TDT4205 - PS 4

Øyvind Skaaden (oyvindps@stud.ntnu.no)

March 19, 2022

1 Three-Address Code

We have the following code in listing 1.1. The translation to TAC can be seen in listing 1.2. I have used the "TAC standard" from the Dragon-Book in section 6.2.1 on page 364-366.

```
1  func catalan(n)
2  begin
3      print factorial(2*n) / (factorial(n+1)*factorial(n))
4      return 0
5  end
6
7  func factorial( n )
8  begin
9      var i, result
10     result := 1
11     i := 1
12     while i < (n+1) do
13     begin
14         result *= i
15         i += 1
16     end
17     return result
18 end
```

Listing 1.1: The code for calculating the n^{th} catalan number.

```
// CATALAN function
// name of parameter: c_n
// temp var on the form: c* where * is number
catalan: // (c_n)
    c1 = 2 * c_n            // Calculate 2·n and store in c1
    param c1                // Prepare c1 to be used in function
    c2 = call factorial, 1  // Call function factorial with 1 parameter (c1)
    c3 = c_n + 1            // Calculate n+1 and store in c3
    param c3                // Prepare c3 to be used in function
    c4 = call factorial, 1  // Call function factorial with 1 parameter (c3)
    param c_n               // Prepare c_n (n) to be used in function
    c5 = call factorial, 1  // Call function factorial with 1 parameter (c_n)
    c6 = c4 * c5            // Calculate c4 * c5, actually (n+1)!n!
    c7 = c2 / c6            // Calculate c2 / c6, actually 2n!/((n+1)!n!)
    param c7                // Prepare c7 with the result to be printed
    call print, 1           // Call print
    return 0


// FACTORIAL function
// name of parameter: f_n
// local variables: result, i
// temp var on the form: f* where * is number
factorial: // (f_n)
    result = 1                          // Prepare result with 1
    i = 1                               // Prepare i with 1
    f1 = f_n + 1                        // Prepare f1 with n+1
factLoop:                               // Start of factorial loop
    ifFalse i < f1 goto aftFactLoop     // Check if i < n+1, if no, end loop
    f2 = result * i                     // Calculate result·i
    result = f2                         // Put that back in result
    f3 = i + 1                          // