

Principles of Programming Languages

Syntactic Operations and Syntactic Properties of Expressions

We covered in the previous lecture how to specify the syntax of a programming language and how to implement the parsing process which turns a stream of characters denoting a program into an Abstract Syntax Tree (AST) value which can be easily processed by a program such as an interpreter or a compiler.

In this lecture, we demonstrate how syntactic properties of expressions can be computed on ASTs and how we can rewrite AST into different ASTs. All of these operations on ASTs work according to the same “recipe” - which consists of following the **structural induction** principle.

We present a first version of the TypeScript program which encodes the following BNF in a set of disjoint union types in TypeScript.

L1 AST in TypeScript

```
<program> ::= (L1 <exp>+) // program(exps:List(exp))
<exp> ::= <define-exp> | <cexp>
<define-exp> ::= (define <var-decl> <cexp>)
                  // def-exp(var:var-decl, val:cexp)
<cexp> ::= <num-exp> // num-exp(val:Number)
          | <bool-exp> // bool-exp(val:Boolean)
          | <prim-op> // prim-op(op:String)
          | <var-ref> // var-ref(var:String)
          | (<cexp> <cexp>*) // app-exp(rator:cexp,
                                     // rands:List(cexp))
<prim-op> ::= + | - | * | / | < | > | = | not
<num-exp> ::= a number token
<bool-exp> ::= #t | #f
<var-ref> ::= an identifier token
<var-decl> ::= an identifier token
```

We define the disjoint union types:

```
type Exp = DefineExp | CExp;  
type CExp = NumExp | BoolExp | PrimOp  
           | VarRef | AppExp;
```

interfaces for each category:

```
interface Program {tag:"Program", exps: Exp[]}  
interface DefineExp {tag:"DefineExp", var: VarDecl, val: CExp}  
interface NumExp {tag:"NumExp", val:number}  
interface BoolExp {tag:"BoolExp", val:boolean}  
interface PrimOp {tag:"PrimOp", op:string}  
interface VarRef {tag:"VarRef", var:string}  
interface VarDecl {tag:"VarDecl", var:string}  
interface AppExp {tag:"AppExp", rator:CExp, rands: CExp[]}
```

Constructors:

```
const makeProgram = (exps: Exp[]):Program =>
  ({tag:"Program", exps: exps});
const makeDefineExp = (v: VarDecl, val: CExp):DefineExp =>
  ({tag:"DefineExp", var:v, val:val});
const makeNumExp = (n: number):NumExp =>
  ({tag:"NumExp", val:n});
const makeBoolExp = (b: boolean):BoolExp =>
  ({tag:"BoolExp", val:b});
const makePrimOp = (op: string):PrimOp =>
  ({tag:"PrimOp", op: op});
const makeVarRef = (v: string):VarRef =>
  ({tag:"VarRef", var:v});
const makeVarDecl = (v: string):VarDecl =>
  ({tag:"VarDecl", var:v});
const makeAppExp = (rator:CExp, rands:CExp[]):AppExp =>
  ({tag:"AppExp", rator: rator, rands: rands});
```

And type predicates:

```
const isProgram = (x:any): x is Program =>
  x.tag === 'Program';
const isDefineExp = (x:any): x is DefineExp =>
  x.tag === 'DefineExp';
const isNumExp = (x:any): x is NumExp =>
  x.tag === 'NumExp';
const isBoolExp = (x:any): x is BoolExp =>
  x.tag === 'BoolExp';
const isPrimOp = (x:any): x is PrimOp =>
  x.tag === 'PrimOp';
const isVarRef = (x:any): x is VarRef =>
  x.tag === 'VarRef';
const isVarDecl = (x:any): x is VarDecl =>
  x.tag === 'VarDecl';
const isAppExp = (x:any): x is AppExp =>
  x.tag === 'AppExp';
```

L1 AST in TypeScript

```
const isExp = (x:any): x is Exp =>
  isDefineExp(x) || isCExp(x);
const isCExp = (x:any): x is CExp =>
  isNumExp(x) || isBoolExp(x) || isPrimOp(x) ||
  isVarRef(x) || isAppExp(x);
```

Note the patterns used in the code:

- A constructor function named **make** which takes as parameters each of the component values of the compound type.

- A type predicate function named `is`. Note the type annotation of this predicate: it takes an any parameter and returns a `x is T` type. Such types are understood by the TypeScript type system and allow the type checker to conclude that a variable has the given type within a scope within a program that is “guarded” by such a predicate. We will see many examples using this facility.

A disjoint union type construct has the following shape:

```
type CExp = NumExp | BoolExp | PrimOp  
          | VarRef | AppExp;
```

which is a union of disjoint types. Each disjoint type is defined as a tagged map. We use the convention of using the key **tag** to enforce the disjointness of these types. Any other key could be used, but we use this one consistently to express our intention of defining disjoint types.

