Course Script

INF 5110: Compiler construction

INF5110, spring 2024

Martin Steffen

Contents

6 Symbol tables **1**
6.1 Introduction . . . 1
6.2 Symbol table design and interface . . . 3
6.3 Implementing symbol tables . . . 5
6.3.1 Hash tables . . . 6
6.4 Block-structure, scoping, binding, name-space organization . . . 11
6.4.1 Block-structured scoping with chained symbol tables . . . 11
6.4.2 Lexical scoping & beyond . . . 11
6.4.3 Same-level declarations . . . 12
6.4.4 Recursive declarations/definitions . . . 15
6.4.5 Static vs. dynamic scope and binding . . . 16
6.5 Symbol tables as attributes in an AG . . . 21
6.5.1 Expressions and declarations: grammar . . . 22

Chapter 6
Symbol tables

What is it about?

Learning Targets of this Chapter

1. symbol table data structure
2. design and implementation choices
3. how to deal with scopes
4. connection to attribute grammars

Contents

6.1 Introduction 1
6.2 Symbol table design and interface 3
6.3 Implementing symbol tables . 5
6.4 Block-structure, scoping, binding, name-space organization 11
6.5 Symbol tables as attributes in an AG 21

6.1 Introduction

Symbol tables are a quite **central** data structure in all compilers. When discussing the architecture of a compiler and how different phases hang together and build upon each other, the symbol table was shown as being used across different phase. It can be seen as a "data base" or repository shared between the phases. It's general and very abstract purpose is to associate attributes or properties to syntactic elements, like names and symbols. (identifiers, symbols).[1] When analyzing the program, during some phases in the compiler, that involves determining some information relating to syntactic elements. Information like that is often needed more than once, also perhaps in different phases. So there is, as often, a trade-off: Storing the information, once calculated, costs memory, not storing it, but recalculating it on demand, costs time.

Most often, the decision is a no-brainer: **storing** the associated information is better. But there are different ways how to store data, of course. If we take for instance type information, in particular the type for variables or symbols as *declared* in the program code, one could store that in the **abstract syntax tree**. Indeed, the abstract syntax tree might contain that information in some form. If there is a variable declaration piece of syntax, it will be part of the AST as a node containing the name of the variable and a node carrying information about the type. The syntax tree thereby contains information

[1] Remember the (general) notion of "attribute" when discussing attribute grammars.

in its structure that associates the name of symbol with its type.[2] Keeping that association inside the AST, in the corresponding node for the variable declaration, however, is however not the best way of realizing that that binding, for more than one reason.

Determining the declared type of the variable will happen more than once and **searching** the AST to figure it out over and over again is inefficient. The AST is simply not an appropriate data structure to **store** and **look-up** associations like that of a variable name or symbol with its type. Another reason is: when proceeding with the compilation through the various phases, the abstract syntax tree may not be around forever. The AST is handed over from the parser to next phase, which could be the type checker. The type checker then traverses the tree and checks whether the syntactically correct AST is also well-typed. But at a later stage, the compiler will no longer work with the AST (or a type-annotated AST), but with other intermediate representations, for instance control-flow graphs and/or intermediate code, and at that point the AST will be forgotten. For instance, types of variables will still be needed for the run-time environment and code generation, as the types will influence the size and the "layout" of the memory needed (and thus how to access the corresponding bits it in terms of memory *addresses*). That is quite later in the compiler where the association between variable names and types is still needed. So even if one wanted, the later phases cannot consult the AST to look-up for information stored in the tree.[3]

So there are good reasons to externalize the associations between names and symbols and information of interest (like types, but there is other information as well) in a data structure tailor-made for efficient **storing** and **looking-up** that association, and that is shared between multiple phases in the compilation process. Inside a compiler that data structure is the **symbol table**. Of course, there will be a point in the later phases or maybe only the very last phase, when variable names (or other names) have "disappeared" as abstraction to refer to memory, at which point also the symbol table has outlived its usefulness and the compiler will do the last finishing steps working only with addresses.

The discussion focused on binding variable names to types, where the binding is established by the variable's (type) declaration. Of course, there are other *named entities* in a compiler, which can be declared. That includes constants, procedures, functions, types, classes etc. All of them are introduced at some point, declared and defined, and most cases given a name[4], and then used at some other places by referring to them by name.

Also, types are not the *only* kind of information that a compiler might associate to names. There can be many different ones for different purposes. One obvious one may be the *address* of the named entity (like the address of a piece of memory for some piece of data, or the start address of the code implementing a function).

---

[2]At least that's the case for languages, that insist that when introducing a variable, the programmer needs at the same time mention the static type of the variable. Java is an example for that. There are however languages that don't require that, but the type system tries to figure out the type itself. That's known as type reconstruction or type inference.

[3]Of course if one made the decision not to use a symbol-table but use the AST to contain the association, then one would feel forced to keep the AST around. But it would be clumsy.

[4]There might also be things like *anonymous* functions and classes etc. for which there is no name, at least no name at the user syntax level. But also those entities will have to be analyzed treated and one needs to remember information about those as well.

So: do I need a symbol table? In theory, alternatives exists; in practice, yes, symbol tables is the way to go; most compilers do use symbol tables. Most often (and in our course), the symbol table is set up once, containing all the symbols that occur in a given program, and then, the semantic analyses (type checking, etc.) update the table accordingly. The symbol table are "static" in that they are part of the compiler, but not the run-time system. There are also some languages, which allow "manipulation" of symbol tables at *run time* (Racket is one (formerly PLT scheme)).

To summarize, the core functionality of a symbol table are to store and retrieve association of information and symbols or identifiers in a compiler. Of course, there are different well-known ways of realizing an appropriate data structure. Section 6.2 discusses issues concerning the design and interface of a symbol table and Section 6.3 sketches aspects of its implementation.

6.2 Symbol table design and interface

It's helpful to think about the interface of a data structure separate from its implementation, i.e., to treat the symbol table as **abstract data type**. Very roughly, a symbol table is "nothing else" than a lookup-table or *dictionary*, associating "keys" with "values". In particular, names (identifiers, symbols) are the keys, and the "attribute(s)" are the associated values. Such a data structure offers two core functions:

> *insert* for adding a new binding and *lookup* retrieve

Of course besides that, there will be more, for instance creating a new empty dictionary or symbol table. In particular, the particularities of organizing names (in scopes and namespaces, and local variables) in the implemented programming language influences the design of the data structure and its interface. So, the symbol table date structure is more complex than a simple dictionary, but that's the core.

Names and identifiers in a language are an **abstraction**: If a variables refers to a piece of data, it's an abstraction of an address in some part of the memory. It's an abstraction in that it hides to the programmer low-level details of where in memory the data resides. Of course this abstraction has to be realized by the compiler, resp. the compiler writer. We will discuss issues surrounding that in the chapter about run-time environments. The abstractions offered by identifiers can be complex. A language may offer different and differently structured *name spaces*, there typically will be **scopes** (and scoping and naming rules for variables may be different from those for, say, classes, etc.).

That means, there is not one single "flat" name space for all kinds of names. Consequently, a symbol table is typically not just a "flat" dictionary, neither conceptually nor the way it's implemented. *Scoping* is something that complicates the design of the symbol table. To be able to capture things like scopes, the symbol table may explicitly or implicitly offer functionality to delete or hide bindings for instance with an explicit *delete*, besides the core functionality of adding and look-up bindings mentioned above. In general the nature of the implemented language influences the design and interface of the symbol table (and also the choice of implementation).

