

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
Linguagem de Programação: Conceitos e Paradigmas

Nome: Esther Maria da Silveira Wanderley

Nome: Isaque Barbosa Martins

Nome: Thuanny Carvalho Rolim de Albuquerque

Nome: Vinícius Oliveira da Silva

Nome: Gabriel Henrique Campos Medeiros

Relatório sobre a Linguagem de Programação produzida

1. Introdução

Neste projeto, implementamos um interpretador em Haskell para a linguagem **LEGOLAN**. O objetivo deste trabalho é explorar e aplicar conceitos de linguagens de programação e paradigmas de implementação de interpretadores. A linguagem criada pelo nosso grupo possui características que focam em estruturas de dados, procurando facilitar suas representações para o programador. O público-alvo da **LEGOLAN** são pesquisadores da área de estruturas de dados, visando facilitar a implementação de algoritmos e operações com determinadas estruturas de dados. O principal diferencial da **LEGOLAN** é que ela busca trazer as principais estruturas de dados (matriz, lista, árvore) como tipos primitivos da linguagem, ou seja, o usuário não precisa se preocupar com a implementação dessas estruturas.

2. Design da Implementação

2.1 Transformação do Código-Fonte em Unidades Léxicas

A transformação do código-fonte em unidades léxicas (tokenização) é a primeira etapa do processo de interpretação. Utilizamos a biblioteca **Alex** para a criação do analisador léxico, que identifica e categoriza os diferentes componentes do código-fonte, como palavras-chave, identificadores, operadores, etc.

- Estrutura Geral:

```
1 tokens :-                                --- Inicia a definicao das regras de
2     tokenizacao.
3     $white+                                -- Ignora espacos em branco (um ou mais).
4     "#".* ; -- Ignora comentarios que comecam com # e vao ate o final da
5     linha.
6     ...
7     $alpha [$alpha $digit \_ \'']*          { \p s -> Id s (getLC p) }
```

- Padrões e Ações:

```
1     ":"                                     { \p s -> Colon (getLC p) }
2
3     --- ":" : Padrao que corresponde ao caractere :
4     ---{ \p s -> Colon (getLC p) } : Acao que cria um token Colon com a posicao
5     (linha e coluna) do caractere. getLC p e uma funcao que obtem a
6     posicao do token.
```

- Exemplos de tokens:

- i. Símbolos e operadores:

```
1  -- cria um token "SemiColon"
2  ";"      { \p s -> SemiColon (getLC p) }
3
4  -- cria um token "Comma"
5  ","      { \p s -> Comma (getLC p) }
6
7  -- cria um token "LeftParenthesis"
8  "("      { \p s -> LeftParenthesis (getLC p) }
9
10 -- cria um token "RightParenthesis"
11 ")"      { \p s -> RightParenthesis (getLC p) }
12
13 -- cria um token "LeftCurlyBrackets"
14 "{"      { \p s -> LeftCurlyBrackets (getLC p) }
15
16 -- cria um token "RightCurlyBrackets"
17 "}"      { \p s -> RightCurlyBrackets (getLC p) }
18
19 -- cria um token " To"
20 "->"     { \p s -> To (getLC p) }
21
22 -- cria um token " Assign"
23 "="      { \p s -> Assign (getLC p) }
```

- ii. Operadores Matemáticos:

```
1
2  -- cria um token "Plus"
3  "+"      { \p s -> Plus (getLC p) }
4
5  -- cria um token "Minus"
6  "-"      { \p s -> Minus (getLC p) }
7
8  -- cria um token "Times"
9  "*"      { \p s -> Times (getLC p) }
10
11 -- cria um token "Divider"
12 "/"      { \p s -> Divider (getLC p) }
13
14 -- cria um token "IntegerDivider"
15 "//"      { \p s -> IntegerDivider (getLC p) }
16
17 -- cria um token "Exponent"
18 "**"      { \p s -> Exponent (getLC p) }
```

iii. Operadores Lógicos:

```
1
2      -- cria um token "And"
3      "&&"      { \p s -> And (getLC p) }
4
5      -- cria um token "Or"
6      "||"      { \p s -> Or (getLC p) }
7
8      -- cria um token "Xor"
9      "^"       { \p s -> Xor (getLC p) }
10
11     -- cria um token "Not"
12     "!"       { \p s -> Not (getLC p) }
13
14     -- cria um token "Less"
15     "<"       { \p s -> Less (getLC p) }
16
17     -- cria um token "LessEqual"
18     "<="     { \p s -> LessEqual (getLC p) }
19
20     -- cria um token "Greater"
21     ">"       { \p s -> Greater (getLC p) }
22
23     -- cria um token "GreaterEqual"
24     ">="     { \p s -> GreaterEqual (getLC p) }
25
26     -- cria um token "Equal"
27     "=="      { \p s -> Equal (getLC p) }
28
29     -- cria um token "Different"
30     "!="      { \p s -> Different (getLC p) }
```

iv. Tipos:

```
1
2      --Cada um cria um token Type com a string correspondente e a
3      posicao.
4
5      "int"      { \p s -> Type s (getLC p) }
6      "float"    { \p s -> Type s (getLC p) }
7      "char"     { \p s -> Type s (getLC p) }
8      "string"   { \p s -> Type s (getLC p) }
9      "bool"     { \p s -> Type s (getLC p) }
10     "list"     { \p s -> Type s (getLC p) }
11     "stack"    { \p s -> Type s (getLC p) }
12     "queue"    { \p s -> Type s (getLC p) }
13     "matrix"   { \p s -> Type s (getLC p) }
14     "graph"    { \p s -> Type s (getLC p) }
15     "tree"     { \p s -> Type s (getLC p) }
16     "n-array"  { \p s -> Type s (getLC p) }
```

v. Estruturas de Controle:

```
1
2  --Cada um cria um token correspondente (If, EndIf, Elif, Else) com a
3     posicao
4
5     "if"                                { \p s -> If (getLC p) }
6     "end_if"                            { \p s -> EndIf (getLC p) }
7     "elif"                              { \p s -> Elif (getLC p) }
8     "else"                              { \p s -> Else (getLC p) }
9     "for"                                { \p s -> For (getLC p) }
10    "end_for"                            { \p s -> EndFor (getLC p) }
11    "while"                              { \p s -> While (getLC p) }
12    "end_while"                          { \p s -> EndWhile (getLC p) }
13    "return"                             { \p s -> Return (getLC p) }
14    "declaration"                       { \p s -> Declaration (getLC p) }
15    "end_declaration"                   { \p s -> EndDeclaration (getLC p) }
16    "func"                              { \p s -> Func (getLC p) }
17    "end_func"                          { \p s -> EndFunc (getLC p) }
18    "main"                              { \p s -> Main (getLC p) }
19    "end_main"                          { \p s -> EndMain (getLC p) }
```

vi. Palavras-Chaves:

```
1
2  -- Cria tokens das palavras-chaves
3
4     "typedef"                           { \p s -> Typedef (getLC p) }
5     "struct"                           { \p s -> Struct (getLC p) }
6     "scan"                             { \p s -> Scan s (getLC p) }
7     "print"                             { \p s -> Print s (getLC p) }
```

vii. Constantes e Literais:

```
1
2  -- Cria um token BoolValue com o valor True or False
3
4     "true"                              { \p s -> BoolValue True (getLC p) }
5     "false"                             { \p s -> BoolValue False (getLC p) }
6
7  -- $digit+ \. $digit+: Corresponde a numeros de ponto flutuante (como
8     123.456) e cria um token FloatValue.
9
10 -- $digit+: Corresponde a inteiros (como 123) e cria um token IntValue.
11
12 -- \" [$white $alpha $digit ! \_ \']* \": Corresponde a strings (como
13     "hello") e cria um token StringValue.
14
15 --\' $printable \': Corresponde a caracteres (como 'a') e cria um token
16     CharValue.
17
18 $digit+ \. $digit+                      { \p s -> FloatValue (read s) (getLC p) }
19 $digit+                                { \p s -> IntValue (read s) (getLC p) }
20 \" [$white $alpha $digit ! \_ \']* \"    { \p s -> StringValue
21     (cleanString s) (getLC p) }
22 \' $printable \'                        { \p s -> CharValue (cleanChar s) (getLC p) }
```