The type definition following the key **tag** is a singleton type - which contains a discriminative string value. For example, the type expression:

tag: "NumExp"

within these type definitions indicates that the key **tag** must have the unique value **"NumExp"** for the map to belong to the type **NumExp**.

There is a constructor function for each disjoint type, but none for the union type. In this sense, we understand that the union type is a sort of abstract type over the disjoint types.

The type predicate for each disjoint type checks for the presence of the appropriate tag. The type predicate for the union types is just a boolean-or of the type predicates of the types it covers.

AST Types are Recursive Types

The AST types we have defined above are recursive. Some of the disjoint types of the disjoint union are atomic, some are compound. The one type that is recursive is CExp (C stands for constituent expression - a constituent is a component of a more complex structure). CExp is defined as follows in the AST definition:

AST Types are Recursive Types

```
<cexp> ::= <num-exp>      // num-exp(val:Number)
        | <bool-exp>      // bool-exp(val:Boolean)
        | <prim-op>       // prim-op(op:String)
        | <var-ref>       // var-ref(var:String)
        | (<cexp> <cexp>*) // app-exp(rator:cexp,
                               //           rands:List(cexp))
```

The only compound expression here is **AppExp**
- which is made up of **CExp** components. The
other types are atomic (**NumExp**, **BoolExp**,
PrimOp, **VarRef**).

This recursive definition is what makes the language **infinite**. It also explain why AST are called abstract syntax **trees**.

As usual - note that the corresponding TypeScript type definitions are based on a disjoint union to enable the recursive type definition.

The parser takes as input a string and returns an AST value which encodes the structure of values recognized in the string.

Parsing is a complex topic which will be covered in much more detail in the Compilation course.

In this course, we take a “shortcut” approach to parsing, because the language we use is based on a general structure called the **S-Expression** (in short S-exp). The fact that Scheme and LISP-like languages use S-exp as the basis for their syntax is part of a general approach towards simplicity and uniformity in these languages.

An S-exp is defined inductively as using the following BNF (we ignore some of the elements used in Lisp such as pairs for simplicity):

```
<S-exp> ::= <AtomicSexp> | <CompoundSexp>  
<AtomicSexp> ::= <number> | <boolean>  
                  | <string> | <symbol>  
<CompoundSexp> ::= '(' <S-exp>* ')'
```

The definition is recursive, leading to deeply nested lists such as:

```
((1) ((2 "a") #t))
```

In Scheme, S-exps are used both to encode programs and data.

To implement our Scheme parser in TypeScript, we rely on an existing Node parser for S-exp. Make sure you install it in your work folder with:

```
> npm install s-expression
```

The Sexp parser takes care of tokenization and handling the nested parentheses in order to construct a parallel structure of nested lists - we have mapped a string into a JavaScript S-exp. Note that the code of the S-exp parsers is not very complex - you can read it all in the module we have downloaded with **npm**. But at this stage, we just skip its complexity.

The S-exp values we obtain as output of the S-exp parser are made up of nested lists of strings only (the only atomic values that appear are strings).

```
parseSexp("(+ 1 (* 2 3))");  
// => [ '+', '1', [ '*', '2', '3' ] ]
```

The S-expression parser we use from **npm** is written in JavaScript - it does not specify the type of the value it returns.

The S-Expression Type

We analyzed the code, and inferred manually the precise type returned by this parser and added this type annotation, which we add as the “contract” that we expect from the library and which can be trusted by the TypeScript type checker. This is achieved by adding a file with extension d.ts in our codebase `src/shared/s-expression.d.ts`:

The S-Expression Type

```
declare module 's-expression' {  
  export type SexpString = String;  
  export type Token = string | SexpString;  
  export type CompoundSexp = Sexp[];  
  export type Sexp = Token | CompoundSexp;  
  
  /*  
    The types returned by the parser are:  
    string - for any token which is not a string,  
             according to the tokenization rules  
             of S-expressions.  
    SexpString - for tokens of the form "..."  
    Sexp[] - for S-expressions that contain  
              sub-expressions (of the form  
              "<s-expr1> ... <s-exprn>")  
  */  
  export default function parse(x: string): Sexp;  
}
```

The S-Expression Type

The S-expression parser interprets all atomic tokens according to Scheme's lexical rules. In particular, it encodes tokens of type string, which are written as balanced double-quotes "... " in a specific TypeScript type called **String** (with capital-S, which is different from the usual string). We call this token type a **SexpString** in our type definition.

The structure of the Sexp type is the usual disjunction between Atomic tokens and Compound expressions. Atomic tokens form the base case of the inductive definition. Compound expressions are encoded as arrays of embedded expressions.