It should also be clear from the context of the discussion: when we speak of the *value* of an attribute we typically don't mean the ultimate run-time value of the symbol, like the concrete integer run-time value of an expression. The value of an attribute is meant in the "meta"-way, the value that the analysis attaches to the entity, for instance its type, its address, etc. (and only in rather rare cases, its run-time level value). The situation is the same as for attribute grammars and indeed, symbol tables can be seen as a data structure realizing "attributes".

The attached information, though, is mostly *semantic* in nature. After all, in the current phase, we have left the syntactic analysis phase behind us, the parser, and doing *semantic analysis*. For instance, when attaching a type like integer to a symbol, it *is* a semantic value, it carries semantic infomation. It's a static abstraction of the possible concrete semantic values that can be stored at run-time in the corresponding variable (and where the value may of course change in an imperative language).

To realize the symbol table, there are *two main philosophies*. One is realizing it in *traditional table(s)*, in one central repository (or more), separate from the AST, supporting in its interface functions like *lookup*(*name*), *insert*(*name*, *decl*), and *delete*(*name*) (and additionally more or refined functionality). The lookup and delete is used for declarations *and* when entering exiting *blocks* of lexical scopes.

Alternatively, one can maintain the declarations and bindings in the AST. That implies that doing a look-up results in a tree *search*. The insert and delete functionality mentioned before could be realized *implict*, i.e., a binding may or may not exists at some point depending on the relative positioning in the tree. Of course there is the question of (lack of) efficiency, we commented on that earlier on the disadvantages of using the AST to contain binding information. But in special cases it may be an option or has been an option, for instance, for representing names of classes in the class hierarchy of object-oriented languages. And indeed, optimizations exists, for instance using a "redundant" extra table outside the AST, similar to the traditional ST, so to avoid to actually searching through the syntax tree.

Language often have different "name spaces". Even a relatively old-school language like C has 4 different name spaces for identifiers. There are different kinds of identifiers, and different rules (for instance wrt. scoping) apply to them. One way to arrange them could be to have different symbol tables, one specially for each name space. Later we will also have the situation (but not caused by different kinds of identifiers), where the symbol table is arranged in such a way that smaller symbol tables (per scope) are linked together where a symbol table of a "surrounding" scope points to a symbol table representing a scope nested deeper. One might see that as having "many" symbol tables, but maybe that's misleading. It's more an internal representation with a linked structure, but that data structure containing many individiual table is better seen conceptually as one symbol table (*the* symbol table of the language), but one with a complex behavior reflecting the lexical scoping of the language. Actually, whether or not one implements it in chaining up a bunch of individual hash tables or similar structures or doing a different representation, is a design choice, both realizing the same external behavior at the interface. In that spirit, also the remark that C has 4 different name spaces (which is true) and therefore a C compiler may make use of 4 symbol tables is a matter of how one sees (and implements)

it: one may as well see and implement it as one symbol table (with 4 different kinds of identifiers which are treated differently).

6.3 Implementing symbol tables

This section touches upon suitable data structures to implement symbol tables. There are of course different ways to implement such data structures, like *dictionaries* or *look-up tables*, etc. It can be based on (simple) association lists or maybe trees, and there are many different forms of suitable trees, including balanced versions like AVL-trees and B-trees, red-black trees, binary-search trees, etc. One also has the choice of functional vs. imperative implementations of corresponding data structures (which influences the interface). Often, the structures underlying symbol tables are **hash tables**. A careful choice of the data structure influences the efficiency of the compiler. In particular and as mentioned, the structure of the symbol table(s) is influenced by the implemented language's **scoping** rules (resp. the structure of the name space in general) etc.[5] As far as data structures are concerned, we will mostly focus on hash-tables, without actually going much into details of hash-table techniques, focusing more on how hash-tables are arranged to reflect aspects such as scoping regimes. We will talk about about scoping later, but here already some examples of block-structured languages. For instance, Listing 6.7 shows blocks in C (but very many languages have block structured lexical scope, together with other scoping mechanisms): there are two declarations of the variable `i`, one in the outer block and one in the inner. Information associated with “the” variable called `i` is actually referring to two different pieces of data, with different information associated. Coincidentally, the type information here is identical, both times it's an integer, but in general this and other attached information depends on where in the code the variable is mentioned. It's **context-sensitive** information, not context-free . . . Anyway, the symbol-table(s), for instance the hash-table(s) must arrange for that, associating the correct and relevant information depending on the different places in the syntax tree.

```
{ int i; ... ; double d;
  void p(...);
  {
     int i;
     ...
  }
  int j;
  ...
```

Listing 6.1: Nested block with lexical scope in C

Blocks also exists in non-programming languages, as shown in the small LaTeX and TeX codes. LaTeX and TeX are chosen for easy trying out the result oneself (assuming that most people have easy access to LaTeX and by implication, TeX).[6]

[5]Also the language used for implementation (and the availability of libraries therein) may play a role.

[6]TeX is the underlying “core” on which LaTeX is put on top. There are other formats on top of TeX (`texi` is another one; `texi` is involved, for instance, type setting the pdf version of the Compila language specification).

```
\def\x{a}
{
  \def\x{b}
  \x
}
\x
\bye
```

Listing 6.2: TEX

```
\documentclass{article}
\newcommand{\x}{a}
\begin{document}
\x
{\renewcommand{\x}{b}
  \x
}
\x
\end{document}
```

Listing 6.3: LATEX

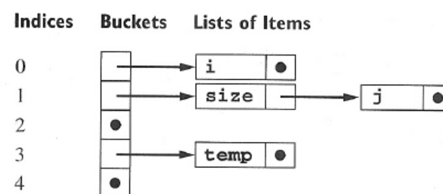
6.3.1 Hash tables

Hash tables are a classical and common implementation for symbol tables. It's a generic term itself, and different forms of hash tables exists. The sections here are not intended as an introduction to hash tables and advanced techniques in that area. A beginner's course impressionistic familiarity with hash tables is sufficient as we don't focus on the internals of those, but on how to arrange them (or other data structure) to reflect aspects such as scoping in a language. Though, *one* internal implementation detail we touch upon, that's the difference between hash-tables using **separate chaining** vs. those using **open addressing**.

Side remark 6.3.1 (Hash-table terminology)**.** There exists alternative terminology (cf. probably mentioned in IN2010 resp. INF2220 in the older numbering scheme, the algorithms & data structures lecture). In older versions of the lecture, separate chaining was discussed under *open hashing* (never versions —I checked only 2023— seem to use the term separate chaining as this script). A method that is related to *open addressing* is discussed in A&D as **linear probing** (or in older version of that lecture **closed hashing**). It's confusing, but that's how it is, and it's just words. □

```
{
  int temp;
  int j;
  real i;
  void size (....) {
    {
        ....
    }
  }
}
```

Listing 6.4: Declarations of variables and procedures



Let's look at the simple code snippet from Listing 6.4. The picture on the right illustrates a possible arrangement based on hash tables. It illustrates one basic fact about hash-table as underlying data structure for dictionaries. Namely the fact that there can be **hash conflicts**. In the example, it's assumed that there is such a hash-conflict between the identifiers `size` and `j`. A hash conflict means, when applying the chosen **hash-function**

onto two keys (here the two identifiers or symbols), the resulting hash values are identical. That's the conflict. One design of hash tables deals with the situation by arranging entries of conflicting hashes into a *linked list*. That's what is illustrated here and that techniques is called **separate chaining**.

Conventionally, the compiler would treat the program from beginning to the end, which means it would enter the information about `j` before doing the same for `size`. It is, in that situation of a linked list, plausible that, after adding both pieces of information in a linked list arrangement, the `size`, being entered last, appears at the head of the list.

We also see that the identifiers for integer **variables** `i`, `j`, etc. are treated the same way as those for **procedures** (here `size`). In particular they are added to the *same* symbol table. There is nothing wrong with that, but, as mentioned before, sometimes identifiers are grouped in different name spaces or, in general, the use of names is generally more complex, so one may choose to handle things with different symbol tables (for different purposes of classes of identifiers etc.)

Scopes are typically arranged in **blocks**. One speaks of **block-structured** scopes and block-structured programming languages. It's actually a pretty old concept, it seriously started with ALGOL60 and almost no language has just one flat, global scope, at least not for variables. A block refers to some lexical "region" in the program code. It's often explicitly delimited by special keywords, like `{` and `}` or `BEGIN` and `END` or similar. For instance the body of a procedure is typically one block. Blocks organize and structure the **scope** of declarations. In particular, scopes can be **nested**.

One also speaks about the scope *of a variable* (or more generally of a name). In most languages often names are introduced by being declared (and defined). If one makes the distinction between declaration and definition, the declaration of the name refers to introducing it, often by specifying its type. That can be done in some languages without also assigning a value to in, which leads to a situation where, when running the program, a variable is undefined ("nil-pointer") or has a default value (or an unspecified value). If one introduces a variable inside a block, the scope of that variable typically is not all of the block, but more precisely the part of the block starting from the point when the variable is declared till the end of the block.

Languages allow typically having the "same" variable declared more than once, like first defined in some block, but then again in a nested block. Also there could be two separate blocks or scopes each using the same variable name. Some languages allow the same variable be declared twice in the same block (though not all).

Anyway, because of that, it's actually imprecise to speak about the scope of a variable, say x. It's more precise to speak about the scope of a declaration of a variable or a particular *occurrence* of the variable in the code. In the same way, one should not even think of those situations as that the "same" variable is declared and it's not "multiple" declarations of one variable. It's different variables that are being declared at different places, they just happen to carry the same name, and they occur in different portions of the program code. In particular, a local variable declared in one procedure is not "the same" local variable in a different procedure.[7] And scoping scoping rules of a language allows such declarations

[7] Unless, one procedure is declared inside the other, which leads to a situation with nested scope, and the variable in question is declared in the outer procedure, and not also declared in the nested one.

with the same name in a meaningful and structured manner.

We mentioned that when introducing a variable or name in a block, the scope of that variable is typically "the rest of that block", so the declaration starts, so to say, the scope of that declaration. It corresponds to a "introduce-before-use" discipline. The point of introduction of declaration of a variable is typically the point where the name or symbol and associated information, for instance its type is entered to the symbol table. A occurrence of the variable in the scope, where the variable is used (for instance for purpose of type checking) is where the symbol table is consulted for looking up associated information, for instance the type.

So far so good, but that picture does not cover all situations resp. languages. One is that not all naming disciplines fall under the "introduce-before-use" discipline. For instance, classes in Java are names that are defined without any specific order. That's particularly visible for public classes, where classes are separate files in some file system without any order. It makes no sense to speak about that to use an instance of a class resp. its methods in some piece of course requires that the class and its methods have to be defined first. Likewise there is no order between methods inside a class, so it does not matter which one is written first.[8]

Likewise, the "introduce-before-use" scheme make not sense when dealing with **mutual recursion**: If one has, say to procedures, and one calls the other and the other one calls the first, then one cannot assume that a function needs to be fully introduced before one can called it the program code (and of course what's been said about classes methods in Java means, mutual recursion of methods or call backs are perfectly fine).

Listing 6.5 shows an example of nested blocks in C (similar to the example from before) and Listing 6.6 an example in Pascal.

[8] As far as classes in Java are concerned, it makes not sense to separate declaration and definition. Writing down a class introduces its name at the same time that the code of its implementation is fixed. Of course one could work with classes and interfaces, where classes implement interface. One could see the interface as "declaration" and a class implementing the interface as definition (which is often a recommended style). But of course the class and the interface must have different names, so it's not really a comparable situation.

```
int i, j;

int f(int size)
{ char i, temp;
  ...
  { double j;
    ..
  }
  ...
  { char * j;
    ...
  }
}
```

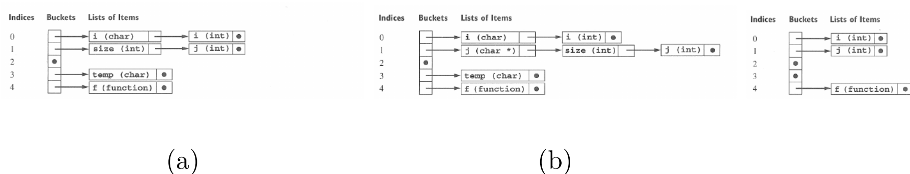
Listing 6.5: Nested blocks with lexical scope (in C)

```
program Ex;
var i,j : integer

function f(size : integer) : integer;
var i, temp : char;
   procedure g;
   var j : real;
   begin
      ...
   end;
   procedure h;
   var j : ^char;
   begin
      ...
   end;

begin (* f's body *)
 ...
end;
begin  (* main program *)
   ...
end.
```

Listing 6.6: Nested procedures in Pascal

The Pascal-example shows a more complex situation and shows a feature of Pascal, which is *not* supported by C, namely **nested declarations of functions or procedures**. As far as scoping and the discussion at the current point in the lecture is concerned, that's not a big issue: just that concerning names for variables, C and Pascal allow nested blocks, but for names representing functions or procedures, Pascal offers more freedom.

The scoping rules of a programming language influences the design and implementation of a language's symbol table, that's a general message here.As preview for a later chapter, we will see how scoping also influences the design of the so-called **run-time environments**. Also in that part, we will discuss how the run-time environments for Pascal are more complex than those for C. And scoping regimes more complex than that of Pascal add even more complexity for symbol tables and in particular to the run-time environment.



(a) (b) (c)

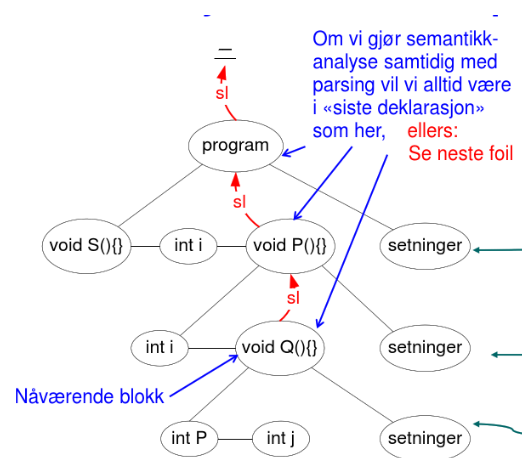
Figure 6.1: Nested blocks and separate chaining

The three pictures from Figure 6.1 correspond to three "points" inside the C program from above. The first one after entering the scope of function `f`. When saying "entering", it's not meant at run-time when calling the function, it's meant when the analysis starts processing the body of the procedure. Inside the body of the function (immediately after entering), the two local variables are available, and of course also the formal parameter `temp`, which can be seen as a local variable, as well. At that point, the global variable `i` of type `int` is no longer "visible" or accessible, any reference to `i` will refer to the local variable `i` at that point.

