CML - Uma Linguagem para *Machine Learning*

Relatório 2 - Trabalho de Conceitos de Linguagem de Programação

Caio Lopes, Leonardo Blanger, Marcelo Silvarolla

16 de maio de 2018

Contents

1 Sintaxe Concreta

Usamos a seguinte notação para nossas regras EBNF:

definição :

concatenação (espaço)

união |

agrupamento (...)

string terminal "..."

string terminal '...'

* zero ou mais

+ um ou mais

/* ... */ comentário

terminação ;

O alfabeto, isto é, o conjunto de símbolos terminais da nossa linguagem, será o conjunto de caracteres ASCII, conforme a tabela:

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

1.1 Palavras Reservadas

```
char else real bool string dataset model if int return void while
skip
|| < <= > >= == != ; { } , = ( ) [ ] !  - + * /
```

1.2 Regras Léxicas

Os espaços são ignorados na análise léxica. Denotaremos os *tokens* por seus nomes em LETRAS MAIÚSCULAS, para os diferenciar dos demais símbolos não-terminais.

TODO: Ajeitar as aspas simples

```
D : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
L : "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z" | "_";
A : L | D;
ES : "\" ("'" | '"' | "\" | "n" | "t");
CHAR : "char";
ELSE : "else";
REAL : "real";
BOOL : "bool";
STRING : "string";
DATASET : "dataset";
MODEL : "model";
IF : "if";
INT : "int";
RETURN : "return";
VOID : "void";
WHILE : "while";
SKIP : "skip";
```

```
IDENTIFIER : L (A)*;

INT_LITERAL : (D)+;

REAL_LITERAL : (D)* "." (D)+ | (D)+ ".";

NONQUOTE_BACKSLASH_NEWLINE :

' ' | '!' | '"' | '#' | '$' | '%' | '' | "'" | '(' | ')' |

'*' | '+' | ',' | '-' | '.' | '/' | '0' | '1' | '2' | '3' |

'4' | '5' | '6' | '7' | '8' | '9' | ':' | ';' | '<' | '=' |

'>' | '?' | '@' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |

'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' |

'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '[' |

'\' | ']' | '^' | '_' | '`' | 'a' | 'b' | 'c' | 'd' | 'e' |

'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' |

'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' |

'z' | '{' | '|' | '}' | '~';

CHAR_LITERAL : "'" (NONQUOTE_BACKSLASH_NEWLINE | '"' | ES)"'";

STRING_LITERAL : '"' (NONQUOTE_BACKSLASH_NEWLINE | "'" | ES)* '"';

AND_OP : "&&";

OR_OP : "||";

LE_OP : "<=";

GE_OP : ">=";

EQ_OP : "==";

NE_OP : "!=";
```

1

[1] `D` de dígito, `L` de letra, `A` de alfanumérico (letra ou dígito), `ES` de *escape sequence*. Note

1.3 Regras Sintáticas

1.3.1 Símbolos Terminais

Os símbolos terminais na análise sintática são os *tokens* gerados na análise léxica acima.

1.3.2 Expressões

Os espaços são ignorados na análise léxica.

```
/* Expressões necessariamente "atômicas":
    Expressões que não causam ambiguidades quando dentro
    de uma expressão maior, mesmo quando a precedência
    das operações não é conhecida
    Estas expressões podem, portanto, ser pensadas
    como identificadores ou literais. */


primary_expression
    : IDENTIFIER
    | literal
    | '{' '}'
    | '{' expression_list '}'
```

que incluímos o *underscore* como letra.