We provide in `src/shared/parser.ts` well-typed TypeScript parsers around the third-party S-expression parser written in JavaScript:

The S-Expression Type

```
import p, { Sexp, SexpString,  
           Token, CompoundSexp } from "s-expression";  
import { makeFailure, makeOk, Result } from "./result";  
import { isString,  
         isArray, isError } from "./type-predicates";  
import { allT } from "./list";
```

The S-Expression Type

```
// s-expression returns strings quoted as "a" as [String: 'a'] objects  
// to distinguish them from symbols - which are encoded as 'a'  
// These are constructed using the new String("a") constructor  
// and can be distinguished from regular strings based  
// on the constructor.  
const isSexpString = (x: any): x is SexpString =>  
  ! isString(x) &&  
  x.constructor &&  
  x.constructor.name === "String";  
  
export const isSexp = (x: any): x is Sexp =>  
  isToken(x) || isCompoundSexp(x);  
export const isToken = (x: any): x is Token =>  
  isString(x) || isSexpString(x);  
export const isCompoundSexp = (x: any): x is CompoundSexp =>  
  isArray(x) && allT(isSexp, x);
```

The S-Expression Type

```
const parse = (x: string): Result<Sexp> => {  
  const parsed = p(x);  
  return isError(parsed) ? makeFailure(parsed.message)  
    : makeOk(parsed);  
}
```

The relevant types of tokens must be recognized by analyzing a **Token** value to decide the type of literal value the token represents (boolean, number or string). We use the following TypeScript type predicates (using the TypeScript type predicate notation `x is T`). These definitions are provided in `src/shared/type-predicates.ts`:

```
// Type utilities  
export const isArray = Array.isArray;  
export const isString = (x: any): x is string =>  
    typeof x === "string";  
export const isNumber = (x: any): x is number =>  
    typeof x === "number";  
export const isBoolean = (x: any): x is boolean =>  
    typeof x === "boolean";  
export const isError = (x: any): x is Error =>  
    x instanceof Error;
```

```
// A weird method to check that a string  
// is a string encoding of a number  
export const isNumericString = (x: string): boolean =>  
    JSON.stringify(+x) === x;
```

Token Types

```
// A predicate for a valid identifier  
// In Scheme, a valid identifier is a token that  
// starts with an alphabetic letter (a-z or A-Z)  
// followed by any number of letters or numbers.  
// As discussed in the Section on Lexical Rules - we  
// use Regular Expressions (regexp) to recognize these.  
export type Identifier = string;  
export const isIdentifier = (x: any): x is Identifier =>  
    /[A-Za-z][A-Za-z0-9]*/i.test(x);
```

The external s-expression parser library can fail when it faces an illegal combination of parentheses or ill-formed tokens. In this case, it returns a value of type **Error**.

In the rest of the course, we will use a functional approach to handle errors, based on the `Result<T>` monad. Therefore, we wrap the call to the library parser in a function that adapts the Error protocol into a `Result<Sexp>`:

Error Handling

```
const parse = (x: string): Result<Sexp> => {  
    const parsed = p(x);  
    return isError(parsed) ? makeFailure(parsed.message)  
                           : makeOk(parsed);  
}
```

The rest of the parser is structured to return values of type `Result<T>`. Given this pattern, we combine functions that return `Result<T>` using the `bind` operator.

For example, instead of writing:

```
const parseL1 = (x: string): Program =>  
  parseL1Program(parse(x));
```

We write:

```
// combine Sexp parsing with L1 parsing using  
// the bind operator  
const parseL1 = (x: string): Result<Program> =>  
    bind(parse(x), parseL1Program);
```

The proper way to read a **bind** combination is:

- First invoke **parse(x)**
- If the returned value is of type **Failure**, return this value without further processing.
- Else, the value is of type **Ok<Sexp>**, then pass the wrapped **Sexp** value to the function **parseL1Program**

In a **bind** call, the composed functions appear in the order in which they are executed – first **parse**, then **parseL1Program**. This is in contrast with the traditional function composition notation **f(g(x))** where **g** is executed before **f** but appears after it.

When using `Result<T>`, we avoid type problems that would occur because of errors. For example, if `parse(x)` were to return an `Error` value, the composition would need to be organized as follows:

Error Handling

```
const parse = (x: string): Sexp | Error => ...;
const parseL1Program = (s: Sexp): Program | Error => ...;

const parseL1 = (x: string): Program => {
  const s = parse(x);
  return isError(s) ? s :
    parseL1Program(s);
}
```

If we read this code, error handling makes us lose the intent that `parseL1` is just a composition of two functions.

This intent is preserved with the `bind` version: `bind` is the **composition** operator for functions that return `Result<T>` values.

The general structure of the parser functions is to traverse an inductive data structure, and to switch according to the type of the parameter.

In TypeScript, we use a switch code structure: to make it functional, we use chained ternary conditionals `e1 ? e2 : e3 ? e4 : ...` which is an expression (as opposed to `switch` or `if else if` which are statements).

We indent such chained ternary conditions in the code to express clearly that they implement a sequence of cases as in a switch.

The type of the parameters are disjoint unions from the AST definitions. The clauses of the switch are all type predicates. We call such tests *guards*.

After a guard, the TypeScript type checker knows that the parameter has the requested type (because type predicates are defined with a return type of the form `x is T`) and takes it into account in the calls of the **then** part of the **if** statement (we call this part of the code the **guarded clause**).

For example, in `parseL1Program`, we receive a parameter of type `Sexp` and traverse it using the following switch pattern:

Type Guards

```
// <Program> -> (L1 <Exp>+)  
export const parseL1Program = (sexp: Sexp): Result<Program> =>  
  sexp === "" || isEmpty(sexp) ?  
    makeFailure("Unexpected empty program") :  
  isToken(sexp) ?  
    makeFailure("Program cannot be a single token") :  
  isCompoundSexp(sexp) ?  
    parseL1GoodProgram(first(sexp), rest(sexp)) :  
  sexp;
```

As usual, the structure of this function follows the structure of the type definition: an **Sexp** value can be either an atomic Token or a CompoundSexp. We deal with specific error conditions by returning **makeFailure** values.

In L_1 , atomic expressions can be either number, boolean or primitive operators. All three are encoded in concrete syntax as different types of Tokens.

Accordingly, the parser function that recognizes atomic expressions enumerates the possible types of tokens and constructs the appropriate AST values for each possible option.

Parsing Atomic Expressions

```
// Atomic -> number | boolean | primitiveOp
export const parseL1Atomic = (token: Token): Result<CExp> =>
  token === "#t" ? makeOk(makeBoolExp(true)) :
  token === "#f" ? makeOk(makeBoolExp(false)) :
  isString(token) && isNumericString(token) ?
    makeOk(makeNumExp(+token)) :
  isString(token) && isPrimitiveOp(token) ?
    makeOk(makePrimOp(token)) :
  isIdentifier(token) ? makeOk(makeVarRef(token)) :
  makeFailure("Invalid atomic token: " + token);

export const isPrimitiveOp = (x: string): boolean =>
  ["+", "-", "*", "/", ">", "<", "=", "not"].includes(x)
```

In the syntax of $L1$, we distinguish between:

- compound expressions that can occur embedded within other expressions (and which we call constituent expressions, in short **CExp**)
- and compound expressions that cannot be embedded within other expressions, but must occur at the toplevel of a program. **define-exp** is the only such case in $L1$.

Accordingly, the parser must process compound expressions depending on the context:

- At the toplevel of a program, one can expect **define-exp** or **c-exp** expressions.
- Within other contexts, one can only expect **c-exp** or atomic expressions.

Compound expressions in the concrete syntax of Scheme-like languages are all **Sexp** arrays, and they are recognized by checking the first element in the array. This makes it easy to check what is the type of a compound **Sexp** given its concrete syntax.

The logic of the traversal of compound expressions is captured in the following functions.

Parsing Compound Expressions

```
// Exp -> <DefineExp> | <Cexp>
export const parseL1Exp = (sexp: Sexp): Result<Exp> =>
  isEmpty(sexp) ?
    makeFailure("Exp cannot be an empty list") :
  isArray(sexp) ?
    parseL1CompoundExp(first(sexp), rest(sexp)) :
  isToken(sexp) ?
    parseL1Atomic(sexp) :
  sexp;
```

Parsing Compound Expressions

```
// Compound -> DefineExp | CompoundCExp
export const parseL1CompoundExp =
  (op: Sexp, params: Sexp[]): Result<Exp> =>
    op === "define"? parseDefine(params) :
    parseL1CompoundCExp(op, params);

// CompoundCExp -> AppExp
export const parseL1CompoundCExp =
  (op: Sexp, params: Sexp[]): Result<CExp> =>
    parseAppExp(op, params);
```

Parsing Compound Expressions

```
// CExp -> AtomicExp | CompoundCExp
export const parseL1CExp = (sexp: Sexp): Result<CExp> =>
  isEmpty(sexp) ?
    makeFailure("CExp cannot be an empty list") :
  isArray(sexp) ?
    parseL1CompoundCExp(first(sexp), rest(sexp)) :
  isToken(sexp) ?
    parseL1Atomic(sexp) :
  sexp;

// AppExp -> ( <cexp>+ )
export const parseAppExp =
  (op: Sexp, params: Sexp[]): Result<CExp> =>
    safe2((rator: CExp, rands: CExp[]) =>
      makeOk(makeAppExp(rator, rands)))
      (parseL1CExp(op), mapResult(parseL1CExp, params));
```

On the basis of the syntactic structure of program expressions, one can specify formally and precisely important properties of the program. We start with an example of such syntactic properties called variable binding.

