Compiler Construction

Chapter 3 – Symbol Tables

Prof. Dr. Jörg Kreiker
joerg.kreiker@informatik.hs-fulda.de

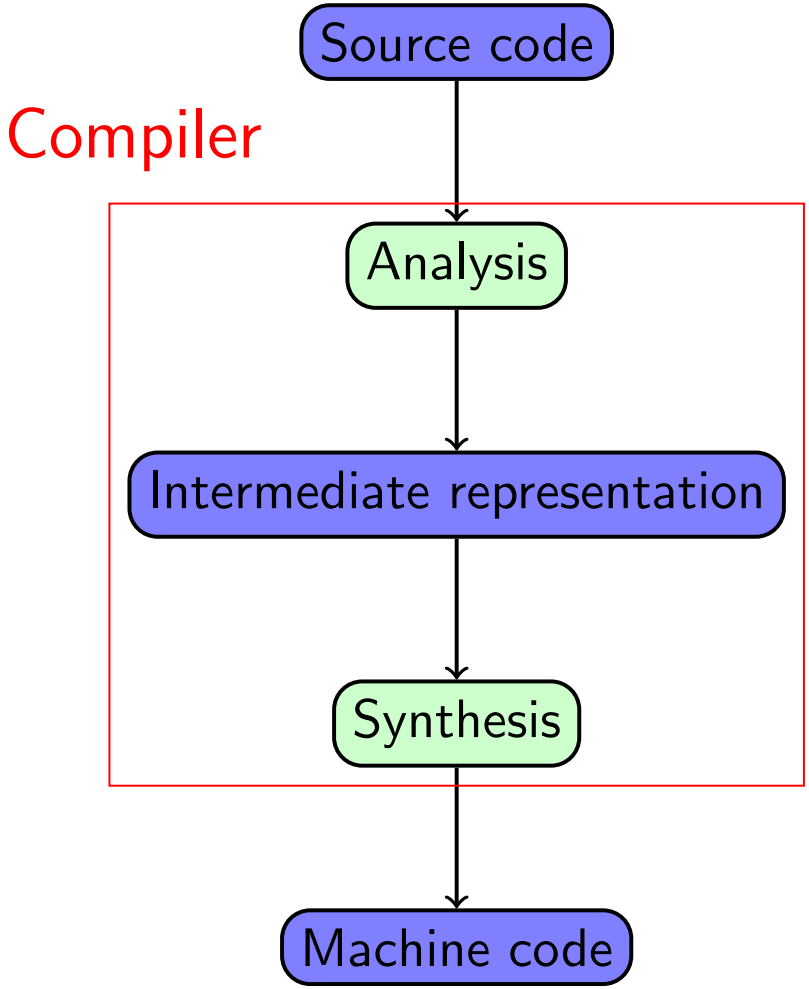
Fachbereich Angewandte Informatik
Hochschule Fulda – University of Applied Sciences