• Identificadores

```

1  -- $alpha [$alpha $digit \_ \'']* : Corresponde a identificadores (como
2  variableName). Começa com uma letra ($alpha) e pode ser seguido por
3  letras, dígitos, sublinhados ou aspas simples.
4
5  -- { \p s -> Id s (getLC p) } : Cria um token Id com a string correspondente
6  e a posicao.
7
8  $alpha [$alpha $digit \_ \'']*          { \p s -> Id s (getLC p) }

```

Este conjunto de regras define como transformar o texto de entrada em uma sequência de tokens, que serão usados pelo analisador sintático para construir a árvore sintática do programa. Cada regra consiste em um padrão de expressão regular e uma ação correspondente para criar o token apropriado com a informação de posição.

```

14 tokens :=
15 $alpha+
16 "a"
17 "b"
18 "c"
19 "d"
20 "e"
21 "f"
22 "g"
23 "h"
24 "i"
25 "j"
26 "k"
27 "l"
28 "m"
29 "n"
30 "o"
31 "p"
32 "q"
33 "r"
34 "s"
35 "t"
36 "u"
37 "v"
38 "w"
39 "x"
40 "y"
41 "z"
42 "A"
43 "B"
44 "C"
45 "D"
46 "E"
47 "F"
48 "G"
49 "H"
50 "I"
51 "J"
52 "K"
53 "L"
54 "M"
55 "N"
56 "O"
57 "P"
58 "Q"
59 "R"
60 "S"
61 "T"
62 "U"
63 "V"
64 "W"
65 "X"
66 "Y"
67 "Z"
68 "0"
69 "1"
70 "2"
71 "3"
72 "4"
73 "5"
74 "6"
75 "7"
76 "8"
77 "9"
78 " "
79 "\n"
80 "\t"
81 "\r"
82 "\f"
83 "\o"
84 "\x"
85 "\y"
86 "\z"
87 "\_"
88 "\'"
89 "\""
90 "\."
91 "\,"
92 "\;"
93 "\:"
94 "\{"
95 "\}"
96 "\["
97 "\]"
98 "\["
99 "\]"
100 "\["
101 "\]"
102 "\["
103 "\]"
104 "\["
105 "\]"
106 "\["
107 "\]"
108 "\["
109 "\]"
110 "\["
111 "\]"
112 "\["
113 "\]"
114 "\["
115 "\]"
116 "\["
117 "\]"
118 "\["
119 "\]"
120 "\["
121 "\]"
122 "\["
123 "\]"
124 "\["
125 "\]"
126 "\["
127 "\]"
128 "\["
129 "\]"
130 "\["
131 "\]"
132 "\["
133 "\]"
134 "\["
135 "\]"
136 "\["
137 "\]"
138 "\["
139 "\]"
140 "\["
141 "\]"
142 "\["
143 "\]"
144 "\["
145 "\]"
146 "\["
147 "\]"
148 "\["
149 "\]"
150 "\["
151 "\]"
152 "\["
153 "\]"
154 "\["
155 "\]"
156 "\["
157 "\]"
158 "\["
159 "\]"
160 "\["
161 "\]"
162 "\["
163 "\]"
164 "\["
165 "\]"
166 "\["
167 "\]"
168 "\["
169 "\]"
170 "\["
171 "\]"
172 "\["
173 "\]"
174 "\["
175 "\]"
176 "\["
177 "\]"
178 "\["
179 "\]"
180 "\["
181 "\]"
182 "\["
183 "\]"
184 "\["
185 "\]"
186 "\["
187 "\]"
188 "\["
189 "\]"
190 "\["
191 "\]"
192 "\["
193 "\]"
194 "\["
195 "\]"
196 "\["
197 "\]"
198 "\["
199 "\]"
200 "\["
201 "\]"
202 "\["
203 "\]"
204 "\["
205 "\]"
206 "\["
207 "\]"
208 "\["
209 "\]"
210 "\["
211 "\]"
212 "\["
213 "\]"
214 "\["
215 "\]"
216 "\["
217 "\]"
218 "\["
219 "\]"
220 "\["
221 "\]"
222 "\["
223 "\]"
224 "\["
225 "\]"
226 "\["
227 "\]"
228 "\["
229 "\]"
230 "\["
231 "\]"
232 "\["
233 "\]"
234 "\["
235 "\]"
236 "\["
237 "\]"
238 "\["
239 "\]"
240 "\["
241 "\]"
242 "\["
243 "\]"
244 "\["
245 "\]"
246 "\["
247 "\]"
248 "\["
249 "\]"
250 "\["
251 "\]"
252 "\["
253 "\]"
254 "\["
255 "\]"
256 "\["
257 "\]"
258 "\["
259 "\]"
260 "\["
261 "\]"
262 "\["
263 "\]"
264 "\["
265 "\]"
266 "\["
267 "\]"
268 "\["
269 "\]"
270 "\["
271 "\]"
272 "\["
273 "\]"
274 "\["
275 "\]"
276 "\["
277 "\]"
278 "\["
279 "\]"
280 "\["
281 "\]"
282 "\["
283 "\]"
284 "\["
285 "\]"
286 "\["
287 "\]"
288 "\["
289 "\]"
290 "\["
291 "\]"
292 "\["
293 "\]"
294 "\["
295 "\]"
296 "\["
297 "\]"
298 "\["
299 "\]"
300 "\["
301 "\]"
302 "\["
303 "\]"
304 "\["
305 "\]"
306 "\["
307 "\]"
308 "\["
309 "\]"
310 "\["
311 "\]"
312 "\["
313 "\]"
314 "\["
315 "\]"
316 "\["
317 "\]"
318 "\["
319 "\]"
320 "\["
321 "\]"
322 "\["
323 "\]"
324 "\["
325 "\]"
326 "\["
327 "\]"
328 "\["
329 "\]"
330 "\["
331 "\]"
332 "\["
333 "\]"
334 "\["
335 "\]"
336 "\["
337 "\]"
338 "\["
339 "\]"
340 "\["
341 "\]"
342 "\["
343 "\]"
344 "\["
345 "\]"
346 "\["
347 "\]"
348 "\["
349 "\]"
350 "\["
351 "\]"
352 "\["
353 "\]"
354 "\["
355 "\]"
356 "\["
357 "\]"
358 "\["
359 "\]"
360 "\["
361 "\]"
362 "\["
363 "\]"
364 "\["
365 "\]"
366 "\["
367 "\]"
368 "\["
369 "\]"
370 "\["
371 "\]"
372 "\["
373 "\]"
374 "\["
375 "\]"
376 "\["
377 "\]"
378 "\["
379 "\]"
380 "\["
381 "\]"
382 "\["
383 "\]"
384 "\["
385 "\]"
386 "\["
387 "\]"
388 "\["
389 "\]"
390 "\["
391 "\]"
392 "\["
393 "\]"
394 "\["
395 "\]"
396 "\["
397 "\]"
398 "\["
399 "\]"
400 "\["
401 "\]"
402 "\["
403 "\]"
404 "\["
405 "\]"
406 "\["
407 "\]"
408 "\["
409 "\]"
410 "\["
411 "\]"
412 "\["
413 "\]"
414 "\["
415 "\]"
416 "\["
417 "\]"
418 "\["
419 "\]"
420 "\["
421 "\]"
422 "\["
423 "\]"
424 "\["
425 "\]"
426 "\["
427 "\]"
428 "\["
429 "\]"
430 "\["
431 "\]"
432 "\["
433 "\]"
434 "\["
435 "\]"
436 "\["
437 "\]"
438 "\["
439 "\]"
440 "\["
441 "\]"
442 "\["
443 "\]"
444 "\["
445 "\]"
446 "\["
447 "\]"
448 "\["
449 "\]"
450 "\["
451 "\]"
452 "\["
453 "\]"
454 "\["
455 "\]"
456 "\["
457 "\]"
458 "\["
459 "\]"
460 "\["
461 "\]"
462 "\["
463 "\]"
464 "\["
465 "\]"
466 "\["
467 "\]"
468 "\["
469 "\]"
470 "\["
471 "\]"
472 "\["
473 "\]"
474 "\["
475 "\]"
476 "\["
477 "\]"
478 "\["
479 "\]"
480 "\["
481 "\]"
482 "\["
483 "\]"
484 "\["
485 "\]"
486 "\["
487 "\]"
488 "\["
489 "\]"
490 "\["
491 "\]"
492 "\["
493 "\]"
494 "\["
495 "\]"
496 "\["
497 "\]"
498 "\["
499 "\]"
500 "\["
501 "\]"
502 "\["
503 "\]"
504 "\["
505 "\]"
506 "\["
507 "\]"
508 "\["
509 "\]"
510 "\["
511 "\]"
512 "\["
513 "\]"
514 "\["
515 "\]"
516 "\["
517 "\]"
518 "\["
519 "\]"
520 "\["