In Scheme as well as in JavaScript, Java and many other languages, variables can occur in two different ways in a program:

- As references
- As declarations

A *variable reference* uses a variable - and refers to its value. For example, in the expression $(+ 1 x)$, x refers to a value that was previously attached to the variable.

Scoping and Binding of Variables

In contrast, a *variable declaration* defines a new variable as an abstraction (a name) for a value. For example, in Scheme, in the expressions `(lambda (x) ...)` or `(let ((x ...)) ...)`, `x` is declared as a new variable. In the `lambda` case, the value of `x` will be provided when the function is invoked; in the `let` case, the value of `x` is provided in the binding location of the `let`-expression.

Scoping and Binding of Variables

Variable declarations usually have limited scope, so that the same variable may be reused in different places in the program. This means that the name `x` in different locations of the program may refer to different variables. In the case of `lambda` and `let`, the declared variables are visible only within the scope of the body of the expressions.

Programming languages come with rules which determine how variable references relate to variable declarations. These are called **binding rules**.

In Scheme, these rules are **syntactic rules** - that is, the relation can be computed by analyzing the AST of the program without executing it.

Another way of saying this is that binding is a **static property** as opposed to a **dynamic property** which would depend on a specific execution of the program.

Static properties are defined through structural induction - that is, they are defined for all possible types of expressions by going over the list of all possible expression types defined in the abstract syntax of the language.

- In an expression
(**lambda** (<variable>) <body>) the
occurrence of <variable> is a declaration
that binds all occurrences of that variable in
<body> unless some intervening
declaration of the same variable occurs.

- In an expression

`(let ((<variable> <value>)) <body>)`

the occurrence of `<variable>` is a declaration that binds all occurrences of that variable in `<body>` unless some intervening declaration of the same variable occurs.

A variable x occurs **free** in an expression E if and only if there is some use of x in E that is not bound by any declaration of x in E .

A variable x occurs **bound** in an expression E if and only if there is some use of x in E that is bound by a declaration of x in E .

For example:

`((lambda (x) x) y)`

- `x` occurs bound
- `y` occurs free

For example:

```
(lambda (y)
  ((lambda (x) x) y))
```

- Now y occurs bound

The algorithm to determine whether a variable occurs free in an expression is encoded as the typical traversal of the AST, using the same recipe as we used to compute the height of an expression (**Eheight**): this is a structural induction over the disjoint union types that define the Scheme AST:

Free and Bound Variables

```
const occursFree = (v: string, e: Exp): boolean =>
  isBoolExp(e) ? false :
  isNumExp(e) ? false :
  isStrExp(e) ? false :
  isLitExp(e) ? false :
  isVarRef(e) ? (v === e.var) :
  isIfExp(e) ? occursFree(v, e.test) ||
                occursFree(v, e.then) ||
                occursFree(v, e.alt) :
  isProcExp(e) ? !(map((p) => p.var, e.args).includes(v)) &&
                some((b) => occursFree(v, b), e.body) :
  isPrimOp(e) ? false :
  isAppExp(e) ? occursFree(v, e.rator) ||
                some((rand) => occursFree(v, rand), e.rands) :
  isDefineExp(e) ? (v !== e.var.var) && occursFree(v, e.val) :
  false;
```

An extension of this algorithm consists of collecting all the variables that are referenced in a given expression.

Collecting Variable References from an Expression

```
export const referencedVars = (e: Program | Exp): VarRef[] =>
  isBoolExp(e) ? Array<VarRef>() :
  isNumExp(e) ? Array<VarRef>() :
  isStrExp(e) ? Array<VarRef>() :
  isLitExp(e) ? Array<VarRef>() :
  isPrimOp(e) ? Array<VarRef>() :
  isVarRef(e) ? [e] :
  isIfExp(e) ? reduce(varRefUnion, Array<VarRef>(),
    map(referencedVars, [e.test, e.then, e.alt])) :
  isAppExp(e) ? union(referencedVars(e.rator),
    reduce(varRefUnion, Array<VarRef>(),
      map(referencedVars, e.rands))) :
  isProcExp(e) ? reduce(varRefUnion,
    Array<VarRef>(),
    map(referencedVars, e.body)) :
  isDefineExp(e) ? referencedVars(e.val) :
  isProgram(e) ? reduce(varRefUnion,
    Array<VarRef>(),
    map(referencedVars, e.exps)) :
  isLetExp(e) ? Array<VarRef>() : // TODO
  Array<VarRef>();
```

Collecting Variable References from an Expression

Note how the structure of this function, is again a traversal of the AST according to the type definition - this function has a structure almost identical to any AST visitor.

By combining **referencedVars** and **occursFree** we can obtain the list of variables that occur free within an expression.

Distinguishing Variable Declaration and Variable References in Abstract Syntax

Since we make a distinction between the two positions of variables, we can change the abstract syntax to represent variable declarations and variable references as two different data types.