May 11, 2017

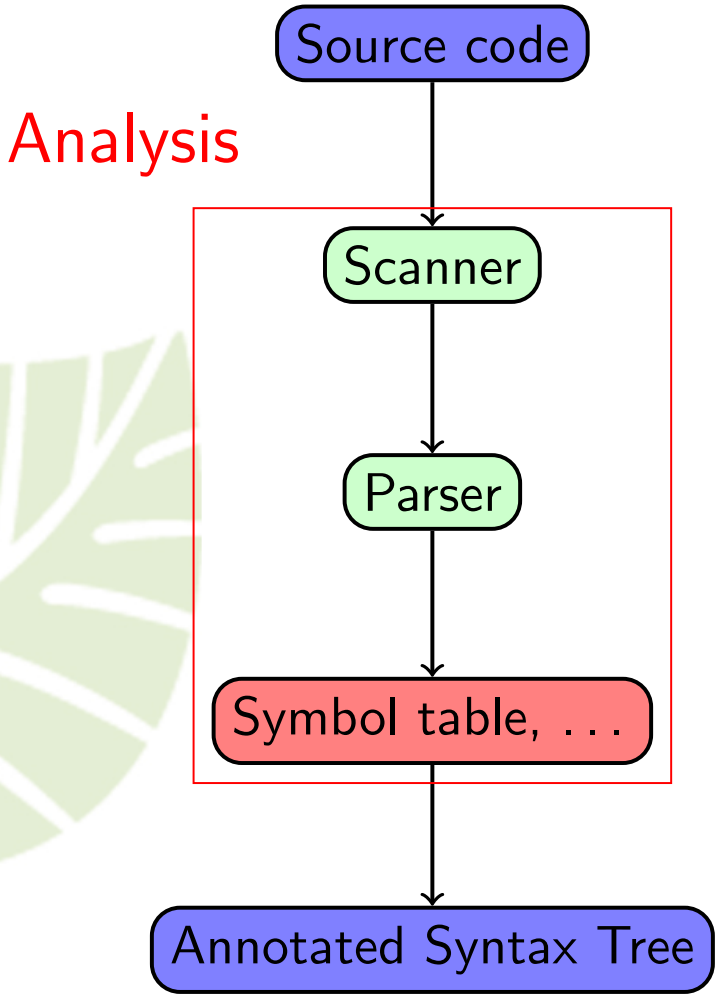
Agenda

- Overview
- From names to numbers: symbol tables
- Declaration $\leftrightarrow$ checking
- Beware!

Compiler Overview



Analysis Phase



Lexical analysis → Token stream

Syntactical analysis → Syntax tree

Semantical analysis

Scope

Consider the following snippet of C code

```
void foo(int a) {
  a++;
  if (a) {
     int a;
     a++;
  }
}
```

- Within the `if` statement, the definition of parameter `a` is shadowed by the local definition
- For code generation, for every use of an identifier, we need to know its definition
- lexical scoping: the definition of a name is valid within the defining block; it is visible only if it is not shadowed within its defining block

Checking Name

For each use of a name, find the matching declaration!

1. Replace names by unique numbers
   - Comparing numbers is faster
   - Save memory
2. Find the matching declaration for each occurrence of a name
   - Need to allocate memory at run-time for each declaration of a name
   - In languages without explicit declarations, an implicit declaration is generated at the first occurrence

Numbers instead of names

Idea

Input Sequence of strings

Output

1. Sequence of numbers
2. Table mapping numbers to strings

Example

Input

	science	is	what	you	know		
	philosophy	is	what	you	do	not	know

Output

0	1	2	3	4	5	1	2	3	6	7	4

0	science
1	is
2	what
3	you
4	know
5	philosophy
6	do
7	not

Implementation

- Partial map S : **String** $\rightarrow$ **int**
- Keep a counter `int count=0;` for the number of distinct words found

Pseudocode

```
int getIndex(String w) {
    if (S(w) undefined)
        S = S + [ w -> count ];
        return count++;
    else return S(w);
}
```

Data structures for partial maps

Assume n pairs of names and unique numbers

- List of pairs
  - insert $\mathcal{O}(1)$
  - find $\mathcal{O}(n)$
- Balanced trees
  - insert $\mathcal{O}(\log(n))$
  - find $\mathcal{O}(\log(n))$
- Hashtables
  - insert $\mathcal{O}(1)$ (on average)
  - find $\mathcal{O}(1)$ (on average)

Symbol Tables

- Check that only declared names are used
- Traverse the syntax tree such that
  - Definitions are visited before uses
  - Currently visible definition is most recently visited
- For each name, manage a stack of valid definitions
- Push declarations onto the stack when met
- Pop declarations, when the scope is left
- When a name is used during traversal, look up the declaration on top of the stack

  found create reference from the point of use to the declaration

  not found error

Example

Nested Scopes

```
int a, b;
b = 5;
if (b>3) {
    int a, c;
    a = 3;
    c = a+1;
    b = c;
} else {
    int c;
    c = a+1;
    b=c;
}
b = a+b;
```

Mapping of names to numbers

0	a
1	b
2	c

Redeclarations

- Programming languages such as `C` disallow redeclaration of the same name within the same block

→ Assign unique numbers to each block

- For each declaration, store the enclosing block number
- If there is another declaration of the same name with identical block numbers: error

Protip: Instead of unique numbers, references to blocks may be used.

Forward Declarations

In order to allow for recursive data structures programming languages allow forward declarations

Crazy Lists

```
struct list0 {
    int info;
    struct list1* next;
}
```

```
struct list1 {
    int info;
    struct list10 next;
}
```

Function Declaration

- The name of recursive functions must be entered into the table before traversing the function body
- This holds for all function names in the case of mutually recursive functions

Caution

- Languages may distinguish a number of identifiers
  - variables
  - type names
  - function names
  - class names
- Overloading
- C typedef parsing problem (avoided here)