521 "\]"
522 "\["
523 "\]"
524 "\["
525 "\]"
526 "\["
527 "\]"
528 "\["
529 "\]"
530 "\["
531 "\]"
532 "\["
533 "\]"
534 "\["
535 "\]"
536 "\["
537 "\]"
538 "\["
539 "\]"
540 "\["
541 "\]"
542 "\["
543 "\]"
544 "\["
545 "\]"
546 "\["
547 "\]"
548 "\["
549 "\]"
550 "\["
551 "\]"
552 "\["
553 "\]"
554 "\["
555 "\]"
556 "\["
557 "\]"
558 "\["
559 "\]"
560 "\["
561 "\]"
562 "\["
563 "\]"
564 "\["
565 "\]"
566 "\["
567 "\]"
568 "\["
569 "\]"
570 "\["
571 "\]"
572 "\["
573 "\]"
574 "\["
575 "\]"
576 "\["
577 "\]"
578 "\["
579 "\]"
580 "\["
581 "\]"
582 "\["
583 "\]"
584 "\["
585 "\]"
586 "\["
587 "\]"
588 "\["
589 "\]"
590 "\["
591 "\]"
592 "\["
593 "\]"
594 "\["
595 "\]"
596 "\["
597 "\]"
598 "\["
599 "\]"
600 "\["
601 "\]"
602 "\["
603 "\]"
604 "\["
605 "\]"
606 "\["
607 "\]"
608 "\["
609 "\]"
610 "\["
611 "\]"
612 "\["
613 "\]"
614 "\["
615 "\]"
616 "\["
617 "\]"
618 "\["
619 "\]"
620 "\["
621 "\]"
622 "\["
623 "\]"
624 "\["
625 "\]"
626 "\["
627 "\]"
628 "\["
629 "\]"
630 "\["
631 "\]"
632 "\["
633 "\]"
634 "\["
635 "\]"
636 "\["
637 "\]"
638 "\["
639 "\]"
640 "\["
641 "\]"
642 "\["
643 "\]"
644 "\["
645 "\]"
646 "\["
647 "\]"
648 "\["
649 "\]"
650 "\["
651 "\]"
652 "\["
653 "\]"
654 "\["
655 "\]"
656 "\["
657 "\]"
658 "\["
659 "\]"
660 "\["
661 "\]"
662 "\["
663 "\]"
664 "\["
665 "\]"
666 "\["
667 "\]"
668 "\["
669 "\]"
670 "\["
671 "\]"
672 "\["
673 "\]"
674 "\["
675 "\]"
676 "\["
677 "\]"
678 "\["
679 "\]"
680 "\["
681 "\]"
682 "\["
683 "\]"
684 "\["
685 "\]"
686 "\["
687 "\]"
688 "\["
689 "\]"
690 "\["
691 "\]"
692 "\["
693 "\]"
694 "\["
695 "\]"
696 "\["
697 "\]"
698 "\["
699 "\]"
700 "\["
701 "\]"
702 "\["
703 "\]"
704 "\["
705 "\]"
706 "\["
707 "\]"
708 "\["
709 "\]"
710 "\["
711 "\]"
712 "\["
713 "\]"
714 "\["
715 "\]"
716 "\["
717 "\]"
718 "\["
719 "\]"
720 "\["
721 "\]"
722 "\["
723 "\]"
724 "\["
725 "\]"
726 "\["
727 "\]"
728 "\["
729 "\]"
730 "\["
731 "\]"
732 "\["
733 "\]"
734 "\["
735 "\]"
736 "\["
737 "\]"
738 "\["
739 "\]"
740 "\["
741 "\]"
742 "\["
743 "\]"
744 "\["
745 "\]"
746 "\["
747 "\]"
748 "\["
749 "\]"
750 "\["
751 "\]"
752 "\["
753 "\]"
754 "\["
755 "\]"
756 "\["
757 "\]"
758 "\["
759 "\]"
760 "\["
761 "\]"
762 "\["
763 "\]"
764 "\["
765 "\]"
766 "\["
767 "\]"
768 "\["
769 "\]"
770 "\["
771 "\]"
772 "\["
773 "\]"
774 "\["
775 "\]"
776 "\["
777 "\]"
778 "\["
779 "\]"
780 "\["
781 "\]"
782 "\["
783 "\]"
784 "\["
785 "\]"
786 "\["
787 "\]"
788 "\["
789 "\]"
790 "\["
791 "\]"
792 "\["
793 "\]"
794 "\["
795 "\]"
796 "\["
797 "\]"
798 "\["
799 "\]"
800 "\["
801 "\]"
802 "\["
803 "\]"
804 "\["
805 "\]"
806 "\["
807 "\]"
808 "\["
809 "\]"
810 "\["
811 "\]"
812 "\["
813 "\]"
814 "\["
815 "\]"
816 "\["
817 "\]"
818 "\["
819 "\]"
820 "\["
821 "\]"
822 "\["
823 "\]"
824 "\["
825 "\]"
826 "\["
827 "\]"
828 "\["
829 "\]"
830 "\["
831 "\]"
832 "\["
833 "\]"
834 "\["
835 "\]"
836 "\["
837 "\]"
838 "\["
839 "\]"
840 "\["
841 "\]"
842 "\["
843 "\]"
844 "\["
845 "\]"
846 "\["
847 "\]"
848 "\["
849 "\]"
850 "\["
851 "\]"
852 "\["
853 "\]"
854 "\["
855 "\]"
856 "\["
857 "\]"
858 "\["
859 "\]"
860 "\["
861 "\]"
862 "\["
863 "\]"
864 "\["
865 "\]"
866 "\["
867 "\]"
868 "\["
869 "\]"
870 "\["
871 "\]"
872 "\["
873 "\]"
874 "\["
875 "\]"
876 "\["
877 "\]"
878 "\["
879 "\]"
880 "\["
881 "\]"
882 "\["
883 "\]"
884 "\["
885 "\]"
886 "\["
887 "\]"
888 "\["
889 "\]"
890 "\["
891 "\]"
892 "\["
893 "\]"
894 "\["
895 "\]"
896 "\["
897 "\]"
898 "\["
899 "\]"
900 "\["
901 "\]"
902 "\["
903 "\]"
904 "\["
905 "\]"
906 "\["
907 "\]"
908 "\["
909 "\]"
910 "\["
911 "\]"
912 "\["
913 "\]"
914 "\["
915 "\]"
916 "\["
917 "\]"
918 "\["
919 "\]"
920 "\["
921 "\]"
922 "\["
923 "\]"
924 "\["
925 "\]"
926 "\["
927 "\]"
928 "\["
929 "\]"
930 "\["
931 "\]"
932 "\["
933 "\]"
934 "\["
935 "\]"
936 "\["
937 "\]"
938 "\["
939 "\]"
940 "\["
941 "\]"
942 "\["
943 "\]"
944 "\["
945 "\]"
946 "\["
947 "\]"
948 "\["
949 "\]"
950 "\["
951 "\]"
952 "\["
953 "\]"
954 "\["
955 "\]"
956 "\["
957 "\]"
958 "\["
959 "\]"
960 "\["
961 "\]"
962 "\["
963 "\]"
964 "\["
965 "\]"
966 "\["
967 "\]"
968 "\["
969 "\]"
970 "\["
971 "\]"
972 "\["
973 "\]"
974 "\["
975 "\]"
976 "\["
977 "\]"
978 "\["
979 "\]"
980 "\["
981 "\]"
982 "\["
983 "\]"
984 "\["
985 "\]"
986 "\["
987 "\]"
988 "\["
989 "\]"
990 "\["
991 "\]"
992 "\["
993 "\]"
994 "\["
995 "\]"
996 "\["
997 "\]"
998 "\["
999 "\]"
1000 "\["
1001 "\]"
1002 "\["
1003 "\]"
1004 "\["
1005 "\]"
1006 "\["
1007 "\]"
1008 "\["
1009 "\]"
1009

```

Figura 1: tokens da LEGOLAN

• Definição de TypeValue :

```

1  --- data TypeValue define um tipo dado algebrico, que pode representar
2  diferentes tipos de valores com informacoes de posicao (linha e coluna)
3  no codigo fonte
4
5
6  data TypeValue =
7      IntType Int (Int, Int) |
8      FloatType Float (Int, Int) |
9      StringType String (Int, Int) |
10     CharType Char (Int, Int) |
11     BoolType Bool (Int, Int) |
12     ListType (Int, [TypeValue]) (Int, Int) |
13     StructType [(String, TypeValue)] (Int, Int)
14     deriving (Eq)

```

Estes dados TypeValue são utilizados para armazenar os valores de cada tipo específico de nossa linguagem, carregando também informações adicionais, como o tamanho de uma

lista.