Distinguishing Variable Declaration and Variable References in Abstract Syntax

This is reflected in this updated BNF - where we define the category **<cexpLA>** for “expression with lexical address”:

```
<cexpLA> ::= <number>                / num-exp(val:number)
| <boolean>                          / bool-exp(val:boolean)
| <string>                            / str-exp(val:string)
| ( quote <sexp> )                    / literal-exp(val:sexp)
| <var-ref>                          / var-ref(var:string)
| ( lambda ( <var-decl>* ) <cexpLA>+ )
  / proc-expLA(params:List(var-decl), body:List(cexpLA))
| ( if <cexpLA> <cexpLA> <cexpLA> )
  / if-expLA(test: cexpLA, then: cexpLA, else: cexpLA)
| ( <cexpLA> <cexpLA>* )
  / app-expLA(rator:cexpLA, rands:List(cexpLA))
```

Distinguishing Variable Declaration and Variable References in Abstract Syntax

The atomic expression types can be reused from the previous AST definition (number, boolean, string). Literal expressions are also unchanged.

Distinguishing Variable Declaration and Variable References in Abstract Syntax

We distinguish between `<var-decl>` and `<var-ref>` as two distinct categories, which are both mapped in concrete syntax to identifiers. But the appropriate category is selected based on the context of the identifier: in the parameter list of a lambda-expression, identifiers are interpreted as `<var-decl>`, elsewhere, as `<var-ref>`.

Distinguishing Variable Declaration and Variable References in Abstract Syntax

Compound expressions have the same structure as in the original syntactic definition, but refer to the new type `<cexpLA>` instead of `<cexp>`.

Determining the Scope of Variable Declarations

In the lexically scoped language we are used to, the same variable name can be used in different scopes to refer to different variables.

For example:

```
((lambda (x) (* x x)) ; 1  
  ((lambda (x) (+ x x)) ; 2  
   2))
```

The variable references in line 1 refer to the declaration in the first **lambda** in line 1, and those in line 2, to the second **lambda** declaration in line 2.

Determining the Scope of Variable Declarations

```
(lambda (x y)                               ; 1  
  ((lambda (x) (+ x y))                     ; 2  
   (+ x x)) 1)                             ; 3
```

- The variable reference `x` in line 2 refers to the declaration in line 2;
- The variable reference `y` in line 2 refers to the declaration in line 1;
- The variable reference `x` in line 3 refers to the declaration in line 1.

These relations between variable reference and variable declarations are static properties - they only depend on the syntactic structure of the expression.

One way to disambiguate variable references is to replace them with their **lexical address**: the lexical address determines in an unambiguous manner the variable declaration to which a variable reference is bound.

To define lexical address it helps to define the **contour** of a sub-expression within an embedding expression: each time a new declaration scoped is defined (using a **lambda** or **let** expression in our language), a new contour is defined. Contours are embedded into each other. Contours correspond to the scope of the declaration.

To define lexical address it helps to define the **contour** of a sub-expression within an embedding expression: each time a new declaration scoped is defined (using a **lambda** or **let** expression in our language), a new contour is defined. Contours are embedded into each other. Contours correspond to the scope of the declaration.

In this example, there is a contour started at line 1 with the lambda declaration, and a second embedded contour in line 2.

```
(lambda (x y) ; 1
  ((lambda (x) (+ x y)) ; 2
    (+ x x)) 1) ; 3
```

Variable references can refer to the declarations in the contours in which they appear - starting from the inner declaration, and looking outwards.

For example, in line 2, the `x` reference looks up to the declaration in the inner contour in line 2; the `y` reference looks up to the external declaration in the outer contour in line 1.

```
(lambda (x y)                ; 1
  ((lambda (x) (+ x y))      ; 2
   (+ x x)) 1)              ; 3
```

To indicate these relations, we define a lexical address as a tuple `[var : depth pos]` where:

- `var` is the name of the variable
- `depth` is the number of contours that are crossed to reach the variable declaration
- `pos` is the offset of the variable within the declaration.

For example, the lexical addresses annotations for the expression:

```
((lambda (x) (* x x)) ; 1  
  ((lambda (x) (+ x x)) ; 2  
   2))
```

is:

```
((lambda (x) (* [x : 0 0] [x : 0 0])) ; 1  
  ((lambda (x) (+ [x : 0 0] [x : 0 0])) ; 2  
   2))
```

For example, the lexical addresses annotations for the expression:

```
(lambda (x y) ; 1
  ((lambda (x) (+ x y)) ; 2
    (+ x x)) 1) ; 3
```

is:

```
(lambda (x y) ; 1
  ((lambda (x) (+ [x : 0 0] [y : 1 1])) ; 2
    (+ [x : 0 0] [x : 0 0])) 1) ; 3
```

Note that the variable references `+` and `*` in these examples are not bound to any declaration. This is because they occur free in the expression.