O não-terminal `NONQUOTE_BACKSLASH_NEWLINE` define o conjunto de todos os caracteres ASCII imprimíveis que não são uma aspa simples, aspa dupla, barra inversa ou quebra de linha.

```
    | '(' expression ')'
    | array_access
    | IDENTIFIER '(' ')'
    | IDENTIFIER '(' expression_list ')'
    ;

literal
    : INT_LITERAL
    | REAL_LITERAL
    | BOOL_LITERAL
    | CHAR_LITERAL
    | STRING_LITERAL
    ;

/* Expressões "não-atômicas" */
expression_list
    : expression
    | expression_list ',' expression
    ;

expression
    : logical_or_expression
    | IDENTIFIER '=' expression
    | array_access '=' expression
```

```
    ;

array_access
    : IDENTIFIER '[' expression ']'
    | array_access '[' expression ']'
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;

logical_and_expression
    : relational_expression
    | logical_and_expression AND_OP relational_expression
    ;

relational_expression
    : additive_expression
    | additive_expression '<' additive_expression
    | additive_expression '>' additive_expression
    | additive_expression LE_OP additive_expression
    | additive_expression GE_OP additive_expression
    | additive_expression EQ_OP additive_expression
```

```
    | additive_expression NE_OP additive_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

multiplicative_expression
    : unary_minus_expression
    | multiplicative_expression '*' unary_minus_expression
    | multiplicative_expression '/' unary_minus_expression
    ;

unary_minus_expression
    : neg_expression
    | '-' neg_expression
    ;

neg_expression
    : primary_expression
    | '!' neg_expression
    ;
```

1.3.3 Comandos

```
command
    : compound_command
    | expression_command
    | selection_command
    | iteration_command
    | jump_command
    | SKIP ';'
    ;

compound_command
    : '{' declaration_or_command_list '}'
    ;

declaration_or_command_list
    : declaration_or_command
    | declaration_or_command_list declaration_or_command
    ;

declaration_or_command
    : declaration
    | command
    ;
```

```
expression_command
    : ';'
    | expression ';'
    ;

selection_command
    : IF '(' expression ')' compound_command ELSE compound_command
    | IF '(' expression ')' compound_command
    ;

iteration_command
    : WHILE '(' expression ')' command
    ;

jump_command
    : RETURN ';'
    | RETURN expression ';'
    ;
```

1.3.4 Declarações

```
declaration
    : type_specifier IDENTIFIER ';'
    | type_specifier IDENTIFIER '=' expression ';'
```

```
    | type_specifier IDENTIFIER '(' ')' ';'
    | type_specifier IDENTIFIER '(' parameter_declaration_list ')' ';'
    ;

parameter_declaration_list
    : parameter_declaration
    | parameter_declaration_list ',' parameter_declaration
    ;

parameter_declaration
    : type_specifier IDENTIFIER
    ;

type_specifier
    : VOID
    | CHAR
    | INT
    | REAL
    | BOOL
    | STRING
    | DATASET
    | MODEL
    | type_specifier '[' ']'
    ;
```

1.3.5 Definições

```
function_definition
    : type_specifier IDENTIFIER '(' ')' compound_command
    | type_specifier IDENTIFIER '('
    parameter_declaration_list ')' compound_command
    ;
```

1.3.6 Programa

```
program
    : declaration_or_function_definition
    | program declaration_or_function_definition
    ;

declaration_or_function_definition
    : function_definition
    | declaration
    ;
```

1.4 Exemplo

TODO: EXEMPLO

2 Semântica

2.1 Domínios Sintáticos

- `Identifier` := conjunto de todas as sequências finitas de terminais deriváveis a partir do não-terminal `IDENTIFIER` acima.
- `Literal` := conjunto de todas as sequências finitas de terminais deriváveis a partir do não-terminal `literal` acima.
- `ArrayAccess` $defeq$ conjunto de todas as sequências finitas de terminais deriváveis a partir do não-terminal `arr_access` acima.
- `Expression` := conjunto de todas sequências finitas de terminais deriváveis a partir do não-terminal `exp` abaixo.
- `Command` := conjunto de todas as sequências finitas de terminais deriváveis a partir do não-terminal `cmd` abaixo.
- `Declaration` := conjunto de todas as sequências finitas de terminais deriváveis a partir do não-terminal `dec` abaixo.
- `FunctionDefinition` := conjunto de todas as sequências finitas de terminais deriváveis a partir do não-terminal `function_def` abaixo.
- `Program` := conjunto de todas as sequências finitas de terminais deriváveis a partir do não-terminal `prog` abaixo.

