Aufgaben zum Thema Compiler

Aufgabe 1

Die Programmiersprache Unlambda ist eine auf dem SKI-Kalkül basierende Programmiersprache. Ihre grundlegenden Bausteine sind die bekannten Kombinatoren
S = λx. λy. λz. x z (y z), K = λx. λy. x und I = λx. x.
Die "Klammerung" wird durch den Backtick ‘ erreicht. Dieser steht für eine einfache Funktionsanwendung und wird in Präfixnotation verwendet. Beispielsweise bedeutet ‘SK, dass S auf K angewendet wird. Alle Funktionen haben (formal) genau einen Parameter, Funktionen mit mehr Parametern (wie S und K) werden mit Currying realisiert. ‘‘KSI wendet K auf S an und das Ergebnis davon auf I (entspricht (K S I) und ergibt daher S), ‘K‘SI hingegen entspricht (K (S I)).
Weitere Konstrukte aus Unlambda werden hier nicht betrachtet. Zusätzlich wird noch eine Art "Variablen" eingeführt (geschrieben als ein in eckige Klammern eingeschlossener nichtnegativer Integer, z.B. "[42]"). Diese werden nicht ausgewertet und können als Platzhalter für Parameter verwendet werden.
In dieser Aufgabe soll ein Compiler für diese Unlambda-ähnliche Sprache geschrieben werden. Das Grundgerüst steht im Ordner "Compiler-A1" zur Verfügung. Sie können zum Übersetzen des ganzen Projekts das Makefile verwenden.

1. Ergänzen Sie den mit Flex geschrieben Lexer (lexer.l) um folgende Token-Typen:

   - **TOKEN_S** das Zeichen ’S’
   - **TOKEN_K** das Zeichen ’K’
   - **TOKEN_I** das Zeichen ’I’
   - **LB** das Zeichen ’[’
   - **RB** das Zeichen ’]’
   - **BACKTICK** das Zeichen ’‘’
   - **VARIABLE** ein int, bestehend aus einer nichtleeren Ziffernfolge. Hier soll der Wert des ints im Token gespeichert werden.

   Hinweise:

   - In Flex ist der aktuelle String in der Variable yytext gespeichert.
   - Das Parsen eines Integers ist mit "sscanf(string, "%d", &var)" möglich.

2. Ergänzen Sie die Datei "parser.c" um die nötigen Parse-Funktionen und den initialen Aufruf des Parsers (in der main-Methode). Knoten des Baumes werden durch die struct "node" beschrieben. Der Typ des Knotens wird durch das Feld "type" beschrieben. Die möglichen Werte sind "S", "K" und "I" für die entsprechenden Blattknoten, "APP" für eine Funktionsanwendung (wobei in left und right die beiden Unterbäume gespeichert werden) und "VAR" für eine Variable (hier wird der Wert in var_name gespeichert).

Die Grammatik sieht wie folgt aus (Terminale sind unterstrichen):
R → P NEWLINE
P → TOKEN_S | TOKEN_K | TOKEN_I | BACKTICK P P | LB VARIABLE RB
Bauen Sie damit den Syntaxbaum auf. Beachten Sie, dass Sie den Speicher für die Knoten mit malloc allokieren müssen. Sie müssen sich nicht um das freigeben des Speichers kümmern.
"yylex()" liefert das nächste Token. Brechen Sie bei Parserfehlern mit "error()" ab.
Als Kontrolle gibt das Programm den von Ihnen generierten Syntaxbaum mit gewohnter Klammerschreibweise aus (d.h. wenn Sie z.B. das Programm ‘S‘KI einlesen, sollte (S(KI)) ausgegeben werden).

3. Zusatzaufgabe (ohne Lösung): Implementieren Sie einen Interpreter für diese Sprache. Schreiben Sie dazu eine Funktion, die einen Auswertungsschritt durchführt. In einem Schritt wird die linkeste Funktion, die ausreichend Parameter hat (K bspw. benötigt zwei Parameter), ausgewertet und der neue Baum zurückgegeben. Ist dies nicht möglich, soll NULL zurückgegeben werden. Schreiben Sie darauf aufbauend eine Funktion, die ein Programm schrittweise auswertet, bis keine solche Reduktion mehr möglich ist.