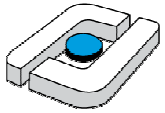

Hochschule Osnabrück
University of Applied Sciences

The Tiger Programming Language

(Appel, Appendix A)

Lecture "Compiler Construction"

Prof. Dr.-Ing. Markus Weinhardt

Labor für Digital- und Mikroprozessortechnik

Source: Lecture Notes "Compiler Construction", © T. Grust, Univ. Konstanz

The Tiger Language

- The **source language Tiger** is a small—yet interesting and complete—procedural programming language.

Tiger feature	**in C?**
usual control constructs (`while`, `if`)	☐
built-in types `int`, `string`	☐
user-defined types	☐
arrays with reference semantics	☐
record values with reference semantics	☐
no (explicit) pointers	☐
nested scopes (`let`)	☐
nested functions	☐
small standard library of support functions	☐

Tiger Programs

01.tig

```
1  /* sample Tiger program */
2
3  print ("Roar!\n")
4
5  /* known escape sequences:
6     \n        newline
7     \t        TAB
8     \^c       control character c (e.g., \^G = BEL)
9     \ddd      ASCII character with decimal code ddd
10    \"        double quote (")
11    \\        the backslash character
12    \...\     line continuation
13 */
```

02.tig

```
1  /* standard library:
2   * conversion int <-> string, string concatenation
3   */
4
5  for c := 0 to 2 do
6          print (concat (concat ("Roar #", chr (c + ord ("0"))), "\n"))
7
```

Tiger Programs

03.tig

```
1  /* function declaration, function call */
2
3  let function digit (c : int) : string =
4      chr (c + ord ("0"))
5  in
6      for c := 0 to 2 do
7          print (concat (concat ("Roar #", digit (c)), "\n"))
8  end
```

04.tig

```
1  /* nested scopes */
2
3  let function digit (c : int) : string =
4      let var zero := ord ("0")
5      in
6          chr (c + zero)
7      end
8  in
9      for c := zero to 2 do
10         print (concat (concat ("Roar #", digit (c)), "\n"))
11 end
```

Tiger Programs

05.tig

```
1  /* procedure declaration, procedure call, sequencing */
2
3  let function newline () =
4      print ("\n")
5  in
6     print ("foo");
7     print ("bar");
8     newline ()
9  end
```

06.tig

```
1  /* nested scopes, hiding */
2
3  let var v := "foo"
4  in
5      print (v);
6      let var v := "bar"
7      in  print (v)
8      end;
9      print (v);
10     print ("\n")
11 end
```

Tiger Programs

07.tig

```
1  /* sequenced expressions (...;...) */
2
3  let var x := "foo"
4      var y := "bar"
5      var z := ""
6  in
7      print ((z := concat (x, y); z := concat (z, "\n"); z))
8  end
```

- Similarly, the value of `let ... in` e_1 `;...;` e_n `end` is the value of e_n (if e_n has a value).

Tiger Programs

08.tig

```
1  /* arrays,
2   * array access,
3   * arrays have reference semantics
4   */
5
6
7  let type strings = array of string
8
9      var a := strings [10] of "foo"
10     var b := a
11 in
12     print (a[0]);
13     print (a[9]);
14     b[5] := "bar";
15     print (a[5]);
16     print ("\n")
17 end
```

- Compare this to the C semantics of the type t [10] for some C type t.

Tiger Programs

09.tig

```
1  /* ''matrices'',
2   * separate type and function/variable name spaces
3   */
4
5  let type vec = array of string
6      type mat = array of vec
7
8      var row := vec [10] of "foo"
9      var mat := mat [10] of row
10 in
11     let var r1 := mat[2]
12         var r2 := mat[3]
13     in
14         r1[5] := "bar";
15         print (r2[5])
16     end
17 end
```

-  Which string value will the above Tiger program print?

Tiger Programs

10.tig

```
1  /* records,
2   * record constants,
3   * record access,
4   * nil,
5   * records (like arrays) have reference semantics
6   */
7
8  let
9      type strings = { hd: string, tl: strings }
10
11     var xs := strings { hd = "foo", tl = nil }
12     var ys := xs
13 in
14     print (ys.hd);
15     xs.hd := "bar";
16     print (ys.hd);
17     print ("\n")
18 end
```

Tiger Programs

11.tig

```
1  /* build a string from a linked list of digits,
2   * nested functions
3   */
4
5  let type ints = { hd: int, tl: ints }
6
7      function digits (ds : ints) : string =
8          let
9              function digit (d : int) : string =
10                 chr (d + ord ("0"))
11         in
12             if ds = nil then ""
13                         else concat (digit (ds.hd), digits (ds.tl))
14         end
15
16     var i := ints { hd = 4, tl = ints { hd = 2, tl = nil } }
17 in
18     print (digits (i));
19     print ("\n")
20 end
```

Tiger Programs

12.tig

```
1  /* read a string from the terminal, print its reverse image,
2   * standard library: getchar (), substring (s,fst,len), size (s)
3   */
4
5  let
6      function reverse (s : string) : string =
7          if size (s) <= 1 then s
8          else
9              concat (reverse (substring (s, 1, size (s) - 1)),
10                     substring (s, 0, 1))
11
12     /* read string from terminal until newline seen */
13     function getstring () : string =
14         let var c := getchar()          /* c is string of length 1 */
15         in
16             if c = "\n" then ""
17                         else concat (c, getstring ())
18         end
19 in
20     print (reverse (getstring ()));
21     print ("\n")
22 end
```

Additional Remarks

- Tiger supports `while` e_1 `do` e_2 as well as `break` with the "obvious" semantics.

Obvious?

break.tig

```
1  /* semantics of nested break */
2
3  let function f () =
4        let function g (i : int) : int =
5              if i = 3 then break
6        in
7            for i := 0 to 9 do
8              (g (i); print (chr (i + ord ("0"))))
9        end
10 in
11     f ()
12 end
13
```

Additional Remarks

- Functions may be (mutually) recursive.
- Type declarations may be (mutually) recursive *as long as any cycle in a group of recursive type declarations passes through an array or record type.*

Why that?

Example:

forest.tig

```
1
2  let type forest = { hd: tree, tl: forest }
3      type tree   = { node: int, children: forest }
4  in
5     tree { node = 1,
6            children = forest { hd = tree { node = 2,
7                                            children = nil },
8                                tl = forest { hd = tree { node = 3,
9                                                          children = nil },
10                                             tl = nil }
11                               }
12           }
13 end
```