2.2 Sintaxe Abstrata

A sintaxe em EBNF acima contém detalhes irrelevantes para a semântica, especificando formalmente a associatividade e precedência dos operadores. Aqui, apresentamos uma sintaxe simplificada, a sintaxe abstrata, que, apesar de ambígua, é suficiente para a definição da semântica. Estamos essencialmente seguindo o capítulo 9 do livro de Kenneth Slonneger e Barry L. Kurtz "Formal Syntax and Semantics of Programming Languages"[2]. Observe que `IDENTIFIER` e `literal` estão definidos na sintaxe concreta acima.

2.2.1 Expressões

```
id : IDENTIFIER;
lit : literal;
exp_list : exp | exp_list "," exp;
arr_access : id | arr_access "[" exp "]";
exp : id | lit | "{" "}" | "{ exp_list "}" |
     exp "||" exp | arr_access "=" exp | exp "&&" exp |
     exp "==" exp | exp "!=" exp | exp "<" exp | exp "<=" exp |
     exp ">" exp | exp ">=" exp | "-" exp | exp "+" exp |
     exp "-" exp | exp "*" exp | exp "/" exp |
     "(" exp ")" | exp "[" exp "]" | id "(" ")" |
     id "(" exp_list ")" | "!" exp;
```

[2]Disponível em `http://www.divms.uiowa.edu/~slonnegr/plf/Book/Chapter9.pdf`

2.2.2 Comandos

```
cmd : comp_cmd | exp_cmd |
       sel_cmd | iter_cmd |
       jump_cmd | SKIP ";";
comp_cmd : "{" (dec | cmd)+ "}";
exp_cmd : exp ";";
sel_cmd : IF "(" exp ")" comp_cmd ELSE comp_cmd |
                 IF "(" exp ")" comp_cmd;
iter_cmd : WHILE "(" exp ")" cmd;
jump_cmd : RETURN ";" | RETURN exp ";";
```

2.2.3 Declarações

Note que `type_specifier` está definido na sintaxe concreta acima.

```
type_spec : type_specifier;
dec : type_spec id ";" |
       type_spec id "=" exp ";" |
       type_spec id "(" ")" ";" |
       type_spec id "(" param_dec_list ")" ";";
param_dec : type_spec id;
param_dec_list : param_dec | param_dec_list "," param_dec;
```

2.2.4 Definições

```
fun_def : type_spec id "(" ")" comp_cmd |
```

```
type_spec id "(" param_dec_list ")" comp_cmd;
```

2.2.5 Programa

```
prog : dec_or_fun_def | prog dec_or_fun_def;
dec_or_fun_def : fun_def | dec;
```

2.3 Domínios Semânticos

Denotamos por $X \to Y$ o conjunto de funções parciais de X em Y. Escrevemos $f : X \to Y$ para dizer que f é função parcial de X em Y. Ademais, se $n \in \mathbb{N}$, $X^n := X \times \ldots (n \text{ vezes }) \ldots \times X := \{f \mid f \text{ é função total do conjunto } \{0, 1, \ldots, n-1\} \text{ em } X\}$ é o conjunto de tuplas (isto é, vetores) de n elementos de X. Por exemplo, $\mathbb{R} \to \mathbb{R}$ é o conjunto de todas as funções da reta na reta, enquanto $\mathbb{R}^n$ é o espaço euclidiano n-dimensional.

- $Int := \{\ldots, -2, -1, 0, 1, 2, \ldots\} = \mathbb{Z}$
- $Real := \mathbb{R}$
- $Bool := \{true, false\}$
- $Char := \{0, \ldots, 127\}$, onde os números de 0 a 127 são interpretados conforme o padrão ASCII, e.g., 43 representa '+', 49 representa '1', etc. Para mais detalhes, ver tabela no início do relatório.

