4.4 Name analysis

Identifiers identify program entities in the program text (**statically**).

The **definition** of an identifier b introduces a **program entity** and **binds** it to the **identifier**.
The binding is valid in a certain range of the program text: the **scope of the definition**.

Name analysis task: Associate the **key of a program entity to each occurrence of an identifier** (consistent renaming) according to **scope rules** of the language.

Hiding rules for languages with nested structures:

- **Algol rule**: The definition of an identifier b is valid in the **whole smallest enclosing range**; but not in inner ranges that have a definition of b, too. (e. g. Algol 60, Pascal, Java, ... with additional rules)
- **C rule**: The definition of an identifier b is valid in the **smallest enclosing range from the position of the definition** to the end; but not in inner ranges that have another definition of b from the position of that definition. (e. g. C, C++, Java, ... with additional rules)

Ranges are syntactic constructs like **blocks, functions, modules, classes, packets**
- as defined for the particular language.

Implementation of name analysis:
Operations of the environment module are called in suitable tree contexts.

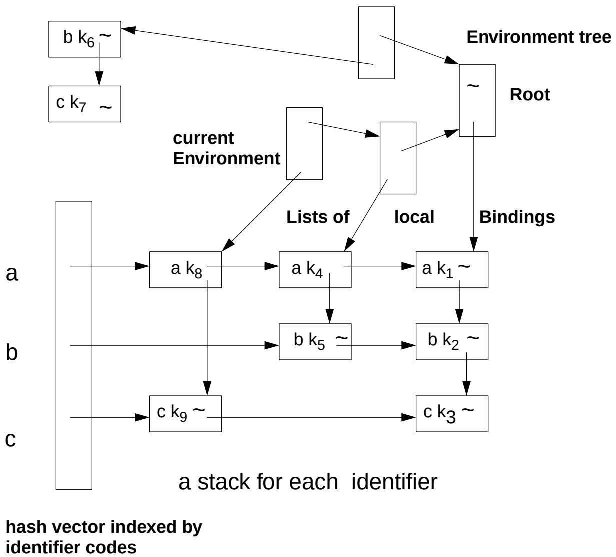
Environment module

Implements the abstract data type **Environment**:
hierarchically nested sets of **Binding**s **(identifier, environment, key)**

Functions:

NewEnv ()	creates a new Environment e, to be used as root of a hierarchy
NewScope (e_1)	creates a new Environment e_2 that is nested in e1. Each binding of e_1 is also a binding of e_2 if it is not hidden there.
BindIdn (e, id)	introduces a binding (id, e, k) if e has no binding for id; then k is a new key representing a new entity; in any case the result is the binding triple (id, e, k)
BindingInEnv (e, id)	yields a binding triple (id, e_1, k) of e or a surrounding environment of e; yields NoBinding if no such binding exists.
BindingInScope (e, id)	yields a binding triple (id, e, k) of e, if contained directly in e, NoBinding otherwise.

Data structure of the environment module



k_i: key of the defined entity

Environment operations in tree contexts

Operations in tree contexts and the order they are called model scope rules.

Root context:
```
Root.Env = NewEnv ();
```

Range context that may contain definitions:
```
Range.Env = NewScope (INCLUDING (Range.Env, Root.Env));
```
accesses the next enclosing Range or Root

defining occurrence of an identifier `IdDefScope`:
```
IdDefScope.Bind = BindIdn (INCLUDING Range.Env, IdDefScope.Symb);
```

applied occurrence of an identifier `IdUseEnv`:
```
IdUseEnv.Bind = BindingInEnv (INCLUDING Range.Env, IdUseEnv.Symb);
```

Preconditions for specific scope rules:
Algol rule: all **`BindIdn()`** of all surrounding ranges before any **`BindingInEnv()`**
C rule: **`BindIdn()`** and **`BindingInEnv()`** in textual order

The resulting **bindings are used for checks and transformations**, e. g.

- no applied occurrence without a valid defining occurrence,
- at most one definition for an identifier in a range,
- no applied occurrence before its defining occurrence (Pascal).

Semantic error handling

Design rules:

Error reports **related to the source code**:

- any explicit or implicit **requirement of the language definitions** needs to be checked by an operation in the tree
- check has to be associated to the **smallest relevant context** yields precise source position for the report; propagate information to that context if necessary
- **meaningfull error report**
- **different reports for different violations**, do not connect texts by **or**

All **operations specified for the tree are executed**, even if errors occur:

- introduce **error values**, e. g. `NoKey, NoType, NoOpr`
- operations that **yield results** have to yield a reasonable one in case of error,
- operations have to accept **error values as parameters**,
- **avoid messages for avalanche errors** by suitable extension of relations, e. g. every type is compatible with `NoType`

5. Transformation

Create **target tree** to represent the program in the intermediate language.

Intermediate language spcified externally or designed for the abstract source machine.

