CS301 Compiler Design and Implementation **Handout # 13**
Prof. Lyn Turbak **September 23, 2003**
Wellesley College

S-Expressions, Parsing, and Unparsing

1 Concrete Syntax

The presentation of INTEX described an abstract syntax for the language based on nodes for abstract syntax trees (ASTs). We saw that it was straightforward to express such trees using OCAML datatype constructors. These give us one concrete syntax for writing down INTEX programs, but it is not particularly concise or convenient!

For example, consider the OCAML encoding of the Fahrenheit-to-Celsius converter:

```
Pgm(1, BinApp(Div,
              BinApp(Mul,
                     BinApp(Sub,Arg(1),Lit(32)),
                     Lit(5)),
              Lit(9)))
```

A far more concise concrete notation for this example might be something like `1:($1-32)*5/9`. Here the `1:` is a header for a program with one argument, `$1` denotes a reference to this argument, and arithmetic is expressed via standard infix notation and the usual associativity and precedence of operators (with explicit parentheses used to override the defaults).

In practice, these concise notations are very convenient, but to use them we must solve a non-trivial **parsing problem**: how do we automatically derive ASTs for these sorts of notations? Later this semester will study techniques for solving the parsing problem. However, the techniques are somewhat complex and it takes a fair bit of time to understand their theoretical underpinnings and learn the software tools associated with them. This would be a big detour in our goal of completing a simple compiler within a few weeks.

What we'd like is a compromise: a notation that is reasonably concise but is so easy to convert to an AST that we can learn the relevant techniques in a day. Fortunately, there is a simple and standard notation that fits this goal nicely: s-expressions.

2 S-Expressions

2.1 Overview

A **symbolic expression** (**s-expression** for short) is a simple notation for representing tree structures using linear text strings containing matched pairs of parentheses. Each leaf of a tree is a **symbolic tokens**, which (to first approximation) is any sequence of characters that does not contain a left parenthesis (`(`), a right parenthesis (`)`), or a whitespace character (space, tab, newline, etc.).[1] Examples of symbolic tokens include `x`, `this-is-a-token`, `anotherKindOfToken`, `17`, `3.14159`, `4/3*pi*r^2`, `a.b[$2]%3`, `'Q'`, and `"a (string) token"`. A node in a tree is represented by a pair of parentheses surrounding zero or more s-expressions that represent the node's subtrees. For example, the s-expression

```
((this is) an ((example) (s-expression tree)))
```

designates the structure depicted in Fig. 1. Whitespace is necessary for separating symbolic tokens that appear next to each other, but can be used liberally to enhance (or obscure!) the readability of the structure. Thus, the above s-expression could also be written as

[1] But as we shall see, string and character literals *can* contain parentheses and whitespace characters.

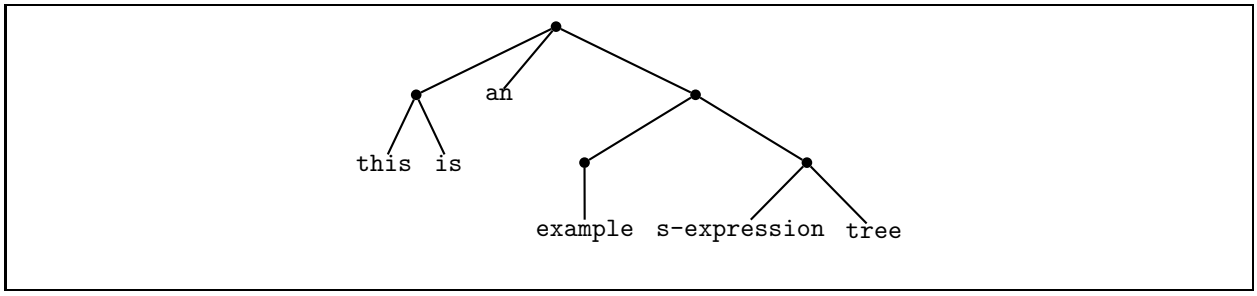

Figure 1: Viewing `((this is) an ((example) (s-expression tree)))` as a tree.

```
((this is)
 an
 ((example)
  (s-expression
   tree)))
```

or (less readably) as

```
                (
     (           this
is) an  (    (          example
           ) (
s-expression tree        )
  )
          )
```

without changing the structure of the tree.