  Definimos, para cada conjunto X, o conjunto $X[]$ dos vetores de elementos de X pondo $X[] := \bigcup_{n=0}^{+\infty} X^n$

$rotulo(X)$ denota o conjunto $\{rotulo(x) : x \in X\}$ de elementos de X rotulados pela palavra *rotulo*. Por exemplo,

$$int(Int) = \{\ldots, int(-2), int(-1), int(0), int(1), int(2), \ldots\}.$$

Isto serve para que possamos tomar a união, por exemplo, $int(Int) \cup char(Char)$, que é

$$\{\ldots, int(-2), int(-1), int(0), int(1), int(2), \ldots, char(0), char(1), \ldots, char(127)\},$$

e saber de que conjunto cada elemento provém. Se simplesmente tomássemos a união sem rótulos, $Int \cup Char = Int$, não saberíamos, por exemplo, se $5 \in Int \cup Char$ provém do conjunto Int ou do conjunto $Char$, isto é, não saberíamos o tipo do valor 5.

- $String = Char[]$

  Observe que $String$ é simplesmente o conjunto dos vetores de $Char$'s. Usaremos os rótulos *string* e *array* para distingui-los.

- $Dataset = \{0, \ldots, n\} \times \{1, \ldots, m\} \to int(Int) \cup real(Real) \cup string(String)$

- $Model = Dataset \to Dataset$

- $Array = int(Int)[] \cup real(Real)[] \cup bool(Bool)[] \cup char(Char)[]$
  $\cup\ string(String)[] \cup dataset(Dataset)[] \cup model(Model))$
  $\cup\ int(Int)[][] \cup real(Real)[][] \cup bool(Bool)[][] \cup char(Char)[][]$
  $\cup\ string(String)[][] \cup dataset(Dataset)[][] \cup model(Model)[][]$
  $\cup\ int(Int)[][][] \cup \ldots$ [3]

- $Input := \bigcup_{n=0}^{+\infty} Dataset^n$
- $Output := \bigcup_{n=0}^{+\infty} Dataset^n$
- $Function = \bigcup_{n=0}^{+\infty} (Location^n \rightarrow Store \times Input \times Output)$
- $StorableValue := int(Int) \cup real(Real) \cup bool(Bool) \cup char(Char) \cup string(String) \cup dataset(Dataset) \cup model(Model) \cup array(Array)$
- $ExpressibleValue := StorableValue$
- $DenotableValue := var(Location) \cup fun(Function)$
- $Location := \mathbb{N} := \{0, 1, 2, \ldots\}$
- $Environment := \texttt{Identifier} \rightarrow DenotableValue \cup \{unbound\}$[4]
- $Store := Location \rightarrow StorableValue \cup \{unused\} \cup \{undefined\}$[5]
- $\Sigma := State := Environment \times Store \times Input \times Output$

[3]Note que *int*[] é meramente um rótulo, um nome, enquanto que Int[] é o conjunto de vetores de Int's.

[4]O valor *unbound* indica que o identificador não foi associado a uma posição de memória ou função, isto é, não foi declarado.

[5]O valor *unused* indica que a localização não está sendo utilizada por nenhuma variável. Já o valor $undefined$ indica que a localização está sendo utilizada por uma variável que não foi inicializada.

Obs.: acrescentamos o valor especial *error* em todos os domínios, para indicar erro no programa e supomos que erros são propagados pelas funções semânticas.

2.4 Funções Semânticas

Todas as funções abaixo de fato podem precisar do estado completo Σ do programa, incluindo *Environment*, *Store*, *Input* e *Output*. Porém:

- Uma `Expression` não precisa devolver o *Environment*, já que não o modifica. Basta, então, devolver *Store*, *Input*, *Output* e, obviamente, um $ExpressibleValue$.
- Idem para um `Command`
- Uma `Declaration` pode alterar o estado completo: *Environment*, *Store*, *Input* e *Output*.
- Uma `FunctionDefinition` pode alterar apenas o *Environment*, ligando um $Identifier$ de função com a $Function$ correspondente.
- Um `Program` pode alterar o estado completo: *Environment*, *Store*, *Input* e *Output*.

