Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia

Interpretação e Compilação de Linguagens
2011/2012

Linguagem O-Blaise

Hélder Marques
31944

11 de Junho de 2012

Índice

Capítulo 1

Introdução

O objectivo deste trabalho é o de desenvolver uma linguagem, implementando o interpretador, compilador e sistema de tipos. Para além da implementação da linguagem *Blaise* foi também traçado como objectivo implementar a linguagem *O-Blaise*, incluíndo compilador e sistema de tipos. Adicionalmente, os extras que também se queriam implementar eram o da optimização das instuções *box* e *unbox*, apontadores, e das declarações de tipos.

Dos extras pretendidos apenas os apontadores não foram implementados devido a limitações de tempo. Quanto ao extra da optimização do *box* e *unbox* teve de ser imposta a restrição de não poder haver comentários no código compilado para evitar que ficassem entre estas instruções não sendo optimizadas pelo método implementado. Em relação à declaração de tipos, a funcionalidade está completamente implementada permitindo declaração de tipos recursivos.

Relativamente às linguagens *Blaise* e *O-Blaise* foram implementados o interpretador, o compilador e o sistema de tipos para ambas as linguagens.

Para uma melhor criação de testes foram também implementados comentários na linguagem. Comentários de uma linha são iniciados por // e comentários em bloco são iniciados por /* e terminados com */.

Capítulo 2

Descrição

O trabalho foi desenvolvido de forma a que apenas exista um ficheiro executável sendo que é passado para este uma flag indicando se é para interpretar ou compilar o código fonte fornecido. O código pode ser fornecido através do caminho para o ficheiro, indicando a seguir à flag qual é, ou, caso seja fornecido nenhum caminho, o programa fica à espera que seja introduzido no *stdin* o código fonte. O trabalho foi dividido em módulos de forma a permitir uma fácil manutenção do código. Para tal foram gerados os seguintes módulos:

main.ml :	Módulo principal para executar o programa
blaise_lexer.mll :	Módulo com a especificação dos tokens reconhecidos pela linguagem
blaise_parser.mly :	Módulo com a especificação da gramática da linguagem
ivalue.ml :	Módulo onde estão definidos os valores de de retorno do interpretador, o valor por omissão para cada tipo, as operações básicas sobre os valores e uma função para representar os valores em forma de string
blaise_iType.ml :	Módulo onde estão definidos os tipos das linguagens, as várias operações que podem ser feitas sobre os tipos e uma função para representar os vários tipo em forma de string
blaise_syntax.ml :	Módulo onde estão definidos os nós da AST das linguagens, funções para obter o tipo de um nó e funções para representar os vários nós em forma de string
blaise_typechk.ml :	Módulo onde está a função de tipificação dos nós da AST
blaise_semantics.ml :	Módulo onde está definida a função de avaliação das linguagens
blaise_compiler.ml :	Módulo onde está definida a função de compilação das linguagens

A interpretação/compilação de um programa divide-se várias fases. A primeira é verificar se o programa passa no parser, ou seja, se estrutura do programa está de acordo com a gramática das linguagens. Após passar esta fase é executado o verificador de tipos que verifica se o programa está semanticamente correcto e caso não esteja imprime uma mensagem de erro. Caso o programa passe na verificação de tipos é então, no caso do interpretador, avaliado o programa e impresso no *stdin* o resultado da avaliação, e no caso do compilador, compilado o programa e impressas no *stdin* as instruções *CIL* geradas.

2.1 Sintaxe

2.1.1 Sintaxe concreta da linguagem *O-Blaise*

```
P ::= program Id; B.

B ::= TD; C; V; D1 ; ...; Dn ; BS

TD ::= type t1 = T1; ...; tn = Tn;

C ::= const x1 = E1 ; ...; xn = En

V ::= var x1 ,...  ,xk:T1 ;...  ;xk ,...  ,xn:Tk

BS ::= begin S end
BS_O ::= BS | S
```

```
E  ::=
   | number
   | string
   | true
   | false
   | self
   | new E
   | E + E
   | E - E
   | -E
   | E * E
   | E / E
   | E % E
   | E = E
   | E <> E
   | E < E
   | E > E
   | E <= E
   | E >= E
   | E and E
   | E or E
   | not E
   | Id
   | E(E1 ,E2 ,...  ,En )
   | { a1 = E1 ,...  ,an = En }
   | E.a
   | [E1,...  , En ]
   | E[E]
   | (E)
```

```
D  ::=
   | function Id(x1:T1 ,...,xn:Tn ):T B
   | procedure Id(x1:T1 ,...,xn:Tn ) B
   | class Id B

S  ::=
   | E := E
   | result := E
   | while E do BS_O
   | if E then BS_O else BS_O
   | if E then BS_O
   | write(E1, ..., En )
   | writeln(E1, ..., En )
   | read(x1, ..., xn )
   | readln(x1, ..., xn )
   | E(E1, E2, ..., En )
   | S ; S

T  ::=
   | Integer
   | String
   | Bool
   | Id
   | Fun(T1, ..., Tn ):T
   | Proc(T1, ..., Tn )
   | Array(number,T)
   | Record(a1:T1, ..., an:Tn )
   | Class(X)(m1:T1, ..., mn:Tn )
   | Object(X)(m1:T1, ..., mn:Tn )
```

2.1.2 Sintaxe abstracta da linguagem *O-Blaise*

$EXPR$:

