Sequence 3.1 – Variables and scopes

P. de Oliveira Castro S. Tardieu

What is a variable?

- A variable is an entity in which a value can be stored.
- The variable may correspond to a place in memory (RAM).
- The variable may correspond to a processor register.
- The variable may be intermittently stored in memory and in one or several registers.

The compiler job is to ensure consistency between uses and modifications of the variable regardless of where it is stored.

How are variables declared and used in Tiger?

- Variables are declared using the `var` keyword as part of a `let` expression.
- Variables are statically typed: their type is determined during the compilation process and cannot change later.
- Variable types can be explicit (given by the programmer) or implicit (inferred from the context by the compiler).

```
let var a : int := 3    /* Explicit typing */
    var b := 4          /* Implicit typing */
in
  a := a * 2;           /* Now a contains 6 */
  a + b                 /* Evaluates as 10 */
end
```

Variable lifetime

- A *scope* captures the set of variables defined in a program portion.
- At any time, zero, one or more scopes are visible and are used to locate variables.
- In Tiger, new scopes are created by the `let` expression and by function declarations.
- A variable exists and can be referred to from its point of declaration to the end of the scope it is declared into. The span between its declaration to the end of the scope is called the variable *lifetime*.

```
let var a := 3       /* a exists after this line */
    var b := a + 4   /* b exists after this line */
in
  ... /* a and b can be used here */
end  /* a and b do longer exist after this line */
```

Variable lookup

- Variables are looked up from the innermost scope to the outermost scope.

```
1  let var a := 3
2      var b := a + 4  /* a from L.1 is used */
3  in
4    let var a := 50
5    in
6      a := a * 3;     /* a from L.4 is used */
7      a + 1
8    end
9  end
```

- We say that the second declaration of a in line 4 temporarily *masks* any existing declaration of a such as the one in line 1.

Variable lookup and types

- Since an innermost declaration of a variable with the same name as an existing one masks the previous one, they must be considered as completely different entities.
- A variable masking a similarly named one can therefore be of a different type.

```
1  let var a := "Hello"
2  in
3    let var a := 3 in
4      ...   /* Here, a is an integer */
5    end;  /* Integral a lifetime end */
6    print(a);  /* This prints the string Hello */
7  end
```

Functions also create a scope

- Function declarations also create a new scope, containing the function parameters.

```
let var a := 3
    function f(a: string) = print(a)
    var b := a + 4
in  /* The innermost scope contains a, f, and b */
  f("The value of b is ");
  print_int(b)
end
```

- Inside the function declaration scope, a is a string.
- Outside the function declaration scope, a is an integer.
- The program above prints: The value of b is 7

Example of multi-level lookups

- When a name cannot be found in the innermost scope, the compiler looks it up in the scope containing the innermost scope.
- It goes up a level until the variable is found or there is no englobing scope.

```
1  let var a := 3
2      var b := 4
3  in
4    let var a := 10
5    in
6      a + b /* Uses a from L.4 and b from L.2: 14 */
7    end
8  end
```

Mutually recursive functions

- In order to allow mutually recursive functions, Tiger makes functions declared contiguously in the same block visible from each other regardless of their order of declaration.

```
let function f(a: int): int = g(a+1)
    function g(a: int): int = f(a-1)
    var b := 4
    function h(a: int): int = f(a) + g(a) + b
in
  f(3)
end
```

- `f` and `g` can see each other.
- `h` is not declared contiguously to `f` and `g` because of `b`. `h` can see `f` and `g`, but those cannot see `h`.

Conclusion

- Variables have a lifetime determined by their lexical scope, from the point of declaration to the end of the current scope.
- New scopes are created when `let` expressions or function declarations are encountered, and last until the end of the `let` expression or function declaration.
- A variable declared in a scope masks similarly named variables until the end of its lifetime.
- Function declarations grouped together allow for mutually recursive functions.