$$E : \texttt{Expression} \to (\Sigma \to Store \times Input \times Output \times ExpressibleValue)$$

$$C : \texttt{Command} \to (\Sigma \to Store \times Input \times Output)$$

$$Dec : \texttt{Declaration} \to (\Sigma \to \Sigma)$$

$$Def : \texttt{FunctionDefinition} \to (\Sigma \to Environment)$$

$$P : \texttt{Program} \to (\Sigma \to \Sigma)$$

Além disso, teremos as seguintes funções auxiliares (algumas de 0 argumentos, como $emptyEnv$):

TODO: especificar funções predefinidas de CML

$value : Literal \to int(Int) \cup real(Real) \cup bool(Bool) \cup char(Char) \cup string(String)$[6]

$emptyEnv : Environment$

$extendEnv : Environment \times Identifier \times DenotableValue \to Environment$

$applyEnv : Environment \times Identifier \to DenotableValue \cup \{unbound\}$

$emptySto : Store$

$updateSto : Store \times Location \times \big(StorableValue \cup \{undefined, unused\}\big) \to Store$

$applySto : Store \times Location \to StorableValue \cup \{undefined, unused\}$

$allocate : Store \to Store \times Location$

$deallocate : Store \times Location \to Store$

2.5 Equações Semânticas

Quando mais de uma equação é fornecida para uma mesma função semântica, supõe-se que a primeira que servir para o argumento é executada. As demais são ignoradas. Há aqui, portanto, uma analogia com linguagens funcionais como Haskell e SML, que possuem *pattern matching*.

[6]Por exemplo:

$$value(\texttt{101}) = int(101)$$

$$value(\texttt{true}) = bool(true)$$

$$value(\texttt{5.2}) = real(5,2)$$

$$value(\texttt{"Maria"}) = string(\text{“}Maria\text{”})$$

2.5.1 Funções auxiliares

TODO: definir value e funções predefinidas de CML

$value$

$emptyEnv\ I = unbound$

$extendEnv(env, I, dval)\ I_1 = \big(\textbf{if}\ I_1 = I\ \textbf{then}\ dval\ \textbf{else}\ env(I_1)\big)$

$applyEnv(env, I) = env(I)$

$emptyStore\ loc = unused$

$updateSto(sto, loc, val)\ loc_1 = \big(\textbf{if}\ loc_1 = loc\ \textbf{then}\ val\ \textbf{else}\ sto(loc_1)\big)$

$applySto(sto, loc) = sto(loc)$

$allocate\ sto = (updateSto(sto, loc, undefined), loc)$

$\quad\textbf{where}\ loc = minimum\{k \mid sto(k) = unused\}$

$deallocate(sto, loc) = updateSto(sto, loc, unused)$

2.5.2 Programa

P [[`dec`]] $\sigma = Dec$ [[`dec`]] σ

P [[**int main()** `comp_cmd`]] $(env, sto, in, out) = (env_2, sto_2, in_2, out_2)$

where $env_1 = Def$ [[**int main()** `comp_cmd`]] (env, sto, in, out)

and $(env_2, sto_2, in_2, out_2) = C$ [[$comp\_cmd$]] (env_1, sto, in, out)

P [[`type_spec id(param_dec_list) comp_cmd`]] $(env, sto, in, out) =$

(env_1, sto, in, out)

where $env_1 = Def$ [[`type_spec id(param_dec_list) comp_cmd`]] (env, sto, in, out)

P [[`type_spec id() comp_cmd`]] $(env, sto, in, out) =$

(env_1, sto, in, out)

where $env_1 = Def$ [[`type_spec id() comp_cmd`]] (env, sto, in, out)

P [[`prog dec_or_fun_def`]] $\sigma = P$ [[`dec_or_fun_def`]](P [[`prog`]] σ)

2.5.3 Expressões

Falta a atribuição, as expressões que envolvem acessar posições de vetor, as chamadas de funções, e as expressões que envolvem arrays literais.