`Number:`	$int \rightarrow EXPR$
`String:`	$string \rightarrow EXPR$
`Boolean:`	$bool \rightarrow EXPR$
`Array:`	$EXPR\ list \times ITYPE \rightarrow EXPR$
`Record:`	$(string \times EXPR)\ list \times ITYPE \rightarrow EXPR$
`New:`	$EXPR \times ITYPE \rightarrow EXPR$
`Add:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Sub:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Compl:`	$EXPR \times ITYPE \rightarrow EXPR$
`Mult:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Div:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Mod:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Eq:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Neq:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Gt:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Lt:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Gteq:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Lteq:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`And:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Or:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`Not:`	$EXPR \times ITYPE \rightarrow EXPR$
`Id:`	$string \times ITYPE \rightarrow EXPR$
`GetArray:`	$EXPR \times EXPR \times ITYPE \rightarrow EXPR$
`GetRecord:`	$EXPR \times string \times ITYPE \rightarrow EXPR$
`CallFun:`	$EXPR \times EXPR\ list \times ITYPE \rightarrow EXPR$

$STAT$:

`Assign:`	$EXPR \times EXPR \times EXPR \times ITYPE \rightarrow STAT$
`While:`	$EXPR \times STAT \times ITYPE \rightarrow STAT$
`If_Else:`	$EXPR \times STAT \times STAT \rightarrow STAT$
`If:`	$EXPR \times STAT \times ITYPE \rightarrow STAT$
`Write:`	$EXPR\ list \times ITYPE \rightarrow STAT$
`WriteLn:`	$EXPR\ list \times ITYPE \rightarrow STAT$
`Read:`	$string\ list \times ITYPE\ list \times ITYPE \rightarrow STAT$
`ReadLn:`	$string\ list \times ITYPE\ list \times ITYPE \rightarrow STAT$
`Seq:`	$STAT \times STAT \times ITYPE \rightarrow STAT$
`CallProc:`	$EXPR \times EXPR\ list \times ITYPE \rightarrow STAT$

$DECL\_BLOCK$:

`Types:`	$(string \times ITYPE)\ list \rightarrow DECL\_BLOCK$
`Consts:`	$(string \times EXPR)\ list \times ITYPE \rightarrow DECL\_BLOCK$
`Vars:`	$(ITYPE \times string\ list)\ list \rightarrow DECL\_BLOCK$
`Operations:`	$OPER\ list \times ITYPE \rightarrow DECL\_BLOCK$

$OPER$:

`Function:`	$string \times (string \times ITYPE)\ list \times DECL\_BLOCK\ list\ \times$ $STAT \times ITYPE \rightarrow OPER$
`Procedure:`	$string \times (string \times ITYPE)\ list \times DECL\_BLOCK\ list\ \times$ $STAT \times ITYPE \rightarrow OPER$
`Class:`	$string \times DECL\_BLOCK\ list \times STAT \times ITYPE$

$PROGRAM$:

`Program:`	$string \times DECL\_BLOCK\ list \times STAT \times ITYPE \rightarrow PROGRAM$

$ITYPE$:

`TNumber:`	$void \rightarrow ITYPE$
`TString:`	$void \rightarrow ITYPE$
`TBoolean:`	$void \rightarrow ITYPE$
`TFun:`	$ITYPE\ list \times ITYPE \rightarrow ITYPE$
`TProc:`	$ITYPE\ list \rightarrow ITYPE$
`TArray:`	$int \times ITYPE \rightarrow ITYPE$
`TRecord:`	$(string \times ITYPE)\ list \rightarrow ITYPE$
`TRef:`	$ITYPE \rightarrow ITYPE$
`TClass:`	$string \times (string \times ITYPE)\ list \rightarrow ITYPE$
`TObject:`	$string \times (string \times ITYPE)\ list \rightarrow ITYPE$
`TType_id:`	$string \rightarrow ITYPE$
`TUnit:`	$void \rightarrow ITYPE$
`TNone:`	$string \rightarrow ITYPE$
`TUndefined:`	$void \rightarrow ITYPE$

A árvore sintáctica abstracta é construída pelo parser sendo que no campo do tipo do nó, excepto quando é indicado explicitamente qual é o tipo, é colocado $TUndefined$ para indicar que o nó em questão ainda não foi tipificado.

2.2 Interpretador

2.2.1 Valores do interpretador

$IVALUE$:

`StringValue:`	$string \rightarrow IVALUE$
`NumberValue:`	$int \rightarrow IVALUE$
`BooleanValue:`	$bool \rightarrow IVALUE$
`ArrayValue:`	$IVALUE\ array \rightarrow IVALUE$
`RecordValue:`	$IVALUE\ map \rightarrow IVALUE$
`RefValue:`	$IVALUE\ ref \rightarrow IVALUE$
`FunValue:`	$(string \times ITYPE)\ list \times DECL\_BLOCK\ list \times STAT \times ITYPE \times (IVALUE\ map)\ ref \rightarrow IVALUE$
`ProcValue:`	$(string \times ITYPE)\ list \times DECL\_BLOCK\ list \times STAT \times (IVALUE\ map)\ ref \rightarrow IVALUE$
`CustomValue:`	$void \rightarrow IVALUE$
`NoneValue:`	$void \rightarrow IVALUE$

2.2.2 Resultado

`ENV` : $IVALUE\ map$

$evalExp$:	$ENV \times bool \times EXPR \rightarrow IVALUE$
$evalState$:	$ENV \times STAT \rightarrow void$
$evalOpers$:	$ENV \times OPER \rightarrow ENV$
$evalDecls$:	$ENV \times DECL\_BLOCK \rightarrow ENV$
$evalAllDecls$:	$DECL\_BLOCK \times DECL\_BLOCK \times DECL\_BLOCK \times DECL\_BLOCK \times ENV \rightarrow ENV$
$evalProgram$:	$PROGRAM \rightarrow void$

O resultado da interpretação de um nó da árvore abstracta é um valor dos definidos acima.

2.2.3 Estruturas auxiliares

Para auxiliar a avaliação de um programa é usado um mapa no qual está o mapeamento entre quais os identificadores que existem e qual o seu valor. Existe ainda um *buffer* para se guardar as linhas lidas do *input* para serem usadas na avaliação dos nós *Read* e *ReadLn*.