Upon entering the first nested local scope, a second variable j is entered (making the global variable j inaccessible). That situation is *not* shown in the pictures. New, when *leaving* the mentioned scope, one way of dealing with the situation is that the additional second j of type double is *removed* from the hash-table again (shortening the corresponding linked chain). What is shown is a situation inside the *second* nested scope with another variable j (now a char pointer). Since the first nested local scope has been left at that point, the corresponding j "has become history", and the hash table of the third picture only contains the global j variable (which is inaccessible) and the now relevant second local j variable.

```
lookup (string n) {
   k = current , surrounding block
   do           // search for n in  decl for block k;
      k = k.sl   // one nesting level  up
   until found or k == none
}
```

Listing 6.7: Using the syntax tree for lookup following (static links)



The notion of **static link** will be discussed later, in connection with the so-called run-time system and the run-time *stack*. There we go into more details, but the idea is the same as here: find a way to "locate" the relevant scope. If scopes are nested, connect them via some "parent pointer", and that pointer is known as static link (again, different names exists for that, unfortunately).

Alternative representation

- arrangement different from 1 table with stack-organized external chaining
- each *block* with its **own** hash table.
- standard hashing within each block
- **static links** to link the block levels

⇒ "tree-of-hash-tables"

- AKA: *sheaf-of-tables* or *chained symbol tables* representation

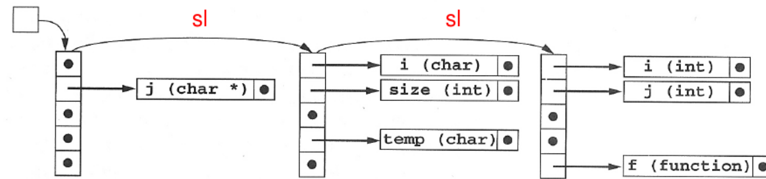


Figure 6.2: Alternative representation (see Listing 6.5)

Note that the current scope and the most nested one is at the left-hand side of the table, and the static-link always points to the (uniquely determined) surrounding scope (if any). The scope on the right-hand side is global scope.

One may more generally say for this representation: it's one *symbol table* per block, as this form of organization can generally be done for symbol tables data structures where hash tables is just one of many possible data structure to implement look-up tables.

6.4 Block-structure, scoping, binding, name-space organization

In this section, we shortly have a look of blocks and scoping, mostly lexical scoping. The concept of scopes is probably known from other lectures (and for from experience with programming languages), and we discuss some aspects in connection with symbol tables.

6.4.1 Block-structured scoping with chained symbol tables

- remember the *interface*
- look-up: following the static link (as seen)
- **Enter** a block
  - create new (empty) symbol table
  - set static link from there to the "old" (= previously current) one
  - set the current block to the newly created one
- at **exit**
  - move the *current block* one level up
  - note: no *deletion* of bindings, just made *inaccessible*

As mentioned in the previous section: The notion of static links will be encountered later again when dealing with *run-time* environments and for analogous purposes: identifying lexical scopes in "block-structured" languages.

6.4.2 Lexical scoping & beyond

As mentioned, block-structured lexical scoping is central in many programming languages (ever since ALGOL60 . . .), but other scoping mechanisms exist (and exist side-by-side). One is of course dynamic binding, which we will explore later in the context of run-time environments.

But there are also variations on the theme used for instance in many object-oriented language. Let's take C++as example. There, so called member functions (think "methods"...) are *declared* inside a class, but *defined* outside. The class name can be seen a **name** of a scope. Still: method are supposed to be able to access names defined in the *scope of the class* definition (i.e., other members, and that's done using `this`).

```
class A {
  ... int f(); ... // member function
}

A::f() {}      // def. of f ``in'' A
```

Listing 6.8: C++ class and member function

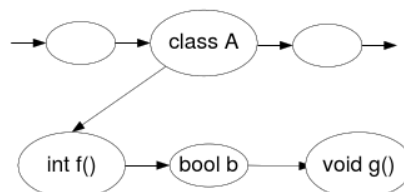
```
class A {
    int f() {...};
    boolean b;
    void h() {...};
}
```

Listing 6.9: Java analogon

scope resolution in C++

- class *name* introduces a **name for the scope**[9] (not only in C++)
- scope resolution operator `::`
- allows to explicitly refer to a "scope"'

- to implement
  - such flexibility,
  - also for *remote access* like `a.f()`
- declarations are kept separately for each block (e.g. one hash table per class, record, etc., appropriately chained up).



6.4.3 Same-level declarations

Declarations of entities (variables, procedures, classes ...) occur inside one scope. Global declarations can be considered to occur in one surrounding global scope. One scope contains normally multiple declarations. Characteristic for declarations in nested blocks is that inside the inner scope, the declarations take precedence over those in the surrounding block, should there be declarations using the same name. It's always the closest declaration (in terms on the nesting structure of scopes) that counts, as explained. In the following we discuss how to treat *same-level* declarations inside one scope, focusing mainly on two alternatives: sequential or simultaneous (or collateral).

[9]Besides that, class names themselves are subject to scoping themselves, of course ...

Multiple declarations of the same name

Before we look at those two alternatives, let's discuss other aspects concerning same-level declarations. One is how to treat multiple declarations of the *same* identifier at the same level. Like in the situation in Listing 6.19, where `i` is used for an integer variable as well as name of a type.

```
typedef int i
int i;
```

Listing 6.10: Same level declarations

One simple way is to simply **forbid** it :-) For instance in C, it's (largely) forbidden; it's a bit more complex there, distinguishing between declarations, declarations combined with definitions, tentative declarations . . . , but it's not relevant for us. Forbidding multiple declarations or definitions is also the solution followed in the *compila24* language of the oblig. That can of course be easily achieved: Before using the *insert* procedure to add a binding, one simply checks first (with, say, *lookup*) whether such a binding for the name exists already.

But one can also allow it. The declarations show up in the block in some textual order, and then the convention is that a declaration make a previous one with the same name inaccessible. It's as if a new nested scope is introduced implicitly which starts at the new declaration and lasts till the end of the "official" surrounding scope.

Those treatments can be more involved or fine-grained. For instance how to handle multiple declarations using the same identifier, but concerning different kinds of language elements. Like declaring a variable and a procedure with the same name, declaring a variables and introducing a type with the same name etc. Again, one could forbid such practice, or at least partially, for simplicity. It may even be that the language would not even allow to write multiple declarations of that kind, because of lexical conventions of the language. For instance, names of types have to start with a capital letter and variables have to start with a lower case letter. So already the lexer (together with the parser) makes it impossible to declare a variable and a type with the same name.

As a side remark: When one wishes, for instance to distinguish variables names from function names, one has to take into account what the programming language actually supports when programming with variables and functions. In many languages, variables are variables and functions are functions. However, there is the concept of function variables. In particular, in functional languages, there is no differences between functions and (other values). Consequently there is no difference conceptually between names for "conventional" values like integers, and function abstractions, which counts among values as well. So corresponding names would live in the same name space.

Sequential vs. simultaneous (or collateral) declarations

Now to the distinction illustrated in the following with a number of examples (in different languages). Many languages insist when using a name, the name must be declared, one cannot use undeclared names. Often it means, one need to declare something *before* one

can use it. “Before” means roughly earlier in the program text, but only roughly so, since there is also the issue of scopes. Being declared in an earlier line in the program does not mean one can make use of the declared variable until the last line of the program, since the scope may not cover the code till then end of course. Like a local variable inside a function or a method of course can be referred to only inside the function or method body, not forever after being locally declared.

But inside one flat scope, some declarations appear earlier than others, and there is a clear notion of “before” and “after” as we assume a *same-level* situation inside one scope.

If one declares one item after the other, then it’s natural that, what is declared first can be used in subsequent declarations. That is clearly a **sequential** form of declarations and corresponds to a declare-before-use pattern.

Alternatively, declarations can be treated as occurring all **at the same time.** This is called a **collateral** or **simultaneous** form of declarations. In a setting like that, the declaration occurring second can make use of what’s been declared first in the code, and also vice versa. That allows **mutually recursive** declarations and definitions. Thus, it’s a quite important form of declaration.

Also when defining, for instance, a function using *direct* recursion (not just indirect), there is the aspect of simultaneousness. Then defining resp. declaring the function, the body of the function already mentions the function, which is “currently” being introduced. That’s the very nature of a recursion definition.

Listing 6.11 shows a sequential way of declarations (and definitions) in C. The variable `i` is declared and defined twice; for simplicity, let’s call it just “defined twice” (C would allow declarations also without assigning a value at the same time, hence the distinction).

The second definition is still in the same scope, but not at the same level. Therefore, it’s two different variables (both called `i`) residing at different locations in memory. The first one in the global, static part of the memory, the second one on the stack. Indeed, the local variable can exists in multiple incarnations at run-time, in different activation records (or stack frames) of the procedure. We will have a closer look at that in the chapter about run-time environments. Now to the real point of Listing 6.11, the sequential treatment of definitions, namely those at the local level of the function. Being *sequential*, the definition of `j` refers to the prior local definition of `i`, which means `j` obtains the value 3.

```
int i = 1;
void f(void)
  { int i = 2, j = i+1,
     ...
  }
```

Listing 6.11: Sequential declarations in C

Simultaneous or collateral definitions work differently. See for instance the example from Listing 6.12 (in ocaml). The example is comparable to the one in C (though without making use procedure-local definitions). The simultaneous declarations of (the second mentioning of) `i` and of `j` is indicated by the keyword `and`. The first definition of `i` and the simultaneous definitions of `i` and `j` are treated *sequentially*, as in the C-example. Since the second `i` and the `j` are defined simultaneously in the second line (but after the

definition from the first line), the j uses the value of i from before, and thus j is defined to contain 2.

```
let i = 1;;
let i = 2 and j = i+1;;

print_int(j);;
```

Listing 6.12: Simultaneous declarations in ocaml/ML

Finally, two examples in Scheme (Listing 6.13 and 6.14). The default construction with let does simultaneous definitions (though it cannot be used for recursive procedure definitions). If one wants sequential definition, Scheme also supports let*.

```
(let ((x 0))
  (let ((x 42) (y x))
    (display y)))    ;; "0"
```

Listing 6.13: Simultaneous (in Scheme)

```
(let ((x 0))
  (let* ((x 42) (y x))
    (display y)))       ;; "42"
```

Listing 6.14: Sequential (in Scheme)

6.4.4 Recursive declarations/definitions

Recursive definitions and declarations are important and common. Functions, procedures, and methods are often defined recursively; via direct recursion or via indirect or mutual recursion. Recursion can also happen at type level. For instance, the order in which classes are introduced in a program does not play a role. This means, a definition of one class, say C_1, can mention a second class C_2, when defining its members, declaring their types, and conversely, the definition of C_2 can do the same, mentioning C_1. So there is no "order" between the classes. At least not an order in the sense discussed here (sequential); there may be an inheritance order between classes, but that's a different issue. In Java, each public class resides in its own source code file, already for that reason, there is no order between their definition in terms of their mentioning in some lines of code.

Back to functions or procedures: Listings 6.15 and 6.16 illustrate direct resp. indirection recursive definitions.

```
int gcd(int n, int m) {
  if   (m == 0) return n;
  else return gcd(m,n % m);
}
```

Listing 6.15: Direct recursion in a function definition

```
void f(void) {
    ... g() ... }
void g(void) {
  ... f() ...}
```

Listing 6.16: Indirect recursion in a function definition

As discussed, a recursive definition is connected with the *simultaneous* treatment of declarations. For direction recursion, as in the gcd example, it simply means that when the body of the gcd function is treated, information about the gcd, like declaring its input and output types must already been available, i.e. entered into the symbol table. That's not a big deal; the procedure header is mentioned (and can be treated) before the body anyway. So that feels more like a sequential situation anyway. As an aside: that it can be treated in a sequential manner is also helped by the fact that the procedure header

declares the types of the two formal parameters explicitly to be integers, as well as the return type. In a language, which allowed the programmer to leave out those explicit type declarations, things would get more involved.

But mutual recursive situations one *really* needs to treat the definitions as simultaneous. Languages primarily working with sequential declarations (or slightly old-fashioned languages) may resort to some "tricks" resp. expect some assistance from the programmer. For instance, in the Example from Listing 6.16, the compiler may insist on a little help from the programmer to add crucial type information about `g` before `f`. So `g` is "declared" first which does not require mentioning `f`, then `f` is (declared and) defined, and finally, `g` is defined in a way consistent with its (and `f`'s) declaration. This is sometimes called a `prototype` (which I think is not a good terminology, because prototypes can mean other things as well).

```
void g(void);  /* function prototype decl. */

void f(void) {
    ... g() ... }
void g(void) {
  ... f() ...}
```

Listing 6.17: Indirect recursion in a function definition (prototype)

In Pascal, an analogous mechanism is known as **forward declaration**. Other languages would treat all function definitions (inside a block, inside a module, or similar) as (potentially) mutually recursive. The compiler will figure it out without being alerted by special forward syntax. Still other languages use special syntax for simultaneous definitions which is used for mutually recursive function definitions. Go is a language that allows recursion without requiring special syntax or help from the user, ocaml and some other functional languages use special syntax. We have seen the use of `and` earlier already.

```
package main

func f(x int) (int) {
        return g(x) +1
}
func g(x int) (int) {
        return f(x) -1
}
func main() {
        f(0)
}
```

Listing 6.18: Mutal recursion (Go)

```
let rec  f (x:int): int  =
     g(x+1)
and g(x:int) : int =
     f(x+1);;
```

Listing 6.19: Mutal recursion (ocaml)

6.4.5 Static vs. dynamic scope and binding

So far we have focused on languages with block-structured lexical scope, simultaneous definitions or otherwise. Later, in the chapter about run-time environments, we will likewise focus on how memory for lexical or static bindings is arranged. That focus is justified in that the concept of lexical, nested scopes is *central*. It does not mean lexical scopes are the only way. An alternative to static (or lexical) scoping is, not surprisingly, *dynamic scoping*. Connected to that is are the concepts of *static* vs. *dynamic* binding;

bindings occur within a scope. When scopes are nested, a binding "belongs" to more than one scope.

We will discuss the issue of static vs. dynamic binding mostly with examples involving variables occurring in blocks of scopes. The question of static vs. dynamic binding also occurs elsewhere, for instance in class-based object-oriented languages like Java, *methods* are dynamically bound as def, and the dynamicity of the binding is wrt. "blocks" in which the methods are mentioned. We will have a look at dynamic method binding on object-oriented languages as well later.