$E\ [[\texttt{id}]]\ (env, sto, in, out) =$ **if** $val = undefined$ **then** $error$ **else** (sto, in, out, val)

where $val = applySto(sto, loc)$

where $loc = applyEnv(env, \texttt{id})$

$E[[\texttt{lit}]](env, sto, in, out) = (sto, in, out, value(\texttt{lit}))$

$E[[\{\}]]\ (env, sto, in, out) = (sto, in, out, array(\emptyset))$

$E[[\{\texttt{exp}\}]]\ (env, sto, in, out) = (sto_1, in_1, out_1, array(val))$

where $(sto_1, in_1, out_1, val) = E[[\texttt{exp}]](env, sto, in, out)$

$E[[\texttt{exp\_list}, \texttt{exp}]]\ (env, sto, in, out) = (sto_2, in_2, out_2, array(extendArray(arr, val)))$

where $(sto_1, in_1, out_1, arr) = E[[\{\texttt{exp\_list}\}]]\ (env, sto, in, out)$

where $(sto_2, in_2, out_2, val) = E[[\texttt{exp}]]\ (env, sto_1, in_1, out_1)$

$$
\begin{aligned}
E[[\texttt{e}_1 + \texttt{e}_2]]\sigma &= int(m+n) \textbf{ where } int(m) = E[[\texttt{e}_1]]\sigma \textbf{ and } int(n) = E[[\texttt{e}_2]]\sigma \\
&= real(m+n) \textbf{ where } int(m) = E[[\texttt{e}_1]]\sigma \textbf{ and } real(n) = E[[\texttt{e}_2]]\sigma \\
&= real(m+n) \textbf{ where } real(m) = E[[\texttt{e}_1]]\sigma \textbf{ and } int(n) = E[[\texttt{e}_2]]\sigma \\
&= real(m+n) \textbf{ where } real(m) = E[[\texttt{e}_1]]\sigma \textbf{ and } real(n) = E[[\texttt{e}_2]]\sigma
\end{aligned}
$$

$$
\begin{aligned}
E[[\mathsf{e}_1 - \mathsf{e}_2]]\sigma &= int(m-n) \textbf{ where } int(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } int(n) = E[[\mathsf{e}_2]]\sigma \\
&= real(m-n) \textbf{ where } int(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } real(n) = E[[\mathsf{e}_2]]\sigma \\
&= real(m-n) \textbf{ where } real(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } int(n) = E[[\mathsf{e}_2]]\sigma \\
&= real(m-n) \textbf{ where } real(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } real(n) = E[[\mathsf{e}_2]]\sigma
\end{aligned}
$$

$$
\begin{aligned}
E[[\mathsf{e}_1 * \mathsf{e}_2]]\sigma &= int(m \times n) \textbf{ where } int(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } int(n) = E[[\mathsf{e}_2]]\sigma \\
&= real(m \times n) \textbf{ where } int(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } real(n) = E[[\mathsf{e}_2]]\sigma \\
&= real(m \times n) \textbf{ where } real(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } int(n) = E[[\mathsf{e}_2]]\sigma \\
&= real(m \times n) \textbf{ where } real(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } real(n) = E[[\mathsf{e}_2]]\sigma
\end{aligned}
$$

$$
\begin{aligned}
E[[\mathsf{e}_1/\mathsf{e}_2]]\sigma = \ & \textbf{if } n = 0 \textbf{ then } \mathit{error} \textbf{ else } \mathit{int}(m \div n) \\
& \textbf{where } \mathit{int}(m) = E[[\mathsf{e}_1]] \textbf{ and } \mathit{int}(n) = E[[\mathsf{e}_2]]\sigma \\
= \ & \textbf{if } n = 0 \textbf{ then } \mathit{error} \textbf{ else } \mathit{real}(m \div n) \\
& \textbf{where } \mathit{real}(m) = E[[\mathsf{e}_1]] \textbf{ and } \mathit{int}(n) = E[[\mathsf{e}_2]]\sigma \\
= \ & \textbf{if } n = 0 \textbf{ then } \mathit{error} \textbf{ else } \mathit{real}(m \div n) \\
& \textbf{where } \mathit{int}(m) = E[[\mathsf{e}_1]] \textbf{ and } \mathit{real}(n) = E[[\mathsf{e}_2]]\sigma \\
= \ & \textbf{if } n = 0 \textbf{ then } \mathit{error} \textbf{ else } \mathit{real}(m \div n) \\
& \textbf{where } \mathit{real}(m) = E[[\mathsf{e}_1]] \textbf{ and } \mathit{real}(n) = E[[\mathsf{e}_2]]\sigma
\end{aligned}
$$