- **Instância da Classe show para TypeValue**

```
1  --- "instance Show TypeValue" define como 'TypeValue' deve ser convertido
    em uma string para exibicao
2
3  --- a funcao show implementa a conversao para cada construtor de
    'TypeValue', exibindo apenas o valor, sem a posicao
4  instance Show TypeValue where
5      show (IntType val pos) = show val
6      show (FloatType val pos) = show val
7      show (StringType val pos) = val
8      show (CharType val pos) = show val
9      show (BoolType val pos) = show val
10     show (ListType (len, val) pos) = show val
```

- **Definição de 'Token'**

data Token: Define um tipo de dado algébrico chamado Token, representando diferentes tipos de tokens que o lexer pode reconhecer, cada um com sua posição no código fonte.

```

1  data Token =
2      Id String (Int, Int)
3      Colon (Int, Int)
4      SemiColon (Int, Int)
5      Comma (Int, Int)
6      LeftParenthesis (Int, Int)
7      RightParenthesis (Int, Int)
8      LeftCurlyBrackets (Int, Int)
9      RightCurlyBrackets (Int, Int)
10     To (Int, Int)
11     Assign (Int, Int)
12     Plus (Int, Int)
13     Minus (Int, Int)
14     Times (Int, Int)
15     Divider (Int, Int)
16     IntegerDivider (Int, Int)
17     Exponent (Int, Int)
18     And (Int, Int)
19     Or (Int, Int)
20     Xor (Int, Int)
21     Not (Int, Int)
22     Less (Int, Int)
23     LessEqual (Int, Int)
24     Greater (Int, Int)
25     GreaterEqual (Int, Int)
26     Equal (Int, Int)
27     Different (Int, Int)
28     Type String (Int, Int)
29     If (Int, Int)
30     EndIf (Int, Int)
31     Elif (Int, Int)
32     Else (Int, Int)
33     For (Int, Int)
34     EndFor (Int, Int)
35     While (Int, Int)
36     EndWhile (Int, Int)
37     Return (Int, Int)
38     Declaration (Int, Int)
39     EndDeclaration (Int, Int)
40     Func (Int, Int)
41     EndFunc (Int, Int)
42     Main (Int, Int)
43     EndMain (Int, Int)
44     IntValue Int (Int, Int)
45     FloatValue Float (Int, Int)
46     StringValue String (Int, Int)
47     CharValue Char (Int, Int)
48     BoolValue Bool (Int, Int)
49     Typedef (Int, Int)
50     Struct (Int, Int)
51     Scan String (Int, Int)
52     Print String (Int, Int)
53
54     deriving (Eq, Show) --Permite a comparacao de igualdade e a conversao
                           para string dos valores de Token.

```

```

data TypeValue =
  IntType Int (Int, Int) |
  FloatType Float (Int, Int) |
  StringType String (Int, Int) |
  CharType Char (Int, Int) |
  BoolType Bool (Int, Int) |
  ListType ([TypeValue]) (Int, Int) -- Tamanho, lista de valores, posição
  StructType [(String, TypeValue)] (Int, Int)
  deriving (Eq)

instance Show TypeValue where
  show (IntType val pos) = show val
  show (FloatType val pos) = show val
  show (StringType val pos) = val
  show (CharType val pos) = show val
  show (BoolType val pos) = show val
  show (ListType (len, val) pos) = show val
  show (StructType (len, val) pos) = show val

data Token =
  ID String (Int, Int) |
  Colon (Int, Int) |
  SemiColon (Int, Int) |
  Comma (Int, Int) |
  LeftParenthesis (Int, Int) |
  RightParenthesis (Int, Int) |
  LeftCurlyBrackets (Int, Int) |
  RightCurlyBrackets (Int, Int) |
  Yo (Int, Int) |
  Assign (Int, Int) |
  Plus (Int, Int) |
  Minus (Int, Int) |
  Times (Int, Int) |
  Divider (Int, Int) |
  IntegerDivider (Int, Int) |
  Exponent (Int, Int) |
  And (Int, Int) |
  Or (Int, Int) |
  Xor (Int, Int) |
  Not (Int, Int) |
  Less (Int, Int) |
  LessEqual (Int, Int) |
  Greater (Int, Int) |
  GreaterEqual (Int, Int) |
  Equal (Int, Int) |
  Different (Int, Int) |
  Type String (Int, Int) |
  If (Int, Int) |
  Endif (Int, Int) |
  Elif (Int, Int) |
  Else (Int, Int) |
  For (Int, Int) |
  Endfor (Int, Int) |
  While (Int, Int) |
  Endwhile (Int, Int) |
  Return (Int, Int) |
  Declaration (Int, Int) |
  Enddeclaration (Int, Int) |
  Func (Int, Int) |
  Endfunc (Int, Int) |
  Main (Int, Int) |
  Endmain (Int, Int) |
  IntValue Int (Int, Int) |
  FloatValue Float (Int, Int) |
  StringValue String (Int, Int) |
  CharValue Char (Int, Int) |
  BoolValue Bool (Int, Int) |
  Typedef (Int, Int) |
  Struct (Int, Int) |
  Scan String (Int, Int) |
  Print String (Int, Int)
  deriving (Eq, Show)

getIC (AlexPa _ l c) = (l, c)

getTokens fn = unsafePerfomID (getTokensAux fn)

getTokensAux fn = do {fh <- openFile fn ReadMode;
  > <- hGetContents fh;

```

Figura 2: TypeValue e Token da LEGOLAN

2.2 Memória da Linguagem

A estrutura de memória *MemoryState* gerencia variáveis, funções, escopos e estruturas dentro de um programa. Ela permite a inserção, atualização e remoção de variáveis em tabelas de símbolos globais e locais, além de manter flags para controle de execução e análise de estruturas.

- Definição do Tipo *MemoryState*

```
1 type MemoryState = (  
2     Bool, -- Flag  
3     [(Token, TypeValue)], -- Symtable  
4     [(Token, [(Token, TypeValue)], [Token])], -- Funcs  
5     [(Token, TypeValue)], -- Structs  
6     CallStack, -- Callstack  
7     Bool, -- StructFlag  
8     Bool -- FuncFlag  
9 )  
10 type CallStack = [(Token, [(Token, TypeValue)], [Token])]
```

- **Bool (Flag)** : Uma flag generalizada que indica se o bloco de código atual deve ser analisado semanticamente.
- **[(Token, TypeValue)] (Symtable)**: Esta lista de tuplas é referente a tabela de símbolos do programa principal, sendo uma lista de pares de tokens e valores de tipo. Cada par representa uma variável e seu valor.
- **[(Token, [(Token, Token)], [Token])] (Funcs)**: Esta memória é utilizada para armazenar a sintaxe da implementação de subprogramas do código, onde o primeiro elemento é um Token Id referente ao nome do subprograma, temos uma lista de variáveis locais onde, inicialmente, é armazenado valores padrões somente para os parâmetros, e uma lista de Tokens referente aos *statements* locais da função.
- **[(Token, TypeValue)] (Structs)**: Uma pilha de registros, que armazena os registros criados pelo usuário, onde o Token é o nome do registro, e o TypeValue é o StructType possuindo as variáveis do registro.
- **[(Token, [(Token, Token)], [Token])] (CallStack)**: Uma pilha de funções instanciadas. Esta memória é utilizada assim que uma função é chamada em um programa principal, ou local. É resgatado a sintaxe armazenada em *funcs* e instanciada com os valores reais sobre os parâmetros, além de definir o input do parsec para os stnts da função instanciada.
- **Bool (StructFlag)**: Uma flag que indica se estamos realizando uma definição ou assinatura de estrutura.
- **Bool (FuncFlag)**: Uma flag que indica se estamos realizando o parser de uma função. Com esta flag, somos capazes de declarar e atribuir variáveis locais, ao invés de fazê-lo globalmente.