In this case, we annotate them as `[var free]` as follows:

```
((lambda (x) ([* free] [x : 0 0] [x : 0 0]))) ; 1  
((lambda (x) ([+ free] [x : 0 0] [x : 0 0]))) ; 2  
2))
```

```
(lambda (x y) ; 1
  ((lambda (x) ([+ free] [x : 0 0] [y : 1 1])) ; 2
    ([+ free] [x : 0 0] [x : 0 0])) 1) ; 3
```

Since the relation between a variable reference and its corresponding variable declaration is unambiguous according to the syntax of the language, we can design an algorithm which computes the lexical address of all variable references in an expression.

Computing the Lexical Address of Variable References

In order to design this algorithm, we must consider how we perform the task of finding the declaration that matches a variable reference.

The best way to visualize this process is to observe the AST as a tree. When we traverse the tree top-down, and reach a variable-reference node - we are in a leaf-position (because variable reference nodes have no constituent sub-expressions - they are leaves in the AST).

Computing the Lexical Address of Variable References

To locate the corresponding declaration, we must traverse the AST upwards from this variable reference leaf, until we find a **ProcExp** node. When we find this node, we check the parameters list of the **ProcExp** and verify whether the variable name occurs in the parameters list. If it does, this is the declaration that matches the variable reference.

We must also identify the position of the variable name within the parameters - this gives us a lexical address of
`(<var> : 0 pos)`.

If the variable is not found in the parameters list, we continue climbing up the tree - but each time we cross a “contour” (that is, we cross a **ProcExp** node), we increase the **depth** parameter by one.

Another way to achieve this matching, if we need to compute the lexical address of all variable references is to traverse the AST top-down and keeping track, each time we traverse a contour (a **ProcExp** node) of the list of visible variable declarations.

Computing the Lexical Address of Variable References

For example, in the expression:

```
(lambda (x y) ; 1
  ((lambda (x) ([+ free] [x : 0 0] [y : 1 1])) ; 2
   ([+ free] [x : 0 0] [x : 0 0])) 1) ; 3
```

We start at the top of the AST, the list of visible variable declarations is empty. We then cross the node `(lambda (x y) ...)` - the list is now `(x y)`. Any match we may find now will be with depth 0. So we actually maintain a list of lexical addresses:

```
([x : 0 0] [y : 0 1])
```

Computing the Lexical Address of Variable References

```
(lambda (x y) ; 1
  ((lambda (x) ([+ free] [x : 0 0] [y : 1 1])) ; 2
    ([+ free] [x : 0 0] [x : 0 0])) 1) ; 3
```

We continue the traversal of the AST and reach the contour `(lambda (x) ...)` in line 2. We now update the list of visible variable declarations to be `([x : 0 0])` which has now priority, followed by `([x : 1 0] [y : 1 1])`.

Computing the Lexical Address of Variable References

```
(lambda (x y) ; 1
  ((lambda (x) ([+ free] [x : 0 0] [y : 1 1])) ; 2
    ([+ free] [x : 0 0] [x : 0 0])) 1) ; 3
```

Meaning, we incremented the depth of the visible variables - because in the scope of the new contour, to reach **x** and **y** from the outer declaration requires going “up to depth 1” instead of 0.

Computing the Lexical Address of Variable References

```
(lambda (x y) ; 1
  ((lambda (x) ([+ free] [x : 0 0] [y : 1 1])) ; 2
   ([+ free] [x : 0 0] [x : 0 0])) 1) ; 3
```

This means that when we cross a contour, the list of visible lexical addresses becomes $([x : 0 \ 0] \ [x : 1 \ 0] \ [y : 1 \ 1])$. Note that this list is sorted by depth. In this example, the first $[x : 0 \ 0]$ “hides” the previous declaration $[x : 1 \ 0]$ - which is what is expected.

Given this way of maintaining the list of visible lexical addresses for accessible variable declarations, we can define the algorithm to retrieve the lexical address of a variable reference.

We start with the definition of the types needed to encode lexical addresses:

Computing the Lexical Address of Variable References

```
type LexAddress = FreeVar | LexicalAddress;  
const isLexAddress = (x: any): x is LexAddress =>  
    isFreeVar(x) || isLexicalAddress(x);
```

Computing the Lexical Address of Variable References

```
interface FreeVar {  
  tag: "FreeVar";  
  var: string;  
}  
  
const isFreeVar = (x: any): x is FreeVar =>  
  typeof x === "object" && x.tag === "FreeVar";  
  
const makeFreeVar = (v: string): FreeVar =>  
  ({tag: "FreeVar", var: v});
```

