



University of Porto
Department of Computer Science

SK Combinator Interpreter
for
Lambda Calculus

Henrique Teixeira
June 27, 2025

Abstract

This article shows a comprehensive introduction to the field of Lambda Calculus followed by one of Combinatorial Logic. Then, it highlights the applications for the interpreter built. Finally, the design of the interpreter and techniques used are discussed.

Keywords: Computer Science, Lambda Calculus, Combinatorial Logic, Interpreter

Contents

1	Introduction and Context Review	1
1.1	Lambda Calculus	1
1.2	Combinatorial Logic	3
1.3	Translation Rules	4
1.4	Examples	4
2	Lambda Calculus and SK Structs	5
3	Interpretation and Translation	7
4	Discussion and Conclusion	14
	References	15

Chapter 1

Introduction and Context Review

1.1 Lambda Calculus

Lambda Calculus is a field in Computer Science created by Alonzo Church and the foundation behind functional programming. It represents data and programs in a set of functions. Everything that is computable can be represented by a function, more precisely, a lambda expression.

Definition 1. *The grammar for Lambda Calculus Expressions (LCE) is the following:*

$$\begin{aligned} LCE &\rightarrow APP \mid ABS \\ APP &\rightarrow APP \, TERM \mid TERM \\ ABS &\rightarrow \lambda x. LCE \\ TERM &\rightarrow (LCE) \mid x \\ &\forall x \in \Sigma \end{aligned}$$

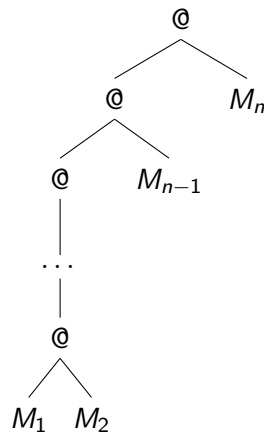
Σ is the set of all terminal symbols, which corresponds to the alphabet of the language. Usually, we use minor case letters to represent variables, like x and y , and upper case letter, like M and N , to represent terms.

We write $M \equiv N$ to state that M and N are identical terms.

This is a left-recursive grammar and because of that the application of terms can be seen as follows:

$$((\dots (M_1 \, M_2) \dots) \, M_{n-1}) \, M_n$$

And the syntax tree as:



There are some important definitions for our work.

Definition 2. $BV(M)$, the set of all bound variables in M , is given by

$$\begin{aligned} BV(x) &= \emptyset \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \\ BV(MN) &= BV(M) \cup BV(N) \end{aligned}$$

Definition 3. $FV(M)$, the set of all free variables in M , is given by

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \end{aligned}$$

Definition 4. $M[L/y]$, the result of substituting L for all free occurrences of y in M , is given by

$$\begin{aligned} x[L/y] &\equiv \begin{cases} L, & x \equiv y \\ x, & \text{otherwise} \end{cases} \\ (\lambda x.M)[L/y] &\equiv \begin{cases} (\lambda x.M), & x \equiv y \\ (\lambda x.M[L/y]), & \text{otherwise} \end{cases} \\ (MN)[L/y] &\equiv (M[L/y]N[L/y]) \end{aligned}$$

A substitution $M[N/x]$ is said to be safe if $BV(M) \cap FV(N) = \emptyset$.

Definition 5. **Conversions** are rules for manipulation of lambda abstractions (ABS). There exist 3 types: α -conversion, β -conversion and η -conversion.

$$\begin{aligned} (\lambda x.M) &\rightarrow_{\alpha} (\lambda y.M[y/x]) \\ (\lambda x.M)N &\rightarrow_{\beta} M[N/x], \text{ It is valid provided the substitution is safe.} \\ (\lambda x.Mx) &\rightarrow_{\eta} M, \text{ It is valid provided } x \notin FV(M) \end{aligned}$$

Definition 6. **Reduction** $M \rightarrow N$ is the process of applying a conversion to some subterm of M in order to create N . If a term admits no reductions, then it is in normal form. Some terms cannot be reduced to normal form (ex.: $(\lambda x.xx)(\lambda x.xx) \rightarrow_{\beta} (\lambda x.xx)(\lambda x.xx)$).

Definition 7. We write $M \twoheadrightarrow N$ if $M \rightarrow M_1 \rightarrow \dots \rightarrow M_k \equiv N$, $k \geq 0$. \twoheadrightarrow is the relation \rightarrow^* , the reflexive/transitive closure of \rightarrow .

Theorem 1. The Church-Rosser Theorem

$$\forall M, N \in LCE, M = N \longrightarrow \exists L \in LCE, M \twoheadrightarrow L \wedge N \twoheadrightarrow L$$

This theorem provides the consistency to lambda calculus, as we cannot reach two different normal forms following two different reduction paths.

With this, we can now construct programs that are a set of lambda expressions.