2.2.4 Como funciona

A avaliação do programa começa com a avaliação do bloco de declarações. No interpretador é ignorada a secção de declaração de tipo porque apenas é necessário no verificador de tipos para validar a correcção do programa. Na parte das declarações de constantes são avaliadas as expressões asssociadas a cada identificador e adicionada ao ambiente essa relação. De seguida são avaliadas as variáveis em que a única coisa que é necessária é adicionar ao ambiente a associação do identificador fornecido ao valor de omissão do tipo que corresponde à variável. Os valores por omissão da linguagem são:

default_type : $ITYPE \rightarrow IVALUE$

TNumber	$\rightarrow$	0
TString	$\rightarrow$	" "
TBoolean	$\rightarrow$	*false*
TArray(n, t)	$\rightarrow$	$[a_0, ..., a_{n-1}] : a_i = default\_type(t)$
TRecord($[(s_0, t_0), ..., (s_n, t_n)]$)	$\rightarrow$	$\{ s_0 : default\_type(t_0), ..., s_n : default\_type(t_n) \}$
TObject(x, $[(s_0, \_), ..., (s_n, \_)]$)	$\rightarrow$	$\{ s_0 : TNone(\text{"dummy"}), ..., s_n : TNone(\text{"dummy"}) \}$

No caso de tipos declarados pelo utilizador o valor por omissão é um valor especial que não tem valor nenhum e que apenas serve para indicar que é uma variável de um tipo declarado.

Após a avaliação das variáveis é a vez de se interpretar as operações que na linguagem se dividem em funções, procedimentos e classes. A interpretação de funções e procedimentos é igual, para isso é apenas necessário criar uma referência para o ambiente actual, criar um *closure* com os tipos dos argumentos, os nós correspondentes ao bloco de declarações dentro da operação, o nó correspondente ao corpo da operação e, no caso da função, o tipo de retorno da mesma. É ainda adicionada a referência para o ambiente para haver a possibilidade de haver operações recursivas. No final é associado ao ambiente o *closure* com o nome fornecido.

Em relação às classes o procedimento que se faz é, através de *syntactic sugar*, transformar a classe numa função geradora de objectos. Para tal é percorrida a classe uma vez para se obter a lista dos métodos disponíveis, depois é criado o tipo de retorno da função criada que será um *TRecord* em que os *fields* terão os nomes dos métodos e cada um terá a *closure* associada a cada método. De seguida é adicionado ao bloco das variáveis o *self* que será o exactamente o *Record* que será retornado pela função para permitir aos vários métodos terem todos os outros métodos disponíveis através do *Record self*. Finalmente são adicionados ao corpo da *Class* a atribuição à variável *self* e o retorno do mesmo e é criado um novo nó *Function* com os parâmetros necessários e é chamada a avaliação de uma função com este nó.

Por fim é necessário avaliar o corpo do programa. De seguida explicam-se os nós mais complicados ou diferentes da linguagem dada nas aulas:

Assign

No *Assign* a parte mais complicada é a de atribuir a uma variável a cópia do valor que está do lado direito da atribuição. Para valores simples isto é facilmente conseguido obtendo a referência e atribuíndo-lhe o valor do outro valor. Quanto aos vectores e registos os valores são copiados índice a índice ou campo a campo chamando recursivamente a função que trata da atribuição de valores. No caso de procedimentos e funções o valor do *closure* da direita é copiado integralmente para a referência do lado esquerdo. Nos objectos como são traduzidos para *Records* a cópia é efectuada campo a campo.

Read

A leitura de dados de *input* é efectuada usando um buffer ao qual vamos buscar os *tokens* a serem lidos. Se for efectuado um *Read* e não existir nada no *buffer* fica à espera de dados no *stdin*. Quando existe um *ReadLn* o comportamento é igual sendo que no final o *buffer* é esvaziado.

CallProc

A chamada de um procedimento tem muitas semelhanças com a avaliação de um programa. Primeiro é avaliado o primeiro parâmetro do nó para obtermos o *Closure* do procedimento. De seguida são percorridos os argumentos presentes no *Closure* para se associar ao ambiente os parâmetros com o respectivo identificador. Depois são avaliados os blocos de declarações do procedimento e finalmente avalia-se o corpo do procedimento no ambiente resultante da avaliação dos blocos de declarações.

CallFun

A chamada de uma função é muito semelhante à chamada de um procedimento. No caso da função antes de se avaliar o corpo da mesma é necessário associar, ao ambiente no qual este vai ser avaliado, uma variável com o identificador *result* com o valor por omissão do tipo de retorno da função. Após a avaliação do corpo é necessário procurar no ambiente pelo valor da variável *result* e devolver esse valor.

New

A criação de um novo objecto é simplesmente uma chamada à função geradora de objectos da *class* pretendida que foi declarada no bloco de declarações.

Desreferênciação implícita

A desreferênciação implícita é efectuada recorrendo ao parâmetro booleano da função de avaliação de expressões sendo este enviado para a função *to_result*. Se este for falso a função *to_result* desreferencia o valor recebido se este for uma referência senão a função retorna sempre o próprio valor em qualquer outro caso.

Vectores

A criação de vectores trata-se de avaliar as expressões correpondentes a cada posição e guardar os seus valores num vector primitivo do *OCaml*. No caso dum vector variável é criado um vector com o tamanho indicado e é colocado em cada posição uma cópia variável do valor por omissão do tipo do vector. Para avaliar o acesso a uma posição de um vector basta avaliar a expressão correspondente ao índice e extrair o inteiro que deve resultar da sua avaliação e usar a biblioteca *Array* para aceder à posição do array desejada. Se o índice se encontrar fora dos limites do vector é lançada a excepção *Index_out_of_bounds*.

Registos

Para se criar um é necessário avaliar as expressões dos vários campos. Para guardar o valor dos campos é usado um mapa de *IVALUE*. No caso dum registo variável é necessário guardar em cada campo uma cópia variável do valor por omissão do tipo de cada campo. Para se aceder a um campo de um registo é necessário avaliar a expressão correspondente ao registo e, usando o módulo *RecordMap*, obter o valor correspondente ao campo pretendido. Caso o campo não exista é lançada a excepção *Element_not_found_in_record*.