- Flags de Execução

```
1 -- Function to set the flag to True  
2 setFlagTrue :: MemoryState -> MemoryState  
3 setFlagTrue (flag, vars, funcs, structs, callstack, structflag) = (True,  
4     vars, funcs, structs, callstack, structflag)  
5 -- Function to set the flag to False  
6 setFlagFalse :: MemoryState -> MemoryState  
7 setFlagFalse (flag, vars, funcs, structs, callstack, structflag) = (False,  
8     vars, funcs, structs, callstack, structflag)  
9 isFlagTrue :: MemoryState -> Bool  
10 isFlagTrue (flag, _, _, _, _, _) = flag
```

- **setFlagTrue e setFlagFalse**: Funções que alteram a *flag* de execução para True ou False, respectivamente.
- **isFlagTrue**: Função que verifica se a *flag* de execução está definido como True.

- **Flags de Estrutura**

```

1  -- Function to set the flag to True
2  setStructFlagTrue :: MemoryState -> MemoryState
3  setStructFlagTrue (flag, vars, funcs, structs, callstack, structflag) =
4      (flag, vars, funcs, structs, callstack, True)
5
6  -- Function to set the flag to False
7  setStructFlagFalse :: MemoryState -> MemoryState
8  setStructFlagFalse (flag, vars, funcs, structs, callstack, structflag) =
9      (flag, vars, funcs, structs, callstack, False)
10
11 isStructFlagTrue :: MemoryState -> Bool
12 isStructFlagTrue (_, _, _, _, _, structflag) = structflag

```

- **setStructFlagTrue** e **setStructFlagFalse**: Funções que alteram a *structFlag* para True ou False, respectivamente.

- **isStructFlagTrue**: Função que verifica se a *structFlag* está definido como True.

- **Flags de funções**

```

1  -- Function to set the flag to True
2  setFuncFlagTrue :: MemoryState -> MemoryState
3  setFuncFlagTrue (flag, vars, funcs, structs, callstack, structflag,
4      funcFlag) = (flag, vars, funcs, structs, callstack, structflag, True)
5
6  -- Function to set the flag to False
7  setFuncFlagFalse :: MemoryState -> MemoryState
8  setFuncFlagFalse (flag, vars, funcs, structs, callStack, structFlag,
9      funcFlag) =
10      let newFuncFlag = (not (null callStack) && funcFlag)
11      in (flag, vars, funcs, structs, callStack, structFlag, newFuncFlag)
12
13 isFuncFlagTrue :: MemoryState -> Bool
14 isFuncFlagTrue (_, _, _, _, _, _, funcFlag) = funcFlag

```

- **setFuncFlagTrue** e **setFuncFlagFalse**: Funções que alteram a *funcFlag* para True ou False, respectivamente.

- **isFuncFlagTrue**: Função que verifica se a *funcFlag* está definida como True.

2.3 Representação de Símbolos, Tabela de Símbolos e Funções Associadas

A tabela de símbolos é uma estrutura de dados crucial que armazena informações sobre variáveis no programa principal. Implementamos uma tabela de símbolos utilizando uma tupla (**Token,TypeValue**), que permite inserções e consultas eficientes.

- **symtableGet**

```

1  symtableGet :: (Token) -> MemoryState -> Maybe TypeValue
2  symtableGet _ (_, [], _, _, _, _) = fail "variable not found"
3  symtableGet (Id idStr1 pos1) (_, (Id idStr2 pos2, value2) : listTail, _, _,
4      _, _) =
5      if idStr1 == idStr2
6      then Just value2
7      else symtableGet (Id idStr1 pos1) (False, listTail, [], [], [], False)
8  symtableGet _ _ = Nothing

```

- **symtableGet**: Recebe um Token ID e o MemoryState. Verifica se a variável existe na tabela de símbolos e, se existir, retorna seu valor. Caso contrário, lança um erro para o usuário.

- **symtableInsert**

```

1  symtableInsert :: (Token, TypeValue) -> MemoryState -> MemoryState
2  symtableInsert newSymbol@(newId, _) state@(flag, symtable, funcs, structs,
3      callstack, structflag, funcFlag) =
4      case symtableGet newId state of
5          Nothing -> (flag, symtable ++ [newSymbol], funcs, structs, callstack,
6              structflag, funcFlag)
7          Just _ -> error $ "Variable " ++ show newId ++ " already declared."

```

- **symtableInsert**: Insere um novo par (Token ID, TypeValue) na tabela de símbolos. Verifica se a variável já não existe nela e, se existir, lança um erro.

- **symtableUpdate**

```

1 symtableUpdate :: (Token, TypeValue) -> MemoryState -> MemoryState
2 symtableUpdate _ (_, [], _, _, _, _) = error "variable not found"
3 symtableUpdate (Id idStr1 pos1, value1) (flag, (Id idStr2 pos2, value2) :
4   listTail, funcs, structs, callstack, structflag, funcFlag) :
5   | idStr1 == idStr2 = (flag, (Id idStr1 pos2, value1) : listTail, funcs,
6     structs, callstack, structflag, funcFlag)
7   | otherwise =
8     let (flag', updatedSymtable, funcs', structs', callstack',
9       structflag', funcFlag') = symtableUpdate (Id idStr1 pos1, value1)
10      (flag, listTail, funcs, structs, callstack, structflag, funcFlag)
11     in (flag', (Id idStr2 pos2, value2) : updatedSymtable, funcs',
12       structs', callstack', structflag', funcFlag')

```

- **symtableUpdate**: Recebe uma tupla (Token ID, TypeValue) e atualiza o valor na tabela de símbolos, se o Token ID já estiver na tabela.

- **symtableRemove**

```

1 symtableRemove :: (Token, TypeValue) -> MemoryState -> MemoryState
2 symtableRemove _ (_, [], _, _, _, _) = error "variable not found"
3 symtableRemove (id1, v1) (flag, (id2, v2) : listTail, funcs, structs,
4   callstack, structflag, funcFlag) :
5   | id1 == id2 = (flag, listTail, funcs, structs, callstack, structflag,
6     funcFlag)
7   | otherwise =
8     let (flag', updatedSymtable, funcs', structs', callstack',
9       structflag', funcFlag') = symtableRemove (id1, v1) (flag,
10      listTail, funcs, structs, callstack, structflag, funcFlag)
11     in (flag', (id2, v2) : updatedSymtable, funcs', structs',
12       callstack', structflag', funcFlag')

```

- **symtableRemove**: Recebe uma tupla (Token ID, Token Value) e remove -a da tabela de símbolos, se o Token ID já existir na tabela.

2.4 Tratamento de Estruturas Condicionais e de Repetição

As estruturas condicionais (if, else) e de repetição (while, for) são tratadas por meio de funções específicas que analisam e executam os blocos de código associados.

- Estruturas Condicionais:

As estruturas condicionais são essenciais para controlar o fluxo de execução do programa. No nosso interpretador, as instruções condicionais são processadas de forma a verificar a validade das expressões associadas e, com base nisso, decidir qual bloco de código deve ser executado.

