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.

Lecture Compiler I WS 2001/2002 / Slide 87

Objectives:

Understand task of name analysis

In the lecture:

Explanations and examples for

- hiding rules (see "Grundlagen der Programmiersprachen"),
- name analysis task: consistent renaming

Suggested reading:

Kastens / Übersetzerbau, Section 6.2, 6.2.2

Questions:

- Assume consistent renaming has been applied to a program. Why are scope rules irrelevant for the resulting program?

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.

Lecture Compiler I WS 2001/2002 / Slide 88

Objectives:

Learn the interface of the Environment module

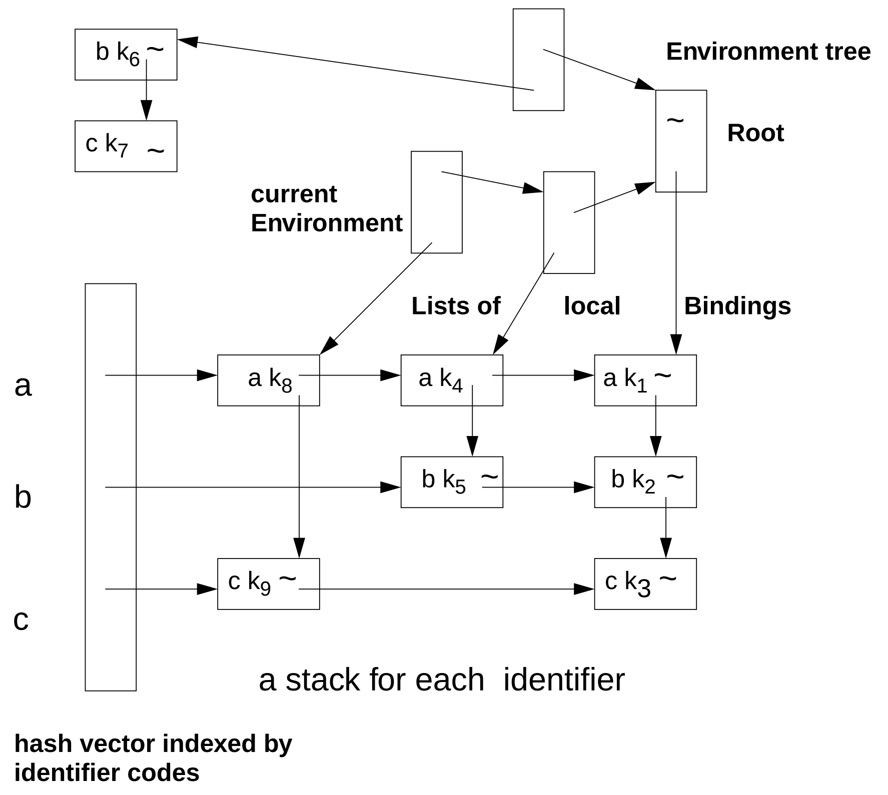
In the lecture:

- Explain the notion of Environment,
- Explain the example of CI-89,
- show that the module is generally applicable.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.2

Data structure of the environment module



k_i: key of the defined entity

Lecture Compiler I WS 2001/2002 / Slide 89

Objectives:

An efficient data structure

In the lecture:

Explanations and examples for

- Explain the concept of identifier stacks.
- Demonstrate the effect of the operations.
- O(1) access instaed of linear search.
- Explain how the current environment is changed using operations Enter and Leave, which insert a set of bindings into the stacks or remove it.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.2

Questions:

- In what sense is this data structure efficient?
- Describe a program for which a linear search in definition lists is more efficient than using this data structure.
- The efficiency advantage may be lost if the operations are executed in an unsuitable order. Explain!
- How can the current environment be changed without calling Enter and Leave explicitly?

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).

Lecture Compiler I WS 2001/2002 / Slide 90

Objectives:

Apply environment module in the program tree

In the lecture:

- Explain the operations in tree contexts.
- Show the effects of the order of calls.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.1

Assignments:

Use Eli module for a simple example.

Questions:

- How do you check the requirement "definition before application"?
- How do you introduce bindings for predefined entities?
- Assume a simple language where the whole program is the only range. There are no declarations, variables are implicitly declared by using their name. How do you use the operations of the environment module for that language?

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`**

Lecture Compiler I WS 2001/2002 / Slide 91

Objectives:

Design rules for error handling

In the lecture:

Explanations and examples

Suggested reading:

Kastens / Übersetzerbau, Section 6.3

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**.