Passagem por valor

Sempre que exista uma chamada de uma função ou de um procedimento com argumentos este devem ser passados por valor para que não seja possível efectuar afectações sobre os valores dos parâmetros. Em relação ao valores simples não é necessário fazer muita coisa, apenas é necessário remover o nó *RefValue* caso o valor seja variável. Quanto aos vectores e registos é feita uma cópia integral dos mesmos sendo que se estes forem variáveis, ou seja, se estiver dentro de um nó *RefValue*, são removidos todos esses nós na cópia resultante.

Algumas operações foram implementadas com base noutras usando *syntactic sugar* para evitar o excessivo número de operações que têm de ser definidas.

2.3 Compilador

2.3.1 Resultado

`ENV` : $(int\ map \times int\ ref)\ list$

compile_expr :	$ENV \times bool \times EXPR \rightarrow string\ list$
compile_stat :	$ENV \times STAT \rightarrow string\ list$
compile_oper :	$ENV \times OPER \rightarrow string\ list \times string\ list \times ENV$
compile_decl :	$ENV \times DECL\_BLOCK \rightarrow string\ list \times string\ list \times ENV$
compile_all_decls :	$DECL\_BLOCK \times DECL\_BLOCK \times DECL\_BLOCK \times$ $DECL\_BLOCK \times ENV \rightarrow string\ list \times string\ list \times ENV$
compile_program :	$PROGRAM \rightarrow string\ list$

O resultado da compilação de um programa é uma lista de *strings* com as instruções *CIL* para executar o programa. Nesta lista já vêm as declarações das funções e já vem optimizada sem operações de *box, unbox* e *ldobj* ao mesmo tipo seguidas.

2.3.2 Estruturas auxiliares

Para a compilação de um programa as estruturas auxiliares usadas são, para guardar a informação de quais os endereços associados a cada identificador e quantas declarações já houve em cada *Stackframe*, uma lista de pares em que o primeiro campo é o mapa com o mapeamento entre os identificadores e qual o endereço que lhe foi atribuído e o segundo campo o número de identificadores mapeados, um mapa com o mapeamento entre os identificadores dos tipos definidos pelo utilizador e a que tipo correspondem, um contador para poder atribuir novos números a *labels* necessárias para a execução de *ifs* e *whiles* em *CIL* e finalmente uma lista de inteiros para guardar o número de variáveis locais criadas em cada função, procedimento ou classe.

Para a execução do programa em *CIL* são usadas as seguintes classes em *C#*:

StackFrame:	Classe usada para guardar o *static link* e os valores das constantes, variáveis e argumentos de uma função, procedimento ou classe. Esta classe tem o apontador para o *static link* e dois vectores com os valores dos argumentos e das declarações locais.
Closure:	Classe usada para representar um closure guardando o *stackframe* de quando foi declarada a operação e o apontador para a declaração da função em *CIL*.
Cell:	Classe usada para representar uma célula, onde serão guardados os valores simples que sejam variáveis. O valor é guardado numa variável do tipo *object*
Record:	Classe usada para representar um registo. Esta classe possiu um dicionário com entradas do tipo (*string,object*) para guardar o mapeamento entre o nome dos campos e os seus valores. Esta classe possui ainda um método para obter uma cópia constante caso seja um registo variável e um método para copiar o conteúdo do registo para outro registo fornecido como argumento.

Array:	Classe usada para representar um vector. Esta classe possiu um vector de objectos para guardar os valores do vector. Tal como a classe *Record* esta classe também tem um método para obter uma cópia constante e outro método para copiar o seu conteúdo para outro vector passado como argumento. É também possível inicializar o vector com n posições e passando, opcionalmente, o valor por omissão a colocar em todas as posições.
Reader:	Classe usada para auxiliar com a leitura de valores do *stdin*. Esta classe contém um *buffer* para guardar uma linha lida do *stdin* e um contador para saber quantos *tokens* já foram lidos. Possui métodos para ler inteiros, booleanos e *strings* do *stdin* e um método para ler uma linha que basicamente esvazia o *buffer*.

2.3.3 Como funciona

A compilação de um programa começa com a criação de um novo ambiente e começo de uma nova contagem de declarações locais. De seguida são compilados os blocos de declarações. No bloco de declaração de tipos é percorrida a lista das declarações e são adicionadas ao ambiente dos tipos a associação entre o identificador e o tipo a que corresponde. Na declaração de constantes são incrementadas as declarações locais e associados os identificadores a endereços novos no *stackframe* actual. De seguida é compilada a expressão de cada constante juntamente com a compilação das instruções para colocar no *StackFrame* a nova constante. A compilação das variáveis é semelhante à compilação de constantes sendo que em vez de compilar a expressão fornecida é compilado o valor por definição para cada variável. Quanto aos procedimentos, para a sua compilação é necessário obter um novo identificar para a operação, compilar o *closure*, começar um novo ambiente, adicionar ao ambiente a ligação do nome a um endereço para poder haver recursividade e fazer um *backup* do ambiente de tipos para ser restaurado depois de compilado o corpo do procedimento. Depois é ncessário atribuir a cada parâmetro um novo endereço e adicioná-lo ao ambiente, de seguida compilam-se os blocos de declarações e finalmente o corpo do procedimento. Por fim restaura-se o ambiente de tipos, termina-se o ambiente do procedimento e compilam-se as instruções para adicionar ao *stackframe* o novo procedimento. Para as funções o processo é parecido com o procedimento sendo que antes de avaliar os blocos de declarações é necessário adicionar ao ambiente uma variável *result* com o tipo de retorno da função. Após se compilar o corpo da função é necessário compilar as intruções para se obter o resultado da função e colocá-lo na pilha. Quanto às classes a técnica usada é igual à do interpretador sendo no final chamada a compilação da função criada. Após compilar os blocos de declaração resta compilar o corpo do programa e no final optimizar o programa compilado percorrendo todas as instruções geradas e removendo *box, unbox* e *ldobj* ao mesmo tipo seguidos.