S-expressions were pioneered in LISP as a notation for data as well as programs (which we have seen are just particular kinds of tree-shaped data!). We shall see that s-expressions are an exceptionally simple and elegant way of solving the parsing problem.[2] For this reason, the mini-languages we study will (at least initially) have a concrete syntax based on s-expressions.

The fact that LISP dialects (including SCHEME) have a built-in primitive for parsing s-expressions (`read`) and treating them as literals (`quote`) makes them particularly good for manipulating programs (in any language) written with s-expressions. It is not quite as convenient to manipulate s-expression program syntax in other languages, such as OCAML, but we shall see that it is still far easier than solving the parsing problem for more general notations.

2.2 S-Expression Representations of Sum-of-Product Trees

The abstract syntax trees we are trying to model (such as the one in Fig. 2 don't quite have the tree structure for s-expressions depicted above. In particular, ASTs are **sum-of-product trees** in which each node is labeled with a tag indicating the summand represented by the node, while the simplest way of interpreting s-expressions involves label-less nodes.

But this difference is easy to address. To model sum-of-product trees with s-expressions, we adopt the simple **prefix convention** in which the first s-expression in a parenthesized sequence is a token that is the summand node label and the remaining s-expressions are arbitrary s-expressions that represent the product components. For example, using this convention with our conversion program yields the following s-expression:

[2]There are detractors who hate s-expressions and claim that LISP stands for **L**ots of **I**rritating **S**illy **P**arenthesis. Apparently such people lack a critical aesthetic gene that prevents them from appreciating beautiful designs. Strangely, many such people seem to prefer the far more verbose encoding of trees in XML notation discussed later. Go figure!

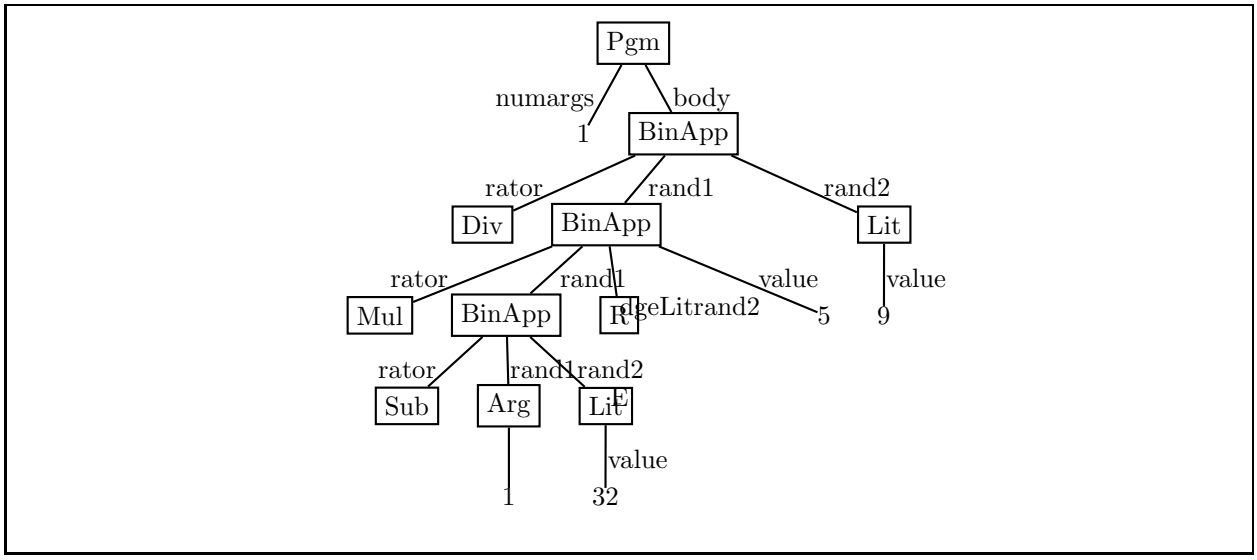

Figure 2: AST for the Fahrenheit-to-Celsius converter.

```
(pgm 1
     (binapp div
             (binapp mul
                     (binapp sub (arg 1) (lit 32))
                     (lit 5))
             (lit 9)))
```

This is still more verbose than we'd like, so we'll use some tricks to make the notation more concise/readable:

- If we assume that numbers stand for themselves, we can avoid the explicit `lit` tag. Thus, we will shorten `(lit 32)` to `32`.
- Since we must be able to distinguish argument references from integer literals, we *cannot* similarly shorten `(arg 1)` to `1`. But we can use a shorter tag name, such as `$`, in which case `(arg 1)` becomes `($ 1)`.
- The only non-leaf node is a binary application, so we can dispense with the `binapp` tag without introducing ambiguity. Using traditional operator symbols (`+`, `-`, `*`, `/`, `%`) in place of names (`add`, `sub`, `mul`, `div`, `rem`) further shortens the notation. For example, `(binapp sub ($ 1) 32)` becomes `(- ($ 1) 32)`.
- To distinguish INTEX program from other programs in other mini-languages we will study, we replace `pgm` by `intex`. This is not shorter, but helps to disambiguate programs from different languages.

The result of applying all of the above tricks is

```
(intex 1 (/ (* (- ($ 1) 32) 5) 9),
```

which is significantly shorter that the OCAML notation or the unoptimized s-expression notation. This is the s-expression notation that we will adopt for INTEX. We will make similar abbreviations in other languages. Note that the syntax of LISP dialects is effectively determined by this process

– prefix tags are used everywhere except for literals (e.g., numbers, booleans, strings, characters) and for applications (which are written without an explicit `apply` tag, as in `(fact 5)` rather than `(apply fact 5)`).

It is worth noting that there are other common notations for representing sum-of-product trees. The most popular of these are the HTML and XML document description languages. In these languages, summand tags appear in begin/end markups and product components are encoded both in the association lists of markups as well as in components nested within the begin/end markups. For instance, Fig. 3 shows how the Fahrenheit-to-Celsius expression might be encoded in XML. The reader is left to ponder why XML, which at one level is a verbose encoding of s-expressions, is a far more popular standard for expressing structured data than s-expressions.

```
<arithop>
  <op name="/"/>
  <rand1>
    <arithop>
      <op name="*"/>
      <rand1>
        <lit num=5/>
      </rand1>
      <rand2>
        <arithop>
          <op name="-"/>
          <rand1>
            <arg index=1/>
          </rand1>
          <rand2>
            <lit num=32/>
          </rand2>
        </arithop>
      </rand2>
    </arithop>
  </rand1>
  <rand2>
    <lit num=9/>
  </rand2>
</arithop>
```

Figure 3: The Fahrenheit-to-Celsius expression in XML notation.

2.3 Representing S-Expressions in OCaml

As with any other kind of tree-shaped data, s-expressions can be represented in OCaml as values of an appropriate datatype. The OCaml datatype representing s-expression trees is presented in Fig. 4.

Recall that the leaves of s-expression trees are symbolic tokens. There are five kinds of symbolic tokens, distinguished by type:

1. integer literals (constructed via `Int`);
2. floating point literals (constructed via `Flt`);
3. string literals (constructed via `Str`);

```
type sexp =
        Int of int
      | Flt of float
      | Str of string
      | Chr of char
      | Sym of string
      | Seq of sexp list
```

Figure 4: OCaml s-expression datatype.

4. character literals (constructed via `Chr`); and

5. symbols (i.e. name tokens, constructed via `Sym`);

The nodes of s-expression trees are represented via the `Seq` constructor, whose `sexp list` argument denotes any number of s-expression subtrees.

For example, the s-expression given by the concrete notation

```
(stuff (17 3.14159) ("foo" 'c' bar))
```

would be expressed in OCaml as:

```
Seq [Sym("stuff");
     Seq [Int(17); Flt(3.14159)];
     Seq [Str("foo"); Chr('c'); Sym("bar")]]
```

As another example, the s-expression notation for the Fahrenheit-to-Celsius Intex program,

```
(intex 1 (/ (* (- ($ 1) 32) 5) 9),
```

would be expressed in OCaml as:

```
Seq [Sym("intex");
     Int(1);
     Seq [Sym("/");
          Seq [Sym("*");
               Seq [Sym("-");
                    Seq [Sym("$"); Int(1)]
                    Int(32)]
               Int(5)]
          Int(9)]
```

3 Converting Between S-Expressions and Intex

In the context of studying programming languages, the main benefit of s-expressions is that they are a generic tree dataype that simplifies the processes of converting concrete program strings to trees (**parsing**) and converting trees to strings (**unparsing**). As an example, we shall consider the parsing and unparsing of Intex programs. We shall assume that the concrete syntax for Intex programs is the optimized s-expression sum-of-products syntax discussed in Sec. 2.2.

