/G53CMP: Lecture 7

Contextual Analysis: Scope II

Henrik Nilsson
University of Nottingham, UK
This Lecture

An Illustrative Identification Algorithm in Haskell

- LTXL Syntax and Semantics, particularly scope rules.
- Abstract syntax representation
- Environment/Symbol Table representation and operations.
- The Identification Algorithm
Recap: Identification

*Identification* is the task of relating each applied identifier occurrence to its declaration or definition:

```java
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```
Recap: Identification

Identification is the task of relating each applied identifier occurrence to its declaration or definition:

```java
public class C {
    int x, n;
    void set(int n) {
        x = n;
    }
}
```

In the body of `set`, the one applied occurrence of
- `x` refers to the *instance variable* `x`
Identification is the task of relating each applied identifier occurrence to its declaration or definition:

```java
public class C {
    int x, n;
    void set(int n) {
        x = n;
    }
}
```

In the body of `set`, the one applied occurrence of
- `x` refers to the **instance variable** `x`
- `n` refers to the **argument** `n`. 
Identification for LTXL

We are now going to study a concrete Haskell implementation of identification for \textit{LTXL}:

\textit{Less Trival eXpression Language}

- \textit{LTXL} \approx \textit{TXL} + typed definitions + \textit{if}-expression + new operators
- Slides only show highlights: complete code available on-line.
LTXL CFG (1)

\[ \text{LTXLP} \rightarrow \text{Exp} \]

\[ \text{Exp} \rightarrow \text{Exp} \text{ || } \text{Exp} | \text{Exp} \text{ && } \text{Exp} | \text{Exp} < \text{Exp} | \text{Exp} == \text{Exp} | \text{Exp} > \text{Exp} | \text{Exp} + \text{Exp} | \text{Exp} - \text{Exp} | \text{Exp} * \text{Exp} | \text{Exp} / \text{Exp} | \text{PrimaryExp} \]
Operator precedence is used to disambiguate. In increasing order of precedence:

1. `||`
2. `&&`
3. `<`, `==`, `>`
4. `+`, `−`
5. `∗`, `/`
PrimaryExp → LitInt
  | Ident
  | \ PrimaryExp
  | – PrimaryExp
  | if Exp then Exp else Exp
  | ( Exp )
  | let Defs in Exp
LTXL CFG (4)

\[
\begin{align*}
\text{Defs} & \rightarrow \text{Def} ; \text{Defs} \\
& \mid \text{Def} \\

\text{Def} & \rightarrow \text{Type } \textbf{Ident} = \text{Exp} \\

\text{Type} & \rightarrow \textbf{int} \\
& \mid \textbf{bool}
\end{align*}
\]
let

    int a = 10;
    bool b = a < 2

in let

    int c = a * 10;
    bool a = a == 42;
    int d = if a then 1 else 2

in

    if a && b then c else 42
LTXL Scope Rules

1. The scope of a variable is all subsequent definitions and the body of the `let`-expression in which the definition of the variable occurs. A variable is *not* in scope in the RHS of its own definition.
LTXL Scope Rules

1. The scope of a variable is all subsequent definitions and the body of the `let`-expression in which the definition of the variable occurs. A variable is *not* in scope in the RHS of its own definition.

2. A definition of a variable hides, for the extent of its scope, any definition of a variable with the same name from an outer `let`-expression.
1. The scope of a variable is all subsequent definitions and the body of the `let`-expression in which the definition of the variable occurs. A variable is *not* in scope in the RHS of its own definition.

2. A definition of a variable hides, for the extent of its scope, any definition of a variable with the same name from an outer `let`-expression.

3. At most one definition may be given for a variable in the list of definitions of a `let`-expression.
LTXL Example 1 (again)

Which scope rules are used where?

let

  int a = 10;
  bool b = a < 2
in let

  int c = a * 10;
  bool a = a == 42;
  int d = if a then 1 else 2
in

  if a && b then c else 42
Which scope rules are used where?

```plaintext
let
  int a = 10;
  bool b = a < 2
in let
  int c = a * 10;
  bool a = a == 42;
  int d = if a then 1 else 2
in
  if a && b then c else 42
```
LTXL Example 1 (again)

Which scope rules are used where?

