Compiladores - Projecto

iJava

João Ricardo Lourenço, Nº 2011151194
Joaquim Pedro Bento Gonçalves Pratas Leitão, Nº 2011150072

2 de Junho de 2014

Relatório

Índice

1 Introdução

No presente trabalho pretende-se desenvolver um compilador para a linguagem $iJava$, um pequeno subconjunto da linguagem Java (versão 5.0). Por ser um subconjunto de uma outra linguagem, todos os programas que respeitem as regras impostas em $iJava$ são também, garantidamente, programas válidos em $Java$[1].

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

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

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

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

É também possível passar parâmetros (literais inteiros) ao nosso programa através da linha de comandos. É o método *main* que vai receber esses parâmetros, armazenando-os num array de objetos do tipo *String*. No entanto, este tipo de dados não está incluído na lista de tipos permitidos em $iJava$, pois a sua utilização apenas é permitida no método *main*, com a mera finalidade de obter os parâmetros passados ao programa aquando da sua invocação. Deste modo, este array apenas suporta duas operações: o operador .length e a obtenção de um inteiro a partir dos seus elementos com $Integert.parseInt$.

O desenvolvimento do compilador foi dividido em três fases distintas.

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

Seguiram-se a realização da *Análise Sintática* e *Análise Semântica*, compostas por quatro etapas principais:

- **Tradução da gramática-fonte** (fornecida em notação $EBNF$) para o yacc e realização da **Análise Sintática** do programa, permitindo assim reconhecer se as sequências de *tokens* que o constituem pertencem à linguagem, permitindo-nos assim detetar eventuais erros de sintaxe.
- **Construção da árvore de sintaxe abstrata**, etapa realizada em simultâneo com a **Análise Sintática**. A árvore de sintaxe abstrata irá

[1]No entanto, certas restrições semânticas tornam certos códigos compiláveis e executáveis em iJava, mas não em Java. Um exemplo é o de código não acessível, que não é aceite pelo compilador javac, mas é válido em iJava.

representar o nosso programa a compilar, recorrendo a uma estrutura em árvore para representar as estruturas sintáticas das cadeias que o constituem.

- **Construção das tabelas de símbolos**, utilizadas para armazenar informações relevantes sobre a classe (seus atributos e métodos), bem como sobre cada método definido pelo programador (mais concretamente o tipo de retorno e os argumentos).

- **Verificação de erros semânticos**, etapa principal da **Análise Semântica**, onde são realizadas verificações de tipos, garantindo que para cada operação a realizar não existem incompatibilidades de tipos entre os operandos nela envolvidos.

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

2 Análise Lexical

Tal como referimos anteriormente, na *Análise Lexical* procedemos à identificação dos *tokens* da nossa linguagem. Para isso utilizámos a ferramenta *lex*, responsável por gerar analisadores lexicais para linguagens.

Assim, no nosso analisador, sempre que é detectada a presença de um comentário no programa a compilar, seja do tipo //... (comentários de apenas uma linha) ou do tipo /∗...∗/ (comentários multi-linha), os caracteres incluídos nesse comentário são ignorados.

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

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

2.1 Tokens

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

- **ID**: Sequências alfanuméricas (maiúsculas e minúsculas) começadas por uma letra, podendo conter também símbolos como "_" e "$". Este *token* pode também ser descrito na forma da sua expressão regular: *letra(letra—[0-9])**, sendo o *token* **letra** da nossa autoria definido por: $[a-z] \mid [A-Z] \mid$ "_"|"$"
- **INTLIT**: Sequências de dígitos decimais e hexadecimais (incluindo a-f e A-F) precedidas de $0x$. Este *token* pode também ser descrito na forma da seguinte expressão regular: *[0-9]+—0x[0-9a-fA-F]+*
- **BOOLLIT**: $true \mid false$
- **INT**: int
- **BOOL**: $boolean$
- **NEW**: new
- **IF**: if
- **ELSE**: $else$
- **WHILE**: $while$

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

Para além dos *tokens* apresentados, definimos outros *tokens*, que passamos a especificar:

- **NEWLINE**: *Token* correspondente ao caracter de mudança de linha, $\backslash n$
- **WHITESPACE**:*Token* correspondente ao caracter de espaço em branco
- **OPEN_COMMENT**: *Token* correspondente ao início de um comentário multi-linha, $/*$
- **CLOSE_COMMENT**: *Token* correspondente ao fecho de um comentário multi-linha, $*/$
- **SINGLE_LINE_COMMENT**: *Token* utilizado para detetar a ocorrência de um comentário de uma linha apenas

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

- **AND** ("&&") e **OR** ("||"), originados a partir do *token* **OP1**
- **LE** ("¡"), **GE** ("¿"), **EQ** ("=="), **NEQ** ("!="), **LEQ** ("¡=") e **GEQ** ("¿="), originados a partir do *token* **OP2**
- **PLUS** ("+") e **MINUS** ("-"), originados a partir do *token* **OP3**
- **MULT** ("*"), **DIV** ("/") e **MOD** ("%"), originados a partir do *token* **OP4**

2.2 Comentários

Para identificarmos a ocorrência de comentários nos programas a compilar recorremos aos *tokens* *OPEN_COMMENT*, *CLOSE_COMMENT* e *SINGLE_LINE_COMMENT*.

Quando detectamos o *token* *OPEN_COMMENT* é criado um novo estado no analisador lexdical, que indica a existência de um comentário multi-linha. A esse estamos demos o nome *MULTI_LINE_COMMENT_S*.

Uma vez neste estado, todos os caracteres e *tokens* identificados são ignorados, com exceção do *token* de fecho do comentário multi-linha (*CLOSE_COMMENT*) e do *token* de fim do ficheiro, $<< EOF >>$, disponível na ferramenta utilizada para desenvolver o analisador (*lex*).

Caso seja identificado o *token CLOSE_COMMENT*, o estado do analisador é reposto, passando este a ter o seu estado por defeito. A utilização desta técnica permite-nos terminar após o primeiro $*/$, como desejado.

Por outro lado, se for detectado $<< EOF >>$ temos uma situação em que um comentário multi-linha não foi devidamente terminado, pelo que é gerado um erro lexical, terminando a execução do analisador e sendo o utilizador informado da ocorrência do erro e da localização no programa fornecido do comando que inicia o comentário.

Se, por alguma razão, o *token CLOSE_COMMENT* for identificado quando o analisador não se encontra no estado *MULTI_LINE_COMMENT_S* é detectada também a ocorrência de um erro lexical, uma vez que na nossa linguagem não é possível a existência do token */* sem que antes tenha sido colocado um /*. Mais uma vez, assim que o erro lexical é detectado, o utilizador é informado com uma mensagem que indica a posição no programa onde se deu o erro, e o analisador termina a sua execução.

Ao detectarmos o *token SINGLE_LINE_COMMENT* vamos ignorar todos os caracteres e *tokens* que se lhe seguirem, até que seja reconhecido o *token* de mudança de linha (*NEWLINE*). Desta forma estamos a descartar toda a restante linha do programa, após a ocorrência de //, tal como seria desejado no tratamento de comentários de uma linha apenas.

2.3 Tratamento de Erros Lexicais

Tal como referimos nos pontos anteriores, sempre que o analisador desenvolvido detecta a ocorrência de um erro lexical (seja por existência de um *token* não permitido ou por não término de um comentário multi-linha), é impressa uma mensagem de erro que indica a posição do erro no programa a compilar (indicando a linha e coluna onde o erro ocorreu).

Quando é detectado um caracter ilegal, o analisador imprime a mensagem de erro, prosseguindo a sua execução na linha seguinte do programa a compilar até ser lido todo o conteúdo do programa.

Tendo sido detectada a ocorrência de um ou mais erros lexicais, após ler todo o programa, o analisador termina a sua execução, não sendo realizado mais nenhum passo da compilação.

3 Análise Sintática

A Análise Lexical permite-nos identificar os *tokens* da linguagem, isto é, os seus "átomos". No entanto, interessa-nos garantir que os *tokens* identificados estejam organizados de acordo com a estrutura sintática da linguagem.

Para validar essa organização dos *tokens* necessitamos de utilizar uma gramática, que nos indica como devem estar organizados sintaticamente os *tokens* da linguagem. Para além disso, utilizámos a ferramenta *YACC (Yet Another Compiler Compiler)*, um gerador de analisadores sintáticos.

Assim, utilizamos o analisador lexical (*lex*) desenvolvido para reconhecer os *tokens* da linguagem, que são transferidos para o *yacc* onde, com base na gramática da linguagem fornecida, podemos verificar se o programa a compilar se encontra organizado de acordo com a estrutura sintática da linguagem.

No analisador sintático foi criada uma *union*, que define a estrutura de uma variável *yylval*, utilizada na comunicação entre o *yacc* e o *lex*. Apresentamos desde já a estrutura da *union* criada:

```
%union
{
    char* token;
    struct _node_t* node;
    int type;
}
```

Os *tokens* detectados no *lex* são comunicados ao analisador sintático através do campo *token* da variável *yylval*.

Para além disso, no analisador sintático declaramos ainda três variáveis externas, para uso partilhado entre o *lex* e o *yacc*: Duas do tipo inteiro, *prev_col* e *prev_line*, e uma do tipo array de caracteres (*char**), chamada *yytext* e que contém a cadeia de caracteres lida pelo analisador, em cada momento. Esta variável é implementada/definida internamente pelo *lex*.

Tal como o nome pode sugerir, *prev_col* e *prev_line* são utilizadas para armazenar o valor da coluna e linha (do programa a compilar) analisadas na iteração anterior da execução do analisador. A justificação para esta escolha prende-se com as situações onde ocorrem erros sintáticos, sendo necessário informar o utilizador da localização no programa desse erro.

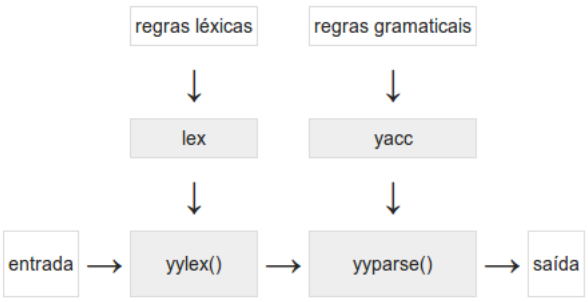

Figura 1: Ligação entre o *lex* e o *yacc*, ferramentas utilizadas para gerar os analisadores lexicais e sintáticos. Retirado de *http://pt.wikipedia.org/wiki/Yacc*

3.1 Gramática

De seguida, apresentamos a gramática da linguagem *iJava*, fornecida no enunciado do projeto, em 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 }

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

3.1.1 Ambiguidade

Um análise mais cuidada da gramática apresentada permite-nos aferir da sua ambiguidade. Por exemplo, se num dado momento da análise sintática pretendermos analisar $2 + 3 * 5$, podemos reduzir esta expressão à variável *EXPR* da gramática por duas formas distintas:

1. Numa primeira abordagem podemos separar a expressão em **Expr** $\rightarrow$ **Expr + Expr** (onde o símbolo *"+"* é proveniente do *token OP3*). De seguida reduziríamos o primeiro símbolo *Expr* para o *token INTLIT* correspondente, neste caso, ao literal *"2"*. O segundo símbolo *Expr* seria desdobrado em **Expr** $\rightarrow$ **Expr * Expr** (onde o símbolo *"*"* é proveniente do *token OP4*). Neste caso os dois símbolos *Expr* seriam reduzidos ao *token INTLIT*, correspondente a cada um dos restantes literais.

2. Numa abordagem alternativa começaríamos por separar a expressão em **Expr** $\rightarrow$ **Expr * Expr**. De seguida desdobraríamos o primeiro símbolo *Expr* em **Expr** $\rightarrow$ **Expr + Expr**. Estes dois novos símbolos *Expr* seriam reduzidos a *INTLIT*, tal como o restante símbolo *Expr*.

Acabámos de provar a ambiguidade da gramática. Tal como esta situação existem muitas outras envolvendo os operadores englobados pelos *tokens OP1, OP2, OP3 e OP4*. Assim, é imperativa a definição de prioridades nos operadores, de forma a eliminar todas as ambiguidades presentes na gramática, permitindo assim a correta realização da análise sintática do programa. A definição de prioridades também nos auxiliará a definir concisamente as prioridades que o Java implementa.

3.2 Alterações à Gramática Fornecida

Uma vez que a gramática apresentada é ambígua e está apresentada em notação **EBNF**, não aceite pela ferramenta utilizada para gerar o analisador sintático, necessitámos de alterar a gramática, de forma a eliminar as suas ambiguidades, tornando-a numa gramática aceite pela ferramenta utilizada.

3.2.1 Definição de Novos *Tokens*

Como já referimos, a primeira alteração realizada prende-se com a separação dos operadores englobados pelos *tokens OP1, OP2, OP3 e OP4*, criando um novo *token* para cada operador, definindo de seguida a prioridade dos diferentes operadores representados pelos *tokens* criados.

As prioridades dos operadores foram definidas na ferramenta *yacc*, de acordo com a sua notação. De seguida apresentamos a definição, em notação *yacc* da prioridade dos operadores identificados:

```
%nonassoc THEN
%nonassoc ELSE
%nonassoc REDUCEEXPRESSON1
%left OR
%left AND
%left EQ NEQ
%left LE GE LEQ GEQ
%left PLUS MINUS
%left MULT DIV MOD
%right ASSIGN
%left OBRACE
%right UNARY_HIGHEST_VAL
%left OSQUARE DOTLENGTH
```

Na notação *yacc* a prioridade de um *token* é definida com as instruções *%left* e *%right*, correspondendo respetivamente à associatividade à esquerda ou à direita. Alternativamente podemos também utilizar a instrução *%nonassoc*, indicando que o operador não tem qualquer associatividade. Para além disso a prioridade de um operador é tanto maior quanto posterior for a sua definição.

Na situação apresentada indicamos, por exemplo, que os operadores *THEN* e *ELSE* não têm qualquer associatividade, tendo o operador *ELSE* maior prioridade do que o operador *THEN*.

Chamamos a atenção para os *tokens* *REDUCEEXPRESSION1*, *THEN*, *ELSE*, e*UNARY_HIGHEST_VAL*, que passamos a detalhar com maior detalhe nas secções que se seguem.

3.2.2 Tokens *THEN* e *ELSE* e o conflito *IF-ELSE*

Um dos conflitos com os quais nos deparámos centrou-se, exactamente, no caso das instruções de controlo de fluxo *if* e *else*.

Analisaremos as regras da gramática da linguagem nas quais estas instruções se encontram:

- *Statement* → *IF OCURV Expr CCURV Statement %prec THEN*
- *Statement* → *IF OCURV Expr CCURV Statement ELSE Statement*

O analisador, ao ler uma sequência de *tokens* correspondente a *IF OCURV Expr CCURV Statement*, e tendo colocado o símbolo *Statement* no topo da pilha (através de uma operação de reduce), encontra uma situação

de não-determinismo na sua acção. Isto porque, com os *tokens* lidos, este poderá realizar um reduce pela primeira regra analisada, colocando o símbolo *Statement* no topo da pilha. Alternativamente, poderá também realizar um *shift* pela segunda regra analisada , esperando ler *ELSE Statement*.

Coloca-se, então, a questão de qual a acção que o analisador deve realizar. Efectivamente, a resposta a esta questão não é trivial, visto que se o próximo *token* lido corresponder a um *ELSE* deveremos ter realizado um *shift* previamente, mas se o próximo caracter não corresponder a um *ELSE*, a realização do *reduce* seria a mais adequada.

Assim, interessa-nos, sempre que possível, realizar um *shift*. Dado que o *token ELSE* apresenta maior prioridade que o *token THEN*, recorremos ao comando *%prec THEN* na primeira regra, para indicar que essa regra tem a mesma precedência que *THEN*. Como este *token* tem menor precedência que *ELSE*, o nosso conflito *shift-reduce* fica resolvido.

3.2.3 Token *UNARY_HIGHEST_VAL* e as precedências dos operadores unários

Numa situação algo semelhante à analisada no ponto anterior para o caso *IF-ELSE*, também os operadores unários requerem alguma atenção no que respeita às prioridades das operações a eles associadas.

De facto, sempre que detectamos a ocorrência de um operador unário aliado a uma expressão, é do nosso interesse que este seja imediatamente associado à expressão. Por exemplo, no caso: $-2 + 3$ pretendemos obter o resultado da soma de 3 ao número negativo -2, e não o simétrico do valor da soma de 2 e 3. Assim, definimos um *token*, *UNARY_HIGHEST_VAL*, ao qual demos o segundo maior valor de prioridade, associando-o a todas as regras que envolvem os operadores unários da linguagem. Desta forma, garantimos que este *token* tem uma prioridade superior à de todos os outros *tokens*, com excepção do operador *.length* (reservado a arrays), e do sinal de parêntesis, que por definição matemática tem uma maior precedência que qualquer operador unário.

3.2.4 Token *REDUCEEXPRESSON1* e as Expressões Indexáveis

Como já referimos em secções anteriores deste relatório, na nossa linguagem possuímos expressões indexáveis e não-indexáveis, sendo que as primeiras podem sofrer indexação, ao contrário das segundas.

Por essa razão tivemos necessidade de separar estes dois *"tipos"* de expressões na gramática, permitindo-nos fazer algumas verificações quanto à possibilidade, ou não, de indexar uma dada expressão.

No entanto, como não permitimos mais do que uma indexação (dupla indexação não é permitida na nossa linguagem), pretendemos impedir que uma expressão indexada uma vez não o seja novamente. Por outras pala-

vras, sempre que encontrarmos uma expressão indexável que já tenha sido indexada queremos reduzi-la a uma expressão ("normal"), de impedindo-a de ser novamente indexada.

Assim, de uma forma semelhante ao que explicámos nas últimas duas secções, definimos um *token* ao qual demos o nome de *REDUCEEXPRESSON1*, atribuindo-lhe um valor de prioridade, que associámos à regra da gramática que nos permite reduzir uma expressão indexável a uma simples expressão, não indexável: *Expr* → *exprIndexable*. Tal foi conseguido com o comando *%prec REDUCEEXPRESSON1*, já abordado em secções anteriores.

3.2.5 Alterações nas Regras da Gramática

Após a criação de novos *tokens* e definição das respetivas prioridades procedemos a modificações na gramática, visando:

- Lidar com situações como *Expr* → *ID OCURV [Args] CCURV* ou *Args* → *Expr* { *COMMA Expr* }, correspondentes a *tokens* opcionais e a 0 ou mais repetições de uma dada sequência de *tokens*
- Distinguir expressões indexáveis, isto é às quais podemos aplicar o operador **"["**, das restantes operações não indexáveis

3.2.6 *Tokens* Opcionais

Estas alterações justificam-se pelo facto de que, evitando a existência de símbolos anuláveis, a criação da *Árvore de Sintaxe Abstrata* se tornar mais fácil, dado que as regras anuláveis implicam a criação de nós nulos na *Árvore de Sintaxe Abstrata*, o que adiciona alguma complexidade e dificuldade à sua criação, manutenção e utilização. Por essa razão, e com o principal objetivo de não adicionar demasiada complexidade à criação da *Árvore de Sintaxe Abstrata*, optámos por eliminar essas situações, sempre que possível.

Assim, as situações correspondentes a *tokens* opcionais na gramática foram substituídas pelas respetivas regras, com e sem os *tokens* opcionais. No exemplo apresentado, a regra *Expr* → *ID OCURV [Args] CCURV* seria substituída pelas regras *Expr* → *ID OCURV CCURV* e *Expr* → *ID OCURV Args CCURV*.

Por sua vez, as repetições de *tokens* 0 ou mais vezes obrigaram-nos a introduzir um novo símbolo na gramática, que garante a ocorrência da repetição de *tokens* pelo menos uma vez, ou obrigando a que os *tokens* não sejam derivados a partir do símbolo da gramática em questão

Assim, no caso apresentado anteriormente, correspondente à regra *Args* → *Expr* { *COMMA Expr* }, introduzimos o símbolo *RealArguments*, substituindo a regra por *Args* → *RealArguments*. Por sua vez, o símbolo *RealArguments* possui duas regras, sendo que numa considera apenas uma

ocorrência de *Expr* (não ocorrendo nenhuma repetição de *COMMA Expr*), e na outra obriga a que a sequência de *tokens COMMA Expr* ocorra pelo menos uma vez. Assim, introduzimos as regras:

Args → *RealArguments* *RealArguments* → *Expr* *RealArguments* → *Expr COMMA RealArguments*

3.2.7 Expressões Indexáveis

À semelhança do que acontece em *Java*, também em *iJava* existem expressões passíveis de serem indexadas e outras onde a operação de indexação não é possível. A nível sintáctico, optámos por estruturar a detecção destes erros de acordo com o seguinte catálogo:

- **Operações Não-Indexáveis:** Declarações de arrays (inteiros e booleanos), expressões compostas apenas por símbolos terminais (literais inteiros ou booleanos) e expressões onde já é realizada indexação (ou seja, indexáveis)
- **Expressões Indexáveis:** Todas as restantes

Realçamos que nem todas as restantes expressões são indexáveis, mas que a verificação da correcta indexação destas é delegada para a análise semântica.

3.2.8 Expressões Não-Indexáveis

Dadas as características da linguagem descritas em secções anteriores, uma vez que em *iJava* apenas está disponível a utilização de arrays unidimensionais, não podemos considerar as declarações de arrays indexáveis. Caso o fizéssemos, a operação *new int[5][2]* seria válida na nossa linguagem.

De facto, poderíamos entender esta instrução como a declaração de um array uni-dimensional de inteiros, e o posterior acesso à sua terceira posição. No entanto, esta operação corresponde, na linguagem *Java* à declaração de um array bi-dimensional de inteiros, não permitido em *iJava*.

Assim, consideramos a operação *new int[5][2]* inválida na linguagem *iJava*, o que nos permite concluir que todas as declarações de arrays não são indexáveis. Para além disso, caso um programador pretenda declarar um array uni-dimensional e aceder de seguida a uma das suas posições, na mesma linha de código, também o poderá fazer em *iJava*, devendo envolver a declaração do array numa expressão indexável, através da utilização de parêntesis: emph(new int[5])[2].

3.2.9 Expressões Indexáveis

Todas as expressões que não declarações de arrays ou terminais são passíveis de serem indexadas na nossa linguagem.

Queremos, contudo, chamar a atenção para situações não permitidas na linguagem, como por exemplo *a[0]* caso *a* seja uma variável do tipo inteiro ou booleano.

De facto, a nível sintático esta expressão é considerada válida, sendo apenas na *Análise Semântica* (a detalhar mais à frente) realizadas verificações dos tipos de dados envolvidos em cada operação, assegurando a adequação dos mesmos à operações a realizar. Naturalmente que o exemplo descrito levará à deteção de um erro semântico, que será posteriormente relatado ao utilizador.

3.2.10 Gramática Final

Apresentamos de seguida a gramática final da linguagem, após efetuar todas as alterações referidas:

```
1  Start:  Program
2    ;
3
4  Program: CLASS ID OBRACE Declarations CBRACE
5      |    CLASS ID OBRACE CBRACE
6    ;
7
8  Declarations: Declarations FieldDecl
9      |      Declarations MethodDecl
10     |      FieldDecl
11     |      MethodDecl
12     ;
13
14 FieldDecl: STATIC VarDecl
15   ;
16
17 VarDecls: VarDecl
18     |      VarDecls VarDecl
19     ;
20
21 MethodDecl: PUBLIC STATIC MethodType ID OCURV
22               FormalParams CCURV OBRACE VarDecls
23               Statements CBRACE
24
25     |         PUBLIC STATIC MethodType ID OCURV
26               FormalParams CCURV OBRACE VarDecls CBRACE
27
28     |         PUBLIC STATIC MethodType ID OCURV
29               FormalParams CCURV OBRACE
30               Statements CBRACE
31
32     |         PUBLIC STATIC MethodType ID OCURV
33               FormalParams CCURV OBRACE CBRACE
34     ;
35
36 MethodType: Type
37     |         VOID
```

```
38      ;
39
40  Statements: Statement
41      |       Statement Statements
42      ;
43
44  FormalParams: RealParams
45      |         STRING OSQUARE CSQUARE ID
46      |
47      ;
48
49  RealParams: Type ID
50      |       Type ID COMMA RealParams
51      ;
52
53  VarDecl: Type IDs SEMIC
54      ;
55
56  IDs:  ID
57      | IDs COMMA ID
58      ;
59
60  Type:  INT OSQUARE CSQUARE
61      |  BOOL OSQUARE CSQUARE
62      |  INT
63      |  BOOL
64      ;
65
66  Statement: OBRACE CBRACE
67      |      OBRACE Statements CBRACE
68      |      IF OCURV Expr CCURV Statement %prec THEN
69      |      IF OCURV Expr CCURV Statement ELSE Statement
70      |      WHILE OCURV Expr CCURV Statement
71      |      PRINT OCURV Expr CCURV SEMIC
72      |      ID ASSIGN Expr SEMIC
73      |      ID OSQUARE Expr CSQUARE ASSIGN Expr SEMIC
74      |      RETURN SEMIC
75      |      RETURN Expr SEMIC
76      ;
77
78  Expr:      NEW INT OSQUARE Expr CSQUARE
79      |      NEW BOOL OSQUARE Expr CSQUARE
80      |      exprIndexable %prec REDUCEEXPRESSON1
81      ;
82
83  exprIndexable: exprIndexable OSQUARE Expr CSQUARE
84      |          Expr AND Expr
85      |          Expr OR Expr
86      |          Expr LE Expr
87      |          Expr GE Expr
88      |          Expr EQ Expr
89      |          Expr NEQ Expr
90      |          Expr GEQ Expr
91      |          Expr LEQ Expr
```

```
92      |               Expr PLUS Expr
93      |               Expr MINUS Expr
94      |               Expr MULT Expr
95      |               Expr DIV Expr
96      |               Expr MOD Expr
97      |              NOT Expr %prec UNARY_HIGHEST_VAL
98      |               PLUS Expr %prec UNARY_HIGHEST_VAL
99      |               MINUS Expr %prec UNARY_HIGHEST_VAL
100     |               Terminal
101     |              OCURV Expr CCURV
102     |               Expr DOTLENGTH
103     |               PARSEINT OCURV ID OSQUARE Expr
104                     CSQUARE CCURV
105     ;
106
107 Terminal:       ID
108     |           INTLIT
109     |           BOOLLIT
110     |           ID OCURV Args CCURV
111     |           ID OCURV CCURV
112     ;
113
114 Args:           RealArguments
115     ;
116
117 RealArguments:  Expr
118     |            Expr COMMA RealArguments
119     ;
```

4 Construção da Árvore de Sintaxe Abstrata

Uma abstracção poderosa implementada em vários compiladores é a Árvore de Sintaxe Abstracta (AST). Esta árvore serve como representação estandardizada das várias árvores de derivação internas que possam surgir na análise sintáctica de um programa, garantindo que a análise semântica se torna numa camada à parte, cujo elo de articulação é precisamente a AST. É uma forma de representação intermédia ainda de algum alto nível.

Foi-nos especificada uma estrutura para a AST que deveríamos implementar. Procurámos seguir de perto o a AST fornecida, reproduzindo-a cuidadosamente, tendo um "tipo" de nó para cada tipo de nó da AST.

Se o projecto pudesse ser desenvolvido facilmente numa linguagem orientada a objectos como C++, faríamos uso extenso de herança e de polimorfismo no seu decorrer. Em particular, desenvolveríamos uma classe abstracta Node, representando um nó, da qual derivariam outros nós como Statement, etc.

No entanto, dadas as limitações de tempo e de linguagem que tivemos[2], optámos por implementar apenas convenções e pequenos mecanismos com um toque de programação orientada a objectos.

4.1 Representação de nós

A nossa AST é internamente representada por nós e suas ligações. Todos os nós são, no entanto, de um mesmo tipo: a estrutura *node_t*. Esta estrutura pretende, pois, ser de algum modo "polimórfica", tendo comportamentos distintos em contextos distintos, bem como membros que apenas são utilizados em certos contextos. A sua definição, que brevemente analisaremos, é a seguinte:

```
1  struct _node_t {
2
3    /* Type of this node (Program, VarDecl, etc..) */
4    nodetype_t nodetype;
5
6    /* Fixed children used in Statements / Operators. */
7    node_t* n1;
8    node_t* n2;
9    node_t* n3;
10   node_t* n4;
11
12   /* Pointer to the next node, in case this node is part of a
     linked-list */
13   node_t* next;
14
15   /* Type of the node, might be TYPE_INT, TYPE_BOOL, etc. In
     some cases,
```

[2]Sabemos que era possível integrar o *yacc* com C++, mas optámos por não dedicar demasiado tempo a este assunto, privilegiando o desenvolvimento do projecto em si.

```
16          such as MethodDeclaration, it might be TYPE_VOID. */
17      ijavatype_t type;
18
19      /* The id, used with some nodes, such as MethodDecl and
        ParamDecl. */
20      char* id;
21
22      /* This is the type of the tree starting at this node. Or so
         to say, it is
23          its "role". For instance, for the "+" node in 3+5 it is
        TYPE_INT.
24          Filled during semantic analysis */
25      ijavatype_t tree_type;
26  };
```

Para distinguir entre os diferentes nós, implementámos uma variável denominada *nodetype*, do tipo $nodetype_t$, o tipo de um nó, definido na seguinte extensa enumeração:

```
1   typedef enum {
2     NODE_PROGRAM,
3     NODE_VARDECL,
4     NODE_METHODDECL,
5     NODE_METHODPARAMS,
6     NODE_METHODBODY,
7     NODE_PARAMDECLARATION,
8     NODE_STATEMENT_COMPOUNDSTATEMENT,
9     NODE_STATEMENT_IFELSE,
10    NODE_STATEMENT_PRINT,
11    NODE_STATEMENT_RETURN,
12    NODE_STATEMENT_STORE,
13    NODE_STATEMENT_STOREARRAY,
14    NODE_STATEMENT_WHILE,
15    NODE_OPER_OR,
16    NODE_OPER_AND,
17    NODE_OPER_EQ,
18    NODE_OPER_NEQ,
19    NODE_OPER_LT,
20    NODE_OPER_GT,
21    NODE_OPER_LEQ,
22    NODE_OPER_GEQ,
23    NODE_OPER_ADD,
24    NODE_OPER_SUB,
25    NODE_OPER_MUL,
26    NODE_OPER_DIV,
27    NODE_OPER_MOD,
28    NODE_OPER_NOT,
29    NODE_OPER_MINUS,
30    NODE_OPER_PLUS,
31    NODE_OPER_LENGTH,
32    NODE_OPER_LOADARRAY,
33    NODE_OPER_CALL,
```

```
34    NODE_OPER_NEWINT,
35    NODE_OPER_NEWBOOL,
36    NODE_OPER_PARSEARGS,
37    NODE_NULL,
38    NODE_TYPE,
39    NODE_LAST_NODE_TYPE
40  } nodetype_t;
```

Esta enumeração detalha todos os tipos de nós que poderemos ter, inclusivamente o nó "Null", querendo isto dizer que, apesar de podermos representá-lo internamente através de ponteiros para null, optámos por criá-lo explicitamente, facilitando a tarefa de impressão da árvore. O nó "NODE_LAST_NODE_TYPE" é utilizado para indicar o fim de certas tabelas que são usadas internamente (como se de um terminador nulo se tratasse)

Tendo em conta que os nós que representam IDs e literais possuem várias características em comum, foram de algum modo agregados num nó único do tipo $NODE\_TYPE$. Para averiguar concretamente qual o "subtipo" de um nó do tipo $NODE\_TYPE$ (i.e: averiguar se é um ID, BOOLLIT; INTLIT, etc), torna-se necessário estudar o campo "type" da estrutura "node_t", que assume um valor da seguinte enumeração:

```
1   typedef enum {
2     TYPE_INT,
3     TYPE_BOOL,
4     TYPE_INTARRAY,
5     TYPE_BOOLARRAY,
6     TYPE_STRINGARRAY,
7     TYPE_VOID,
8     TYPE_ID,
9     TYPE_INTLIT,
10    TYPE_BOOLLIT,
11    TYPE_STRING,
12    TYPE_METHOD,
13    TYPE_UNKNOWN
14  } ijavatype_t;
```

Vemos que internamente os tipos "TYPE_STRINGARRAY", "TYPE_STRING" e "TYPE_METHOD" existem, apesar de não corresponderem a tipos efectivos da linguagem a implementar. Os dois primeiros são utilizados internamente em verificações semânticas, enquanto que o último, actualmente pouco utilizado, serviu anteriormente para simbolizar métodos no decorrer da construção da AST[3]. Por último, o tipo "TYPE_UNKNOWN" simboliza o desconhecimento de um dado tipo, quer durante o desenvolvimento da

[3]Este tipo mantém-se devido à rapidez de desenvolvimento do projecto e o receio de "quebrar algo que funcionava", pois poderia remover-se, se se alterassem outras tabelas que indirectamente o utilizam também.

AST, quer em métodos que precisem de simbolizar o erro a encontrar um dado tipo[4]. Realçamos que o tipo $ijavatype\_t$ será usado internamente para representar um tipo de dados suportado em iJava, daí a utilidade particular adicional de incluir esta divisão conceptual e de implementação do $node\_t$.

A nomenclatura dos nós também segue um padrão que tenta impor alguma hierarquia e lógica. Por exemplo, os nós que dizem respeito a statements são todos prefixados por "NODE_STATEMENT_".

4.2 Representação da árvore

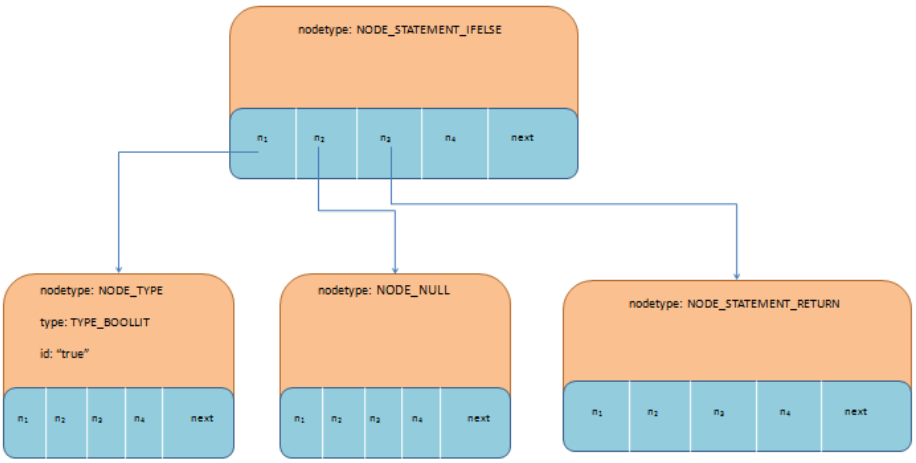

Figura 2: Representação gráfica de um nó do tipo "If". No caso, representa a expressão *if (true) else return;*.

A árvore e o nó estão intimamente relacionados. De facto, uma árvore fica definida como o nó da sua raíz. Dado um dado nó, existe uma árvore com raíz nesse nó.

Os filhos n_1, n_2, n_3 e n_4 são utilizados para indicar nós filhos de um dado nó, nos casos em que se sabe que um dado nó terá um número fixo de filhos, e pela ordem em que surgem na gramática (esquerda, direita). Por exemplo, o nó $NODE\_STATEMENT\_ELSEIF$ tem, segundo a AST fornecida, de ter 3 nós. Desse modo, n_1, n_2 e n_3 estão preenchidos, respectivamente, com a condição, o statement correspondente à verificação da condição ("then") e o statement correspondente à não verificação da condição ('else").

[4]Por exemplo, um método que devolva o tipo de um dado identificador, após consulta na tabela de símbolos, mas que seja invocado para um identificador inválido.

Em anexo (Anexo A), apresentamos uma tabela que detalha a estruturação de todos os nós da AST. A regra "geral" a ter em conta é que sempre que um nó tem um potencial número variável de filhos, então armazena-os numa lista ligada, apontando para o primeiro nó da mesma, geralmente a partir de n_1, mas não necessariamente. Reiteramos que o Anexo A esclarece todas as dúvidas quanto à AST.

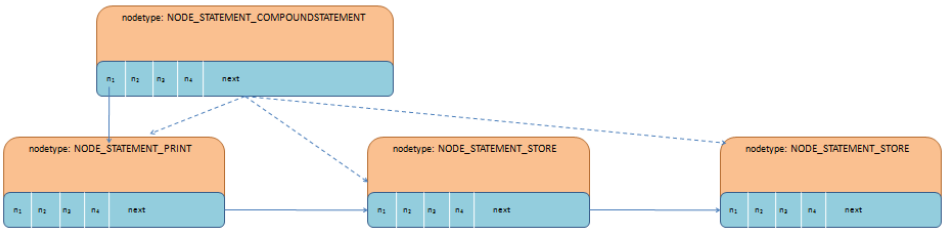

Figura 3: Representação gráfica de um nó do tipo "Compound Statement". A tracejado encontra-se uma representação mais conceptual da árvore, enquanto que as setas preenchidas indicam a verdadeira representação interna da mesma.

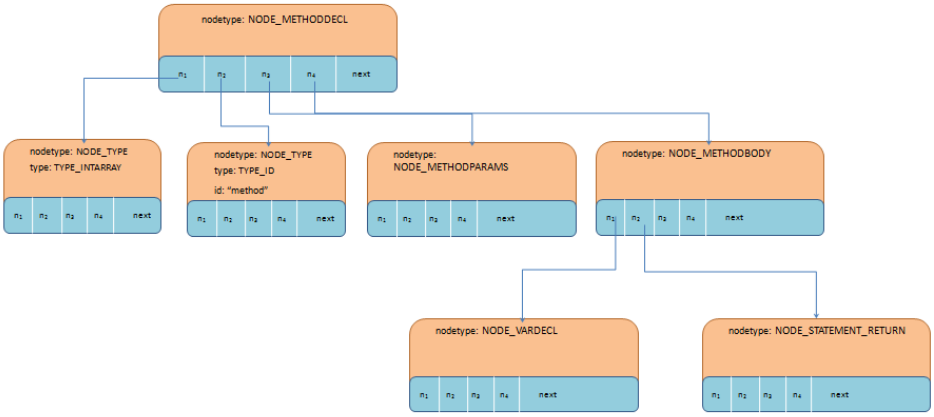

Figura 4: Representação gráfica de um nó do tipo "MethodDeclaration". Este é o único nó que utiliza todos os ponteiros (n_1, n_2, n_3, n_4 e, potencialmente, *next*).

4.3 Construção da árvore

4.3.1 Métodos auxiliares

A par da estrutura *node_t*, existe ainda um conjunto de funções que auxiliam a manipular e criar nós. No espírito da programação orientada a objectos, existe uma função de criação de nós, *node_create*, que apenas serve de base (é invocada por) outras funções, uma por cada nó que existe na AST. Assim, existem, por exemplo, *node_create_vardecl*, *node_create_methodbody*, etc. Em particular, no caso dos operadores binários e unários, recorremos a macros que geram estas funções e suas declarações por nós.

Para ligar nós uns aos outros (utilizando o next), implementámos o método *node_append*, que adiciona a lista começada pelo segundo argumento ao final da lista começada pelo primeiro argumento.

4.3.2 YACC

Utilizando o sistema de pilha interno do $YACC$, fazendo uso das *unions* descritas na análise lexical, criamos nós quando chegamos a uma folha e propagamo-los, subindo na árvore, utilizando o método *node_append* sempre que necessário. Deste modo, como temos um método individual para todos os nós, e uma só estrutura, bem como o sistema de pilha do $YACC$, o código torna-se limpo e fácil de implementar.

4.3.3 CompoundStatements

Os CompundStatements são um caso particular e que se desvia levemente do resto do código. Uma vez que temos um símbolo *Statements*, que permite construir listas de *Statements*, aproveitámos este facto para introduzir uma versão modificada do *node_append*, que aceita argumentos nulos, o *node_statement_append_statement* (e que internamente utiliza o *node_append*), Para além disso, quando encontramos a regra *Statement: OBRACE Statements CBRACE*, invocamos a função *node_create_statement_potential_compoundstatement*, que, ou retorna a lista de statements que recebe (criada durante o processo de criação da árvore do $YACC$), ou então retorna um recém-criado nó do tipo $NODE\_STATEMENT\_COMPOUNDSTATEMENT$, cujo n_1 aponta para a lista passada (caso haja mais de 2 statements na lista). Esta função limita-se, pois, a agir diferenciadamente consoante o número de nós na lista que recebe.

4.3.4 Nós null

Conforme já referido, os nós null são explicitamente criados, com o tipo $NODE\_NULL$, facto que facilita a impressão

4.3.5 Impressão da AST

Uma vez que apenas temos um tipo de nós, $node\_t$, a impressão pode ser feita de forma recursiva. Ao estabelecermos uma tabela de strings que aceita como índices os valores inteiros correspondentes à enumeração que representa o tipo de cada nó, podemos imprimir facilmente a representação em string de cada nó. Por exemplo, para obter o nome do nó x, basta-nos aceder a $tabela[x- > nodetype]$. Na verdade, recorremos a duas tabelas, já que o nó do tipo $NODE\_TYPE$ pode ter diferentes strings que lhe correspomde, e que são estas mapeadas de acordo com outra tabela para a enumeração $ijavatype\_t$.

A impressão é, pois, feita de forma recursiva, passando como argumento a profundidade, seguindo sempre os ponteiros n_1, n_2, n_3 e n_4, primeiramente (para descer na árvore) e $next$, em segundo lugar, para percorrer o mesmo nível da árvore, da esquerda para a direita.

Esta tabela de lookup e o certo "polimorfismo" da nossa estrutura de dados permite, então, uma maneira simples e eficiente de fazer a impressão.

5 Análise Semântica

Até esta fase da compilação, os passos executados, estruturas criadas, etc tinham como principal objetivo certificar que o programa, de facto, estava escrito na linguagem *iJava* adotada.

Na fase da *Análise Semântica* pretendemos alargar esse estudo, procurando aferir se as instruções que constituem o programa possuem, de facto, algum significado em *iJava*.

Assim, nesta fase da compilação, a principal preocupação irá recair sobre as operações a realizar indicadas no programa, verificando a compatibilidade dos tipos de dados envolvidos nestas. Nesse sentido, necessitamos de criar uma estrutura que, de alguma forma, nos permita armazenar e consultar as variáveis (e os respetivos tipos) a que podemos aceder a partir de uma qualquer zona do programa.

Para tal, foi necessário criar e implementar *Tabelas de Símbolos*, que nos permitem aceder a essas mesmas informações. Desta forma podemos verificar os tipos dos dados envolvidos nas diversas operações que constituem o programa, detectando eventuais situações de incompatibilidade entre tipos, inválidas nos programas.

5.1 Tabelas de Símbolos

Uma *Tabela de Símbolos* (ou um *Ambiente*) é, como já referimos, uma estrutura de dados que mapeia identificadores com os seus tipos e localizações.

Como a nossa linguagem apenas permite a execução de programas com uma única classe, que possui métodos e atributos estáticos e públicos, necessitamos de conhecer todos os métodos e atributos que compõem a classe, bem como as variáveis definidas em cada um dos métodos.

Desta forma podemos facilmente detectar situações em que são invocados métodos não declarados na classe, ou em que são utilizados atributos não definidos na classe ou para o método no qual se pretende utilizar o atributo.

Por estas razões mantemos uma *Tabela de Símbolos* para a classe na qual está contido o programa, sendo também mantida uma *Tabela de Símbolos* para cada método definido para a classe.

5.1.1 Criação das Tabelas de Símbolos

Cada *Tabela de Símbolos* é criada a partir da *Árvore de Sintaxe Abstrata*, construída na etapa anterior da compilação.

Assim, para a criação da *Tabela de Símbolos* da classe do programa, percorremos os nós da *Árvore de Sintaxe Abstrata* relativos às declarações de atributos e métodos, criando uma nova entrada na tabela (ou uma tabela

inteiramente nova) para cada método ou atributo declarado, na qual se armazenam o nome do método ou atributo declarado.

No caso da entrada corresponder a um atributo é também armazenado o tipo do atributo. Caso a entrada corresponda à declaração de um método é guardado uma referência para a *Tabela de Símbolos* do método.

Na criação da *Tabela de Símbolos* de um método da classe percorremos os nós da *Árvore de Sintaxe Abstrata* relativos às instruções que compõem o método, criando na sua *Tabela de Símbolos* uma entrada onde armazenamos o seu nome, outra onde é armazenado o tipo de retorno, e uma entrada para cada argumento ou variável local declarada, contendo o respetivo tipo.

5.1.2 Implementação

Na listagem que se segue, apresentamos a estrutura de dados utilizada para representar e implementar cada *Tabela de Símbolos* criada, à qual demos o nome de *sym_t*:

```
1  typedef struct _sym_t sym_t;
2
3  struct _sym_t{
4    ijava_table_type_t node_type;
5    char* id;
6    ijavatype_t type;
7    int is_parameter;
8    sym_t* next;
9    sym_t* table_method;
10   node_t* method_start;
11 };
12
13 typedef enum {
14   CLASS_TABLE,
15   METHOD_TABLE,
16   VARIABLE,
17   METHOD
18 } ijava_table_type_t;
19
```

Chamamos também a atenção do leitor para a definição do tipo *ijava_table_type_t*, apresentado a par da estrutura *sym_t*.

Lembramos ainda que os tipos *ijavatype_t* e *node_t* foram apresentados na definição das estruturas utilizadas na *Árvore de Sintaxe Abstrata*.

Passemos a detalhar um pouco mais a implementação apresentada.

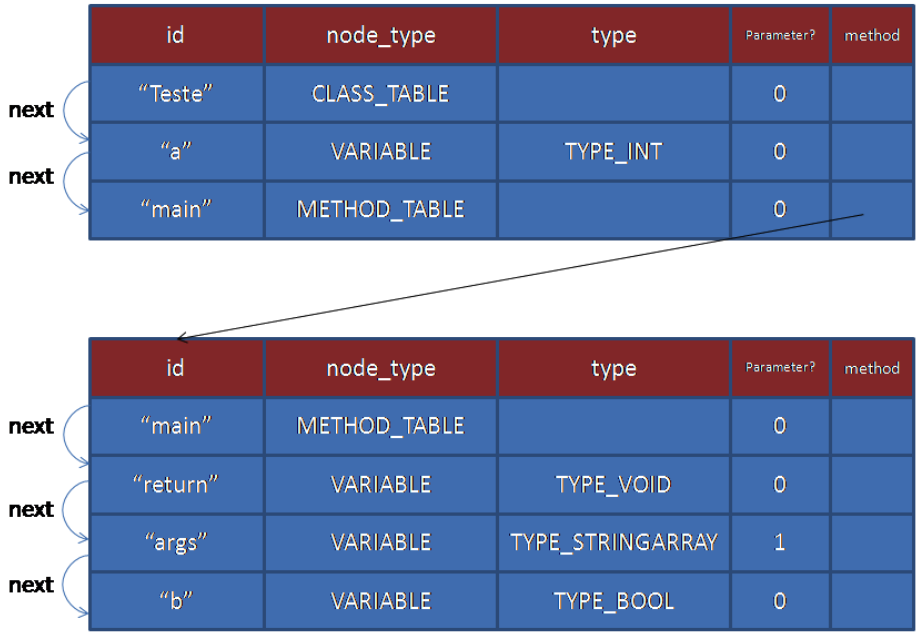

Figura 5: Representação gráfica, conceptual, mas orientada para a implementação das tabelas de símbolos. As tabelas em si são listas ligadas, e cada entrada pode ser uma variável, uma entrada inicial com metadata, ou uma função (apontando para uma próxima tabela). Note-se que a primeira variável tem sempre o nome "return" e corresponde ao tipo de retorno de um método.

Cada entrada da *Tabela de Símbolos* corresponde a uma estrutura do tipo *sym_t*, sendo cada tabela constituída por uma ou mais entradas.

As nossas *Tabelas de Símbolos* não passam de simples listas ligadas, onde o primeiro elemento (a cabeça da lista) caracteriza a tabela, possuindo informação relativa ao seu nome (que coincide com o nome da classe ou do método em questão), armazenado em *id*, e tipo, que se encontra em *node_type*. O tipo indica-nos se a entrada em questão corresponde a uma tabela de uma classe ou de um método, ou a uma declaração de um método ou de uma variável. Os diferentes elementos da lista encontram-se ligados pelo valor armazenado em *next*, um ponteiro para o próximo elemento da lista.

No caso da *Tabela de Símbolos* da classe, os restantes elementos vão corresponder a declarações de atributos ou métodos, possuindo no seu campo *id* o nome do atributo ou método declarado. Os que correspondem a declarações de atributos da classe vão possuir o seu tipo (*int*, *boolean*, etc) armazenado em *type*. Já os que correspondem a declarações de métodos refe-

renciam a *Tabela de Símbolos* desse método através do campo *table_method*. Adicionalmente, estes últimos possuem também no campo *method_start* uma referência para o nó da *Árvore de Sintaxe Abstrata* onde se encontra a declaração do método.

Relativamente à *Tabela de Símbolos* de um método, o seu primeiro elemento está de acordo com o descrito anteriormente. Segue-se um elemento que contém o tipo de retorno do método, armazenado no campo *type*. Posteriormente encontram-se os argumentos do método, no caso de existirem, também eles com a informação relativa ao nome e tipo presente em *id* e *type*, respetivamente. Estes elementos possuem também o valor 1 no campo inteiro *is_parameter*, que indica que se tratam de parâmetros do método. Por fim, encontram-se as variáveis locais, pela ordem de declaração no método, sendo a informação relativa ao seu nome e tipo armazenada nos campos *id* e *type*, respetivamente.

5.1.3 Detecção de erros semânticos nas declarações

É durante a construção das tabelas de símbolos que também verificamos se existem declarações inadequadas (múltiplas, por exemplo). Se, por exemplo, tentarmos inserir duas vezes uma variável com um dado nome numa dada tabela, teremos encontrado uma definição múltipla.

5.1.4 Impressão

A impressão das tabelas de símbolos faz-se novamente com recurso a uma tabela de lookup, como já outrora feito para a AST. Este mecanismo permite-nos facilmente modificar todas as strings que esperamos que apareçam no ecrã aquando do output desta tabela.

5.1.5 Conclusão

Após a análise das estruturas de dados apresentadas podemos retirar algumas conclusões acerca das implicações que a escolha destas estruturas teve na implementação das *Tabelas de Símbolos*.

É clara a utilização da mesma representação para estruturas conceptualmente distintas, como é o caso da tabela de símbolos da classe e dos métodos, ou das declarações de atributos e métodos de uma classe.

De facto, reconhecemos que a utilização de diferentes estruturas seria mais adequado de um ponto de vista conceptual, já que nos permitiria distinguir melhor os elementos que pretendemos representar, evitando a existência de campos não utilizados em algumas estruturas.

No entanto, optámos por manter esta representação dado que a consideramos mais simples ao nível da implementação, permitindo-nos generalizar algumas operações como a impressão dos elementos das diferentes tabelas, e até a procura de um elemento numa tabela.

5.2 Detecção de Erros Semânticos (para além de declarações)

A detecção dos restantes erros semânticos faz-se percorrendo recursivamente a AST, tomando partido de algumas informações armazenadas na tabela de símbolos. A ideia geral do algoritmo consiste em iterar sobre todos os statements e avaliar a sua validade semântica. Para tal, aplicamos um algoritmo recursivo nas expressões que os envolvem. Por exemplo, no statement $System.out.println(a[2])$, percorreremos recursivamente o operando "a[2]"(do tipo $LOADARRAY$) e os seus operandos ("a" e "2"). Expressões complexas são decompostas e analisadas recursivamente, verificando se estão a ser aplicadas a tipos correctos e se os tipos são coerentes entre si.

No caso particular da maior parte dos operadores binários, possuímos inclusivamente o conjunto de macros que funcionam quase como "regras" da aplicação deste operador e que são quase auto-explicativas. Um excerto:

```
1     int is_binary_operator_allowed(nodetype_t oper, ijavatype_t
      lhstype, ijavatype_t rhstype) {
2     ALLOW_BIN_OPER_SAMETYPE_BOOL(NODE_OPER_AND);
3     ALLOW_BIN_OPER_SAMETYPE_BOOL(NODE_OPER_OR);
4
5     ALLOW_BIN_OPER_SAMETYPE_ALL(NODE_OPER_EQ);
6     ALLOW_BIN_OPER_SAMETYPE_ALL(NODE_OPER_NEQ);
7
8     ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_LT);
9     ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_GT);
10    ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_GEQ);
11    ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_LEQ);
12    ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_ADD);
13    ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_SUB);
14    ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_MUL);
15    ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_DIV);
16    ALLOW_BIN_OPER_SAMETYPE_INT(NODE_OPER_MOD);
17
18   ALLOW_BIN_OPER(NODE_OPER_LOADARRAY, TYPE_INTARRAY, TYPE_INT);
19   ALLOW_BIN_OPER(NODE_OPER_LOADARRAY, TYPE_BOOLARRAY, TYPE_INT)
      ;
20
21
22    ALLOW_BIN_OPER(NODE_OPER_PARSEARGS, TYPE_STRINGARRAY,
     TYPE_INT);
23
24
25  return 0;
26 }
27
```

Torna-se claro que este método é como que uma tabela de regras, o que facilita a implementação, a manutenção e, de certo modo, a portabilidade do código.

No entanto, dados os constrangimentos de tempo do projecto, outros operadores e outros statements não foram implementados com o mesmo sistema, não invalidando a correcção dos mesmos.

A função recursiva que progressivamente analisa sub-árvores, obtém os seus tipos e verifica se são aplicáveis no contexto de um dado operador, chama-se *get_tree_type*, apesar de admitirmos que um nome mais apropriado talvez fosse *get_expression_type*[5].

A recursividade pára quando se alcança um nó do tipo $NODE_TYPE$, que tanto pode ter um tipo directamente associado (isto é, ser um literal, como "true" ou "0xAB"), como referenciar um outro id, caso em que se torna necessário aceder à tabela de símbolos[6] para saber que tipo lhe corresponde. A informação dos tipos é depois propagada para as funções chamantes. Torna-se também necessário referir que a operação de invocação de funções também é um dos casos base de fim de recursividade.

Neste processo de análise, é ainda verificada a validade dos literais[7], quando são encontrados no fim da recursividade.

Caso um operador não possa aplicar-se aos tipos que foram já obtidos das sub-árvores da AST, o programa termina com um erro semântico apropriado

Relativamente ao método que percorre os statements, chama-se *recurse_down*, pois é responsável por iniciar o processo recursivo de análise de statements individuais. Para além disso, invoca-se a si mesmo, recursivamente, caso alcance um Compund Statement.

Destacamos ainda que não percorremos de novo toda a AST para fazer esta análise. Em vez disso, durante o processo de construção das tabelas de símbolos, armazenamos ponteiros para os nós da AST que dizem respeito a cada função. Deste modo, apenas precisamos de percorrer todos os métodos que temos na tabela de símbolos e seguir os ponteiros deles para os nós respectivos da AST.

[5] A ideia do nome original advém do facto de estarmos a obter o "tipo" da subárvore passada como argumento

[6] Mais precisamente, acede às várias tabelas de símbolos: a do método actualmente a ser analisado e a da tabela de símbolos da classe

[7] Se não são demasiado largos, e se representam correctamente números.

6 Geração de Código

A geração de código faz-se de uma forma extremamente semelhante à maneira como fazemos a análise sintáctica. Na verdade, ambas as tarefas podiam unir-se numa só, e optámos por não o fazer por questões de facilidade de implementação, modularização de código e constrangimentos de tempo.

De facto, a geração de código consiste em percorrer a AST e, para cada nó, produzir um conjunto de instruções correspondentes à representação intermédia *LLVM IR*, que geralmente se mapeiam quase directamente. Por exemplo os operadores binários correspondem todos a uma única instrução simples, havendo mapeamento directo entre nós da árvore e instruções *LLVM IR*.

A AST é percorrida exactamente da mesma forma que na análise semântica, gerando primeiro o código para obter os resultados de subexpressões. Por exemplo, em $(a+b)+c$, primeiro é gerado o código para obter o valor de $(a+b)$ numa variável temporária, que depois é passada à operação de soma em conjunto com c. Esta lógica recursiva é a que já antes encontrámos durante a análise semântica, com a diferença de que nesta última procurávamos obter os tipos de subexpressões (e, portanto subárvores), enquanto que na geração de código procuramos obter os resultados destas.

6.1 Variáveis temporárias

O nosso código faz uso extensivo de variáveis temporárias pertencentes a funções. Implementámos um método *get_local_var_name*, que nos retorna a próxima variável temporária a utilizar. Sendo assim, retornará progressivamente %1, %2, %3, etc. Este contador de variáveis é reinicializado sempre que entramos na geração de código para uma nova função. Optámos por abstrair este comportamento numa função, em vez de manualmente lidar com o contador de variáveis, para permitir que outros métodos de obter variáveis temporárias possam mais tarde vir a ser implementados – trata-se de uma camada de abstracção adicional.

6.2 Labels

Ao longo do código precisaremos de utilizar labels, emparilhadas com statements de branching. Para tal, implementámos um contador que nunca reinicia e que constrói labels da forma "label1", "label2", etc. Não há, deste modo, qualquer informação "útil" de debugging nos nossos nomes de labels, escolha que tomámos por rapidez de implementação.

6.3 Estruturas de dados

Ao longo do nosso código de geração de *LLVM IR*, a estrutura de dados mais utilizada é o tipo de dados *llvm_var_t*, que representa uma variável

actualmente existente no código gerado. Ela associa informação sobre a sua representação ("*%variavel*", por exemplo), o tipo de dados associado (*TYPE_INT*, por exemplo) e se a variável tem memória associada (globais e pilha) ou não. Esta última noção é internamente referida como "ser value". Consideramos que uma variável é um "value" se não tiver memória associada, isto é, se for temporária. Ao mantermos a informação sobre quais variáveis são temporárias ou não, ficamos a saber se temos de fazer loads, por exemplo. A declaração da estrutura que temos vindo a abordar é a seguinte:

```
1  typedef struct _llvm_var_t {
2    char* repr; /* Representation: %1,@1, etc... */
3    ijavatype_t type; /* TYPE_INTARRAY, TYPE_INT... */
4    int value; /* Raw value or pointer? */
5  } llvm_var_t;
```

A par desta estrutura, existem métodos para a criar e destruir, bem como alguns métodos auxiliares. Por exemplo, existe um método que obtém uma variável não alocada ("temporária" / "loaded"[8]) a partir de uma alocada.

6.4 Retorno de métodos

O "return" foi implementado recorrendo a uma label de retorno (uma por cada função) e uma variável alocada na pilha onde se guadarão os valores a retornar. A variável é declarada como ".return", para garantir que tal nome não existe em iJava. Deste modo, um return é equivalente ao "pseudo-LLVM"[9]:

```
1  store %retorno, %.return
2  goto .return
```

A existência de uma label garante que o retorno por defeito é garantido aquando da alocação da variável de retorno na pilha (é o valor com que a inicializamos), e torna o fluxo do código fácil de analisar. Também nos permite escrever um só epílogo de função, em vez de ter de escrever vários antes de cada statement de return.

Por questões de facilidade de implementação, todos os métodos void retornam um inteiro na nossa implementação.

[8] Ao longo do código, utilizamos também a ideia de uma variável estar "loaded" se tiver $value = 1$, isto é, não necessitar de um *load* para os acessos aos dados.

[9] Utilizamos um híbrido entre LLVM e assembly, em que omitimos o type-checking, para facilitar a leitura.

6.5 Declaração de funções e armazenamento de parâmetros na pilha

Para permitir a modificação de variáveis passadas como argumentos, torna-se necessário armazenar os parâmetros que são passados às funções na pilha. Para tal, optámos por os declarar com um "." no início do nome, e o nome sem o ponto é utilizado para a variável alocada na pilha.

Por exemplo, a seguinte implementação em *iJava*:

```
1  public static void function(int a, int b, boolean c, int[] d)
     {}
```

produz o seguinte código (comentado e indentado a posteriori):

```
1  define i32 @function(i32 %.a, i32 %.b, i1 %.c) {
2    %return = alloca i32, align 4
3    store i32 0, i32* %return              ; Inicializar a zero o
       valor de retorno
4    %a = alloca i32, align 4
5    store i32 0, i32* %a
6    store i32 %.a, i32* %a                   ; Guardar a na pilha
7    %b = alloca i32, align 4
8    store i32 0, i32* %b
9    store i32 %.b, i32* %b                   ; Guardar b na pilha
10   %c = alloca i1, align 4
11   store i1 0, i1* %c
12   store i1 %.c, i1* %c                       ; Guardar c na pilha
```

Observamos também algumas redundâncias do nosso código, como inicializações a zero seguidas de substituições desses mesmos valores. Tal código é aceitável e válido.

6.6 Arrays

Os arrays foram implementados recorrendo a uma estrutura do LLVM em que armazenamos tanto o tamanho como um ponteiro para o array em si. Dado que suportamos dois tipos de arrays, temos duas destas estruturas, emitidas no preâmbulo do ficheiro de output:

```
1  %.IntArray = type { i32, i32* }
2  %.BoolArray = type { i32, i1* }
```

Utilizando as instruções *getelementptr*, store e a keyword *null*, é-nos possível indexar e aceder aos vários arrays. Por uma questão de implementação (e debugging), os arrays são os únicos casos em que utilizamos variáveis temporárias com um nome mais explícito do que apenas um

número. Mais concretamente, se um array se chamar *variavel*, geraremos as variáveis temporárias *.array.variavel* e *..array.variavel*[10].

6.7 Short-circuiting

Tal como requerido pela linguagem iJava, torna-se necessário implementar short-circuiting, isto é, garantir que apenas avaliamos as expressões que temos necessariamente de avaliar. A nossa implementação reside no facto de o seguinte código

```
1 a = b && c;
```

ser equivalente a

```
1 a = b;
2 if ( a ) a = c;
```

ou, de outra forma (mais facilmente automatizável)

```
1 tmp = b;
2 if ( tmp ) tmp = c;
3 a = tmp;
```

Algo semelhante pode ser conseguido para o operador OR, pois, de facto,

```
1 a = b || c;
```

é equivalente a

```
1 tmp = b;
2 if ( !tmp ) tmp = c;
3 a = tmp;
```

É desta forma que implementamos short-circuiting em iJava, recorrendo a uma variável temporária intermédia e utilizando ifs.

Concretamente, e a título de exemplo, o seguinte excerto de código

```
1 a = b && c;
```

[10] A utilização dos pontos evita name clashing com o iJava, enquanto que a palavra "array" evita name-clashing interno com outras variáveis.

produz (indentado e comentado):

```
%1 = alloca i1, align 4    ; tmp
store i1 0, i1* %1           ; Inicializado a zero (escusado)
%2 = load i1* %b
store i1 %2, i1* %1          ; tmp = b
%3 = load i1* %1

; if ( tmp ) goto %label1; else goto %label2
br i1 %3, label %.label1, label %.label2

.label1:
    %4 = load i1* %c
    store i1 %4, i1* %1      ; tmp = c
br label %.label2             ; goto %label2
.label2:
    %5 = load i1* %1
    store i1 %5, i1* %a      ; c = tmp
```

Novamente, neste excerto observam-se pequenas ineficiências no nosso código, como inicializações a zero de variáveis que não precisavam de ser assim inicializadas.

6.8 Prólogo e epílogo

A implementação de uma função, no nosso compilador, possui duas estapas à parte, que consistem na emissão do prólogo e do epílogo.

No prólogo:

- Colocam-se os parâmetros na pilha
- Aloca-se espaço para a variável de retorno

No epílogo:

- Coloca-se o retorno da função

Optámos por fazer esta divisão conceptual porque se relaciona com a implementação mais baixo-nível de código deste género.

6.9 Operador new

O operador new foi implementado internamente recorrendo ao calloc. Escolhemos o calloc e não o malloc porque este inicializa a zero, um pré-requisito do iJava. Este operador cria um novo array com o tamanho e a memória dada e coloca-o numa variável, podendo este ser depois atribuído a outra variável. A título de exemplo, o seguinte excerto de código (assume-se que a é global):

```
a = new int[100];
```

produz (indentado e comentado)

```
  %1 = add i32 0, 100
; Alocar o novo array
%2 = alloca %.IntArray, align 8
     ; (Iniciar a Zero)
     %.array.2 = getelementptr inbounds %.IntArray* %2, i32 0,
     i32 0
     store i32 0, i32* %.array.2, align 4
     %..array.2 = getelementptr inbounds %.IntArray* %2, i32 0,
     i32 1
     store i32* null, i32** %..array.2, align 8

;Armazenar tamanho
%3 = getelementptr inbounds %.IntArray* %2, i32 0, i32 0
store i32 %1, i32* %3, align 4

;Alocar bloco com calloc
%4 = call noalias i8* @calloc(i32 %1, i32 4) nounwind
%5 = bitcast i8* %4 to i32*
%6 = getelementptr inbounds %.IntArray* %2, i32 0, i32 1

;Guardar bloco no array
store i32* %5, i32** %6, align 8

;Copiar array para o array a
%7 = load %.IntArray* %2
store %.IntArray %7, %.IntArray* @a
```

Realçamos o facto de a última etapa deste código ser gerada aquando do processamento do operador de atribuição (statement *STORE*).

No caso de variáveis globais, utilizamos o *zeroinitializer* para garantir inicialização a zero de todos os campos de uma estrutura definida em *LLVM IR*. Por exemplo:

```
@a = global %.IntArray zeroinitializer, align 8
```

6.10 Statements if-else e while

Os statements condicionais e de repetição são implementados recorrendo a labels de maneira bastante autoexplicativa. Podemos reparar que

```
if ( A ) { B } else { C }
```

é equivalente a

```
1  if ( A ) goto thenlabel; else goto elselabel;
2  thenlabel:
3    B;
4    goto endifelselabel;
5  elselabel;
6    C
7  goto endifelselabel;
```

Tal facto permite-nos gerar uma representação intermédia que mapeia quase perfeitamente para esta generalização do if. Se concretizarmos com um pseudo-exemplo:

```
1  if ( A ) { B } else { C }
```

produzirá

```
1   %1 = load i1* @A
2 br i1 %1, label %.label1 , label %.label2
3 .label1:
4 ... B ...
5 br label %.label3
6 .label2:
7 ... C ...
8 br label %.label3
9 .label3:
```

Conforme esperávamos pela nossa simplificação do if em instruções mais pequenas. O mesmo podemos fazer para o while.

De facto,

```
1  while ( A ) { B }
```

é equivalente a

```
1  labeltest:
2    if ( A ) goto labelcontinue; else goto labelendwhile;
3  labelcontinue;
4    B
5  goto labeltest;
```

E, de facto, o anterior "pseudo-código" produz:

```
1 br label %.label1
2 .label1:                    ;labeltest
```

```
%1 = load i1* @A

;if ( A ) goto labelcontinue; else goto labelendwhile;
br i1 %1, label %.label2 , label %.label3
.label2:                    ;labelcontinue
... B ...
br label %.label1  ; goto labeltest;
.label3:                ; labelendwhile
```

6.11 Acesso a parâmetros da linha de comandos

Para aceder aos parâmetros da linha de comandos, temos de lidar com a representação interna do *LLVM IR* destes parâmetros, que é em tudo similar à do C, com um *argc* e um *argv*. Por forma a facilitar a nossa implementação, o nosso código renomeia todas as referências à variável do pseudo-tipo *StringArray* para .args, um nome que sabemos que não entra em conflito com nenhum outro. Também por questões de implementação, o parâmetro *argc* chamar-se-á *.args.length*.

6.11.1 Operador .length

O caso particular do operador .length para *StringArray* é gerido simplesmente acedendo o parâmetro *.args.length*, depois de este ter sido decrementado por uma unidade (para excluir o nome do programa da contagem de argumentos).

6.11.2 Integer.parseInt

O operador *Integer.parseInt* é em si implementado percorrendo o array de strings que o *LLVM* nos fornecem tendo o cuidado de somar um offset de um para evitar aceder ao nome do programa.

O seguinte exemplo demonstra o acesso a parâmetros da linha de comandos:

```
public static void function(String[] args) {
        int c;
        c = args.length;

        c = Integer.parseInt(args[2]);
    }
```

produzindo (indentado e comentado):

```
define i32 @function(i32 %args.length, i8** %args) {
    ;Retorno
    %return = alloca i32, align 4
```

```
    store i32 0, i32* %return

    ;Local
    %c = alloca i32, align 4
    store i32 0, i32* %c

    ;Subtrair ao .length
    %1 = sub i32 %args.length, 1

    ;Armazenar em %c
    store i32 %1, i32* %c

    ; Indice
    %2 = add i32 0, 2

    ; Offset a somar
    %3 = add i32 %2, 1

    ; Aceder ao parametro
    %4= getelementptr inbounds i8** %args, i32 %3
    %5 = load i8** %4
    %6 = call i32 @atoi(i8* %5) nounwind readonly

    ;Armazenar
    store i32 %6, i32* %c

    ;Epilogo
    br label %.return1
    .return1:
    %.return_final = load i32* %return
    ret i32 %.return_final
}
```

6.12 Impressão no ecrã

Imprimir no ecrã é conseguido invocando a função printf com uma de três strings. Caso se trate de um inteiro, é-he passada a string “

Esta tabela e estas strings são emitidas no preâmbulo:

```

; String 'false\n'
@str.false_str = private unnamed_addr constant [7 x i8] c"false
    \0A\00"

; String 'true\n'
@str.true_str = private unnamed_addr constant [7 x i8] c"true\0A
    \00\00"

; Lookup table que mapeia as strings
@str.bools_array = global [2 x i8*] [i8* getelementptr inbounds
    ([7 x i8]* @str.false_str, i32 0, i32 0), i8* getelementptr
```

```
inbounds ([7 x i8]* @str.true_str, i32 0, i32 0)]
```

6.13 Operações aritméticas unárias

A implementação de operações unárias foi simples, pois todas podem ser convertidas em operações binárias. De facto, apesar de até podermos evitar o unário "+", optámos por o reduzir a uma operação binária na mesma. Estas reduções são dadas pelas seguintes igualdades:

$$-a = 0 - a$$

$$+a = 0 + a$$

$$!a = a \ \ xor \ \ true$$

Com estas igualdades podemos, então, reduzir uma expressão unária a uma expressão binária.

6.14 Comparações booleanas

As operações booleanas são implementadas recorrendo ao operador *icmp* e aos seus parâmetros, dados pela tabela seguinte:

Operação	Equivalente (para icmp)
$=$	eq
$\neq$	neq
$<$	slt
$>$	sgt
$\leq$	sle
$\geq$	slg

Por exemplo, o seguinte código

```
1  a = 5 <= 2;
```

produz

```
1 %2 = add i32 0, 5
2 %3 = add i32 0, 2
3 %4 = icmp sle i32 %2, %3
4 store i1 %4, i1* %1
5 %5 = load i1* %1
6 store i1 %5, i1* %a
```

6.15 Restantes operações e statements

A implementação das restantes operações e statements é intuitiva e auto-explicativa. Praticamente todos os operadores têm equivalentes em *LLVM* facilmente mapeáveis e a simples leitura do código é suficiente para compreender.

7 Apreciação do Trabalho