Symbol tables

Symbol tables

Major data structures in a compiler after syntax trees

Symbol tables

Major data structures in a compiler after syntax trees

Major inherited attributes

Symbol tables

Major data structures in a compiler after syntax trees

Major inherited attributes

Typical dictionary data structures

Symbol tables

Major data structures in a compiler after syntax trees

Major inherited attributes

Typical dictionary data structures

Main operations:

- *insert*
- *lookup*
- *delete*

Formal Languages and Compilers Paola Quaglia, 2022

Symbol tables

Major data structures in a compiler after syntax trees

Major inherited attributes

Typical dictionary data structures

Main operations:

- *insert*
- *lookup*
- *delete*

Typical implementations include: linear lists, search trees, hash tables

Symbol tables

Major data structures in a compiler after syntax trees

Major inherited attributes

Typical dictionary data structures

Main operations:

- *insert*
- *lookup*
- *delete*

Typical implementations include: linear lists, search trees, hash tables

Hash tables often the best choice: the main operations can be performed almost in linear time

Hash tables

COLLISION RESOLUTION

Hash tables
COLLISION RESOLUTION

Open addressing: Each bucket has enough space for a single item, insert colliding items in successive buckets

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Hash tables
COLLISION RESOLUTION

Open addressing: Each bucket has enough space for a single item, insert colliding items in successive buckets

Contents of the table limited by the size of the bucket array

Hash tables
COLLISION RESOLUTION

Open addressing: Each bucket has enough space for a single item, insert colliding items in successive buckets

Contents of the table limited by the size of the bucket array

Poor performance

Hash tables
COLLISION RESOLUTION

Hash tables

COLLISION RESOLUTION

Separate chaining: Each bucket is a linear list, and hash collisions are resolved by inserting the new item into the list

Formal Languages and Compilers Paola Quaglia, 2022

Hash tables
COLLISION RESOLUTION
Separate chaining: Each bucket is a linear list, and hash collisions are resolved by inserting the new item into the list

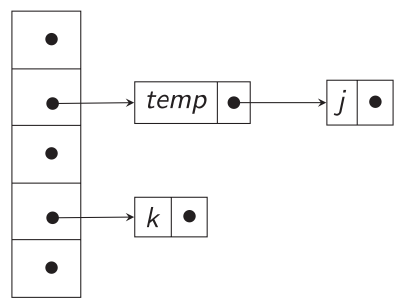
For example, if *temp* and *j* have the same hash value, then:

Hash tables

COLLISION RESOLUTION

Separate chaining: Each bucket is a linear list, and hash collisions are resolved by inserting the new item into the list

For example, if *temp* and *j* have the same hash value, then:



Hash tables

HASH FUNCTIONS

<u>Goal</u>: Convert a character string (the name of the identifier) into an integer in the range $0 \ldots size - 1$ where *size* is the size of the table and of the bucket array

Hash tables

HASH FUNCTIONS

Goal: Convert a character string (the name of the identifier) into an integer in the range $0 \ldots size - 1$ where *size* is the size of the table and of the bucket array

Tactics:

Formal Languages and Compilers Paola Quaglia, 2022

Hash tables

HASH FUNCTIONS

Goal: Convert a character string (the name of the identifier) into an integer in the range $0 \ldots size - 1$ where *size* is the size of the table and of the bucket array

Tactics:

- Convert each character to a nonnegative integer, usually by built-in conversion mechanisms of the compiler implementation language

Hash tables

HASH FUNCTIONS

Goal: Convert a character string (the name of the identifier) into an integer in the range $0 \ldots size - 1$ where *size* is the size of the table and of the bucket array

Tactics:

- Convert each character to a nonnegative integer, usually by built-in conversion mechanisms of the compiler implementation language
- Apply an adequate hash function h (often variable names assigned in groups: $temp1, temp2, \ldots$; take all the characters to avoid too many collisions)

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Hash tables

HASH FUNCTIONS

Goal: Convert a character string (the name of the identifier) into an integer in the range $0 \ldots size - 1$ where *size* is the size of the table and of the bucket array