```plaintext
let
  int a = 10;
  bool b = a < 2
in let
  int c = a * 10;
  bool a = a == 42;
  int d = if a then 1 else 2
in
  if a && b then c else 42
```
LTXL Example 1 (again)

Which scope rules are used where?

let

\[ \text{int } a = 10; \]
\[ \text{bool } b = a < 2 \]

in

let

\[ \text{int } c = a \times 10; \]
\[ \text{bool } a = a == 42; \]
\[ \text{int } d = \text{if } a \text{ then } 1 \text{ else } 2 \]

in

\[ \text{if } a \text{ && } b \text{ then } c \text{ else } 42 \]
LTXL Example 1 (again)

Which scope rules are used where?

```plaintext
let
  int a = 10;
  bool b = a < 2
in let
  int c = a * 10;
  bool a = a == 42;
  int d = if a then 1 else 2
in
  if a && b then c else 42
```
LTXL Example 1 (again)

Which scope rules are used where?

```plaintext
let
  int a = 10;
  bool b = a < 2
in let
  int c = a * 10;
  bool a = a == 42;
  int d = if a then 1 else 2
in
  if a && b then c else 42
```
LTXL Example 1 (again)

Which scope rules are used where?

```haskell
let
  int a = 10;
  bool b = a < 2
in let
  int c = a * 10;
  bool a = a == 42;
  int d = if a then 1 else 2
in
  if a && b then c else 42
```
LTXL Example 1 (again)

Which scope rules are used where?

```markdown
let
  int a = 10;
  bool b = a < 2
in let
  int c = a * 10;
  bool a = a == 42;
  int d = if a then 1 else 2
in
  if a && b then c else 42
```
What about this LTXL example?

```plaintext
let
    int a = 1;
    int b = c * 2;
    bool a = a < 1
in
    a + b
```
What about this LTXL example?

```kotlin
let

int a = 1;
int b = c * 2;
bool a = a < 1
in

a + b
```
What about this LTXL example?

```plaintext
let
    int a = 1;
    int b = c * 2;
    bool a = a < 1
in
    a + b
```

- Not defined: `b` is defined as `c * 2`, but `c` is not defined.
- Defined twice at same scope level: `a` is defined in the `let` block and again later in the `in` block, both with the same value `a < 1`. The second definition is redundant.
The following Haskell data types are used to represent LTXL programs.

```haskell
type Id = String

data Type = IntType
    | BoolType
    | UnknownType
```
data UnOp = Not | Neg

data BinOp = Or |
| And |
| Less |
| Equal |
| Greater |
| Plus |
| Minus |
| Times |
| Divide |
Exp is a *parameterized* type. The *type parameter* \(a\) allows variables to be *annotated* with an attribute of type \(a\). This facility is used by the identification function.

```haskell
data Exp a =
  LitInt Int
| Var Id a
| UnOpApp UnOp (Exp a)
| BinOpApp BinOp (Exp a) (Exp a)
| If (Exp a) (Exp a) (Exp a)
| Let [(Id, Type, Exp a)] (Exp a)
```
Example: The LTXL program

\[
\text{let int } x = 7 \text{ in } x + 35
\]

would be represented like this \textit{before} identification (type \texttt{Exp ()}): 

\[
\text{Let } \left[ ("x", \text{IntType}, \text{LitInt 7}) \right] \\
\text{(BinOpApp Plus} \\
\text{(Var "x" (})) \\
\text{(LitInt 35))}
\]

\textit{After} identification, type will be \texttt{Exp Attr.}
An association list is used to represent the environment/symbol table to keep things simple.
• An **association list** is used to represent the environment/symbol table to keep things simple.
• By ** prepending ** new declarations to the list, and searching from the beginning, we will always find an identifier in the closest containing scope. For example:

\[
\text{lookup } "x" \ [ ("x", a_1), ("y", a_2), ("x", a_3) ] \\
\Rightarrow a_1
\]
An *association list* is used to represent the environment/symbol table to keep things simple.

By *prepending* new declarations to the list, and searching from the beginning, we will always find an identifier in the closest containing scope. For example:

```plaintext
lookup "x"  [("x", a₁), ("y", a₂), ("x", a₃)]
⇒  a₁
```

No need for a "close scope" operation. We are in a pure functional setting ⇒ persistent data.
The environment associates identifiers with **variable attributes**. Our attributes are the **scope level** and the **declared type**.

\[
\text{type Attr} = (\text{Int}, \text{Type})
\]
The environment associates identifiers with **variable attributes**. Our attributes are the **scope level** and the **declared type**.

```plaintext
type Attr = (Int, Type)
```

The environment is just an association list:

```plaintext
type Env = [(Id, Attr)]
```
The environment associates identifiers with **variable attributes**. Our attributes are the **scope level** and the **declared type**.

\[
\text{type } \text{Attr} = (\text{Int}, \text{Type})
\]

The environment is just an association list:

\[
\text{type } \text{Env} = [(\text{Id}, \text{Attr})]
\]

**Note:** our environment does *not* store variable **definitions**.
Example:

```ocaml
let
  int a = 10;  (1)
  int b = a + 42
in let
  bool a = b < 20  (2)
in
  if a then b else 13
```

Env. after (1): `[ ("a", (1, IntType)) ]`
Env. after (2): `[ ("a", (2, BoolType)), ("b", (1, IntType)), ("a", (1, IntType)) ]`
**LTXL Environment (4)**

`enterVar` inserts a variable at the given scope level and of the given type into an environment.

```
enterVar :: Id -> Int -> Type -> Env
           -> Either Env ErrorMsg
```
enterVar inserts a variable at the given scope level and of the given type into an environment.

- Check that no variable with same name has been defined at the same scope level.

\[
\text{enterVar} :: \text{Id} \rightarrow \text{Int} \rightarrow \text{Type} \rightarrow \text{Env} \\
\rightarrow \text{Either Env ErrorMsg}
\]
enterVar inserts a variable at the given scope level and of the given type into an environment.

- Check that no variable with same name has been defined at the same scope level.
- If not, the new variable is entered, and the *resulting environment* is returned.

```
enterVar :: Id -> Int -> Type -> Env
          -> Either Env ErrorMsg
```
enterVar inserts a variable at the given scope level and of the given type into an environment.

- Check that no variable with same name has been defined at the same scope level.
- If not, the new variable is entered, and the resulting environment is returned.
- Otherwise an error message is returned.

enterVar :: Id -> Int -> Type -> Env
-> Either Env ErrorMsg
Aside: The Haskell Type `Either`

The standard Haskell type `Either` comes in handy when one needs to represent a value that has one of two possible types:

```haskell
data Either a b = Left a | Right b
```

A typical example is when a function needs to return one of two kinds of results:

```haskell
foo :: Int -> Either Bool String
foo x | x < 100   = Left (x < 0)
     | otherwise = Right "Too big"
```
LTXL Environment (5)

enterVar i l t env

| not (isDefined i l env)
  = Left ((i, (l, t)) : env)
enterVar \( i \) \( l \) \( t \) env

\[
| \text{not (isDefined } i \ l \text{ env)} \\
| = \text{Left } ((i,(l,t)) : \text{env})
\]
enterVar i l t env
    | not (isDefined i l env)
    = Left ((i, (l, t)) : env)
    | otherwise
    = Right (i ++ " already defined.")
```haskell
enterVar i l t env
    | not (isDefined i l env)
    = Left ((i, (l, t)) : env)
    | otherwise
    = Right (i ++ " already defined.")
where
    isDefined i l [] = False
```

**decl. prepended**
enterVar i l t env
| not (isDefined i l env)
  = Left ((i, (l, t)) : env)
| otherwise
  = Right (i ++ " already defined.")

where

isDefined i l [] = False
isDefined i l ((i’, (l’, _)) : env)
  | l < l’ = error "Should not happen!"
**LTXL Environment (5)**

```haskell
txtl
enterVar i l t env
    | not (isDefined i l env)
    = Left ((i, (l, t)) : env)
    | otherwise
    = Right (i ++ " already defined.")

where
isDefined i l [] = False
isDefined i l ((i', (l', _)) : env)
    | l < l' = error "Should not happen!"
    | l > l' = False
```
enterVar i l t env
  | not (isDefined i l env)
  = Left ((i, (l, t)) : env)
  | otherwise
  = Right (i ++ " already defined.")

where

isDefined i l [] = False
isDefined i l ((i', (l', _)) : env)
  | l < l' = error "Should not happen!"
  | l > l' = False
  | i == i' = True
LTXL Environment (5)

```haskell
enterVar i l t env
  | not (isDefined i l env)
  = Left ((i,(l,t)) : env)
  | otherwise
  = Right (i ++ " already defined.")

where

isDefined i l [] = False
isDefined i l ((i',(l',_)) : env)
  | l < l' = error "Should not happen!"
  | l > l' = False
  | i == i' = True
  | otherwise = isDefined i l env
```

decl. prepended
Let

\[
\text{env} = [("y", (2, \text{IntType})),
\quad\text{("x", (1, \text{IntType}))}]
\]

Then:
Let

\[ \text{env} = \left[ ("y", (2, \text{IntType})), ("x", (1, \text{IntType})) \right] \]

Then:

\[
\begin{align*}
\text{enterVar } "x" & \ 2 \ \text{BoolType} \ \text{env} \\
\Rightarrow \text{Left} & \ [ ("x", (2, \text{BoolType})), ("y", (2, \text{IntType})), ("x", (1, \text{IntType})) ]
\end{align*}
\]
Let

\[
\text{env} = [ ("y", (2, \text{IntType})),
             ("x", (1, \text{IntType})) ]
\]

Then:

\[
\text{enterVar "x" 2 \text{BoolType env}}
\Rightarrow \text{Left } [ ("x", (2, \text{BoolType})),
                  ("y", (2, \text{IntType})),
                  ("x", (1, \text{IntType})) ]
\]

\[
\text{enterVar "y" 2 \text{BoolType env}}
\Rightarrow \text{Right } "y \text{ already defined.}"
\]
lookupVar looks up a variable in an environment.

lookupVar :: Id -> Env

-> Either Attr ErrorMsg
lookupVar looks up a variable in an environment.

- Returns **variable attributes** if found.

\[
\text{lookupVar} :: \text{Id} \rightarrow \text{Env} \rightarrow \text{Either Attr ErrorMsg}
\]
lookupVar looks up a variable in an environment.

- Returns *variable attributes* if found.
- Returns an *error message* otherwise.

\[
\text{lookupVar} :: \text{Id} \rightarrow \text{Env} \\
\rightarrow \text{Either} \quad \text{Attr} \quad \text{ErrorMsg}
\]
lookupVar looks up a variable in an environment.

• Returns variable attributes if found.
• Returns an error message otherwise.

lookupVar :: Id -> Env
  -> Either Attr ErrorMsg
lookupVar i [] = Right (i ++ " not defined.")
lookupVar looks up a variable in an environment.

- Returns **variable attributes** if found.
- Returns an **error message** otherwise.

\[
\text{lookupVar} :: \text{Id} \rightarrow \text{Env} \\
\quad \rightarrow \text{Either Attr ErrorMsg}
\]

\[
\text{lookupVar}\ i\ [] = \text{Right}\ (i\ +\ "\ not defined."
\]

\[
\text{lookupVar}\ i\ ((i',a) : \text{env}) \\
\quad | \ i\ ==\ i'\ =\ \text{Left}\ a
\]
lookupVar looks up a variable in an environment.

- Returns **variable attributes** if found.
- Returns an **error message** otherwise.

```haskell
lookupVar :: Id -> Env
            -> Either Attr ErrorMsg
lookupVar i [] = Right (i ++ " not defined.")
lookupVar i ((i',a) : env)
    | i == i' = Left a
    | otherwise = lookupVar i env
```
lookupVar looks up a variable in an environment.

- Returns **variable attributes** if found.
- Returns an **error message** otherwise.

```haskell
lookupVar :: Id -> Env
           -> Either Attr ErrorMsg
lookupVar i [] = Right (i ++ " not defined.")
lookupVar i ((i',a) : env)
    | i == i'   = Left a
    | otherwise = lookupVar i env
```

Note: returns first decl. found, later decls. hidden!
Let

$$env = [ ("x", (2, \text{BoolType})),
         ("y", (2, \text{IntType})),
         ("x", (1, \text{IntType})) ]$$

Then:
Let

\[
env = [("x", (2, \text{BoolType}))],
      ("y", (2, \text{IntType})),
      ("x", (1, \text{IntType}))]
\]

Then:

\[
\text{lookupVar} \ "y" \ env \\
\Rightarrow \text{Left} \ (2, \text{IntType})
\]
Let

\[ \text{env} = \left[ (\text{"x"}, (2, \text{BoolType})), \right. \]
\[ \left. (\text{"y"}, (2, \text{IntType})), \right. \]
\[ \left. (\text{"x"}, (1, \text{IntType})) \right] \]

Then:

\[ \text{lookupVar "y" env} \Rightarrow \text{Left (2, IntType)} \]
\[ \text{lookupVar "x" env} \Rightarrow \text{Left (2, BoolType)} \]
Let

$$env = [("x", (2,\text{BoolType})),$$

$$\quad ("y", (2,\text{IntType})),$$

$$\quad ("x", (1,\text{IntType})))]$$

Then:

$$\text{lookupVar } "y" \text{ env } \Rightarrow \text{Left } (2,\text{IntType})$$
$$\text{lookupVar } "x" \text{ env } \Rightarrow \text{Left } (2,\text{BoolType})$$
$$\text{lookupVar } "z" \text{ env } \Rightarrow \text{Right } "z \text{ not defined.}"$$
LTXL Identification (1)

Goals of LTXL identification phase:

\[
\text{identification} :: \\
\quad \text{Exp} () \rightarrow (\text{Exp Attr}, \ [\text{ErrorMsg}])
\]
Goals of LTXL identification phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration.