3.1 Parsing Intex Programs

An Intex program expressed as a string of characters written in concrete parenthesized notation can be automatically parsed into an OCaml `pgm/exp` tree via a two stage process:

1. Parse the string into an OCaml `sexp` tree; and

2. Translate the `sexp` tree into a `pgm`/`exp` tree.

Breaking the problem into two stages is a good idea because the solution to the first problem can be reused whenever we adopt s-expression-based notations for the programming languages we study. Moreover, the second stage is considerably easier than the first because it is much simpler to translate from one tree structure to another than to deduce a tree structure from a linear string notation.

For simplicity, we will assume for now that we are given the following two black-box functions for converting between strings and s-expressions:

> `val string2Sexp : string -> Sexp.sexp`
> Parses the given string into an s-expression. Raises an `Sexp.SyntaxError` exception if the given string cannot be interpreted as an s-expression.
>
> `val sexp2String : Sexp.sexp -> string`
> Returns a string representation of the given s-expression.

`string2Sexp` implements the first stage of the parsing process, while `sexp2String` is used for parsing errors and an unparsing. We shall see how to write these functions in Sec. 4.

An implementation of the second stage for Intex parsing is shown in Fig. 5. The `sexp2Pgm` and `sexp2Exp` functions convert `sexp` trees into `pgm` and `exp` trees, respectively. These functions rely heavily on OCaml pattern matching to determine syntactic well-formedness and distinguish expression types. Composing these functions with `string2Sexp` yields functions (`string2Pgm` and `string2Exp`) that parse Intex programs and expressions from strings. For example:

```
# string2Pgm "(intex 2 (/ (+ ($ 1) ($ 2)) 2))";;
- : Intex.pgm = Pgm (2, BinApp (Div, BinApp (Add, Arg 1, Arg 2), Lit 2))
```

3.2 Unparsing Intex Programs

As with parsing, unparsing Intex programs is a two stage process that involves s-expressions as an intermediate form.

1. Translate the `pgm`/`exp` tree into an `sexp` tree;

2. Unparse the `sexp` tree into a string.

Again, the modularity of this approach simplifies the process since the s-expression unparser is reusable.

Using this approach, we can unparse Intex programs and expressions via the `pgm2Sexp` and `exp2Sexp` functions presented in Fig. 6. Composing these with `sexp2String` yields unparsers that convert Intex programs and expressions to trees. For example:

```
# pgm2String (Pgm (2, BinApp (Div, BinApp (Add, Arg 1, Arg 2), Lit 2)));;
- : string = "(intex 2 (/ (+ ($ 1) ($ 2)) 2))"
```

```
exception SyntaxError of string

let rec sexp2Pgm sexp =
  match sexp with
    Sexp.Seq [Sexp.Sym("intex");
              Sexp.Int(n);
              body] ->
      Pgm(n, sexp2Exp body)
  | _ -> raise (SyntaxError ("invalid Intex program: "
                             ^ (sexp2String sexp)))

and sexp2Exp sexp =
  match sexp with
    Int i -> Lit i
  | Seq([Sym "$"; Int i]) -> Arg i
  | Seq([Sym p; rand1; rand2]) ->
      BinApp(string2Primop p, sexp2Exp rand1, sexp2Exp rand2)
  | _ ->  raise (SyntaxError ("invalid Intex expression: "
                              ^ (sexp2String sexp)))

and string2Primop s =
  match s with
    "+" -> Add
  | "-" -> Sub
  | "*" -> Mul
  | "/" -> Div
  | "%" -> Rem
  |  _ -> raise (SyntaxError ("invalid Intex primop: " ^ s))

and string2Exp s = sexp2Exp (string2Sexp s)
and string2Pgm s = sexp2Pgm (string2Sexp s)
```

Figure 5: Functions for parsing INTEX programs and expressions from s-expressions.

```
let rec pgm2Sexp p =
    match p with
      Pgm (n, e) ->
        Seq ([Sym "intex"; Int n; exp2Sexp e])

  and exp2Sexp e =
    match e with
      Lit i -> Int i
    | Arg i -> Seq [Sym "$"; Int i]
    | BinApp (rator, rand1, rand2) ->
        Seq ([Sym (primop2String rator); exp2Sexp rand1; exp2Sexp rand2])

  and primop2String p =
    match p with
      Add -> "+"
    | Sub -> "-"
    | Mul -> "*"
    | Div -> "/"
    | Rem -> "%"

  and exp2String s = sexp2String (exp2Sexp s)
  and pgm2String s = sexp2String (pgm2Sexp s)
```

Figure 6: Functions for unparsing INTEX programs and expressions to s-expressions.

4 Converting Between Strings and S-Expressions

We conclude our discussion of s-expressions by showing how to convert between parenthesized character strings and OCAML `sexp` trees.

4.1 Lexical Conventions

In order to parse s-expression strings, we must first specify the **lexical conventions** for such strings – i.e., the nitty-gritty details governing how they are decomposed into tokens that are parsed into trees.

As explained in Sec. 2.3, there are five kinds of symbolic tokens. Here are the specifications for these tokens:

- *integer tokens*: a non-empty sequence of digits preceded by an optional `+` or `-` sign. E.g. `496`, `+17`, `-273`, `+0`, `-0`;
- *float tokens*: a non-empty sequence of digits containing exactly one `.` and preceded by an optional `+` or `-` sign. E.g. `17.`, `-273.15`, `0.123`, `.123`, `3.14159`;
- *string tokens*: any sequence of characters delimited by a pair of double quotes that does not containing a "raw" double quote character but possibly contains any number of *escape sequences* of the form: `\t` (tab), `\n` (newline), `\r` (carriage return), `\b` (backspace), `\'` (single quote), `\"` (double quote), and `\\` (backslash). E.g. `"A simple string."`, `"A string\\nwith many\t \"es`
- *character tokens*: a single character (including an escape sequence but not including a "raw" single quote), delimited by a pair of single quotes. E.g. `'p'`, `'Q'`, `'\n'`, `'\''`, `'\"'`, `'\\'`;

- *symbol tokens*: any sequence of characters that (1) cannot be interpreted as one of the other kinds of tokens, (2) does not contain any of the following characters: open paren, close paren, open squiggly brace, or any whitespace character (space, tab, newline, carriage return, or backspace). E.g. `x`, `this-is-a-token`, `anotherKindOfToken`, `4/3*pi*r^2`, `a.b[$2]%3`, `symbol-'with'-"quotes"`, `xs'`, `xs''`

Open parens ('(') and close parens (')') are always distinct tokens except when then appear inside comments or string or character tokens. In particular, a symbol token cannot contain a paren. Parens, squiggly-brace delimited comments (see below), and one or more whitespace characters may be used to separate symbolic tokens. For example, all of the following s-expressions are treated exactly the same:

```
(foo (bar baz) quux)

(foo(bar baz)quux)

       ( foo
    (        bar
  baz            )
quux               )

(foocomment 1(barcomment 2baz)comment 3quux)
```

In our concrete s-expression notation, we allow **block comments** that are delimited by squiggly braces ('' and ''). These braces and any characters appearing between them are ignored by the parsing process. For example, the s-expression

```
(postfix 2 Given args a and b,
            calculate 2a - b
   1 get Push a 2 mul Replace a by 2a on top of stack
   1 put Replace a by 2a on bottom of stack
   sub Calculate 2a - b
   )
```

is treated as if it were written

```
(postfix 2 1 get 2 mul 1 put sub)
```

Like any decent language designers, we allow comments that properly nest. For example,

```
(postfix 2
  0 get push a oops, this is wrong!
  1 get push a correctly 2 mul 1 put sub)
```

Surprisingly, many so-called modern programming languages (including C and Java) do not support proper nesting of block comments!

Another possible kind of comment is the **line comment** in which all characters between a special token and the end of line are treated as comments. For example, `//` is the line comment token in C/Java and `;` is the line comment token in Scheme. We do not support line comments in our s-expression notation, but it could easily be extended to do so.

4.2 Parsing S-Expressions

The process of determining the tree structure implied by a linear string of characters is typically decomposed into two stages:

1. decomposing the string into a sequence of tokens (a process known as **scanning**, **lexing**, or **tokenizing**).

2. determining the tree structure implied by the token sequence (a process known as **parsing**).

We shall use this two-stage approach for processing s-expression strings.