An example of a lambda expression is $\lambda x.\lambda y.x$ which can have multiple meanings in terms of computation. We can define the function *true* as the function that produces the first of two elements. This is useful for the logic of if statements, for example. We can also define *false* as the function that returns the second argument. Here are some definitions:

$$\begin{aligned}
\text{true} &\equiv \lambda x. \lambda y. x \\
\text{false} &\equiv \lambda x. \lambda y. y \\
\text{if} &\equiv \lambda x. \lambda y. \lambda f. f x y \\
\text{not} &\equiv \lambda p. p \text{ false true} \\
\text{and} &\equiv \lambda p. \lambda q. p q \text{ false} \\
\text{or} &\equiv \lambda p. \lambda q. p \text{ true } q
\end{aligned}$$

We can also define numbers as functions. The following definitions of natural numbers are also called church numerals.

$$\begin{aligned}
0 &\equiv \lambda f. \lambda x. x \\
1 &\equiv \lambda f. \lambda x. f x \\
&\dots \\
n &\equiv \lambda f. \lambda x. f(\dots(f x)), \text{ where } f \text{ is applied a total of } n \text{ times}
\end{aligned}$$

1.2 Combinatorial Logic

Combinatorial Logic (CL) is a field in Computer Science, with a theory in its own right, related to Lambda Calculus. It consists of sets of operators called combinators. Each one can be defined with a number of arguments and a mapping to a combination and/or copy of them. One of the simplest versions is the set of combinators $\{S, K\}$. CL is not as readable as Lambda Calculus; however, it has the potential to be Turing Complete.

Theorem 2. *The combinator set $\{\{S, K\}, \{S P Q R \rightarrow_w P Q (Q R), K P Q \rightarrow_w P\}\}$ is Turing Complete.*

This combinators can be defined by λ -calculus as

$$\begin{aligned}
\mathbf{S} &\equiv \lambda z. \lambda y. \lambda x. x z (y z) \\
\mathbf{K} &\equiv \lambda x. \lambda y. x
\end{aligned}$$

Some interesting functions which can be defined with this set of combinators:

$$\begin{aligned}
I &\equiv S K K \equiv \lambda x. x, \text{ the identity function} \\
\text{true} &\equiv K \equiv \lambda x. \lambda y. x \\
\text{false} &\equiv K(S K K) \equiv \lambda x. \lambda y. y
\end{aligned}$$

We can transform any Lambda Calculus expressions to a combination of S and K, which gives the interpreter/compiler an easier time with the runtime of programs. One big problem with the representation of lambda expressions is the fact that they can have free variables which, if not previously stated, the behavior and so the resulting computation is undefined.

If we want to write a program with these expressions, we could write a single expression which represents the entire program, but that is not feasible as the programmer would not easily be able to understand what they are doing, however, it would solve the free variable problem. Another way we can solve this is to allow the programmer to set multiple variables to different lambda expressions. The combination of this variables/functions will piece by piece build the intended program.

1.3 Translation Rules

For the this interpreter, the following recursive translation rules were used (T_{CL}):

$$\lambda^T x.x \equiv S K K \quad (1)$$

$$\lambda^T x.P \equiv K P, x \notin FV(P) \quad (2)$$

$$\lambda^T x.P x \equiv P \quad (3)$$

$$\lambda^T x.PQ \equiv S(\lambda^T x.P)(\lambda^T x.Q), x \in FV(P) \wedge x \in FV(Q) \quad (4)$$

$$\lambda^T x.PQ \equiv S(K P)(\lambda^T x.Q), x \notin FV(P) \wedge x \in FV(Q) \quad (5)$$

$$\lambda^T x.PQ \equiv S(\lambda^T x.P)(K Q), x \in FV(P) \wedge x \notin FV(Q) \quad (6)$$

$$T_{CL}(PQ) \equiv T_{CL}(P)T_{CL}(Q) \quad (7)$$

1.4 Examples

Some examples for easier understanding:

$$\begin{aligned} T_{CL}(true) &\equiv \lambda^T x.\lambda^T y.x \\ &\equiv \lambda^T x.(\lambda^T y.x) \\ &\equiv \lambda^T x.(K x) \\ &\equiv \lambda^T x.K x \\ &\equiv K \end{aligned}$$

$$\begin{aligned} T_{CL}(false) &\equiv \lambda^T x.\lambda^T y.y \\ &\equiv \lambda^T x.(\lambda^T y.y) \\ &\equiv \lambda^T x.(S K K) \\ &\equiv K(S K K) \end{aligned}$$

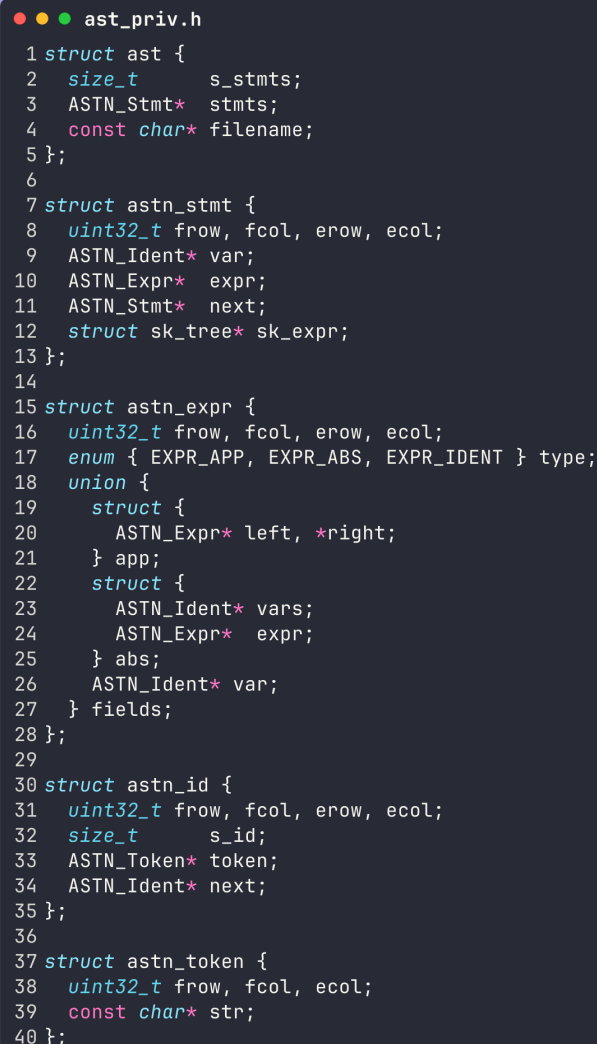
$$\begin{aligned} T_{CL}(and) &\equiv \lambda^T p.\lambda^T q.p q false \\ &\equiv \lambda^T p.(\lambda^T q.p q false) \\ &\equiv \lambda^T p.S(\lambda^T q.p q)(K false) \\ &\equiv \lambda^T p.S(S(K p)(\lambda^T q.q))(K false) \\ &\equiv \lambda^T p.S(S(K p)(S K K))(K false) \\ &\equiv S(\lambda^T p.S(S(K p)(S K K)))(K(K false)) \\ &\equiv S(S(K S)(\lambda^T p.S(K p)(S K K)))(K(K false)) \\ &\equiv S(S(K S)((\lambda^T p.S(K p))(K(S K K)))(K(K false)) \\ &\equiv S(S(K S)((S(K S)(\lambda^T p.K p))(K(S K K)))(K(K false)) \\ &\equiv S(S(K S)((S(K S)K)(K(S K K)))(K(K false)) \end{aligned}$$

$$\begin{aligned} T_{CL}(not) &\equiv \lambda^T p.p false true \\ &\equiv S(\lambda^T p.p false)(K true) \\ &\equiv S(S(\lambda^T p.p)(K false))(K true) \\ &\equiv S(S(S K K)(K false))(K true) \end{aligned}$$

Chapter 2

Lambda Calculus and SK Structs

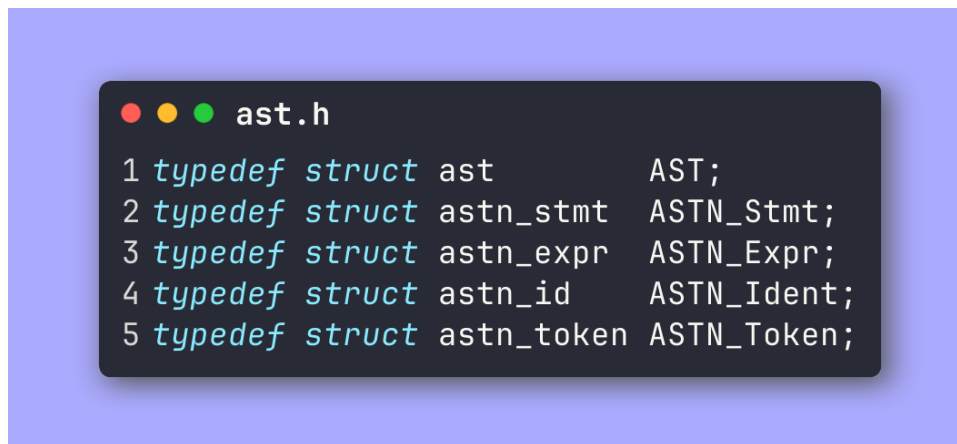
The way I chose to represent the data structures for the lambda expressions is the following:



```
1 struct ast {
2     size_t      s_stmts;
3     ASTN_Stmt*  stmts;
4     const char* filename;
5 };
6
7 struct astn_stmt {
8     uint32_t frow, fcol, erow, ecol;
9     ASTN_Ident* var;
10    ASTN_Expr*  expr;
11    ASTN_Stmt*  next;
12    struct sk_tree* sk_expr;
13 };
14
15 struct astn_expr {
16     uint32_t frow, fcol, erow, ecol;
17     enum { Expr_App, Expr_Abs, Expr_Ident } type;
18     union {
19         struct {
20             ASTN_Expr* left, *right;
21         } app;
22         struct {
23             ASTN_Ident* vars;
24             ASTN_Expr*  expr;
25         } abs;
26         ASTN_Ident* var;
27     } fields;
28 };
29
30 struct astn_id {
31     uint32_t frow, fcol, erow, ecol;
32     size_t      s_id;
33     ASTN-Token* token;
34     ASTN_Ident* next;
35 };
36
37 struct astn_token {
38     uint32_t frow, fcol, ecol;
39     const char* str;
40 };
```

Figure 2.1: Snapshot of the LCE Implementation

For some context, here are the definitions of the "ASTN" pointers:

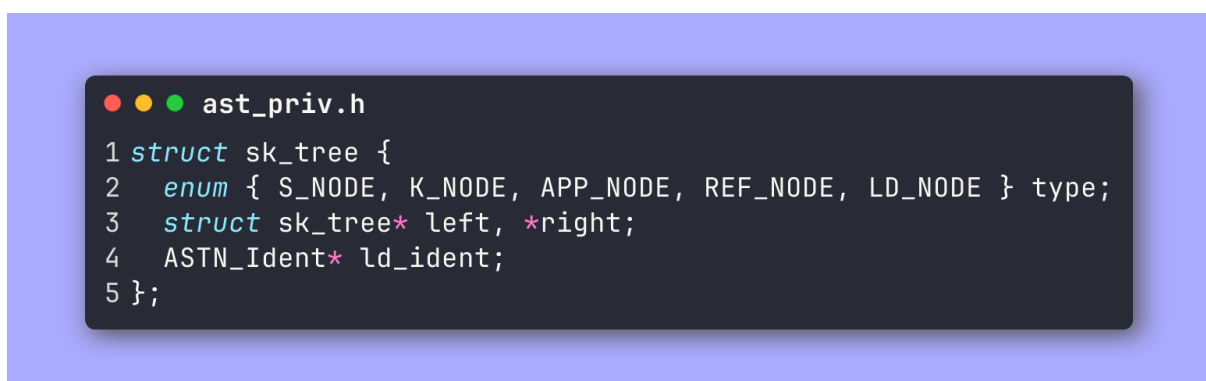
A terminal window with a dark background and light blue text. The title bar shows three colored circles (red, yellow, green) and the filename 'ast.h'. The code contains five typedef statements for ASTN pointers.

```
1 typedef struct ast      AST;
2 typedef struct astn_stmt ASTN_Stmt;
3 typedef struct astn_expr ASTN_Expr;
4 typedef struct astn_id   ASTN_Ident;
5 typedef struct astn_token ASTN-Token;
```

Figure 2.2: Snapshot of the ASTN Definitions

These data structures provide the parser everything for the representation of the entire file input by the programmer. The most important struct, however, is "astn_expr" which holds the data for the entire tree of lambda expressions.

Next follows the data structure chosen for the SK tree representation:

A terminal window with a dark background and light blue text. The title bar shows three colored circles (red, yellow, green) and the filename 'ast_priv.h'. The code defines a struct 'sk_tree' containing an enum 'type' and pointers to 'left', 'right', and 'ld_ident'.

```
1 struct sk_tree {
2     enum { S_NODE, K_NODE, APP_NODE, REF_NODE, LD_NODE } type;
3     struct sk_tree* left, *right;
4     ASTN_Ident* ld_ident;
5 };
```

Figure 2.3: Snapshot of the SK Struct

The SK Tree struct can be one of five types: S, K, application (APP), reference (REF) or lambda (LD) nodes. If a SK Tree node is of the type lambda, then the tree is being formed during the process of converting the lambda expression to the SK. The pointer references a free variable in the expression that corresponds to a higher abstraction in the lambda tree. If the variable is a free variable in the entire expression, then it is defined earlier and so it will turn into a reference to another SK tree. This will be explained next.

Chapter 3

Interpretation and Translation

As the process of translation is very long, I'll start by describing how the interpreter beta reduces a SK Tree.

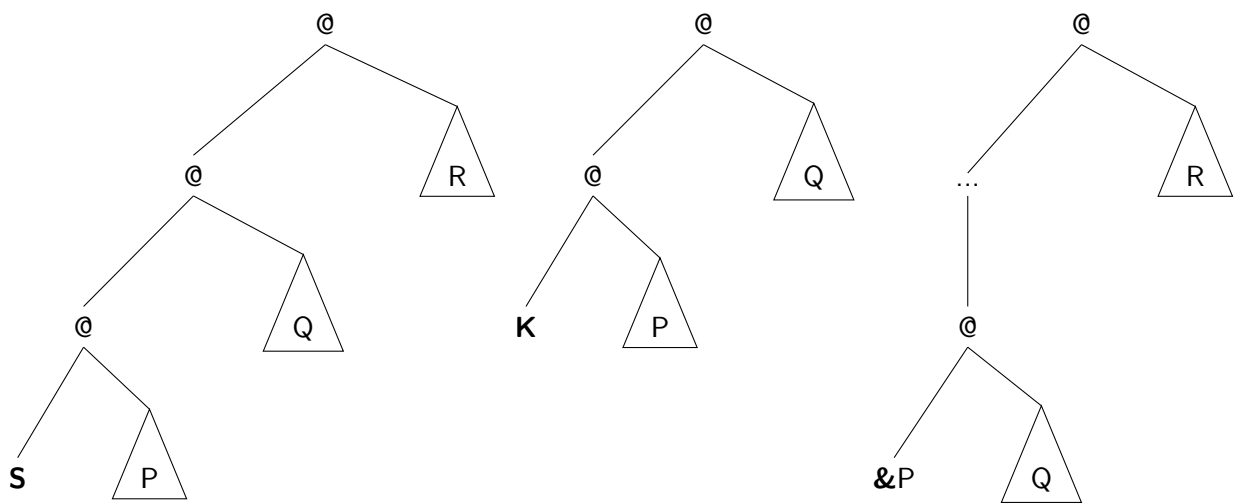
```
1 SK_Tree* skt_beta_redu(Arena arena, SK_Tree* root) {
2     if (root == NULL)
3         return NULL;
4
5     SK_Tree* expr = root;
6     size_t i = 0;
7     for (; i < MAX_BETA_REDUCTIONS; i++) {
8         // skt_print(&root, 1);
9
10        size_t depth = 0;
11        SK_Tree* leftmost = _skt_get_leftmost(expr, &depth);
12        assert(leftmost != NULL);
13
14        if (leftmost->type == K_NODE && depth ≥ 2) {
15            SK_Tree** sub_expr = &expr;
16            for (size_t j = 0; j + 2 < depth; sub_expr = &((*sub_expr)->left), j++);
17            *sub_expr = (*sub_expr)->left->right;
18            continue;
19        }
20        else if (leftmost->type == S_NODE && depth ≥ 3) {
21            SK_Tree* sub_expr = expr;
22            for (size_t j = 0; j + 3 < depth; sub_expr = sub_expr->left, j++);
23
24            sub_expr->left->left->left = sub_expr->left->left->right;
25
26            SK_Tree* ref = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
27            assert(ref != NULL);
28            *ref = (SK_Tree){ .type = REF_NODE, .left = sub_expr->right, .right = NULL, .ld_ident = NULL };
29
30            sub_expr->left->left->right = sub_expr->right;
31
32            SK_Tree* temp = sub_expr->left;
33            sub_expr->left = sub_expr->left->left;
34            temp->left = temp->right;
35            temp->right = ref;
36            sub_expr->right = temp;
37            continue;
38        }
39        else if (leftmost->type == REF_NODE) {
40            SK_Tree** sub_expr = &expr;
41            for (size_t j = 0; j < depth; sub_expr = &((*sub_expr)->left), j++);
42            *sub_expr = skt_copy(arena, leftmost->left);
43            assert(*sub_expr != NULL);
44            continue;
45        }
46
47        break;
48    }
49
50    // skt_print(&root, 1);
51    return expr;
52 }
```

Figure 3.1: Snapshot of the Beta Reduce Function

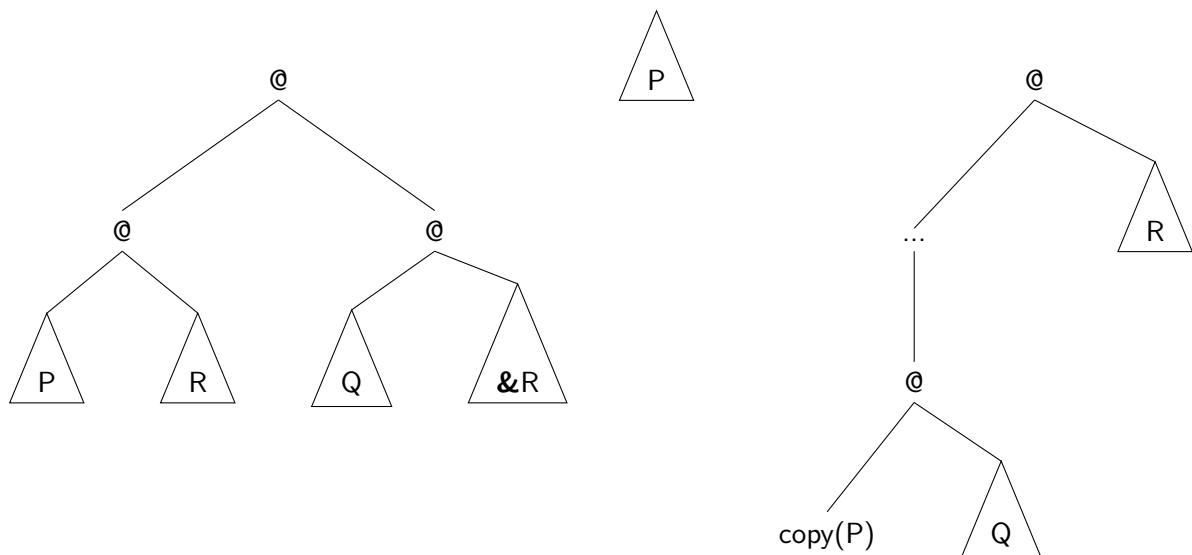
In order to not have to worry about memory, my implementation uses an arena which is a type of memory allocator which requests at runtime a specific amount of memory, specified by the programmer, in its initialization (mine is also implemented in a linked list structure in order to scale if needed - this gives flexibility for the parser and the interpreter part of the program, however, it may become inefficient for large programs which is not the case in this context).

After the SK Tree is produced, we can evaluate the resulting expression and try to reduce it. As some expressions cannot reach normal form as stated in Definition 6, a maximum is predefined in the program's macros (500, in this case).

There exist 3 possible cases: S, K or a & reference.



The resulting trees after the reduction should be then:



This is what is described in each case of the if statement block. If none of these cases are met, then the expression has reached normal form. This function does not guarantee that every SK expression reaches a normal form, but if the expression can reach normal form in under *MAX_BETA_REDUCTIONS*, it will.

```
interpreter.h
1 SK_Tree** ast_convert (Arena, AST*, HashTable);
```

Figure 3.2: Snapshot of the Convert Function Declaration

```
interpreter_priv.h
1 SK_Tree* _ast_expr_convert (Arena, ASTN_Expr*, HashTable, const char*);
```

Figure 3.3: Snapshot of the Private Convert Function Declaration

The second function, the most important, is fairly large to put it here, but the link to the Github Repository will be added at the end of this report. In a broader sense, this recursive function has 3 paths depending on the type of expression, application (EXPR_APP), abstraction (EXPR_ABS) and identifier (EXPR_IDENT).

```
interpreter.c
1 case EXPR_APP: {
2   SK_Tree* app = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
3   assert(app != NULL);
4
5   *app = (SK_Tree){
6     .type = APP_NODE,
7     .left = _ast_expr_convert(arena, expr->fields.app.left, table, filename),
8     .right = _ast_expr_convert(arena, expr->fields.app.right, table, filename)
9   };
10
11   return app;
12 }
```

Figure 3.4: Snapshot of the Convert Function Application Path

In this case, the translation is trivial, corresponding to the translation rule T_{CL} 6. It creates an application node and then recursively adds the left and the right trees.

```

1 case EXPR_IDENT: {
2   ASTN_Stmt* stmt = hashtable_lookup(table, expr->fields.var->token);
3
4   if (stmt != NULL) {
5     if (stmt->sk_expr == NULL) {
6       fprintf(
7         stderr,
8         "[SK CONVERTER]: sub expression was not defined previously to the current statement %s at line %d in file %s. Maybe you declared it later?\n",
9         expr->fields.var->token->str,
10        expr->fields.var->frow,
11        filename
12      );
13      _error_underline(filename, expr->fields.var->frow, expr->fields.var->fcol, expr->fields.var->ecol);
14    }
15
16    SK_Tree* ref = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
17    assert(ref != NULL);
18    *ref = (SK_Tree){ .type = REF_NODE, .left = stmt->sk_expr, .right = NULL, .ld_ident = stmt->var };
19
20    return ref;
21  }
22
23  SK_Tree* wrapper = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
24  assert(wrapper != NULL);
25
26  *wrapper = (SK_Tree){ .type = LD_NODE, .ld_ident = expr->fields.var, .left = NULL, .right = NULL };
27
28  return wrapper;
29 }

```

Figure 3.5: Snapshot of the Convert Function Identifier Path

This case is also trivial. The current expression is an identifier thus there exists two options: either this identifier is a free variable, hence it must be in the hashtable (otherwise we output an error), where a reference to the corresponding SK Tree is created to not occupy more memory; or this is a bounded one (a checking is performed prior in order to guarantee the consistency of the program) and, for this reason, we wrap it around a SK struct for it to be handled later.

```

1 case EXPR_ABS: {
2   ASTN_Ident* var = expr->fields.abs.vars;
3
4   if (expr->fields.abs.expr->type != EXPR_APP) {
5     if (expr->fields.abs.expr->type == EXPR_IDENT) {
6       if (strcmp(expr->fields.abs.expr->fields.var->token->str, var->token->str) == 0) {
7         SK_Tree* app1, *app2, *s, *k1, *k2;
8         app1 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
9         app2 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
10        s = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
11        k1 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
12        k2 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
13        assert(app1 != NULL && app2 != NULL && s != NULL && k1 != NULL && k2 != NULL);
14
15        *s = (SK_Tree){ .type = S_NODE, .left = NULL, .right = NULL, .ld_ident = NULL };
16        *k2 = (SK_Tree){ .type = K_NODE, .left = NULL, .right = NULL, .ld_ident = NULL };
17        *k1 = (SK_Tree){ .type = K_NODE, .left = NULL, .right = NULL, .ld_ident = NULL };
18
19        *app2 = (SK_Tree){ .type = APP_NODE, .left = s, .right = k2, .ld_ident = NULL };
20        *app1 = (SK_Tree){ .type = APP_NODE, .left = app2, .right = k1, .ld_ident = NULL };
21
22        return app1;
23      }
24
25      SK_Tree* k = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
26      assert(k != NULL);
27
28      ASTN_Stmt* stmt = hashtable_lookup(table, var->token);
29      if (stmt != NULL) {
30        if (stmt->sk_expr == NULL) {
31          fprintf(
32            stderr,
33            "[SK CONVERTER]: sub expression was not defined previously to the current statement %s at line %d in file %s. Maybe you declared it later?\n",
34            expr->fields.var->token->str,
35            expr->fields.var->frow,
36            filename
37          );
38          _error_underline(filename, expr->fields.var->frow, expr->fields.var->fcol, expr->fields.var->ecol);
39        }
40
41        SK_Tree* app = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
42        assert(app != NULL);
43
44        SK_Tree* ref = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
45        assert(ref != NULL);
46        *ref = (SK_Tree){ .type = REF_NODE, .left = stmt->sk_expr, .right = NULL, .ld_ident = stmt->var };
47
48        *app = (SK_Tree){ .type = APP_NODE, .left = k, .right = ref, .ld_ident = NULL };
49
50        return app;
51      }
52
53      *k = (SK_Tree){ .type = K_NODE, .left = NULL, .right = NULL, .ld_ident = NULL };
54      return k;
55    }
56  }