```
identification ::
    Exp () -> (Exp Attr, [ErrorMsg])
```
LTXL Identification (1)

Goals of LTXL identification phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration.
  I.e., map unannotated AST $\text{Exp}()$ to annotated AST $\text{Exp Attr}$.

\[
\text{identification} :: \\
\text{Exp}() \rightarrow (\text{Exp Attr}, [\text{ErrorMsg}])
\]
LTXL Identification (1)

Goals of LTXL identification phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration. I.e., map unannotated AST `Exp ()` to annotated AST `Exp Attr`.
- Report conflicting variable definitions and undefined variables.

\[
\text{identification} :: \quad \text{Exp} () \rightarrow (\text{Exp Attr}, [\text{ErrorMsg}])
\]
Example: Before Identification

Let ["x", IntType, LitInt 7]
(BinOpApp Plus
  (Var "x" ())
  (LitInt 35))
Example: Before Identification

Let ["x", IntType, LitInt 7]]
(BinOpApp Plus
 (Var "x" (1, IntType))
 (LitInt 35))

After identification:

Let ["x", IntType, LitInt 7]]
(BinOpApp Plus
 (Var "x" (1, IntType))
 (LitInt 35))
Main identification function:

\[
\text{identification} :: \text{Exp} () \\
\rightarrow (\text{Exp Attr}, \text{[ErrorMsg]}) \\
\text{identification } e = \text{identAux 0 emptyEnv } e
\]

Type signature for auxiliary identification function:

\[
\text{identAux} :: \text{Int} \rightarrow \text{Env} \rightarrow \text{Exp} () \\
\rightarrow (\text{Exp Attr}, \text{[ErrorMsg]})
\]
Variable case:

\[
\text{identAux} \, l \, \text{env} \, (\text{Var} \, i \, \_ \,) = \\
\text{case} \, \text{lookupVar} \, i \, \text{env} \, \text{of} \\
\text{Left} \, a \rightarrow (\text{Var} \, i \, a, \, []) \\
\text{Right} \, m \rightarrow (\text{Var} \, i \, (0, \, \text{UnknownType}), \, [m])
\]
Binary operator application (typical recursive case):

\[
\text{identAux} \ l \ \text{env} \ (\text{BinOpApp} \ \text{op} \ e1 \ e2) = \\
(\text{BinOpApp} \ \text{op} \ e1' \ e2', \ ms1 ++ ms2)
\]

where

\[
(e1', \ ms1) = \text{identAux} \ l \ \text{env} \ e1 \\
(e2', \ ms2) = \text{identAux} \ l \ \text{env} \ e2
\]
Reminder: LTXL scope rules

1. The scope of a variable is all subsequent definitions and the body of the let-expression in which the definition of the variable occurs. A variable is not in scope in the RHS of its definition.

2. A definition of a variable hides, for the extent of its scope, any definition of a variable with the same name from an outer let-expression.

3. At most one definition may be given for a variable in the list of definitions of a let-expression.
Block of definitions (**let**):

\[
\text{identAux } l \ env \ (\text{Let } ds \ e) = \\
(\text{Let } ds' \ e', \ ms1 ++ ms2)
\]

where

\[
(e', ms2) = \text{identAux } l' \ env' \ e
\]
Block of definitions (**let**):

\[
\text{identAux } l \text{ env } (\text{Let } ds \ e) = \\
(\text{Let } ds' \ e', \ ms1 ++ ms2) \\
\text{where} \\
\quad l' = l + 1 \\
\quad (e', ms2) = \text{identAux } l' \text{ env'} \ e
\]
Block of definitions (**let**):

\[
\text{identAux } l \ \text{env } (\text{Let } ds \ e) = \\
(\text{Let } ds’ \ e’, \ ms1 ++ ms2)
\]

where

\[
l’ = l + 1
\]

\[
(ds’, \ env’, \ ms1) = \text{identDefs } l’ \ \text{env } ds
\]

\[
(e’, \ ms2) = \text{identAux } l’ \ \text{env’ } e
\]
Block of definitions (let):

\[ \text{identAux } l \ env \ (\text{Let } ds \ e) = \]
\[ (\text{Let } ds' \ e', \ ms1 ++ ms2) \]