C is a language (one of many) with static scopes. Let's have a look at Listing 6.20.

```
#include <stdio.h>

int i = 1;
void f(void) {
  printf("%d\n",i);
}

void main(void) {
  int  i = 2;
  f();
  return 0;
}
```

Listing 6.20: Static scoping in C

The code contains two definitions of `i`, a global one and one local to `main`. The use of `i` in `f`'s body refers to the global instance of `i`, since `f` is defined on the global level. Additionally, the definition of `i` is done *sequentially* before `f` and the body of `f` does not contain a local (re-)definition of `i` that would overshadow the global one. That's the situation concerning the *definition* of `f`. Now to the **use** of `f`. It's called only one time, inside `main` and in a scope that works with a local definition of `i`. When executing `f`, to which declaration does the printed `i` binds to, which is the relevant scope for it?

Under static binding, relevant is the scope where "statically" the function is defined, its static scope (in this example the global one). What is printed therefore is 1, of course.

Dynamic binding (for variable `i` again) is illustrated in Figure 6.21. In this example, the assignment inside `Q` affects the variable `i` as introduced in line 4, since this is the relevant scope in which `Q` is *called*. The procedure `Q` is called only once in the example. If there were different calls originating from inside different scopes, different `i`'s may be affected. For static scoping, which `i` is meant is statically fixed from the place where the function is defined.

```
1  void Y () {
2    int i;
3    void P() {
4      int i;
5      ...;
6      Q();
7    }
8    void Q(){
9      ...;
10     i := 5;  // which i is meant?
11   }
12   ...;
13
```

```
14     P();
15     ...;
16  }
```

Listing 6.21: Dynamic scoping (pseudo code)

Let's look at some non-pseudo code examples. Let's take TEXor LATEX. Those are more domain-specific languages rather than general purpose languages; though TEX allows loops, conditionals etc., so those languages are actually Turing complete. Also it's clearly a compiler, translating some textual source language into some output format, like dvi, ps, or pdf.

TEX and LATEXalso make use of scopes; one does not have to work with one global scope. In the examples from Listing 6.22 and 6.23, the beginning and the end of an inner scope is marked by { and } (there are other ways to obtain scopes in LATEX, but it's unimportant for us). Both examples basically the same (using the slightly different syntax of TEX and LATEX). What happens is very easy to check by invoking TEX or LATEX and check the generated output, say pdf.

Again, it's very easy to check by invoking TEX or LATEX, or firing off emacs and evaluate the lisp snippet in a buffer, for instance.

```
\def\astring{a1}
\def\x{\astring}
\x
{
  \def\astring{a2}
  \x
}
\x
\bye
```

Listing 6.22: Dynamic binding (TEX)

```
\documentclass{article}
\newcommand{\astring}{a1}
\newcommand{\x}{\astring}
\begin{document}
\x
{
  \renewcommand{\astring}{a2}
  \x
}
\x
\end{document}
```

Listing 6.23: Dynamic binding (LATEX)

The next illustration uses Lisp, in particular, *emacs lisp*. If one has access to emacs, the examples are easy to run and check, simply firing off emacs and evaluate the lisp snippet in a buffer. Emacs Lisp is one well-known Lisp-dialect based on dynamic scoping, though as of emacs version 24, also lexical scoping is supported. It is probably not a coincidence, that the key person behind *emacs* is Richard Stallman, the "last true hacker" from the MIT school of hackers. McCarthy and Minsky were also at the MIT (earlier pioneers than Stallman), and McCarthy is central behind Lisp, and actually also coined the term AI (MIT was the focal point of early AI). For emacs, Stallman is central in kicking it off, hacking its initial versions (with others), mentoring it through many years and giving a spiritual (or ideological?) background as part of the larger free software movement.

The are many lisp dialects, emacs lisp just one of theme. Another important one is Scheme, actually Scheme was the first one (or at least the first significant and most prominent one) with *static* or lexical scoping. Originally, Lisp used dynamic binding. Lisp was way ahead of its time, actually revolutionary (higher-order functions, reflection, garbage collection), one should not forget that it was conceived (and implemented!) in the 50ies (at MIT). Now, resource requirements for Lisp stretched the hardware and compiler concepts of those

days. Note that the very earliest machines did not even have hardware support for stack pointers (Burroughs machines at the beginning of the 60ies where the first that pioneered that) which made even recursion (which uses stack) a costly luxury. And Lisp supported higher-order functions from the start. It took some time (and conceptual and hardware advances) until major lexically-scoped variant of Lisp could establish itself (known as Scheme). Scheme also supports dynamic scoping (though frowns upon it). More "classic" Lisp dialects (like Common Lisp) also support lexical scoping besides dynamically scoping in the meantime.

```
(setq astring "a1")   ;; "assignment"
(defun x() astring)   ;; define "variable" x"
(x)                   ;; read value
(let ((astring "a2"))
      (x))
```

Listing 6.24: Dynamic binding in elisp

To round off the discussion, let's go back to static binding and point something out that probably should be clear anyway, but it can't hurt to rub it in. As said, in static binding, it's the static scope "that counts", where a variables has been declared. But it's *not* about the original **value**. A value may change, and as far as the *value* is concerned, static binding does *not* mean, when used inside a function, for instance as in a situation as in Listing 6.20, that it's the original value that counts. Static binding refers to the association of a variable with a memory location or address, not the association with a particular value.

Listing 6.25 illustrates that, in this case using Go, not C. Both languages use static binding of variables. As shown in the example, unlike C, Go supports nested function definitions.

```
package main
import ("fmt")

var f = func ()   {
  var x = 0
  var g = func() {fmt.Printf(" x = %v", x)}
  x = x + 1
    {
      var x = 40                  // local variable
      g()
      fmt.Printf(" x = %v", x)}
}
func main() {
  f()
}
```

Listing 6.25: Static binding and mutating values (in Go)

The value of `x` printed in the body of `g` is 1, not `0` which is the value at the point when `g` is defined. Overall, what is printed is `x = 1 x = 40`.

Maintaining lexical binding can become challenging. The Example from Listing 6.26 basically does the same as the previous one: static binding and changing the value (the latter part, however, changing the value is not central to the purpose of the example).

```
package main
import ("fmt")

var f = func () (func (int) int)  {
```

```
        var x = 40                    // local variable
        var g = func (y int) int {  // nested function
                return x + 1
        }
        x = x+1                       // update x
        return g                      // function as return value
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (4)
        fmt.Printf(" r = %v", r)
}
```

Listing 6.26: Static binding and higher-order functions (in Go)

It is more complex than the previous one in that it not just use nested function definitions, but makes use of *higher-order* functions. That's another feature supported by Go, but not by C. It does not change what static-binding means, it just makes it harder for the compiler to achieve. We will discuss that in slightly more detail in the chapter about run-time environments. Here we just point out what is responsible for the complications

As said, the example uses higher-order functions. In particular, the function `f` gives back some function, namely the function `g`, and not only that: function `g` is defined *inside* `f`, in particular, `g` is defined inside the scope of `f`. And finally, the nested function `g` refers to `x`, which is also defined inside `f`. Now the problem is that the scope of `f` lives longer than the body of `f` itself. In many languages, one important part of the RTE is the run-time stack, or call stack. It turns out, that in situations like the ones illustrated here, a stack is no longer good enough for providing lexical scoping `f`.

At any rate, lexical scoping in the example results that `r = 42` is printed.

Overloading

If one want to allow that *different*, say, functions or variables carry the **same** name when declared at the same level, one could use the type system (and the parser) to distinguish one from the other. The parser may factor in that some uses of function names may be syntactically forbidden for variable names and/or vice versa. Like `call f (x)` may, in some language be syntactically allowed only if `f` is the name of a function and `x` the name of a variable. So therefore also `call f (f)` may be tractable, with two kinds of `f` declared at the same level (though it's probably ill-advised to exploit that freedom).