Design rules:

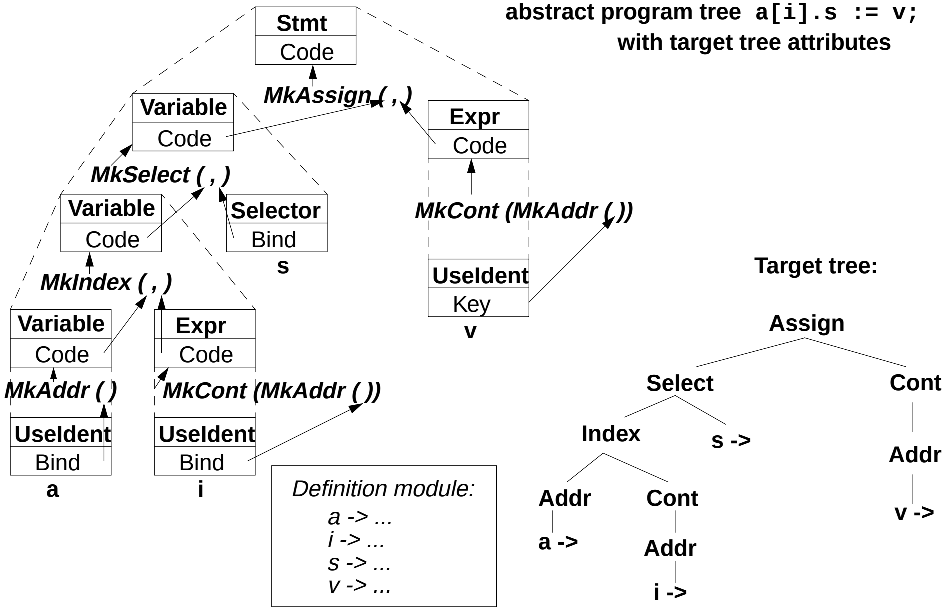
- **simplify the structure**
  only those constructs and properties that are needed for the synthesis phase;
  omit declarations and type denotations - they are kept in the definition module
- **unify constructs**
  e. g. standard representation of loops, or translation into jumps and labels
- **distinguished target operators for overloaded operators**
- **explicit target operators for implicit source operations**
  e. g. type coercion, contents operation for variable access, run-time checks

Transfer **target tree and definition module to synthesis phase**
as data structure, file, or sequence of function calls

For **source-to-source translation** the target tree represents the **target program**.
The target text is produced from the tree by **recursive application of text patterns**.

Example: Target tree construction

abstract program tree `a[i].s := v;`
with target tree attributes



Attribute grammar for target tree construction (CI-93)

```
RULE: Stmt ::= Variable ':=' Expr            COMPUTE
    Stmt.Code = MkAssign (Variable.Code, Expr.Code);
END;
RULE: Variable ::= Variable '.' Selector     COMPUTE
    Variable[1].Code = MkSelect (Variable[2].Code, Selector.Bind);
END;
RULE: Variable ::= Variable '[' Expr ']'     COMPUTE
    Variable[1].Code = MkIndex (Variable[2].Code, Expr.Code);
END;
RULE: Variable ::= UseIdent                  COMPUTE
    Variable.Code = MkAddr (UseIdent.Bind);
END;
RULE: Expr ::= UseIdent                      COMPUTE
    Expr.Code = MkCont (MkAddr (UseIdent.Bind));
END;
```

Generator for creation of structured target texts

Tool PTG: Pattern-based Text Generator

Creation of structured texts in arbitrary languages. Used as computations in the abstract tree, and also in arbitrary C programs. Principle shown by examples:

1. Specify output pattern with insertion points:

```
ProgramFrame:   $
                "void main () {\n"
                $
                "}\n"

Exit:           "exit (" $ int ");\n"

IOInclude:      "#include <stdio.h>"
```

2. PTG generates a function for each pattern; calls produce target structure:

```
PTGNode a, b, c;
a = PTGIOInclude ();
b = PTGExit (5);
c = PTGProgramFrame (a, b);
```

correspondingly with attribute in the tree

3. Output of the target structure:

`PTGOut (c);` or `PTGOutFile ("Output.c", c);`

PTG Patterns for creation of HTML-Texts

concatenation of texts:

```
Seq:          $ $
```

large heading:

```
Heading:      "<H1>" $1 string "</H1>\n"
```

small heading:

```
Subheading:   "<H3>" $1 string "</H3>\n"
```

paragraph:

```
Paragraph:    "<P>\n" $1
```

Lists and list elements:

```
List:         "<UL>\n" $ "</UL>\n"
Listelement:  "<LI>" $ "</LI>\n"
```

Hyperlink:

```
Hyperlink:    "<A HREF=\"" $1 string "\">" $2 string "</A>"
```

Text example:

```
<H1>My favorite travel links</H1>
<H3>Table of Contents</H3>
<UL>
<LI> <A HREF="#position_Maps">Maps</A>
<LI> <A HREF="#position_Train">Train</A>
</UL>
```