$$
\begin{aligned}
E[[-\mathsf{e}]]\sigma &= \mathit{int}(-1 \times m) \textbf{ where } \mathit{int}(m) = E[[\mathsf{e}]]\sigma \\
&= \mathit{real}(-1 \times m) \textbf{ where } \mathit{real}(m) = E[[\mathsf{e}]]\sigma
\end{aligned}
$$

$$E[[\mathsf{e}_1||\mathsf{e}_2]]\sigma = \mathit{bool}(m \vee n) \textbf{ where } \mathit{bool}(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } \mathit{bool}(n) = E[[\mathsf{e}_2]]\sigma$$

$$E[[\mathsf{e}_1\&\&\mathsf{e}_2]]\sigma = \mathit{bool}(m \wedge n) \textbf{ where } \mathit{bool}(m) = E[[\mathsf{e}_1]]\sigma \textbf{ and } \mathit{bool}(n) = E[[\mathsf{e}_2]]\sigma$$

$$E[[\mathsf{e}_1 == \mathsf{e}_2]]\sigma = bool(true) \textbf{ where } E[[\mathsf{e}_1]]\sigma = E[[\mathsf{e}_2]]\sigma$$
$$= bool(false) \textbf{ where } E[[\mathsf{e}_1]]\sigma \neq E[[\mathsf{e}_2]]\sigma$$

$$E[[\mathsf{e}_1 != \mathsf{e}_2]]\sigma = bool(false) \textbf{ where } E[[\mathsf{e}_1]]\sigma = E[[\mathsf{e}_2]]\sigma$$
$$= bool(true) \textbf{ where } E[[\mathsf{e}_1]]\sigma \neq E[[\mathsf{e}_2]]\sigma$$

$$E[[\mathsf{e}_1 < \mathsf{e}_2]]\sigma = bool(true) \textbf{ where } E[[\mathsf{e}_1]]\sigma < E[[\mathsf{e}_2]]\sigma$$
$$= bool(false) \textbf{ where } E[[\mathsf{e}_1]]\sigma \geq E[[\mathsf{e}_2]]\sigma$$

$$E[[\mathsf{e}_1 <= \mathsf{e}_2]]\sigma = bool(true) \textbf{ where } E[[\mathsf{e}_1]]\sigma \leq E[[\mathsf{e}_2]]\sigma$$
$$= bool(false) \textbf{ where } E[[\mathsf{e}_1]]\sigma > E[[\mathsf{e}_2]]\sigma$$

$$E[[\mathsf{e}_1 > \mathsf{e}_2]]\sigma = bool(true) \textbf{ where } E[[\mathsf{e}_1]]\sigma > E[[\mathsf{e}_2]]\sigma$$
$$= bool(false) \textbf{ where } E[[\mathsf{e}_1]]\sigma \leq E[[\mathsf{e}_2]]\sigma$$

$$E[[\mathsf{e}_1 >= \mathsf{e}_2]]\sigma = bool(true) \textbf{ where } E[[\mathsf{e}_1]]\sigma \geq E[[\mathsf{e}_2]]\sigma$$
$$= bool(false) \textbf{ where } E[[\mathsf{e}_1]]\sigma < E[[\mathsf{e}_2]]\sigma$$

$$E[[!\mathsf{e}]]\sigma = bool(true) \textbf{ where } bool(false) = E[[\mathsf{e}]]\sigma$$
$$= bool(false) \textbf{ where } bool(true) = E[[\mathsf{e}]]\sigma$$

$$E[[(\mathsf{e})]]\sigma = E[[\mathsf{e}]]\sigma$$

$$E[[\mathsf{exp}_1[\mathsf{exp}_2]]]\sigma = getValueAtPosition(E[[\mathsf{exp}_1]]\sigma, E[[\mathsf{exp}_2]]\sigma)$$