Such "dual-use" (or multiple-use) of names is also called *overloading*. Outside of compilers, for instance in writing technical texts, it's called *abuse of notation*. And, to awaken the reader's attention to an forthcoming such abuse, it's sometimes introduced with the words "in abuse of notation". Like "in abuse of notation, let n and m refer to nodes in control-flow graph", with the abuse warning because perhaps n and m had, in the text, so far be conventionally used for natural numbers. For the time being, the text re-uses the notation for nodes, hoping the reader can figure it out. Perhaps the writer would not just write n_1, n_2, or n_j referring to different nodes, but even using n_n, referring to the n'th node using both interpretations of n in the same scope (i.e., chapter or section, hoping the reader

can figure out which is which). This is a situation comparable to the `call f(f)` in a programming language, where the compiler may have no problems to figure out which is which (and likewise not recommended).

Often, this overloading in written technical text is avoid by using different “alphabets” or indeed different portions of alphabets or conventions (like the roman alphabet $abc\ldots$ vs. $\alpha\beta\gamma\ldots$ or $\mathfrak{abc}$ vs. $\mathbb{ABC}$ and x, y, z at the end of the latin alphabet for variables and $a, b, c\ldots$ from the beginning for constants. In this script we also use other typographic conventions (bold-facing etc.) to help disambiguation to some extent.

In programming languages as well as in texts a certain amount of overloading occurs, sometimes even unnoticed and not every overload situation will be introduced with an abuse-warning; it’s often clear enough anyway. A modest and careful use of overloading can help understanding a text or a piece of source code. It depends also on the background of the audience. Especially for beginners of some field, when too many concepts are used with the same name, it can be confusing (“strange, earlier, if I remember correctly, XXX referred to such-and-such, but now, this XXX is used strangely, that makes no sense. What then **is** XXX **really** then, why can’t the author give me a definite and unambiguous definition?”. If familiar with the concept or in very obvious situations, one would not think twice, maybe not even notice.

We will pick up on overloading when discussing types and type checking later. Different uses of the same name will generally carry different types, and the type system typically assists in disambiguating the notation. Overloading is then discussed as one particular form of *polymorphism*, a property of type systems.

6.5 Symbol tables as attributes in an AG

Let’s have a short look at how to represent symbol tables (in an easy setting) with attribute grammars. We illustrate it on a fragment of a language covering expressions and declarations.

We have seen similar syntax before, for intance also in the chapter about attribute grammars. There we have seen attribute grammar examples for evaluating expressions and another example, dealing with declarations. The first one used synthesized attributes, the second one mostly inherited ones. The example here does not involve expression evaluation, it focuses on the declaration part, i.e., checking if a variable has been declared earlier. In a typical setting that would also involve type checking, i.e., not just checking when using a variable whether it has been declared before, but that the use of the variable is consistent type-wise with its earlier declaration. Also that typing aspect is absent in the example, but it would be straightforward to add.

With or without type-checking, to check conformity in this setting, the information flows is from the declarations to the uses of a variable. Since the declarations (here) come before the uses, it means, the declarations are higher-up in the syntax tree and the information therefore “flows” downwards. In other words, we are dealing conceptually with *inherited* attributes. We have seen that in the chapter about attribute grammars before. The grammar will not be solely on inherited attributes, there will also be synthesized ones.

Still, symbol tables can be seen as a realization of *inherited attributes*. At least in a simple situation like the one here, with a "declare-before-use" regiment. As discussed, there are more complex forms of declarations, notably those allowing recursion. Of course, also with recursive declarations, one can use symbol tables. Likewise one can also capture those more complex situations by attribute grammars. As discussed, for attribute grammars, cycles in the so-called dependency graphs of parse trees are strictly forbidden. Still one could capture recursive declarations by formalizing a "staged approach" and with additional attributes, like first adding partial information and then going through the abstract syntax tree a second time. In an attribute grammar, that may be done splitting a symbol-table attribute into an attribute capturing the preliminary stage with partially entered information and then the final version. Using two different attributes for that would break the forbidden cyclic dependencies. This way of approaching declaration-checking (or type checking) correspond also the way how one would do it working directly with symbol tables in an implementation (without considering it as an attribute grammar problem).

We don't cover recursive declarations, our attribute grammar will therefore be simpler. The small digression about recursion is added to dispell a possible (wrong) impression, that attribute grammars can only capture declare-before-use situations, due to their acyclicity restriction.

The example, however, deals with one important complication, namely *nested scopes*. This aspect was *not* covered yet in the earlier chapter about attribute grammars. The syntax we will be dealing with can similarly found in various languages. A small piece of code in ocaml is shown in Listing 6.27; remember also the code from Listing 6.12, used earlier when discussing simultanous vs. sequential declarations.

```
let x = 2 and y = 3 in
  (let x = x+2 and y =
       (let z = 4 in x+y+z)
   in print_int (x+y))
```

Listing 6.27: Nested lets (in ocaml)

6.5.1 Expressions and declarations: grammar

Let's start fixing the syntax by giving the grammar in BNF.

$$
\begin{array}{rcl}
S & \rightarrow & exp \\
exp & \rightarrow & (\,exp\,) \mid exp + exp \mid \mathbf{id} \mid num \mid \mathbf{let}\ dec\text{-}list\ \mathbf{in}\ exp \\
dec\text{-}list & \rightarrow & dec\text{-}list,\ decl \mid decl \\
decl & \rightarrow & \mathbf{id} = exp
\end{array}
$$

We want the following informal *rules* what's allowed and what's not for declarations. Those need to be captured by the semantic rules of the attribute grammar:

1. No identical names in the same let-block,
2. used names must be declared,
3. most-closely nested binding counts, and
4. *sequential* (non-simultaneous) declarations ($\neq$ ocaml/ML/Haskell ...)

These rules are illustrated in Listing 6.28. Note that we intend to use a *sequential*, not a simultaneous interpretation of declarations. In the exercises, one task will be to port the sequential treatment here to *simultaneous* declarations.

```
let x = 2, x = 3 in x + 1        (* no, duplicate  *)

let x = 2  in x+y                (* no, y unbound *)

let x = 2 in (let x = 3 in x)    (* decl. with 3 counts *)

let x = 2, y = x+1               (* one after the other *)
in  (let x = x+y,
         y = x+y
     in y)
```

Listing 6.28: Illustration of what's allowed and what not

The attributes used in the grammar are shown in Table 6.1. We also indicate which ones are inherited and which are synthesized. Note the special "status" of the attribute of the terminal symbol **id**. It has a special status in that we assume it injected by the scanner. The issue of attributes of terminals, whether should or should not be inherited, synthesized, or something else has been discussed in the attribute grammar chapter.

Let's discuss two further aspects of the attributes. Both aspects have to do with the fact that attribute grammars are a functional, declarative formalism, working with equations on attributes.

One aspect is *errors* and **error handling**. For the symbol table task, there will be situations that correspond to errors. In an implementation that may covered by raising an *exception* (and perhaps handling it). Exceptions and their treatment is not part of attribute grammars, in particular the fact that raising an exception means, breaking out of the normal, i.e., *un-exceptional* control flow. Exceptions can be part of the interface of the symbol-table. For instance, the attempt to look up a variable which has not been entered may result in a specific exception, likewise trying to enter a double binding at the same nesting level. That way, the programmer of the definedness-checker (or type checker) would not have to write code to check whether the mentioned conditions are met; the symbol-tables makes sure of that and maintain a corresponding invariant by themselves. Perhaps that's a more robust design, but of course the programmer of the checker still needs to write code to *handle* the exceptions properly, for instance like translating the symbol-table exception into more helpful user error message (which again could be done by catching the symbol-table expection and letting the handler (re-)raise another, more informative exception).

In general, evaluating a dependency graph for a parse tree corresponds to a tree traversal, in our setting, with mostly inherited attributes, a traversal "downwards" (and upwards afterwards, using a recursive procedure, like depth-first traversal). But there is no mechanism that would allow to stop the tree traversal after stumbling over an error (like an undeclared variable or a multiple declaration of the same variable at the same level). Instead the traversal has to continue, and one needs to somehow "simulate" the exceptional situation using attributes. That explains the error-attribute. The attribute is synthesized, since the error condition propagates from the place where it occurs up to the root. That's all fine and not actually complicated. However, to handle the non-erronous and the erroneous situation, the "user" of the attribute grammar has to add boiler-plate code all

over the place ("if no error do this else do that"). That clutters the semantic rules and makes the solution slightly unelegant (see the attribute grammar below). We will make a similar remark later in the chapter about type checking (where there is also a section that specifies a simple type system using an attribute grammar).

The second aspect I want to discuss is the **treatment of the symbol table.** We said that symbol tables somehow corresponds to a (mostly) inherited attribute. That's not incorrect. However, Table 6.1 shows that the attribute grammar has *3 attributes* to capture the symbol table. Partly that's caused by the fact that the attributes are on different (non-terminal) symbols. In particular `symtab` is an attribute of *exp*, whereas `intab` and `outtab` are attributes of *dec-list* resp. *decl*. That makes `symbtab` a different attribute from the other two in some way, and we might choose to simply rename `symbtab` to `intab` or `outtab` without changing the solution (though probably `symbtab` is a clearer choice). Anyway, it's not the point I want to make, the point is about the attributes of *decl* and *dec-list*, `intab` and `outtab`. There, we really need to have two *different* attribute names. Declarations and declaration lists **change** the symbol table insofar that new binding(s) are added (unless an "exception" occurs).

That means, the state of the symbol table before the declaration or before a list of declaration is typicalled different from the state afterwards. This is captured by the two different attributes, `intabl` and `outtab`. In many languages, the symbol table would conventionally implemented **imperatively** (though also efficient functional implementations like using red-black trees) exist. For instance, the hash-tables which often underly symbol-tables are conventionally an imperative data structure. That means, in an implementation, handling a declaration is an operation that *changes* the symbol-table. In the attribute grammar here, we specify how a declaration transforms the symbol-table from the state before (`intab`) to its state afterwards (`outtab`, because we are working with equations, which are side-effect free.

symbol	attributes	kind
exp	`symtab`	inherited
	`nestlevel`	inherited
	err	synthesized
dec-list, *decl*	`intab`	inherited
	`outtab`	synthesized
	`nestlevel`	inherited
id	`name`	injected by scanner

Table 6.1: Attributes for the symbol tables

Attributes in attribute grammars are generally typed. We don't explicitly list the types in a separate table; they should mostly be clear: nesting level is an integer, actually a non-negative one, the outermost nesting is counted as level 0, as the attribute grammar shows. The error attribute is a kind of boolean: is there an error, yes or no? As said, in a practical situation (with or without exception), one might choose to refine it with information about what kind of error occured and/or where. The most complex data structure, of course, is the symbol table itself. For that Table 6.2 contains the *interface* and that's is type-related information.

We see in particular in the signature of the `insert` function that it returns a (changed) symbol table, instead of changing the state of the argument `tab` in-place.

return type		
symboltable	`insert(tab,name,lev)`	returns a changed table
bool	`isin(tab,name)`	boolean check
int	`lookup(tab,name)`	gives back *level*
symboltable	`emptytable`	you have to start somewhere
	`errtab`	erroronous table

Table 6.2: Interface of the symbol table

Treatment of nested scopes here

A few words also on the "design" of the symbol tables in connection with nested scopes. In earlier sections, we discussed the issue to some extent (chained symbol tables or specific arrangements in hash-table).

Here, the attribute grammar operates with nesting levels. The nesting level is explicitly handed over as argument when entering a binding in `insert`. That's another way of dealing with nested block structures.

One central production in that context is, of course, the one dealing with let-declarations. The production and the semantic rules dealing with the nesting level are shown in equation (6.1).

$$
\begin{array}{ll}
exp_1 ::= \textbf{let}\ dec\text{-}list\ \textbf{in}\ \ exp_2 & dec\text{-}list.\texttt{nextlevel} = exp_1\,.\texttt{nextlevel} + 1 \\
 & exp_2\,.\texttt{nextlevel} = dec\text{-}list.\texttt{nextlevel}
\end{array}
\tag{6.1}
$$

When processing the let-declaration, the nesting level is increased by one for *both* the declaration list and the body exp_2 of the declaration. Assume that exp_1 occurs at a nesting depth of n. That exp_2 is processed at a nesting level $n+1$ is clear.

For *expressions* in a *dec-list*, they conceptually are occuring at level n, at the *same* level than exp_1, the next nesting level kicks in only in the body. So the level $n+1$ for *dec-list* is to be interpreted as "use $n+1$ as level when adding a new declaration", thus building up (at level n) the bindings *for* the body at level $n+1$.

Grammar Rule	Semantic Rules
$S \rightarrow exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$
$exp_1 \rightarrow exp_2$ **+** exp_3	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$
$exp_1 \rightarrow$ **(** exp_2 **)**	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$
$exp \rightarrow$ **id**	$exp.err =$ **not** $isin(exp.symtab,$**id**$.name)$ } 2
$exp \rightarrow$ **num**	$exp.err =$ **false**
$exp_1 \rightarrow$ **let** $dec\text{-}list$ **in** exp_2	$dec\text{-}list.intab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-}list.outtab$ } 3 $exp_2.nestlevel = dec\text{-}list.nestlevel$ $exp_1.err = (decl\text{-}list.outtab = errtab)$ **or** $exp_2.err$
$dec\text{-}list_1 \rightarrow dec\text{-}list_2$ **,** $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ } 4 $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$
$dec\text{-}list \rightarrow decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ } 4 $dec\text{-}list.outtab = decl.outtab$
$decl \rightarrow$ **id** **=** exp	$exp.symtab = decl.intab$ $exp.nestlevel = decl.nestlevel$ $decl.outtab =$ **if** $(decl.intab = errtab)$ **or** $exp.err$ **then** $errtab$ **else if** $(lookup(decl.intab,$**id**$.name) =$ $decl.nestlevel)$ } 1 **then** $errtab$ **else** $insert(decl.intab,$**id**$.name, decl.nestlevel)$

Figure 6.3: Attribute grammar

Bibliography

Bibliography

[1] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

Index

attribute grammar, 21
  error handling, 23
  symbol table, 21

binding, 11
  dynamic, 17
block structure, 7, 11

class, 15
closed hashing, 6

declaration
  forward, 16
dependence graph, 22
dictionary, 5
dynamic binding, 17

forward declaration, 16

hash conflict, 6
hash table, 5

look-up table, 5

nested procedure, 9

open addressing, 6
open hashing, 6
overloading, 21

PLT scheme, 3
procedure
  nested, 9

Racket, 3
recursive declaration, 22
run-time environment, 9

scope resolution operator, 12
scoping, 11
search tree, 5
separate chaining, 6
static link, 10
symbol table, 1
  interface, 3