- i. 'if' :

A instrução if é tratada da seguinte maneira:

- **Tokenização:** O token if é identificado e a expressão condicional é avaliada.
- **Análise da Condição:** A condição dentro da instrução if é avaliada.
- **Execução do Bloco de Código:** Se a condição for verdadeira, o bloco de código associado ao if é executado.
- **Estruturas Aninhadas:** Estruturas if podem ser aninhadas, e o interpretador trata esses aninhamentos recursivamente.
- **Manipulação da Tabela de Símbolos:** Dependendo do estado da análise (semântica ou sintática), e do escopo sendo tratado (global ou local), a tabela de símbolos necessária é atualizada ou restaurada após a execução do bloco if.

```

1  ifStmt :: ParsecT [Token] MemoryState IO [Token]
2  ifStmt = do
3      ifLiteral <- ifToken
4      expression <- ifParenthesisExpression
5      colonLiteral <- colonToken
6      state <- getState
7      if isFlagTrue state
8          then do
9              if isFuncFlagTrue state
10                 then do
11                     let nLocalVar = getLocalSymtableLength state
12                     let result = evaluateCondition expression
13                     if result
14                         then do
15                             stmtsBlock <- stmts
16                             skip' <- manyTill anyToken (lookAhead endifToken)
17                             endIfLiteral <- endifToken
18                             semiCol1 <- semiColonToken
19                             updateState (removeLocalSymtableUntilLength nLocalVar)
20                             return ([ifLiteral] ++ [expression] ++ [colonLiteral] ++
21                                 stmtsBlock)
22                         else do
23                             skip' <- manyTill anyToken (lookAhead elifToken <|>
24                                 lookAhead elseToken <|> lookAhead endifToken)
25                             elifStmt' <- elifStmt
26                             if null elifStmt'
27                                 then do
28                                     skip' <- manyTill anyToken (lookAhead elseToken <|>
29                                         lookAhead endifToken)
30                                     elseStmt' <- elseStmt
31                                     endIfLiteral <- endifToken
32                                     semiCol <- semiColonToken
33                                     updateState (removeLocalSymtableUntilLength nLocalVar)
34                                     return ([ifLiteral] ++ [expression] ++ [colonLiteral]
35                                         ++ elseStmt')
36                                 else do
37                                     skip' <- manyTill anyToken (lookAhead endifToken <|>
38                                         lookAhead endifToken)
39                                     endIfLiteral <- endifToken
40                                     semiCol <- semiColonToken
41                                     updateState (removeLocalSymtableUntilLength nLocalVar)
42                                     return ([ifLiteral] ++ [expression] ++ [colonLiteral]
43                                         ++ elifStmt')
44                     else do
45                         let nVar = getSymtableLength state
46                         let result = evaluateCondition expression
47                         if result
48                             then do
49                                 stmtsBlock <- stmts
50                                 skip' <- manyTill anyToken (lookAhead endifToken)
51                                 endIfLiteral <- endifToken
52                                 semiCol1 <- semiColonToken
53                                 updateState (symtableRemoveUntilLength nVar)
54                                 return ([ifLiteral] ++ [expression] ++ [colonLiteral] ++
55                                     stmtsBlock)
56                             else do
57                                 skip' <- manyTill anyToken (lookAhead elifToken <|>
58                                     lookAhead elseToken <|> lookAhead endifToken)
59                                 elifStmt' <- elifStmt
60                                 if null elifStmt'
61                                     then do
62                                         skip' <- manyTill anyToken (lookAhead elseToken <|>
63                                             lookAhead endifToken)
64                                         elseStmt' <- elseStmt
65                                         endIfLiteral <- endifToken
66                                         semiCol <- semiColonToken
67                                         updateState (symtableRemoveUntilLength nVar)
68                                         return ([ifLiteral] ++ [expression] ++ [colonLiteral]
69                                             ++ elseStmt')
69                                     else do
70                                         skip' <- manyTill anyToken (lookAhead endifToken <|>
71                                             lookAhead endifToken)
72                                         endIfLiteral <- endifToken
73                                         semiCol <- semiColonToken

```

```

64         updateState (syntableRemoveUntilLenght nVar)
65         return ([ifLiteral] ++ [expression] ++ [colonLiteral]
66               ++ elifStmt')
67     else
68         return []

```

ii. 'elif' :

A instrução elif fornece uma forma de adicionar condições adicionais após um if. A análise e execução seguem estas etapas:

- **Tokenização:** O token elif é identificado e a expressão condicional é avaliada.
- **Análise da Condição:** A condição dentro da instrução elif é avaliada.
- **Execução do Bloco de Código:** Se a condição for verdadeira, o bloco de código associado ao elif é executado.
- **Verificação de Condições Adicionais:** Se a condição do elif for falsa, outras condições elif subsequentes são avaliadas até encontrar uma verdadeira ou chegar ao else.

```

1 elifStmt :: ParsecT [Token] MemoryState IO [Token]
2 elifStmt =
3     ( do
4         elifLiteral <- elifToken
5         expression <- ifParenthesisExpression
6         colonLiteral <- colonToken
7         let result = evaluateCondition expression
8         if result
9             then do
10                 stmtsBlock <- stmts
11                 return ([elifLiteral] ++ [expression] ++ [colonLiteral] ++
12                       stmtsBlock)
13             else do
14                 skip' <- manyTill anyToken (lookAhead elifToken)
15                 elifStmt' <- elifStmt
16                 return elifStmt'
17     )
18 <|> return []

```

iii. 'else' :

A instrução else fornece um bloco de código alternativo que é executado se nenhuma das condições if ou elif forem verdadeiras.

- **Tokenização:** O token else é identificado.
- **Execução do Bloco de Código:** O bloco de código associado ao else é executado incondicionalmente, já que todas as condições anteriores falharam.

```

1 elseStmt :: ParsecT [Token] MemoryState IO [Token]
2 elseStmt =
3     ( do
4         elseLiteral <- elseToken
5         colonLiteral <- colonToken
6         stmtsBlock <- stmts
7         return ([elseLiteral] ++ [colonLiteral] ++ stmtsBlock)
8     )
9     <|> return []

```

– **Abordagem Implementada :**

Utilizamos uma abordagem baseada em análise de tokens com a biblioteca Parsec em Haskell. Esta abordagem permite:

- * **Análise Léxica e Sintática:** Separação clara entre a análise de tokens e a avaliação das condições.

- * **Manutenção da Tabela de Símbolos:** A tabela de símbolos é manipulada para garantir que as variáveis locais e globais sejam tratadas corretamente durante a análise e execução.
- * **Tratamento de Blocos de Código:** Blocos de código são lidos e executados com base nos resultados das avaliações das condições.

Esta implementação garante que as estruturas condicionais sejam tratadas de forma eficiente e correta, mantendo o fluxo de controle do programa conforme esperado.

- **Estruturas de Repetição:**

As estruturas de repetição, como `while` e `for`, são essenciais para executar blocos de código repetidamente com base em uma condição. Abaixo, descrevemos como essas estruturas são tratadas no seu interpretador usando a biblioteca `Parsec` em Haskell.

- **'while':**

A instrução `while` é tratada da seguinte maneira:

- * **Tokenização:** O token `while` é identificado e a expressão condicional é armazenada sintaticamente.
- * **Análise da Condição:** Em um loop, a condição dentro da instrução `while` é avaliada.
- * **Execução do Bloco de Código:** Quando a condição for verdadeira, o bloco de código associado ao `while` é definido como input do parser e executado. Ao finalizar o bloco de código, definimos a expressão como input novamente e a reavaliamos, fazendo isto recursivamente até a condição se tornar falsa.
- * **Manipulação da Tabela de Símbolos:** Variáveis locais e globais são manipuladas corretamente durante a execução do loop.

```

1 whileStmt :: ParsecT [Token] MemoryState IO [Token]
2 whileStmt = do
3   whileLiteral <- whileToken
4   expressionTokens <- manyTill anyToken (lookAhead colonToken)
5   colonLiteral <- colonToken
6   stmtsBlock <- nestedWhileTokens 0
7   endWhileLiteral <- endWhileToken
8   semiCol <- semiColonToken
9
10  memoryState <- getState
11  input <- getInput
12
13  let loop = do
14    memoryState <- getState
15    if isFlagTrue memoryState
16      then do
17        if isFuncFlagTrue memoryState
18          then do
19            let nLocalVar = getLocalSymtableLength memoryState
20            setInput expressionTokens
21            expressionValue <- ifParenthesisExpression
22            let condition = evaluateCondition expressionValue
23            if condition
24              then do
25                setInput stmtsBlock
26                _ <- many stmts
27                updateState (removeLocalSymtableUntilLength
28                           nLocalVar)
29                loop
30              else setInput input
31            else do
32              let nVar = getSymtableLength memoryState
33              setInput expressionTokens
34              expressionValue <- ifParenthesisExpression
35              let condition = evaluateCondition expressionValue
36              if condition
37                then do
38                  setInput stmtsBlock
39                  _ <- many stmts
40                  updateState (symtableRemoveUntilLength nVar)
41                  loop
42                else setInput input
43              else setInput input
44
45  loop
46
47  return ([whileLiteral] ++ expressionTokens ++ [colonLiteral] ++
48         stmtsBlock ++ [endWhileLiteral] ++ [semiCol])
49
50 nestedWhileTokens :: Int -> ParsecT [Token] MemoryState IO [Token]
51 nestedWhileTokens nestDepth = do
52   tokens <- manyTill anyToken (lookAhead endWhileToken <|> lookAhead
53   whileToken)
54   next <- lookAhead anyToken
55   case next of
56     While _ -> do
57       whileToken' <- whileToken
58       nestedFor <- nestedWhileTokens (nestDepth + 1)
59       tokens' <- manyTill anyToken (lookAhead endWhileToken)
60       return (tokens ++ [whileToken'] ++ nestedFor ++ tokens')
61     -> do
62       if nestDepth == 0
63         then return tokens
64         else do
65           endWhile' <- endWhileToken
66           return (tokens ++ [endWhile'])

```

– ‘for’:

A instrução for é tratada da seguinte maneira:

- * **Tokenização:** O token for é identificado a primeira atribuição é analisada semanticamente, a expressão de iteração é armazenada sintaticamente assim como a atribuição iterativa.