Tactics:

- Convert each character to a nonnegative integer, usually by built-in conversion mechanisms of the compiler implementation language
- Apply an adequate hash function h (often variable names assigned in groups: $temp1, temp2, \ldots$; take all the characters to avoid too many collisions)
- Typical good choice: $h = (\Sigma_{i=1}^{n} \phi^{n-i} c_i) \bmod size$

Hash tables

HASH FUNCTIONS

Goal: Convert a character string (the name of the identifier) into an integer in the range $0 \ldots size - 1$ where *size* is the size of the table and of the bucket array

Tactics:

- Convert each character to a nonnegative integer, usually by built-in conversion mechanisms of the compiler implementation language
- Apply an adequate hash function *h* (often variable names assigned in groups: $temp1, temp2, \ldots$; take all the characters to avoid too many collisions)
- Typical good choice: $h = (\Sigma_{i=1}^{n} \phi^{n-i} c_i) \bmod size$
- Where c_i is the numeric value of the ith character, and ϕ is a power of 2, so that multiplication can be a shift

Hash tables

SCOPE

Hash tables

SCOPE

Names have a declaration (constant declarations, variable declarations, function declarations)

Formal Languages and Compilers Paola Quaglia, 2022

Hash tables
SCOPE
Names have a declaration (constant declarations, variable declarations, function declarations)

- Local declarations: Names have a limited scope (are visible in a portion of the program)

Hash tables

SCOPE

Names have a declaration (constant declarations, variable declarations, function declarations)

- Local declarations: Names have a limited scope (are visible in a portion of the program)
- Global declarations: Names are visible in the entire program

Formal Languages and Compilers Paola Quaglia, 2022

Hash tables

SCOPE

Names have a declaration (constant declarations, variable declarations, function declarations)

- Local declarations: Names have a limited scope (are visible in a portion of the program)
- Global declarations: Names are visible in the entire program

The same name can be declared several times, then the use of the name refers to its closest declaration

Hash tables

SCOPE

Names have a declaration (constant declarations, variable declarations, function declarations)

- Local declarations: Names have a limited scope (are visible in a portion of the program)
- Global declarations: Names are visible in the entire program

The same name can be declared several times, then the use of the name refers to its closest declaration

The **scope** of a declaration is a sub-tree of the syntax tree

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Hash tables

SCOPE

Names have a declaration (constant declarations, variable declarations, function declarations)

- Local declarations: Names have a limited scope (are visible in a portion of the program)
- Global declarations: Names are visible in the entire program

The same name can be declared several times, then the use of the name refers to its closest declaration

The **scope** of a declaration is a sub-tree of the syntax tree

Nested declarations give rise to scopes that are nested sub-trees of the syntax tree

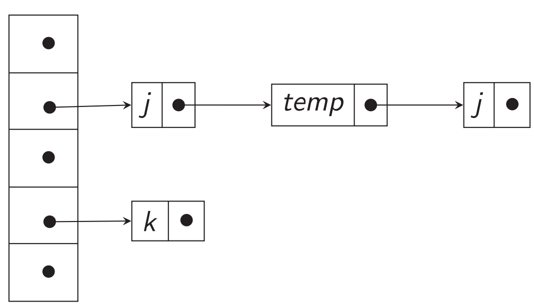
Hash tables

NESTED SCOPE

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

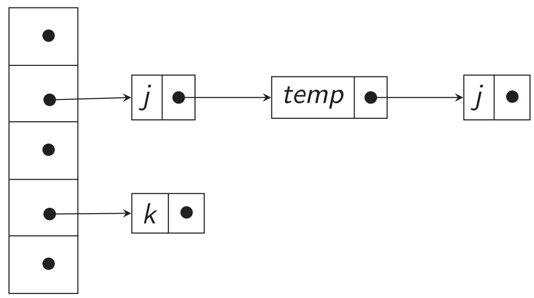
Hash tables

NESTED SCOPE



80 / 97

Hash tables
NESTED SCOPE

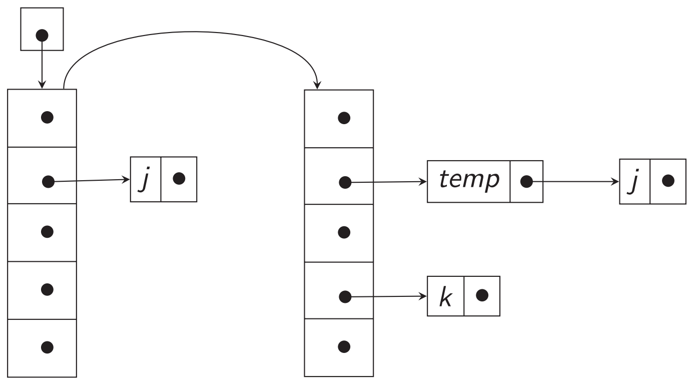


One solution is to manage the hash table in a stack-like manner

Hash tables
NESTED SCOPE

Hash tables

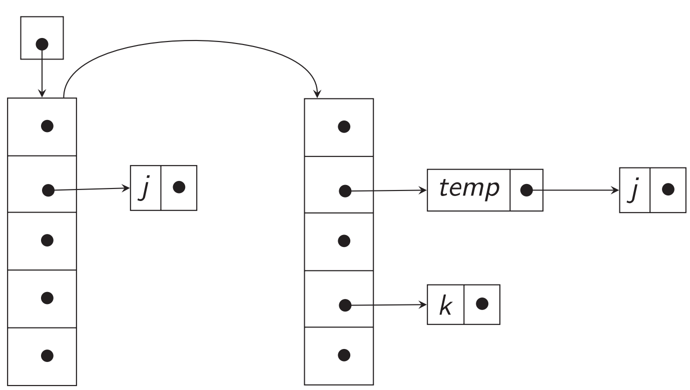
NESTED SCOPE



Formal Languages and Compilers Paola Quaglia, 2022

Hash tables

NESTED SCOPE



Another solution is to build a new symbol table for each scope and to link the tables from inner to outer scopes together

Insert types in the symbol table
TRAINING

Formal Languages and Compilers Paola Quaglia, 2022

Insert types in the symbol table
TRAINING

Design semantic actions to get an SDD for adding the type of the declared identifiers (*int* or *float*) to their entries in the symbol table

82 / 97

Insert types in the symbol table
TRAINING

Design semantic actions to get an SDD for adding the type of the declared identifiers (*int* or *float*) to their entries in the symbol table

$$\begin{array}{l} D \rightarrow TL \\ T \rightarrow int \\ T \rightarrow float \\ L \rightarrow L_1, id \\ L \rightarrow id \end{array}$$

Formal Languages and Compilers Paola Quaglia, 2022

Insert types in the symbol table
TRAINING

Design semantic actions to get an SDD for adding the type of the declared identifiers (*int* or *float*) to their entries in the symbol table

$$\begin{array}{l} D \rightarrow TL \\ T \rightarrow int \\ T \rightarrow float \\ L \rightarrow L_1, id \\ L \rightarrow id \end{array}$$

To update the table, use function $addtype(table\_entry, type\_instance)$ where $type\_instance$ is either *int* or *float*

Insert types in the symbol table
TRAINING

$$
\begin{array}{ll}
D \rightarrow TL & \{L.i = T.type\} \\
T \rightarrow int & \{T.type = int\} \\
T \rightarrow float & \{T.type = float\} \\
L \rightarrow L_1, id & \{L_1.i = L.i;\ addtype(id.entry, L.i)\} \\
L \rightarrow id & \{addtype(id.entry, L.i)\}
\end{array}
$$

Symbol Tables

Interpreters

Interpreters

Once we have abstract syntax trees and symbol tables we can design an interpreter

Interpreters

Once we have abstract syntax trees and symbol tables we can design an interpreter

The interpreter typically consists of one function per syntactic category (non-terminal).

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters

Once we have abstract syntax trees and symbol tables we can design an interpreter

The interpreter typically consists of one function per syntactic category (non-terminal).

Each function takes as arguments an abstract syntax tree, possibly symbol tables, and possibly some other argument, and returns some results.

85 / 97

Interpreters

Once we have abstract syntax trees and symbol tables we can design an interpreter

The interpreter typically consists of one function per syntactic category (non-terminal).

Each function takes as arguments an abstract syntax tree, possibly symbol tables, and possibly some other argument, and returns some results.

Functions can be implemented in any executable language: either machine language or a language compiled to machine language

Interpreters

Once we have abstract syntax trees and symbol tables we can design an interpreter

The interpreter typically consists of one function per syntactic category (non-terminal).

Each function takes as arguments an abstract syntax tree, possibly symbol tables, and possibly some other argument, and returns some results.

Functions can be implemented in any executable language: either machine language or a language compiled to machine language

"The Java source code is first compiled into a binary bytecode using Java compiler, then this bytecode runs on the JVM (Java Virtual Machine), which is a software based interpreter."

Interpreters

Once we have abstract syntax trees and symbol tables we can design an interpreter

The interpreter typically consists of one function per syntactic category (non-terminal).

Each function takes as arguments an abstract syntax tree, possibly symbol tables, and possibly some other argument, and returns some results.

Functions can be implemented in any executable language: either machine language or a language compiled to machine language

"The Java source code is first compiled into a binary bytecode using Java compiler, then this bytecode runs on the JVM (Java Virtual Machine), which is a software based interpreter."

Really?

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE

Ambiguous but good enough

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE

Ambiguous but good enough

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{FunS} \\
\mathit{FunS} & \rightarrow & \mathit{Fun} \mid \mathit{Fun}\ \mathit{FunS} \\
\mathit{Fun} & \rightarrow & \mathit{TypeId}(\mathit{TypeIdS}) = \mathit{Exp} \\
\mathit{TypeId} & \rightarrow & \mathit{int}\ \mathit{id} \mid \mathit{bool}\ \mathit{id} \\
\mathit{TypeIdS} & \rightarrow & \mathit{TypeId} \mid \mathit{TypeId}\ ,\ \mathit{TypeIdS} \\
\mathit{Exp} & \rightarrow & \mathit{num} \mid \mathit{id} \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} = \mathit{Exp} \\
 & & \mathit{if}\ \mathit{Exp}\ \mathit{then}\ \mathit{Exp}\ \mathit{else}\ \mathit{Exp} \mid \\
 & & \mathit{id}(\mathit{ExpS}) \mid \\
 & & \mathit{let}\ \mathit{id} = \mathit{Exp}\ \mathit{in}\ \mathit{Exp} \\
\mathit{ExpS} & \rightarrow & \mathit{Exp} \mid \mathit{Exp}\ ,\ \mathit{ExpS}
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{FunS} \\
\mathit{FunS} & \rightarrow & \mathit{Fun} \mid \mathit{Fun}\ \mathit{FunS}
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$\begin{array}{lcl} \textit{Program} & \to & \textit{FunS} \\ \textit{FunS} & \to & \textit{Fun} \mid \textit{Fun FunS} \end{array}$$

A program is a list of functions

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$\begin{array}{lcl} \textit{Program} & \to & \textit{FunS} \\ \textit{FunS} & \to & \textit{Fun} \mid \textit{Fun FunS} \end{array}$$

A program is a list of functions

Assumptions:

- No function can be declared more than once
- There are separate symbol tables for function and variable identifiers
- The program contains a function called *main* with an integer argument and returning an integer
- The execution of the program starts invoking *main*

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lcl}
\mathit{Fun} & \rightarrow & \mathit{TypeId}(\mathit{TypeIdS}) = \mathit{Exp} \\
\mathit{TypeId} & \rightarrow & \mathit{int\ id} \mid \mathit{bool\ id} \\
\mathit{TypeIdS} & \rightarrow & \mathit{TypeId} \mid \mathit{TypeId}\ ,\ \mathit{TypeIdS}
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lcl}
\mathit{Fun} & \rightarrow & \mathit{TypeId}(\mathit{TypeIdS}) = \mathit{Exp} \\
\mathit{TypeId} & \rightarrow & \mathit{int\ id} \mid \mathit{bool\ id} \\
\mathit{TypeIdS} & \rightarrow & \mathit{TypeId} \mid \mathit{TypeId}\ ,\ \mathit{TypeIdS}
\end{array}
$$

Each function has a result type, and types and names of its arguments.