As compilações mais complicadas são explicadas de seguida:

Assign

Para se compilar uma afectação é necessário recorrer às anotações deixadas pelo verificador de tipo para se saber qual o tipo que se está a afectar. Mediante o tipo que está no nó, e à excepção dos tipos simples em que apenas é necessário fazer um *Set* à célula que representará o lado esquerdo da afectação, trocam-se os objectos no topo da pilha e chama-se o método *CopyTo* da classe respectiva ao tipo das expressões da afectação.

Read

A compilação do *Read* é auxiliada pela classe *Reader* da qual é chamado o método correspondente para ler um valor do tipo que foi anotado na verificação de tipos. No caso de um *ReadLn* basta no final chamar o método *ReadLine* do *Reader* para esvaziar o *buffer*.

CallProc

O processo de compilar a chamada a um procedimento começa por se compilar a expressão correspondente à obtenção do *closure* do procedimento. De seguida é compilada a lista de argumentos do procedimento sendo que no fim de cada um é efectuada a afectação no *stackframe* ao endereço que foi atribuído aquando da compilação da declaração do procedimento. Por fim é compilada a chamada ao procedimento obtendo o *stackframe* actual, criando um novo, iniciando os argumentos do procedimento e colocando-os na *stackframe* e finalmente obtendo-se o apontador para o procedimento que está no *Closure* e compilando a instrução *CIL* de chamada de um procedimento.

CallFun

A compilação da chamada de uma função é igual à chamada de um procedimento visto que, ao contrário do interpretador, não é necessário adicionar ao ambiente a variável *self* nem compilar as instruções para a colocar no topo da pilha como retorno da função porque a adição e a compilação da sua obtenção já foram efectuadas aquando da declaração da função.

Desreferênciação implícita

A desreferênciação é feita usando o método usado no interpretador. Se for necessário desreferenciar uma variável apenas se tem de verificar se o tipo desta é guardado numa célula e caso o seja chamar o método *GetValue* da class *Cell*.

Vectores

Os vectores foram implementados recorrendo à classe auxiliar *Array*. A compilação de um vector constante é feita chamando o construtor da classe *Array* que apenas recebe o tamanho do vector como argumento enquanto que o constructor que recebe o valor por omissão é usado quando se cria um vector variável. Isto deve-se ao facto de quando se declara um vector constante é necessário especificar qual o valor de cada posição logo não faz sentido inicializar cada posição com um valor para, logo de seguida, ser alterado. A construção do vector é muito semelhante ao interpretador, compilam-se as instruções correspondentes a cada posição do vector juntando-se a chamada ao método *Set* da classe *Array* empilhando-se os respectivos argumentos. Para se compilar o acesso a uma posição de um vector é necessário compilar

a expressão que denota o vector ao qual se pretende aceder, a expressão correspondente ao índice pretendido e no final, se necessário, adicionar a chamada ao método *Get* da classe *Cell* para se efectuar a desreferênciação da posição do vector. No caso da posição que se pretende obter não estiver nos limites do vector é lançada a excepção *System.IndexOutOfRangeException* lançada pelo *C#*.

Registos

Os registos são implementados recorrendo à classe auxiliar *Record*. Para se compilar a criação de um registo é chamado o contructor da classe *Record* para criar um novo registo. De seguida são geradas as intruções para cada expressão juntamente com o empilhamento do nome do campo e da chamada ao método *SetValue* da classe *Record*. Para a criação de um registo variavél não é necessário existir um constructor com valores por omissão porque é necessário especificar os tipos de todos os campos logo pode-se colocar em cada campo o valor por omissão para o tipo indicado. No acesso ao campo de um registo é compilada a expressão correspondente ao registo, o empilhamento do nome do campo que se pretende aceder e a chamada ao método *GetValue* da class *Record*. No casos dos registos nunca existe o acesso a um campo não existente porque essa verificação é efetuada pelo verificador de tipos.

Passagem por valor

Para se compilar a passagem de argumentos por valor é necessário recorrer às classes *C#* para se efectuar a cópia constante dos vários argumentos. Ao compilar-se a chamada de a uma função ou procedimento é chamado o método *GetConstCopy* da classe do tipo do argumento caso este seja um vector ou um registo variáveis. Na cópia dos valores serão removidos todos os objectos *Cell* colocando no seu lugar o valor que guardam.

2.4 Sistema de tipos

2.4.1 Resultado

`ENV` : $ITYPE\ map$

typechk_exp :	$ENV \times EXPR \rightarrow EXPR$
typechk_stat :	$ENV \times STAT \rightarrow STAT$
typechk_oper :	$ENV \times OPER \rightarrow string \times OPER \times ENV$
typechk_decl :	$ENV \times DECL\_BLOCK \rightarrow string\ list \times DECL\_BLOCK \times ENV$
typechk_all_decls :	$ENV \times DECL\_BLOCK \times DECL\_BLOCK \times DECL\_BLOCK \times$ $DECL\_BLOCK \rightarrow DECL\_BLOCK\ list \times ENV \times ITYPE$
typechk_program :	$PROGRAM \rightarrow PROGRAM$

O resultado devolvido pelo verificador de tipos é o nó que é passado como parâmetro sendo que este vem tipificado, bem como todos os nós que fazem parte da árvore abstracta.

2.4.2 Estruturas auxiliares

A estrutura de dados utilizada para auxiliar o verificador de tipos é um mapa que representa o ambiente em que se está a tipificar o nó, sendo que este ambiente é passado para todas quase todas as funções do verificador de tipos. Neste ambiente é guardada a informação sobre o tipo dos identificadores declarados e ainda, como extra, para guardar o tipo que é representado pelo identificador que foi declarado na zona de declaração de tipos. Esta estrutura é utilizada quando se pretende tipificar o nó ***Id*** e quando se está a fazer uma comparação de tipos para o caso em que encontramos um identificador de um tipo declarado.

2.4.3 Como funciona

O verificador de tipos começa por tipificar os vários blocos de declarações presentes num programa. A função que tipifica um bloco de declarações devolve uma lista com todos os identificadores declarados em cada bloco para que, após a tipificação de todos os blocos seja chamada a função que verifica se existem nomes duplicados. Caso existam identificadores repetidos é lançada uma excepção. Primeiro tipifica-se a declaração de tipos que apenas adiciona ao ambiente a associação entre o identificador e o tipo que representa para se utilizar quando se efectua a comparação entre tipos e um deles foi declarado pelo utilizador. De seguida são tipificadas as declarações de constantes. Para isso é necessário tipificar cada expressão e associar, no ambiente, o seu tipo ao identificador dado. Depois vem a tipificação de variáveis que também apenas associa no ambiente uma referência para o tipo indicado ao indentificador fornecido. De seguida vêm as operações e as classes. Nas operações o procedimento é muito semalhante. É necessário adicionar ao ambiente os argumentos com os tipos respectivos, depois adicionar a associação entre o nome da operação e o seu tipo para que possa existir funções recursivas e no caso das funções adicionar a ssociação entre a variável *result* e o tipo de retorno da função. De seguida tipificam-se os blocos de declarações e depois, no ambiente devolvido, tipifica-se o corpo da operação. Finalmente verifica-se se nenhum nó está tipificado com *TNone* e devolve-se o nome da operação, o nó tipificado e o ambiente actualizado. No caso das classes o procedimento é igual ao das operações sendo que é precedido pelo varrimento da classe para obter uma lista dos métodos declarados para se puder construir o tipo da classe.

Após a tipificação dos blocos de declarações tipifica-se o corpo do programa. Para se tipificar o corpo recorre-se várias vezes ao método *equals* que verifica se dois tipo são iguais ignorando se têm *TRef* ou não.

Equals

ENV: $ITYPE\ map$

Equals: $ENV \times (ITYPE \times ITYPE)\ list \times ITYPE \times ITYPE \rightarrow bool$

A função *equals* recebe um ambiente, que é onde estão declarados os tipo definidos pelo utilizador para que seja possível expandir sempre que necessário, e uma lista de tipos já comparados que serve para evitar a que se entre num ciclo infinito ao compararem-se dois tipos recursivos. Na função começa-se por se verificar se os dois tipo não foram já comparados usando a lista fornecida como argumento. Se já foram devolve-se *true* porque significa que correu tudo bem durante a expansão e comparação dos tipo e voltámos ao mesmo ponto através da recursividade de tipos. Caso contrário, adiciona-se o par de tipos a lista e comparam-se os mesmos. Caso sejam tipos simples a comparação é trivial e basta devolver o resultado da comparação primitiva do *OCaml*. No caso dos vectores e dos registo a comparação é feita chamando recursivamente a função para cada posição ou campo. Nas operações a igualdade verifica-se quando os tipos dos argumentos são iguais. Quanto aos objectos existe o caso em que é declarado um nome à cabeça do tipo para representar o próprio objecto pelo que esta associação é adicionada ao ambiente e são então comparados todos os métodos dos dois objectos comparando nome, o retorno e os argumentos dos métodos. Finalmente, em relação aos tipos definidos pelo utilizador, quando se encontra um identificador destes procura-se no ambiente o tipo que representa e chama-se novamente a comparação com o novo tipo.

Assign

A tipificação de uma afectação começa por tipificar ambas as expressões e de seguida verifica se a expressão do lado esquerdo resulta num *TRef* e caso o seja compara-se se os dois tipos são iguais.

Caso alguma verificação falhe durante a verificação é retornado o nó com o *TNone*.

Read

Para se tipificar um nó *Read* é necessário verificar se todos os tipos dos valores que se pretendem ler são tipo simples porque estes são os únicos que podem ser lidos. Para auxiliar no compilador é criada uma lista com os tipos das variáveis que se pretendem ler.

CallProc e CallFun

A tipificação da chamada de uma função ou procedimento consiste apenas na verificação se a tipificação da expressão que denota a operação resulta num *closure* válido consoante se está chamar um procedimento ou uma função. Caso o *closure* seja válido é verificado se os tipos das expressões que se pretendem passar como argumentos coincidem com os tipos dos argumentos da operação que está a ser chamada.

New

Na tipificação da construção de um novo objecto é apenas necessário verificar se a expressão denota uma classe válida e caso o seja tipificar o nó *New* com o tipo do objecto que instancia a classe indicada.

Desreferênciação implícita

Na verificação de tipos a desreferênciação implícita é efectuada quando se pretende usar o valor da expressão por isso não é necessário ter o parâmetro booleano na tipificação de expressões. Para desreferenciar uma expressão quando necessário é usada a função *unref_iType* que apenas verifica se o tipo

passado como argumento é um *TRef* e, caso o seja, devolve o tipo que está dentro do *TRef*, caso contrário devolve o próprio tipo.

Vectores

Na criação de um vector constante é necessário verificar se todas as expressões que representam os vários valores do vector têm o mesmo tipo. Para tal é percorrida a lista de expressões fornecida, são tipificados todos os seus nós e contadas quantas expressões são para se saber qual o tamanho do vector. Se todos os tipos coincidirem é então tipificada a expressão com o tipo *TArray* com o tamanho obtido da contagem de expressões e o tipo dos vários elementos. Caso alguma verificação falhe o tipo do vector fica como *TNone*.

Registos

A tipificação da criação de um registo consiste simplesmente em percorrer todas as expressões dos campos e verificar que estão bem tipificadas. Durante este processo é necessário guardar uma lista com todos os tipos para se criar no final o tipo do registo. No acesso a um campo de um registo apenas é necessário verificar que a tipificação da primeira expressão tem tipo *TRecord* e procurar na lista de campos pelo campo que pretendido e devolver o seu tipo.