The following OCaml datatype represents the five kinds of tokens that will be used to communicate between the scanner and the parser.

```
type token =
| ATOM of string
| STRING of string
| CHAR of char
| LPAREN
| RPAREN
```

The `ATOM` constructor is used to represent integer, floating point, and symbol tokens. The other two kinds of symbolic tokens, strings and characters, are handled by the `STRING` and `CHAR` constructors. Parens are represented via the `LPAREN` and `RPAREN` constructors. No constructors are associated with comments because comments are ignored by the scanning process.

4.2.1 Scanning S-Expressions

In this section, we show how to scan an s-expression string into tokens. In the case of s-expressions, this process is easy enough that we can write it directly "by hand". But writing such scanners by hand is usually complex, tedious, and error-prone. Later this semester, we will study tools that automate the process of writing scanners from high-level token descriptions. Such tools make it possible to easily create scanners. Moreover, the resulting scanners are more likely to be correct and efficient than the ones we write by hand!

Our scanner will be implemented in the form of the following function:

> `val string2Tokens: string -> token list`
> Returns the list of s-expression tokens scanned from the given string. Signals an `IllFormedSexp` exception if the lexical conventions are violated.

A first cut at an implementation of `string2Tokens` is presented in Fig. 7. This version handles only parenthesized strings of atomic (integer, float, and symbol) tokens. Below, we will extend this version to handle comments, string literals, and character literals.

The scanner walks over the given string, character by character, by incremening a string index `i` into the given string `s`. Whitespace is ignored and open and close parens are turned into tokens. Whenever the initial character of an atomic token is discovered, the scanner enters "atomic token mode" by calling the `scanSymbol` function, which determines the index of the last character of the symbol. The fact that OCaml functions are tail recursive and nested (here `scanSymbol` is defined within `string2Tokens` makes them perfect for encoding complex iterations as mutually recursive tail recursive functions. The scanning process in Fig. 7 is trickier to express as a traditional loop because it is necessary to keep track of the mode the process is in.

```
exception IllFormedSexp of string

(* Note: string2Tokens is a very compelling example of block structure --
   e.g. defininig local functions inside another function definition. *)
let string2Tokens s =
  let len = String.length s in
  let rec scanTokens i =
    if i >= len then
      []
    else match String.get s i with
      ' ' | '\t' | '\n' | '\r' | '\b' -> scanTokens(i+1) (* ignore whitespace *)
    |        '(' -> (LPAREN :: scanTokens(i+1))
    | ')' -> (RPAREN :: scanTokens(i+1))
    |        '\'' -> raise (IllFormedSexp "don't handle character literals yet")
    |        '\"' -> raise (IllFormedSexp "don't handle string literals yet")
    |        '{' -> raise (IllFormedSexp "don't handle comments yet")
    | _ -> scanSymbol i (i+1) (* get a symbol *)

  and scanSymbol start k =
    if k >= len then
      [ATOM(String.sub s start (k-start))]
    else let c = String.get s k
         in if List.memq c [' ';'\t';'\n';'\r';'\b';'(';')';'{'] then
              (ATOM(String.sub s start (k-start))::scanTokens(k))
            else
              scanSymbol start (k+1)

  in scanTokens 0
```

Figure 7: A first cut at scanning an s-expression string into tokens.

We can handle character literals by changing the single-quote dispatch within `string2Tokens` to be

```
| '\'' -> scanChar (i+1) (* start of char *)
```

and by adding the `scanChar` and `escaped` functions defined in Fig. 8.

```
and scanChar k =
  if k >= len then
    raise (IllFormedSexp "Sexp: input ended before end of char literal")
  else
    let c = String.get s k in
      if (c = '\') then (* begin escape sequence *)
        if k+2 >= len then
          raise (IllFormedSexp "Sexp: input ended before end of char literal")
        else if (String.get s (k+2)) = '´' then
          CHAR(escaped(String.get s (k+1)))::scanTokens(k+3)
        else
          raise (IllFormedSexp
                 ("Sexp: ill-formed char literal: "
                  ^ "'\" ^ (String.sub s (k+1) 2)))
      else if k+1 >= len then
        raise (IllFormedSexp "Sexp: input ended before end of char literal")
      else if (String.get s (k+1)) = '´' then
        CHAR(c)::scanTokens(k+2)
      else
        raise (IllFormedSexp
               ("Sexp: ill-formed char literal: "
                ^ "'" ^ (String.sub s k 2)))

and escaped c = (* convert escaped char to special char *)
  match c with
    't' -> '\t'
  | 'n' -> '\n'
  | 'r' -> '\r'
  | 'b' -> '\b'
  | '\\' -> '\\'
  | '\"' -> '\"'
  | '\'' -> '\''
  |  _ -> raise (IllFormedSexp
                 ("Sexp: unrecognized escape sequence:"
                   ^  "\\" ^ (String.make 1 c)))
```

Figure 8: Handling character literals within `string2Tokens`.

String literals are handled by changing the double-quote dispatch in `string2Tokens` to:

```
        | '"' -> scanString (i+1) (i+1) [] (* start of string *)
```

and by adding the `scanString` function defined in Fig. 9.

```
and scanString start k revChars =
      (* Look for end of strings quotes. Handle escapes along the way.
         revChars is reversed list of chars seen so far *)
      if k >= len then
        raise (IllFormedSexp
               ("Sexp: input ended before end of string:\n"
                ^ (StringUtils.implode(List.rev revChars))))
      else
        let c = String.get s k
         in if c = '"' then (* close double-quote ending string *)
              STRING(StringUtils.implode(List.rev revChars))::scanTokens(k+1)
            else if c = '\' then (* begin escape sequence *)
              if (k+1) >= len then
                raise (IllFormedSexp
                       ("Sexp: input ended before end of string:\n"
                        ^ (StringUtils.implode(List.rev (c::revChars)))))
              else
                scanString start (k+2) (escaped(String.get s (k+1))::revChars)
            else (* continue reading string *)
              scanString start (k+1) (c::revChars)
```

Figure 9: Handling string literals within `string2Tokens`.

Finally, for block comments we change the open-squiggly dispatch in `string2Tokens` to:

```
        | '{' -> scanBlockComment (i+1) 0 (* beginning of block comment *)
```

and add the `scanBlockComment` function defined in Fig. 10.

4.2.2 Parsing S-Expressions Tokens

Once we have a list of s-expression tokens, we arrange them into an `sexp` tree, as shown in Fig. 11.

4.3 Unparsing S-Expressions

Figures 12– **??** give code for unparsing s-expressions into strings.

```
and scanBlockComment k nestLevel =
      (* Ignore characters in comments. Handle nesting levels
         appropriately to avoid summary execution by lyn. *)
      if k >= len then
        raise (IllFormedSexp
               "Sexp: input ended before end of block comment")
      else
        let c = String.get s k
         in if c = '}' then
              if nestLevel = 0 then
                scanTokens (k+1)
              else
                scanBlockComment (k+1) (nestLevel - 1)
            else if c = '{' then
              scanBlockComment (k+1) (nestLevel + 1)
            else
              scanBlockComment (k+1) nestLevel
```

Figure 10: Handling block comments within `string2Tokens`.

```
(* fromToks : token list -> (sexp * token list) *)
  let rec fromToks toks =
    match toks with
      [] -> raise (IllFormedSexp "Sexp: no tokens")
    | (STRING(s)::ts) -> (Str(s), ts)
    | (CHAR(c)::ts) -> (Chr(c), ts)
    | (ATOM(s)::ts) ->
        (try
          (Int(int_of_string(s)), ts)
         with
          (Failure("int_of_string")) ->
            (try
              (Flt(float_of_string(s)), ts)
             with
              (Failure("float_of_string")) ->
                (Sym(s), ts)))
    | (LPAREN::ts) ->
       (match fromToksList ts with
          (sexps,ts') -> (Seq(sexps), ts'))
    | (RPAREN::_) ->  raise (IllFormedSexp "Sexp: unmatched right paren")

   (* fromToksList : token list -> (sexp list * token list) *)
   (* Collects all sexps before next right paren *)
   and fromToksList toks =
     match toks with
       [] -> raise (IllFormedSexp "Sexp: no tokens")
     | (RPAREN::ts) -> ([],ts)
     | _ -> (match fromToks toks with
               (sexp,ts) ->
                 (match fromToksList ts with
                    (sexps,ts') -> (sexp::sexps,ts')))

  let string2Sexp s =
    match fromToks (string2Tokens s) with
      (sexp, []) -> sexp
    | _ -> raise (IllFormedSexp "Sexp: extra chars after end of sexp")

  let file2Sexp filename =
    string2Sexp(File.file2String(filename))

  let rec string2Sexps s = toks2Sexps (string2Tokens s)

  and toks2Sexps toks =
    match toks with
      [] -> []
    | _ -> (match fromToks toks with
              (sexp, toks') -> sexp :: (toks2Sexps toks'))

  let file2Sexps filename =
    string2Sexps(File.file2String(filename))
```