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lcl}
\mathit{Fun} & \rightarrow & \mathit{TypeId}(\mathit{TypeIdS}) = \mathit{Exp} \\
\mathit{TypeId} & \rightarrow & \mathit{int\ id} \mid \mathit{bool\ id} \\
\mathit{TypeIdS} & \rightarrow & \mathit{TypeId} \mid \mathit{TypeId}\ ,\ \mathit{TypeIdS}
\end{array}
$$

Each function has a result type, and types and names of its arguments.

The body of a function is an expression

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lcll}
Exp & \rightarrow & num & \%\ \text{number} \\
 & & \mid id & \%\ \text{variable} \\
 & & \mid Exp + Exp & \%\ \text{sum} \\
 & & \mid Exp = Exp & \%\ \text{comparison} \\
 & & \mid if\ Exp\ then\ Exp\ else\ Exp & \%\ \text{conditional} \\
 & & \mid id(ExpS) & \%\ \text{function call} \\
 & & \mid let\ id = Exp\ in\ Exp & \%\ \text{exp. with local declaration} \\
ExpS & \rightarrow & Exp & \\
 & & \mid Exp\ ,\ ExpS &
\end{array}
$$

89 / 97

Interpreters
EXAMPLE FUNCTIONAL LANGUAGE, ctd

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters
EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{eval\_ExpS}(\mathit{ExpS}, \mathit{vtab}, \mathit{ftab}) = \texttt{case}\ \mathit{ExpS}\ \texttt{of}} \\
\mathit{Exp} & : & [\,\mathit{eval\_Exp}(\mathit{Exp}, \mathit{vtab}, \mathit{ftab})\,] \\
\mathit{Exp}\ ,\ \mathit{ExpS} & : & \mathit{eval\_Exp}(\mathit{Exp}, \mathit{vtab}, \mathit{ftab}) :: \\
 & & \mathit{eval\_ExpS}(\mathit{ExpS}, \mathit{vtab}, \mathit{ftab})
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{eval\_ExpS}(\mathit{ExpS}, \mathit{vtab}, \mathit{ftab}) = \texttt{case}\ \mathit{ExpS}\ \texttt{of}} \\
\mathit{Exp} & : & [\, \mathit{eval\_Exp}(\mathit{Exp}, \mathit{vtab}, \mathit{ftab})\,] \\
\mathit{Exp}\, ,\ \mathit{ExpS} & : & \mathit{eval\_Exp}(\mathit{Exp}, \mathit{vtab}, \mathit{ftab}) :: \\
 & & \mathit{eval\_ExpS}(\mathit{ExpS}, \mathit{vtab}, \mathit{ftab})
\end{array}
$$

Where :: adds an element to the front of a list, *vtab* is the symbol table for variables, *ftab* is the table for functions

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lll}
\multicolumn{3}{l}{eval\_Exp(Exp, vtab, ftab) = \texttt{case}\ Exp\ \texttt{of}} \\
num & : & num.lexval \\
id & : & v = lookup(vtab, getname(id)) \\
 & & if\ v = unbound\ then\ error()\ else\ v \\
\ldots & & \\
id(ExpS) & : & def = lookup(ftab, getname(id)) \\
 & & if\ def = unbound \\
 & & then\ error() \\
 & & else \\
 & & \quad args = eval\_ExpS(ExpS, vtab, ftab) \\
 & & \quad eval\_Fun(def, args, ftab)
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lll}
\multicolumn{3}{l}{eval\_Exp(Exp, vtab, ftab) = \texttt{case}\ Exp\ \texttt{of}} \\
\ldots & & \\
\ldots & & \\
let\ id = Exp_1\ in\ Exp_2 & : & v_1 = eval\_Exp(Exp_1, vtab, ftab) \\
& & vtab' = insert(vtab, getname(id), v_1) \\
& & eval\_Exp(Exp_2, vtab', ftab)
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Function calls:

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Function calls:

- Check if number of actual parameters matches number of arguments

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Function calls:

- Check if number of actual parameters matches number of arguments
- Check if types of actual parameters match types of arguments

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Function calls:

- Check if number of actual parameters matches number of arguments
- Check if types of actual parameters match types of arguments
- Install a symbol table for the function name space

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Function calls:

- Check if number of actual parameters matches number of arguments
- Check if types of actual parameters match types of arguments
- Install a symbol table for the function name space
- Evaluate the function body using that table

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Function calls:

- Check if number of actual parameters matches number of arguments
- Check if types of actual parameters match types of arguments
- Install a symbol table for the function name space
- Evaluate the function body using that table

- Check if the result type matches the return type of the function

93 / 97

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters
EXAMPLE FUNCTIONAL LANGUAGE, ctd

$$
\begin{array}{lcl}
\mathit{Fun} & \rightarrow & \mathit{TypeId}(\mathit{TypeIdS}) = \mathit{Exp} \\
\mathit{TypeId} & \rightarrow & \mathit{int\ id} \mid \mathit{bool\ id} \\
\mathit{TypeIdS} & \rightarrow & \mathit{TypeId} \mid \mathit{TypeId}\ ,\ \mathit{TypeIdS}
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Interpreters
EXAMPLE FUNCTIONAL LANGUAGE, ctd

Building blocks for the definition of symbol tables

95 / 97

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Building blocks for the definition of symbol tables

$$TypeId \quad \rightarrow \quad int\ id \mid bool\ id$$

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Building blocks for the definition of symbol tables

$$TypeId \quad \rightarrow \quad int\ id \mid bool\ id$$

$$\begin{array}{lcl} \multicolumn{3}{l}{get\_TypeId(TypeId) = \texttt{case}\ TypeId\ \texttt{of}} \\ \quad int\ id & : & (getname(id), int) \\ \quad bool\ id & : & (getname(id), bool) \end{array}$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd

Building a symbol table (if due)

Interpreters
EXAMPLE FUNCTIONAL LANGUAGE, ctd

Building a symbol table (if due)

$$\mathit{TypeIdS} \quad \rightarrow \quad \mathit{TypeId} \mid \mathit{TypeId}\ ,\ \mathit{TypeIdS}$$

Formal Languages and Compilers Paola Quaglia, 2022

Interpreters
EXAMPLE FUNCTIONAL LANGUAGE, ctd

Building a symbol table (if due)

$$\mathit{TypeIdS} \quad \rightarrow \quad \mathit{TypeId} \mid \mathit{TypeId}\ ,\ \mathit{TypeIdS}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{build\_TypeIdS(\mathit{TypeIdS}, args) = \texttt{case}\,(\mathit{TypeIdS}, args)\,\texttt{of}} \\
(\mathit{TypeId}, v) & : & (x, t) = get\_TypeId(\mathit{TypeId}) \\
 & & \mathit{if}\ type(v) = t\ \mathit{then}\ insert(emptyT, x, v)\,\mathit{else}\ error() \\
(\mathit{TypeId}, v :: vs) & : & (x, t) = get\_TypeId(\mathit{TypeId}) \\
 & & vtab = build\_TypeIdS(\mathit{TypeIdS}, vs) \\
 & & \mathit{if}\ lookup(vtab, x) = unbound\ \mathit{and}\ type(v) = t \\
 & & \mathit{then}\ insert(vtab, x, v) \\
 & & \mathit{else}\ error() \\
\_ & : & error()
\end{array}
$$

Interpreters

EXAMPLE FUNCTIONAL LANGUAGE, ctd