerado um erro semântico.

- **Erro 9**

  - No caso de *store array*, o erro pode ser análogo ao da indexação (Ver abaixo);
  - Quando os operadores "+", -", "*", "/", "%", ≪", ≫", ≪=", ≫="são aplicados a tipos diferentes de *integer*;
  - Quando os operdores "!="e "=="são aplicados a tipos diferentes;
  - Quando os operadores "&&"e "||"são aplicados a tipos diferentes de *Boolean*
  - Caso o *Integer.parseInt* não seja aplicado a um *array* de *Strings* indexado por um *integer*, então é gerado um erro deste tipo;
  - É gerado um erro deste tipo quando é feita a indexação a um tipo diferente de *int[]* ou *bool[]*. Igualmente, caso a expressão de indexação não seja do tipo inteiro é gerado um erro semântico desta natureza;

5 Geração de Código

A última fase, que se segue à fase semântica, é a geração de código. No âmbito do nosso projecto, foi implementada a geração de código intermédio *LLVM*. Após a geração de um ficheiro *.ll*, pode então interpretar-se com o comando *lli* ou compilar com o comando *llc* seguido de um qualquer compilador de *Assembly* o código gerado pelo nosso compilador.

Nesta secção serão apenas mencionadas as funcionalidades mais complexas, para manter a brevidade deste relatório. As restantes funcionalidades, por serem comparativamente triviais, foram omitidas.

5.1 Ifs e Whiles

Para a implementação destas estruturas de controlo foram utilizadas *named labels*, com o cuidado de usar pontos na sua nomenclatura, para evitar conflitos com identificadores do programa a ser compilado.

O *if-else* consiste em 3 *labels*, uma para o *then*, uma para o *else* e uma que representa o fim do *if-else*. A *then*, contém o código que deve ser executado se a condição for verdadeira, sendo que na *else* se encontra o código que deve ser executado se esta for falsa. A *label* que representa o fim do *if-else* é para onde qualquer um dos segmentos anteriores salta no final da sua execução. Esta *label* precede o código que se segue a esta estrutura de controlo de fluxo.

O *while* é muito análogo ao *if-else*, tendo também 3 *labels*. A primeira é o início da estrutura, onde se encontra a condição de paragem. Caso esta condição seja verdadeira, o programa executa o que se encontra na segunda *label*, *do*, voltando no final deste segmento à *label start*. Caso a condição de paragem seja falsa, a execução salta para a *label end*, que precede o código que se segue ao *while*.

5.2 Returns

Como não é efectuada qualquer análise da presença de *returns* numa função, ou da existência de *returns* em todos os ramos de execução de uma função, foi necessário encontrar uma forma de nos certificarmos que qualquer função retorna, independentemente da existência ou não de *returns explícitos*.

Para solucionar este problema, todos os métodos possuem um return por omissão, colocado no final da função. Este retorna 0 no caso de o tipo de retorno ser *inteiro* ou *boolean*, *null* no caso de ser um ponteiro ou uma estrutura com o primeiro campo a 0 e o segundo a *null* no caso de ser um *array*.

5.3 ParseInt

Para o *Integer.parseInt*, foi usada a função *atoi()* da biblioteca de *C*, através da *API* de chamada deste tipo de funções a partir de *LLVM*.

5.4 Prints

Para a impressão foi necessário definir *strings* auxiliares para serem usadas na função *printf()* da biblioteca de *C*.

No caso de inteiros, foi utilizada a *string "%d\n"*, sendo passada essa *string* como *format string* da função *printf()* e o inteiro a imprimir como segundo argumento dessa mesma função.

No caso de *booleans*, utilizou-se um *array* com a string *"false\n"* na primeira posição e *"true \n"* na segunda. Desta forma, e tendo em conta que em *LLVM false* = 0 e *true* = 1, podemos usar o valor que queremos imprimir como índice do *array* para obter a *format string* a passar à função *printf()*.

5.5 Short Circuiting

Inicialmente tentámos implementar esta funcionalidade com recurso a nós *phi* da representação intermédia *LLVM*. No entanto, após problemas com a correcta indicação das *labels* que precedem a instrução *phi* desistimos desta abordagem.

No entanto, essa abordagem seria correcta e mais eficiente em termos de memórida alocada na stack, não fazendo uso de nenhuma memória deste tipo. Chegámos tardiamente à conclusão de que, com uma *label* adicional que representasse sempre a saída de qualquer código que fosse executado apenas caso não existisse *short circuiting*, poderiamos ter solucionado este problema.

A nossa abordagem implementada foi utilizar uma estrutura de controlo de fluxo semelhante a um *if*, mas que necessita de recorrer a memória da pilha.

Listing 18: Short Circuiting em ANDs

```
a && b

res = a;
if(a)
    res = b;
```

Listing 19: Short Circuiting em ORs

```
a || b

res = a;
if(!a)
    res = b;
```

5.6 Arrays e .length

Para poder ter a funcionalidade de *Java* existente no operador *.length*, implementámos os *arrays* como sendo estruturas, em que na primeira posição se encontra o tamanho do vector e na segunda o ponteiro em si.

Desta forma, o operador *.length* consiste apenas na obtenção do valor guardado na primeira posição da estrutura. Este valor deve ser actualizado sempre que existir um *new* com o valor correcto do novo tamanho do array, sendo inicializado a 0 aquando da declaração da variável.

A indexação de um array continua a ser uma simples indexação, em que apenas se deve indexar o valor na segunda posição da estrutura.

5.7 Inicialização de Arrays

Usando a função *calloc* de *C* forçamos a inicialização de *arrays* de inteiros a 0 e de *arrays* de *booleans* a *false*, tal como acontece em *Java*.

6 Possíveis Melhorias

Uma das possíveis melhorias que poderiamos implementar seria a libertação de toda a memória alocada na *heap* no processo de compilação de um qualquer programa. Apesar de termos o código já estruturado para que, em qualquer cenário de saída do programa, não tivémos, infelizmente, tempo para implementar esta correcta libertação.

Outra melhoria digna de ser mencionada é o facto de não utilizarmos *unions* nos nós da *AST* em casos em que o uso destas estruturas pouparia claramente espaço. A razão pela qual não implementámos desta forma os nós da *AST* foi não complicar em demasia as funções de criação de nós da *AST*, que nesse caso teriam de ter sido implementadas com cuidado para não sobrepor campos importantes ao inicializar campos irrelevantes para o tipo de nó em questão.