where

\[ l' = l + 1 \]
\[ (ds', \ env', \ ms1) = \text{identDefs } l' \ env \ ds \]
\[ (e', \ ms2) = \text{identAux } l' \ env' \ e \]

Note that \text{identDefs} returns an \textit{updated environment} to be \textit{used} when checking the \textit{body} of the let (rule 1).
identDefs l env [] = ([], env, [])
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
LTXL Identification (8)

\[
\text{identDefs} \ l \ \text{env} \ [\ ] = ([], \ \text{env}, \ [\ ]) \\
\text{identDefs} \ l \ \text{env} \ ((i,t,e) : \ ds) = \\
\hspace{1em} ((i,t,e') : \ ds', \ \text{env''}, \ ms1++ms2++ms3) \\
\text{where} \\
\hspace{1em} (e', \ ms1) = \text{identAux} \ l \ \text{env} \ e
\]

\(i \ \text{not in scope (rule 1)}\)
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
where
  (e', ms1) = identAux l env e
  (env', ms2) =
  case enterVar i l t env of
    Left env' -> (env', [])
    Right m  -> (env, [m])
identDefs l env [] = ([], env, [])
identDefs l env ((i, t, e) : ds) =
  ((i, t, e') : ds', env'', ms1++ms2++ms3)
where
  i not in scope (rule 1)
  (e', ms1) = identAux l env e
  (env', ms2) = impl./checks rules 2 & 3
  case enterVar i l t env of
    Left env' -> (env', [])
    Right m   -> (env, [m])
  (ds', env'', ms3) = i in scope (rule 1)
  identDefs l env' ds
Efficient Symbol Table Implementation

Lists don’t make for very efficient symbol tables. Insertion (at head) is fast, $O(1)$, but lookup is $O(n)$, where $n$ is the number of symbols.

Some more efficient options:
Efficient Symbol Table Implementation

Lists don’t make for very efficient symbol tables. Insertion (at head) is fast, $O(1)$, but lookup is $O(n)$, where $n$ is the number of symbols.

Some more efficient options:

- Balanced trees:
Lists don’t make for very efficient symbol tables. Insertion (at head) is fast, $O(1)$, but lookup is $O(n)$, where $n$ is the number of symbols.

Some more efficient options:

- Balanced trees:
  - Insertion and lookup are both $O(\log n)$. 
Efficient Symbol Table Implementation

Lists don’t make for very efficient symbol tables. Insertion (at head) is fast, $O(1)$, but lookup is $O(n)$, where $n$ is the number of symbols.

Some more efficient options:

- Balanced trees:
  - Insertion and lookup are both $O(\log n)$.
  - One way of handling nested scopes would be a stack of trees.
Efficient Symbol Table Implementation

- Hash tables:
Efficient Symbol Table Implementation

- Hash tables:
  - Insertion and lookup are both $O(1)$ as long as the ratio between the number of symbols and the hash table size is kept below a small constant factor.
Efficient Symbol Table Implementation

- Hash tables:
  - Insertion and lookup are both $O(1)$ as long as the ratio between the number of symbols and the hash table size is kept below a small constant factor.
  - Algorithms such as **linear hashing** allows the table to grow and shrink gracefully, guaranteeing near optimal performance.
Efficient Symbol Table Implementation

- Hash tables:
  - Insertion and lookup are both $O(1)$ as long as the ratio between the number of symbols and the hash table size is kept below a small constant factor.
  - Algorithms such as linear hashing allows the table to grow and shrink gracefully, guaranteeing near optimal performance.

See e.g. Aho, Sethi, Ullman (1986) for further details.