Figure 11: Parsing s-expression tokens into strings.

```
let rec sexps2String sexps =
    String.concat "\n\n" (List.map sexp2String sexps)

  and sexp2String sexp = sexp2String' 80 sexp

  and sexp2String' width sexp = String.concat "\n" (sexp2Strings width sexp)

  (* Unparse sexp as a list of pretty-printed lines *)
  (* When possible, try to have all lines be <= width chars wide *)
  and sexp2Strings width sexp =
    match sexp with
      (* For leaf tokens (ints, floats, symbols, chars, strings)
         we "lose" if they're bigger than width. No way to address this. *)
      Int i -> [string_of_int i]
    | Flt f -> [string_of_float f]
    | Sym s -> [s]
    | Chr c -> (* Don't introduce escape sequences *)
       ["'" ^ (String.make 1 c) ^ "'"]
    | Str s -> (* Don't introduce escape sequences *)
       ["\"" ^ s ^ "\""]
    | Seq [] -> ["()"]
    | Seq (sexp1::sexps) ->
        match sexp2Strings (width - 1) (* account for "(" *)
                           sexp1 with
          [s1] -> (* First sexp fits on single line.
                     Try to get format
                       (s1 s2 s3 ... sn)
                     or
                       (s1 s2
                           s3
                           ...
                           sn
                           )
                  *)
            squeeze width s1 sexps
        | strs -> (* Resort to shape
                       (s1
                        s2
                        s3
                        ...
                        sn
                        )
                  *)
            seq2Strings
              (strs @ (List.concat
                         (List.map (sexp2Strings (width - 1)) sexps)))
```

Figure 12: Unparsing s-expressions into strings, part 1

```
and squeeze width s1 sexps =
     let len1 = String.length s1 in
     let rest1 = List.concat
                   (List.map (sexp2Strings (width - len1 - 2))
                             sexps) in
       if (len1 + (totalLen rest1) + 3) <= width then
         (* everything fits on one line. The "3" accounts
            for initial '(', the ' ' between s1 and rest1,
            and the final ')' *)
         ["(" ^ s1
              ^ " "
              ^ (String.concat " " rest1)
              ^ ")"]
       else if (len1 + (maxLen rest1) + 2) <= width then
         (* everything fits into shape
              (s1 s2
                  s3
                  ...
                  sn
                  )
            The "2" accounts for the initial '(' and
            the ' ' between s1 and rest1 *)
         (match rest1 with
            [] -> ["(" ^ s1 ^ ")"]
          | (str1 :: strs) ->
              ("(" ^ s1 ^ " " ^ str1)
              :: (List.map (prefix (len1 + 2)) (* 1 for '(', 1 for ' ' *)
                           (strs @ [")"])))
          )
       else (* must resort to shape
                 (s1
                  s2
                  s3
                  ...
                  sn
                  )
               This requires backtracking on assumed width
             *)
         seq2Strings (s1 ::
                      (List.concat
                        (List.map (sexp2Strings (width - 1))
                                  sexps)))
```

Figure 13: Unparsing s-expressions into strings, part 2

```
and seq2Strings strings =
    match strings with
      [] -> ["()"]
    | (fst :: rst) ->
        ("(" ^ fst) ::
        ((List.map (fun s -> " " ^ s) rst)
         @ [" )"])

  and prefix n str = (* prefix str with n spaces *)
    (String.make n ' ') ^ str

  and totalLen strs = (* Total length of strings in list of strings *)
                      (* Count 1 for spaces between strings *)
    (List.length strs) - 1
    + (List.fold_left
        (fun n str -> n + (String.length str))
        0
        strs)

  and maxLen strs = (* Length of longest string in list of strings *)
    List.fold_left
     (fun n str ->  max n (String.length str))
     0
     strs
```

Figure 14: Unparsing s-expressions into strings, part 3