Passagem por valor

A passagem de argumentos por valor não é tipificada porque quando se avalia o corpo de uma função nou procedimento os nomes dos parâmetros são associados como constantes não permitindo a sua afectação logo na chamada apenas é necessário verificar que os tipos coincidem ignorando os nós *TRef*.

Classes e Objectos

Com a introdução de objectos apenas foi necessário introduzir o caso de, no acesso a um registo, a expressão denotar um objecto pelo que o procedimento é igual, basta ir à lista de métodos do nó *TObject* e devolver o tipo do método.

Definição de tipos

A definição de foi implementada adicionando ao ambiente de tipo a associação entre o nome do tipo e o tipo que representa. Quando é encontrado um identificador de um tipo que foi definido pelo utilizador apenas é subtituído pelo tipo que representa e é verificado novamente com o tipo correcto.

Operações permitidas

`Add, Eq, Neq:`	*`strings`* `e inteiros`
`Sub, Mult, Div, Mod, Compl, Gt, Lt, Gteq, Lteq:`	`inteiros`
`And, Or, Not:`	`booleanos`

Capítulo 3

Exemplos

Para testar o programa foram usados testes criados por outros alunos à medida que se foi implementando novas funcionalidades. Foram também criados testes muito simples à medida que se ia programando para se testarem as funcionalidades. Apenas os testes mais elaborados foram sendo guardados. Alguns dos testes usados foram adaptações de outros programas, nomeadamente o problema $3n + 1$ e o jogo do galo que eram programas que estavam programados em *Pascal*. Não são conhecidas as alterações que foram necessárias porque o programa foi fornecido já na linguagem *Blaise*. Estão em anexo alguns dos testes utilizados para testar o programa.

Capítulo 4

Anexos

4.1 Testes

Expressões

Source:

```
program Test_basic_expressions;
begin

// Testing add
if 3+1 <> 4 then
        writeln("Should be 4");

if (-1)+1 <> 0 then
        writeln("Should be 0");

if "Hello" + " World!" <> "Hello World!" then
        writeln("Should be \"Hello World!\"");

//Testing sub
if 3-1 <> 2 then
        writeln("Should be 2");

if 1-2 <> -1 then
        writeln("Should be -1");

//Testing mult
if 19*0 <> 0 then
        writeln("Should be 0");

if 12*(-1) <> -12 then
        writeln("Should be -12");

//Testing div
if 3/1 <> 3 then
        writeln("Should be 3");

if 4/2 <> 2 then
        writeln("Should be 2");

if 3/2 <> 1 then
        writeln("Should be 2");

//Testing mod
if 3%1 <> 0 then
        writeln("Should be 0");

if 4%2 <> 0 then
        writeln("Should be 0");

if 3%2 <> 1 then
        writeln("Should be 1");
```

```
//Testing compl
if −0 <> 0 then
        writeln("Should be 0");

if −2 <> −2 then
        writeln("Should be −2");

if −(−3) <> 3 then
        writeln("Should be 3");

//Testing Neq
if 2 <> 2 then
        writeln("should be equal");

if "ola" <> "ola" then
        writeln("Should be equal");

//Testing Equal
if 2 = 3 then
        writeln("Should be different");

if "bla" = "bla1" then
        writeln("Should be different");

//Testing greater
if 2 > 3 then
        writeln("Should be less");

if −4 > 3 then
        writeln("Should be less");

//Testing less
if 3 < 2 then
        writeln("Should be greater");

if 3 < −4 then
        writeln("Should be greater");

//Testing greater or equal
if 2 >= 3 then
        writeln("Should be less");

if −4 >= 3 then
        writeln("Should be less");

//Testing and
if true and false then
        writeln("Should be false");

if false and false then
        writeln("Should be false");

if true and true then
        write("")
else
        writeln("Should be true");

//Testing or
if true or false then
        write("")
else
        writeln("Should be true");

if false or false then
        writeln("Should be false");

if true or true then
        write("")
else
        writeln("Should be true");

//Testing not
if not true then
        writeln("Should be false");

if not (not false) then
        writeln("Should be false");

if not false then
        write("")
else
        writeln("Should be true");
```

```
if not (not true) then
        write("")
else
        writeln("Should be true");

writeln("Test ended")
end.
```

Output:

```
Test ended
```

Funções

Source:

```
program functionsWithFunctions;
var x:Integer;

function f3(f : Fun(Integer):Integer):Integer
begin
        result := f(x)+x
end;

function f0(y:Integer):Integer
        var z:Integer;

        function f1(w:Integer):Integer
        begin
                result := w+y+x+z
        end;

        function f2(w:Integer):Integer
        begin
                result := f3(f1)
        end;

begin
        result := f2(f1(y))
end;

function f4(x:Integer):Integer
begin
        result := x+1
end;

begin
        x := 1;
        write(f4(f0(1)))
end.
```

Output:

```
5
```

Funções como variável

Source:

```
program ola;
var d:Fun(Integer):Integer;
function f(x:Integer):Integer
const d=20;
begin
        result := x + 1
end;
begin
        d := f;
        writeln(d(2))
end.
```

Output:

```
3
```

Declaração de tipos

Source:

```
program o;
type a=Array(2,b); b=Integer;
var x:a;
begin
x := [1, 2]
end.
```