```

Figure 3.6: Snapshot of the Convert Function Abstraction Path - Sub-expression Identifier

Finally, the abstraction case as more depth to it. Firstly, we separate the cases of the sub-expression being an application or not. If it isn't, then if it is an identifier and it is the variable of abstraction, then we return the identity ($SK\ K$) - $T_{CL}\ 1$ -, otherwise we do the same thing as in the previous identifier expression case.

```

1  SK_Tree* sub_expr = _ast_expr_convert(arena, expr->fields.abs.expr, table, filename);
2  if (!_ast_in_free_var_set(expr->fields.abs.expr, var)) {
3      SK_Tree* app = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
4      assert(app != NULL);
5
6      SK_Tree* k = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
7      assert(k != NULL);
8
9      *k = (SK_Tree){ .type = K_NODE, .left = NULL, .right = NULL, .ld_ident = NULL };
10
11     *app = (SK_Tree){
12         .type = APP_NODE,
13         .left = k,
14         .right = sub_expr,
15         .ld_ident = NULL
16     };
17
18     return app;
19 }
20
21 return _ast_expr_convert_sk(arena, sub_expr, expr->fields.abs.vars);
22 }

```

Figure 3.7: Snapshot of the Convert Function Abstraction Path - Sub-expression Abstraction

If the sub-expression is an abstraction as well, then we convert it to SK. If the abstraction variable is not in the free variable set, this corresponds to rule $T_{CL}\ 2$, otherwise the variable exists in the sub-expression and we just convert it recursively to SK.

```

1  bool var_free_left = _ast_in_free_var_set(expr->fields.abs.expr->fields.app.left, var);
2  bool var_free_right = _ast_in_free_var_set(expr->fields.abs.expr->fields.app.right, var);
3
4  if (!var_free_left && !var_free_right) {
5      SK_Tree* k = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
6      assert(k != NULL);
7      *k = (SK_Tree){
8          .type = K_NODE,
9          .left = NULL,
10         .right = NULL,
11         .ld_ident = NULL
12     };
13
14     SK_Tree* app = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
15     assert(app != NULL);
16     *app = (SK_Tree){
17         .type = APP_NODE,
18         .left = k,
19         .right = _ast_expr_convert(arena, expr->fields.abs.expr, table, filename),
20         .ld_ident = NULL
21     };
22
23     return app;
24 }
25
26 if (!var_free_left && var_free_right && expr->fields.abs.expr->fields.app.right->type == EXPR_IDENT)
27     return _ast_expr_convert(arena, expr->fields.abs.expr->fields.app.left, table, filename);

```

Figure 3.8: Snapshot of the Convert Function Abstraction Path - Sub-expression Application 1

In case the sub-expression is an application MN , we start by checking if the variable is free in each. If it is not in both, then we just return $K(MN)$, by rule T_{CL} 2. Otherwise, if we have that it is not free on left and it is on the right with it being an identifier, then we have the case T_{CL} 3, so we just return the conversion of the left tree.

```

1  SK_Tree* app1 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
2  assert(app1 != NULL);
3
4  SK_Tree* app2 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
5  assert(app2 != NULL);
6
7  SK_Tree* s = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
8  assert(s != NULL);
9
10 if (!var_free_left) {
11     SK_Tree* k = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
12     assert(k != NULL);
13     *k = (SK_Tree){
14         .type = K_NODE,
15         .left = NULL,
16         .right = NULL,
17         .id_ident = NULL
18     };
19
20     SK_Tree* app3 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
21     assert(app3 != NULL);
22     *app3 = (SK_Tree){
23         .type = APP_NODE,
24         .left = k,
25         .right = _ast_expr_convert(arena, expr->fields.abs.expr->fields.app.left, table, filename),
26         .id_ident = NULL
27     };
28
29     *app2 = (SK_Tree){
30         .type = APP_NODE,
31         .left = s,
32         .right = app3,
33         .id_ident = NULL
34     };
35 } else {
36     ASTN_Expr* sub_expr = astn_copy_expr(arena, expr->fields.abs.expr->fields.app.left);
37     assert(sub_expr != NULL);
38
39     ASTN_Expr* left = (ASTN_Expr*)arena_alloc(arena, sizeof(struct astn_expr));
40     assert(left != NULL);
41     *left = (ASTN_Expr){
42         .frow = expr->frow,
43         .fcol = expr->fcol,
44         .erow = expr->erow,
45         .ecol = expr->ecol,
46         .type = EXPR_ABS,
47         .fields.abs.vars = astn_copy_ident(arena, expr->fields.abs.vars),
48         .fields.abs.expr = sub_expr
49     };
50
51     *app2 = (SK_Tree){
52         .type = APP_NODE,
53         .left = s,
54         .right = _ast_expr_convert(arena, left, table, filename),
55         .id_ident = NULL
56     };
57 }
58
59 if (!var_free_right) {
60     SK_Tree* k = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
61     assert(k != NULL);
62     *k = (SK_Tree){
63         .type = K_NODE,
64         .left = NULL,
65         .right = NULL,
66         .id_ident = NULL
67     };
68
69     SK_Tree* app3 = (SK_Tree*)arena_alloc(arena, sizeof(struct sk_tree));
70     assert(app3 != NULL);
71     *app3 = (SK_Tree){
72         .type = APP_NODE,
73         .left = k,
74         .right = _ast_expr_convert(arena, expr->fields.abs.expr->fields.app.right, table, filename),
75         .id_ident = NULL
76     };
77
78     *app1 = (SK_Tree){
79         .type = APP_NODE,
80         .left = app2,
81         .right = app3,
82         .id_ident = NULL
83     };
84 } else {
85     ASTN_Expr* sub_expr = astn_copy_expr(arena, expr->fields.abs.expr->fields.app.right);
86     assert(sub_expr != NULL);
87
88     ASTN_Expr* right = (ASTN_Expr*)arena_alloc(arena, sizeof(struct astn_expr));
89     assert(right != NULL);
90     *right = (ASTN_Expr){
91         .frow = expr->frow,
92         .fcol = expr->fcol,
93         .erow = expr->erow,
94         .ecol = expr->ecol,
95         .type = EXPR_ABS,
96         .fields.abs.vars = astn_copy_ident(arena, expr->fields.abs.vars),
97         .fields.abs.expr = sub_expr
98     };
99
100    *app1 = (SK_Tree){
101        .type = APP_NODE,
102        .left = app2,
103        .right = _ast_expr_convert(arena, right, table, filename),
104        .id_ident = NULL
105    };
106 }
107
108 return app1;
109 }