Computing the Lexical Address of Variable References

```
interface LexicalAddress {  
  tag: "LexicalAddress";  
  var: string;  
  depth: number;  
  pos: number;  
}  
  
const isLexicalAddress = (x: any): x is LexicalAddress =>  
  typeof x === "object" && x.tag === "LexicalAddress";  
  
const makeLexicalAddress =  
  (v: string, depth: number, pos: number): LexicalAddress =>  
    ({tag: "LexicalAddress", var: v, depth: depth, pos: pos});  
  
const makeDeeperLexicalAddress =  
  (la: LexicalAddress): LexicalAddress =>  
    makeLexicalAddress(la.var, la.depth + 1, la.pos);
```

Computing the Lexical Address of Variable References

The following procedure implements the algorithm to retrieve the lexical address of a variable reference:

```
const getLexicalAddress =  
  (v: VarRef, lexAddresses: LexAddress[]): LexAddress => {  
    const loop = (addresses: LexAddress[]): LexAddress => {  
      isEmpty(addresses) ? makeFreeVar(v.var) :  
      v.var === first(addresses).var ? first(addresses) :  
      loop(rest(addresses));  
    }  
    return loop(lexAddresses);  
  }
```

Computing the Lexical Address of Variable References

Note how we mark the variable as occurring free when it is not found in any of the visible declarations.

Observe how we implement iteration by defining a local recursive procedure called **loop** and invoke it inside the main body of the procedure.

The algorithm to compute the lexical address of all variable references is thus implemented as follows.

The function is used as follows:

Traversing the Whole AST

```
const f = (s: string): Result<any> =>  
  bind(LA.parseLA(s),  
    (cexpla: LA.CExpLA) =>  
      bind(LA.addLexicalAddresses(cexpla),  
        (cexpla: LA.CExpLA) => makeOk(LA.unparseLA(cexpla))))
```

Traversing the Whole AST

```
f("(lambda (x) x)");  
// => { tag: 'Ok',  
//      value: ["lambda", ["x"], ["x", ":", 0, 0]] }
```

```
f("(lambda (x) (lambda (y) (+ x y)))");  
// => { tag: 'Ok',  
//      value: ["lambda", ["x"],  
//                  ["lambda", ["y"],  
//                      [["+ ", "free"],  
//                      ["x", ":", 1, 0],  
//                      ["y", ":", 0, 0]]]] }
```

Recall that we introduced the let-expression in the previous lectures as a **syntactic abbreviation**. This means that when we define the operational semantics of the language, we do not need to define a new computation rule for this expression type, instead we indicate that this expression is equivalent to a combination of other syntactic constructs that mean the same thing.

In the case of `let` the syntactic transformation leading to a simpler equivalent construct is:

`(let ((var1 val1) ...) body)`



`((lambda (var1 ...) body) val1 ...)`

For example:

```
(let ((x 1) (y 2)) (+ x y))
```

⇓

```
((lambda (x y) (+ x y)) 1 2)
```

Such syntactic transformations are implemented by mapping AST values containing let-expressions to semantically equivalent AST values that only contain lambda applications.

```
const rewriteLet = (e: LetExp): AppExp => {  
  const vars = map(b => b.var, e.bindings);  
  const vals = map(b => b.val, e.bindings);  
  return makeAppExp(makeProcExp(vars, e.body),  
                    vals);  
}
```

This definition only applies on a single
let-expression.

To perform the transformation at all levels within an arbitrary expression, we must visit the AST top-down and apply the transformation wherever needed. This is implemented in this function, which has the typical structural induction structure of traversing all possible AST values:

Rewriting ASTs

```
const rewriteAllLet = (exp: Program | Exp): Program | Exp =>
  isExp(exp) ? rewriteAllLetExp(exp) :
  isProgram(exp) ? makeProgram(map(rewriteAllLetExp, exp.exps)) :
  exp;

const rewriteAllLetExp = (exp: Exp): Exp =>
  isCExp(exp) ? rewriteAllLetCExp(exp) :
  isDefineExp(exp) ? makeDefineExp(exp.var,
                                   rewriteAllLetCExp(exp.val)) :
  exp;
```

Rewriting ASTs

```
const rewriteAllLetCExp = (exp: CExp): CExp =>
  isAtomicExp(exp) ? exp :
  isLitExp(exp) ? exp :
  isIfExp(exp) ? makeIfExp(rewriteAllLetCExp(exp.test),
                           rewriteAllLetCExp(exp.then),
                           rewriteAllLetCExp(exp.alt)) :
  isAppExp(exp) ? makeAppExp(rewriteAllLetCExp(exp.rator),
                             map(rewriteAllLetCExp, exp.rands)) :
  isProcExp(exp) ? makeProcExp(exp.args,
                               map(rewriteAllLetCExp, exp.body)) :
  isLetExp(exp) ? rewriteAllLetCExp(rewriteLet(exp)) :
  exp;
```

Observe how the same exact programming pattern is used as for the case of computing lexical addresses - in the form of a transformation function for nodes of type **LetExp** and a walker function which traverses a complete AST from root to leaves and applies a transformation to each node.