Stack interactiva

Source:

```
program objects;
type
        List1 = Object(Size():Integer, GetElem(Integer):Integer, AddElem(Integer));
        Node1 = Object(SetVal(Integer), GetVal():Integer, Next():Node1, SetNext(Node1));
var
        list:List1;
        command:String;

class Node
        var val:Integer; next:Node1;

        procedure SetVal(x:Integer)
        begin
                val := x
        end;

        function GetVal():Integer
        begin
                result := val
        end;

        function Next():Node1
        begin
                result := next
        end;

        procedure SetNext(new_next:Node1)
        begin
                next := new_next
        end;

        begin
                val := 0
        end;

class List
        var head:Node1; size:Integer;

        function Size():Integer
        begin
                result := size
        end;

        function GetElem(i:Integer):Integer
        var aux:Integer; curr:Node1;
        begin
                if i > size - 1 or i < 0 then
                        result := -1
                else
                begin
                        aux := 0;
                        curr := head;
                        while aux <> i do
                        begin
                                aux := aux + 1;
```

```
                        curr := curr.Next()
                    end;
                    result := curr.GetVal()
                end
        end;

        procedure AddElem(e:Integer)
        var new_node:Node1;
        begin
                new_node := new Node;
                new_node.SetVal(e);
                if size <> 0 then
                        new_node.SetNext(head);
                head := new_node;
                size := size + 1
        end;

        begin
         size := 0
        end;

        procedure readCommand()
        begin
                write("Command: ");
                readln(command)
        end;

        procedure addElem()
        var e:Integer;
        begin
                write("Enter element: ");
                readln(e);
                list.AddElem(e);
                writeln("Element added")
        end;

        procedure getElem()
        var i, e:Integer;
        begin
                writeln("Enter index: ");
                readln(i);
                e := list.GetElem(i);
                writeln("Element at position ", i, ": ", e)
        end;

        procedure size()
        begin
                writeln("Size of list: ", list.Size())
        end;

begin
        list := new List;
        writeln(list.Size());
        readCommand();
        while(command <> "quit") do
        begin
                if command = "add" then
                        addElem()
                else if command = "get" then
                        getElem()
                else if command = "size" then
                        size();
                readCommand()
        end
end.
```

Vector de vector

Source:

```
program ArrayOfArray;
const b = [2,5,8,9];
var a : Array(4, Array(4, Integer));
begin
        a[2][0] := 50;
        a[1] := b;
        writeln(a[0][0]," ",a[0][1]," ",a[0][2]," ",a[0][3]);
        writeln(a[1][0]," ",a[1][1]," ",a[1][2]," ",a[1][3]);
```

```
        writeln(a[2][0]," ",a[2][1]," ",a[2][2]," ",a[2][3]);
        writeln(a[3][0]," ",a[3][1]," ",a[3][2]," ",a[3][3])
end.
```

Output:

```
0 0 0 0
2 5 8 9
50 0 0 0
0 0 0 0
```

Vectores

Source:

```
program Arrays;

const
        cai = 1;
        cbi = 2;

        cas = "String A";
        cbs = "String B";

        cab = false;
        cbb = true;

var
        vai: Integer;
        vbi: Integer;

        vas: String;
        vbs: String;

        vab: Bool;
        vbb: Bool;

        arr: Array(5, Integer);

begin
        writeln("vai ", vai);
        writeln("arr[2] ","= ", arr[2]);
        writeln("arr[vai] ","= ", arr[vai]);

        vai := 3;

        arr[2] := 60;
        arr[1] := 90;

        writeln("arr[2] ","= ", arr[2]);
        writeln("arr[var - 2] ","= ", arr[(vai - 2)]);

        writeln()
end.
```

Output:

```
vai 0
arr[2] = 0
arr[vai] = 0
arr[2] = 60
arr[var - 2] = 90
```

Objectos

Source:

```
program ObjectTest;
var c : Object(inc(), get() : Integer, dup());

class Counter
var val : Integer;

        procedure inc()
        begin
                val := val + 1
        end;

        function get() : Integer
        begin
                result := val
        end;

        procedure dup()
        begin
                self.inc();
                self.inc()
        end;

begin
        val := 0
end;

begin
        c := new Counter;
        c.inc();
        c.inc();
        writeln(c.get());
        c.dup();
        writeln(c.get())
end.
```

Output:

```
2
4
```

Source:

```
program o;
        type a = Object(getX():Integer, setX(Integer));
        var a1, a2 :a;

        class aC
                var x:Integer;

                function getX():Integer
                begin
                        result := x
                end;
                procedure setX(new_x:Integer)
                begin
                        x := new_x
                end;
                begin
                        x := 0
                end;
        begin
                a1 := new aC;
                a1.setX(2);
                writeln(a1.getX());
                a2 := a1;
                a2.setX(3);
                writeln(a1.getX(), " ", a2.getX())
        end.
```

Output:

```
2
3 3
```

Classes

Source:

```
program OtherClassTest;
type
        h = Class(me() : h1, other() : w);
        h1 = Object(me() : h1, other() : w);
        w = Object(print(), me() : w);
var classCounter : h;
        c : h1;

class OtherClass
var i : Integer;

        procedure print()
        begin
                write("lalala")
        end;

        function me() : w
        begin
                result := self
        end;

begin
        i := 0
end;

class Counter
var i : Integer;

        function me() : h1
        begin
                result := self
        end;

        function other() : w
        begin
                result := new OtherClass
        end;

begin
        i := 0
end;

begin
        classCounter := Counter;
        c := new classCounter;
        (c.other()).print()
end.
```

Output:

```
lalala
```

Erros de tipificação

Source:

```
program o;
const x = 4;
var x:Integer;
begin
writeln(x)
end.
```

Output:

```
Fatal error: exception Blaise_typechk.Type_check_error("Id already declared: x")
```

Source:

```
program o;
var x:Integer;
procedure f()
begin
writeln(2)
end;
begin
x := f()
end.
```

Output:

```
Invalid Program:
```

Source:

```
program o;
type mine = Object(get():Integer);
var o:mine;

class MineC
var c:Integer;
function get():Integer
begin
result := c
end;

procedure set(x:Integer)
begin
c := x
end;

begin
c := 0
end;

begin
o := new MineC
end.
```

Output:

```
Fatal error: exception Invalid_argument("List.fold_left2")
```