Corrigir isto

$E[[\texttt{arr\_access} = \texttt{exp}]](env, sto, in, out) = (env, sto_2, in_1, out_1)$

where $(sto_1, in_1, out_1, expVal) = E[[\texttt{exp}]]\ (env, sto, in, out)$

where $(sto_2, in_2, out_2) = updateArray(env, sto_1, in_1, out_1), \texttt{arr\_access}, expVal)$

where $updateArray : \Sigma \times ArrayAccess \times ExpressibleValue \rightarrow Store \times Input \times Output$

where $updateArray((env, sto, in, out), \texttt{id}, val) = (updateSto(sto, loc, val), in, out)$

where $loc = applyEnv(env, \texttt{id})$

and $updateArray((env, sto, in, out), \texttt{arr\_access}[\texttt{exp}], val) = (sto_f, in_f, out_f)$

where $(sto_1, in_1, out_1, array(arr)) = E[[\texttt{arr\_access}]]\ (env, sto, in, out)$

where $(sto_2, in_f, out_f, int(pos)) = E\ [[\texttt{exp}]]\ (env, sto_1, in_1, out_1)$

where $sto_f = updateSto(sto_2, arr_{pos}, val)$

2.5.4 Comandos

$$C[[\texttt{skip;}]]\sigma = \sigma$$

$$C[[\{\texttt{cmd}\}]]\sigma = C[[\texttt{cmd}]]\sigma$$

$$C[[C_1C_2]]\sigma = C[[\texttt{C}_2]](C[[\texttt{C}_1]]\sigma)$$

$$C[[\texttt{exp;}]]\sigma = (sto', env)$$
$$\textbf{where } (sto', val) = E[[\texttt{exp}]]\sigma$$
$$\textbf{and } (sto, env) = \sigma$$

$$C[[\texttt{if (exp) cmd}]]\sigma = \textbf{ if } bool(m) \textbf{ then } C[[\texttt{cmd}]](sto', env) \textbf{ else } (sto', env)$$
$$\textbf{where } (bool(m), sto') = E[[\texttt{exp}]]\sigma$$
$$\textbf{and } (sto, env) = \sigma$$

$$C[[\texttt{if (exp) cmd}_1 \texttt{ else cmd}_2]]\sigma = \textbf{ if } bool(m) \textbf{ then } C[[\texttt{cmd}_1]](sto', env) \textbf{ else } C[[\texttt{cmd}_2]](sto', env)$$
$$\textbf{where } (bool(m), sto') = E[[\texttt{exp}]]\sigma$$
$$\textbf{and } (sto, env) = \sigma$$

$$C[[\texttt{while (exp) cmd}]]\sigma = loop(\sigma)$$
$$\textbf{where } loop(sto, env) = \textbf{ if } bool(m) \textbf{ then}$$
$$loop(C[[\texttt{cmd}]](sto', env), env)$$
$$\textbf{else } (sto', env)$$
$$\textbf{and } (bool(m), sto') = E[[\texttt{exp}]]\sigma$$
$$\textbf{and } (sto, env) = \sigma$$

3 Desafios enfrentados

Percebemos que digitar as equações semânticas em LaTeXé difícil por elas ficarem grandes e saírem da tela. Solucionamos o problema escolhendo nomes curtos para os símbolos da sintaxe abstrata e para as funções de avaliação. Seria conveniente se houvesse um ambiente no qual descrever a semântica sem necessidade de cuidar de detalhes de alinhamento.