```

Figure 3.9: Snapshot of the Convert Function Abstraction Path - Sub-expression Application 2

We end now with the last cases corresponding to the rules T_{CL} 4, 5 and 6, which is what is represented in the code.

Chapter 4

Discussion and Conclusion

The interpreter developed demonstrated efficiency in terms of memory and time for the conversion and reduction of lambda expressions, though further testing on larger programs would be necessary to fully ascertain scalability. The use of an arena allocator for memory management proved effective in simplifying memory concerns for the interpreter and parser parts of the program, offering flexibility to scale as needed, particularly within the context of typical program sizes for this project, although I maintain my position that it the interpreter might need changes to handle larger and more repeated programs. The chosen data structures for both Lambda Calculus expressions and SK tree representation, including the `ast` and `sk_tree` structs, proved flexible and facilitate easy modification for processing more abstract and larger programs.

The beta reduction function, relying on the logic of reduction by the leftmost branch, behaved robustly for a variety of distinct expressions. This reliable behavior is fundamentally supported by the Church-Rosser Theorem, which guarantees that despite different reduction paths, the system will always converge to a unique normal form, if one exists. The translation process, governed by a set of recursive rules (T_{CL}), is inherently more complex due to its multi-case nature, differentiating based on variable freeness and expression type. This complexity is a necessary aspect of converting Lambda Calculus's variable binding into the simpler, variable-free combinatory logic. Furthermore, the interpreter directly addresses the critical problem of handling free variables, ensuring defined computational behavior by allowing programmers to set multiple variables to different lambda expressions, which piece by piece build the intended program, ultimately leading to SK tree references for free variables.

For future work, some key points to analyze would be: how to implement these SK data structures and process them in smarter ways in order to take advantage of modern hardware, given that this type of program interpretation is no longer as widely used; and how to create SK files for a Java-like virtual machine that are small in size, countering a common problem this representation has in terms of the growth of expressions compared to their lambda calculus equivalents. Additionally, exploring the techniques implemented for compression of output files could be further optimized to maximize storage efficiency.

In conclusion, this project successfully designed and implemented an SK Combinator Interpreter that effectively translates and reduces Lambda Calculus expressions. By leveraging the foundational principles of Combinatorial Logic and addressing practical considerations like memory management and free variable handling, the interpreter provides a simple framework for understanding and working with these computational models. While the inherent complexities of translation and potential growth of SK expressions present avenues for future optimization, the current implementation serves as a foundation for further research into efficient program representation and interpretation within the realm of functional programming.

References

- [1] H. Teixeira, "SK-combinators-interpreter," GitHub. [Online]. Available: <https://github.com/UShouldRun/SK-combinators-interpreter>.
- [2] Doisinkidney, " λ , S, K, I," doisinkidney.com. Blog post. [Online]. Available: <https://doisinkidney.com/posts/2020-10-17-ski.html>. Published: Oct. 17, 2020. Accessed on: May, 2025.
- [3] I. Zerny, "On graph rewriting, reduction, and evaluation in the presence of cycles," *Higher-Order Symbolic Computation*, vol. 26, pp. 63–84, 2013.
- [4] L. C. Paulson, "Foundations of Functional Programming," Computer Laboratory, University of Cambridge, UK, Course Notes, Jun. 2000.
- [5] H. Barendregt, "The Impact Of The Lambda Calculus In Logic And Computer Science," *The Bulletin of Symbolic Logic*, vol. 3, no. 2, pp. 181–215, Jun. 1997.