- * **Análise da Condição:** Em um loop, a condição dentro da instrução for é avaliada.
- * **Execução do Bloco de Código:** Se a condição for verdadeira, o bloco de código associado ao for é definido como input e executado. Após seu término, a atribuição iterativa é analisada semanticamente e em seguida o bloco de expressão.
- * **Manipulação da Tabela de Símbolos:** Variáveis locais e globais são manipuladas corretamente durante a execução do loop.

```

1  forStmt :: ParsecT [Token] MemoryState IO [Token]
2  forStmt = do
3      forLiteral <- forToken
4      leftParenthesis <- leftParenthesisToken
5      assign' <- assign
6      semiCol' <- semiColonToken
7      expressionTokens <- manyTill anyToken (lookAhead semiColonToken)
8      semiCol'' <- semiColonToken
9      updateAssign <- manyTill anyToken (lookAhead rightParenthesisToken)
10     rightParenthesis <- rightParenthesisToken
11     colon' <- colonToken
12     stmtsBlock <- nestedForTokens 0
13     endFor <- endForToken
14     semiCol <- semiColonToken
15     memoryState <- getState
16     input <- getInput
17     let loop = do
18         memoryState <- getState
19         if isFlagTrue memoryState
20             then do
21                 if isFuncFlagTrue memoryState
22                     then do
23                         let nLocalVar = getLocalSymtableLength memoryState
24                         setInput expressionTokens
25                         expressionValue <- relatOrLogicExpression
26                         let condition = evaluateCondition expressionValue
27                         if condition
28                             then do
29                                 setInput stmtsBlock
30                                 _ <- many stmts
31                                 setInput updateAssign
32                                 assign'' <- assign
33                                 updateState (removeLocalSymtableUntillLength
34                                     nLocalVar)
35                                 loop
36                             else setInput input
37                         else do
38                             let nVar = getSymtableLength memoryState
39                             setInput expressionTokens
40                             expressionValue <- relatOrLogicExpression
41                             let condition = evaluateCondition expressionValue
42                             if condition
43                                 then do
44                                     setInput stmtsBlock
45                                     _ <- many stmts
46                                     setInput updateAssign
47                                     assign'' <- assign
48                                     updateState (symtableRemoveUntillLength nVar)
49                                     loop
50                                 else setInput input
51                             else setInput input
52     loop
53
54     return ([forLiteral] ++ [colon'] ++ stmtsBlock ++ [endFor] ++
55         [semiCol])
56
57 nestedForTokens :: Int -> ParsecT [Token] MemoryState IO [Token]
58 nestedForTokens nestDepth = do
59     tokens <- manyTill anyToken (lookAhead endForToken <|> lookAhead
60         forToken)
61     next <- lookAhead anyToken
62     case next of
63         For _ -> do
64             forToken' <- forToken
65             nestedFor <- nestedForTokens (nestDepth + 1)
66             tokens' <- manyTill anyToken (lookAhead endForToken)
67             return (tokens ++ [forToken'] ++ nestedFor ++ tokens')
68         _ -> do
69             if nestDepth == 0
70                 then return tokens
71             else do
72                 endFor' <- endForToken
73                 return (tokens ++ [endFor'])

```

– **Abordagem Implementada**

Utilizamos uma abordagem baseada em análise de tokens com a biblioteca Parsec em Haskell. Esta abordagem permite:

- * Análise Léxica e Sintática: Separação clara entre a análise de tokens e a avaliação das condições.
- * Manutenção da Tabela de Símbolos: A tabela de símbolos é manipulada para garantir que as variáveis locais e globais sejam tratadas corretamente durante a análise e execução.
- * Tratamento de Blocos de Código: Blocos de código são lidos e executados com base nos resultados das avaliações das condições.

Esta implementação garante que as estruturas de repetição sejam tratadas de forma eficiente e correta, mantendo o fluxo de controle do programa conforme esperado.

2.5 Tratamento de Subprogramas

Os subprogramas, incluindo funções e procedimentos, são blocos de código que podem ser chamados e executados a partir de diferentes pontos em um programa. Eles permitem a modularização do código, promovendo reutilização e facilitando a manutenção. Em linguagens de programação, os subprogramas são definidos com um nome e um conjunto de parâmetros e podem retornar um valor (no caso de funções).

- **Chamada de Função:**

A chamada de função envolve os seguintes passos:

- Tokenização e Análise: Identificação do identificador da função, parênteses e parâmetros.
- Verificação de Parâmetros: Verificação se existe uma função com o identificador fornecido, que aceite o mesmo número e tipo de parâmetros.
- Manipulação da Pilha de Chamadas: Atualização do estado para refletir a nova chamada de função, incluindo a atualização da pilha de chamadas.
- Execução da Função: Avaliação das instruções da função.
- Manipulação do Estado Pós-Chamada: Atualização do estado após a execução da função, incluindo a passagem de valores de retorno.

```

1      funcStmt :: ParsecT [Token] MemoryState IO [Token]
2  funcStmt = do
3      id <- idToken
4      leftpar <- leftParenthesisToken
5      parameters' <- parametersExprBlock
6      rightPar <- rightParenthesisToken
7      semiCol <- semiColonToken
8      state <- getState
9      input <- getInput
10     let parametersValues = parametersValuesFromIDs parameters' state
11     let funcMemoryInstance@(idFunc, funcMemory, funcStmts) =
12         checkFunctionParameters id parametersValues state
13
14     updateState (callStackPush funcMemoryInstance)
15     state' <- getState
16
17     updateState setFuncFlagTrue
18     setInput funcStmts
19     stmts' <- stmts
20     setInput input
21
22     newState <- getState
23     let topStack = callStackGet newState
24
25     updateState (const (callStackPop newState))
26     removedState <- getState
27
28     updateState (passResultValue parameters' topStack)
29     updatedState <- getState
30
31     updateState setFuncFlagFalse
32     updateState setFlagTrue
33     return stmts'

```

- Expressão de função:

As expressões de função são tratadas de forma semelhante às chamadas de função, com a diferença de que elas retornam um valor específico.

```

1      funcExpr :: ParsecT [Token] MemoryState IO Token
2  funcExpr = do
3      id <- idToken
4      leftpar <- leftParenthesisToken
5      parameters' <- parametersExprBlock
6      rightPar <- rightParenthesisToken
7      state <- getState
8      input <- getInput
9
10     let parametersValues = parametersValuesFromIDs parameters' state
11     let funcMemoryInstance@(idFunc, funcMemory, funcStmts) =
12         checkFunctionParameters id parametersValues state
13
14     updateState (callStackPush funcMemoryInstance)
15     state' <- getState
16
17     updateState setFuncFlagTrue
18     setInput funcStmts
19     stmts' <- stmts
20     setInput input
21
22     newState <- getState
23     let topStack = callStackGet newState
24
25     updateState (const (callStackPop newState))
26     removedState <- getState
27
28     updateState (passResultValue parameters' topStack)
29     updatedState <- getState
30
31     updateState setFuncFlagFalse
32     updateState setFlagTrue
33     return (stmts' !! (length stmts' - 2))

```

- Principais Componentes:

- Verificação de Parâmetros:

A função **checkFunctionParameters** é responsável por verificar se uma função com o identificador fornecido existe e se aceita os parâmetros fornecidos.

```
1      checkFunctionParameters :: String -> [Value] -> MemoryState ->
      (String, FunctionMemory, [Token])
2  checkFunctionParameters id parameters state = -- Implementacao da
      verificacao de parametros
```

- Manipulação da Pilha de Chamadas:

A manipulação da pilha de chamadas envolve operações como `callStackPush`, `callStackPop`, e `callStackGet`, que gerenciam a pilha de chamadas.

```
1      callStackPush :: FunctionMemoryInstance -> MemoryState ->
      MemoryState
2  callStackPush instance state = -- Implementacao da operacao de push na
      pilha
3
4  callStackPop :: MemoryState -> MemoryState
5  callStackPop state = -- Implementacao da operacao de pop na pilha
6
7  callStackGet :: MemoryState -> FunctionMemoryInstance
8  callStackGet state = -- Implementacao da operacao de get na pilha
```

- Simulação de Passagem por referência :

Para simular a passagem por referência no nosso interpretador, utilizamos a técnica de passagem por valor resultado. Isso significa que os valores dos parâmetros são copiados para dentro da função, e os resultados são copiados de volta para os parâmetros originais após a execução da função. Esse método permite que as funções possam modificar os valores dos parâmetros, simulando o comportamento da passagem por referência.

A implementação de subprogramas no seu interpretador é crucial para permitir a modularização e reutilização de código. A abordagem baseada em Parsec em Haskell permite uma análise sintática e semântica eficiente das chamadas de função e expressões de função, garantindo que o fluxo de controle do programa seja mantido corretamente. Além disso, a simulação da passagem por referência através da passagem por valor resultado proporciona flexibilidade na manipulação de parâmetros dentro das funções.

2.6 Verificações Realizadas

Implementamos diversas verificações para assegurar a corretude do programa, como foram apresentados acima, funções como:

- `symtableInsert`
- `symtableGet`
- `symtableUpdate`
- `symtableRemove`
- `callStackGet`
- `callStackPop`
- `callStackUpdateTop`

Dentre outras, são funções que fazem verificação de erros de tipos, e existências de variáveis.

3. Instruções de Uso do Interpretador

Para utilizar o interpretador, siga os passos abaixo:

- Compilação do Léxico e geração do Parser:

```
1 alex tokens.x
```

- Compilação do Interpretador:
Compile o interpretador utilizando o GHC (Glasgow Haskell Compiler):

```
1 ghc parser.hs
```

- Execução do Interpretador:
Execute o interpretador passando o arquivo de código-fonte da linguagem criada:

```
1 ./parser
```

- Exemplos de Entrada:

```
1 ./parser exemplos/exemplo_a_sua_escolha
```

Aqui estão alguns exemplos de programas escritos na linguagem criada:

— exemplo1_declarations.txt

```
1 declaration:
2 a: int;
3 b: int;
4 c: int;
5 d: int;
6 end_declaration
```

— exemplo2_uma_atribuicao.txt

```
1 declaration:
2 a: int;
3 c: int;
4 end_declaration
5
6 main:
7 #a: int;
8 b: int;
9 d: int;
10 a = 10;
11 end_main
```

— exemplo3_atribuicao_tipos_simples.txt

```
1 declaration:
2 a: int;
3 b: string;
4 c: char;
5 d: float;
6 e: bool;
7 end_declaration
8
9 main:
10 a = 10;
11 b = "oi";
12 c = 'o';
13 d = 10.102;
14 e = false;
15 end_main
```

— exemplo4_atribuicao_por_expressao.txt

```
1 declaration:
2 b: int;
3 end_declaration
4
5 main:
6 b = 10 + 5;
7 end_main
```

– exemplo5_atribuicoes_por_expressoes.txt

```
1  declaration:
2      a: int;
3      b: int;
4      c: int;
5      d: float;
6      e: int;
7      f: int;
8      problem_1: float;
9      g: bool;
10     h: int;
11     i: int;
12     j: bool;
13     k: bool;
14     l: bool;
15     m: bool;
16     n: bool;
17     o: bool;
18     p: bool;
19     q: bool;
20     r: bool;
21     s: bool;
22     t: bool;
23     u: bool;
24     v: bool;
25     w: bool;
26     x: bool;
27     y: bool;
28     z: bool;
29 end_declaration
30
31 main:
32     a = 10 + 5; # 15
33     b = 10 - 5; # 5
34     c = 10 * 2; # 20
35     d = 10 / 2; # 5.0
36     e = 10 // 2; # 5
37     f = 10 ** 2; # 100
38     problem_1 = 10.1 ** 2 - 5.5 + 5; # 95
39     g = !false; # True
40     h = (10 + 5); # 15
41     i = 10; # 10
42     j = 5 < 10 + 5; # True
43     k = 5 + 5 < 10 + 5; # True
44     l = 5 - 5 < 10 - 5; # True
45     m = 5 * 5 > 5; # True
46     n = 5/5 >= 1; # True
47     o = 10 // 2 <= 5; # True
48     p = 10 ** 2 != 99; # True
49     r = (10 ** 2) == (10 ** 2); # True
50     s = 10 > 5; # True
51     t = (10 > 5) && (5 < 10); # True
52     u = (10 != 5) || (5 > 10); # True
53     v = (10 < 5) && (5 < 10); # False
54     w = (10 < 5) && (11 < 6); # False
55     x = (10 != 10) || (10 == 10); # True
56     y = (10 != 10) || (9 != 9); # False
57     #z
58 end_main
```

– exemplo5.1_problema_1.txt

```
1  declaration:
2      problem_1: float;
3      a: float;
4      b: int;
5      c: float;
6      d: int;
7  end_declaration
8
9  main:
10     a = 10.1;
11     b = 2;
12     c = 5.5;
13     d = 5;
14     problem_1 = a ** b - c + d; # 101.51
15     print(problem_1);
16 end_main
```

– exemplo6_if_stmts.txt

```
1      declaration:
2      a: int;
3      b: int;
4      c: int;
5  end_declaration
6
7  main:
8      if(11 < 10):
9          a = 5;
10     elif(10 > 11):
11         b = 10;
12     elif(10 > 9):
13         c = 15;
14     else:
15         a = 15;
16         b = 15;
17         c = 15;
18     end_if;
19 end_main
```

– exemplo7_while_stmts.txt

```
1      declaration:
2      a: int;
3      b: int;
4      c: int;
5  end_declaration
6
7  main:
8      a = 5;
9      while(a < 10):
10         a = a + 1;
11     end_while;
12     print(a);
13 end_main
```

– exemplo8_for_stmts.txt

```
1      declaration:
2      a: int;
3      x: int;
4  end_declaration
5
6  main:
7      for(x = 0; x < 10; x = x + 1):
8          a = a + 5;
9      end_for;
10
11     print(a);
12 end_main
```


– exemplo9_scan.txt

```
1      declaration:
2      a: int;
3  end_declaration
4
5  main:
6      a = scan("Oi ");
7  end_main
```

– exemplo10_problema_2.txt

```
1      declaration:
2      entry: int;
3      count1: int;
4      count2: int;
5      count3: int;
6      count4: int;
7  end_declaration
8
9  main:
10     entry = scan("Informe um valor que deseja verificar");
11     while(entry >= 0):
12         if ((entry >= 0) && (entry <= 25)):
13             count1 = count1 + 1;
14         elif ((entry >= 26) && (entry <= 50)):
15             count2 = count2 + 1;
16         elif ((entry >= 51) && (entry <= 75)):
17             count3 = count3 + 1;
18         elif ((entry >= 76) && (entry <= 100)):
19             count4 = count4 + 1;
20         end_if;
21         entry = scan("Informe um valor que deseja verificar");
22     end_while;
23
24     print(count1);
25     print(count2);
26     print(count3);
27     print(count4);
28 end_main
```

– exemplo11_print.txt

```
1      declaration:
2      entry: int;
3      numero: float;
4      teste: string;
5      cara : char;
6      flasg : bool;
7  end_declaration
8
9  main:
10     entry = 1;
11     numero = 63.89;
12     teste = "teste";
13     cara = 'k';
14     flasg = true;
15     print(teste);
16     print(entry);
17     print(numero);
18     print(cara);
19     print(flasg);
20 end_main
```