Lecture Compiler I WS 2001/2002 / Slide 92

Objectives:

Properties of intermediate languages

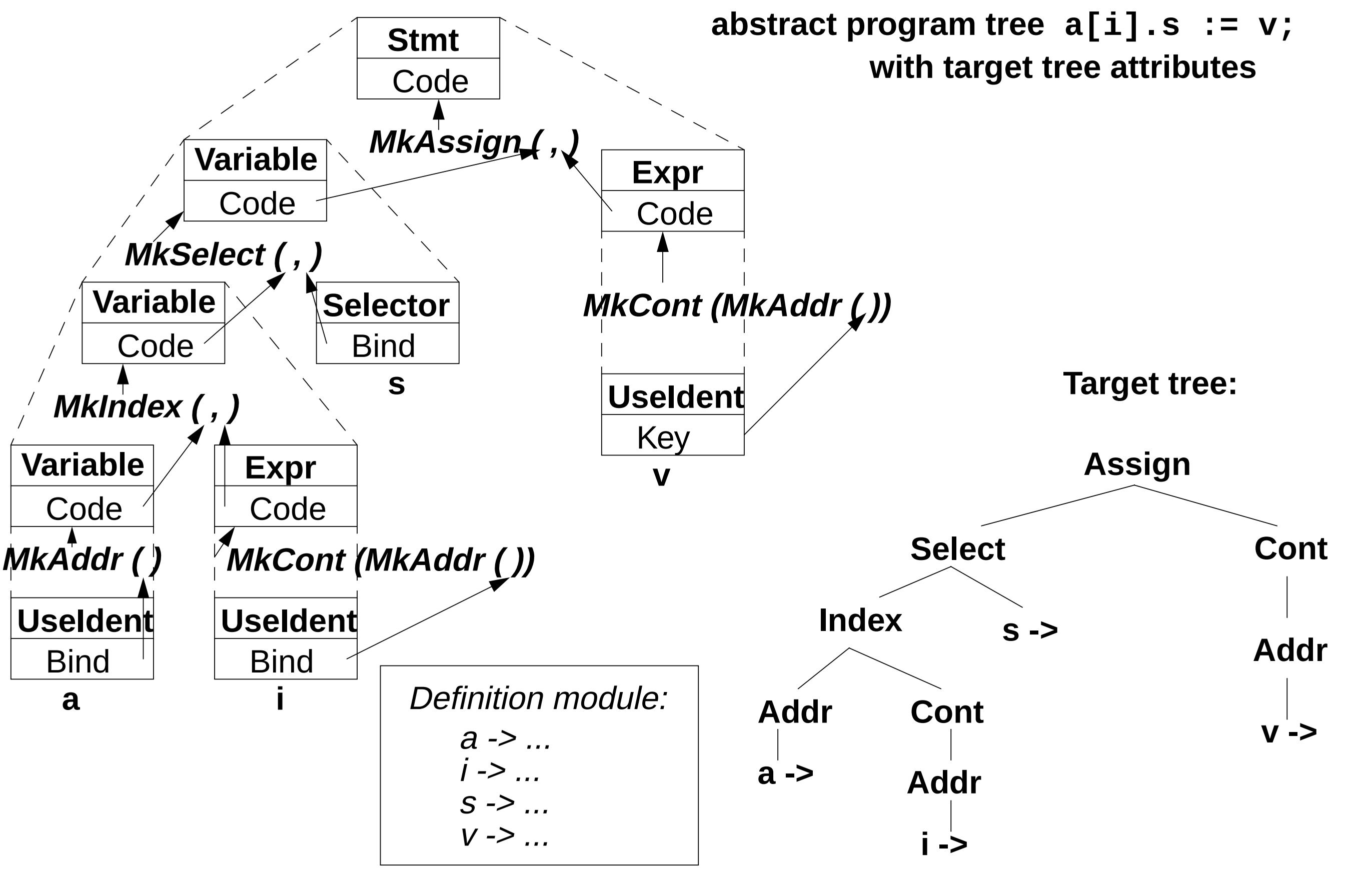
In the lecture:

Example for a target tree on CI-93

Suggested reading:

Kastens / Übersetzerbau, Section 6.4

Example: Target tree construction



Lecture Compiler I WS 2001/2002 / Slide 93

Objectives:

Recognize the principle of target tree construction

In the lecture:

Explain the principle using the example.

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;
```

Lecture Compiler I WS 2001/2002 / Slide 94

Objectives:

Attribute grammar specifies target tree construction

In the lecture:

Explain using the example of CI-93

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);`

Lecture Compiler I WS 2001/2002 / Slide 95

Objectives:

Principle of producing target text using PTG

In the lecture:

Explain the examples

Questions:

- Where can PTG be applied for tasks different from compilers?

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>
```


Lecture Compiler I WS 2001/2002 / Slide 96

Objectives:

See an application of PTG

In the lecture:

Explain the patterns

Questions:

- Which calls of pattern functions produce the example text given on the slide?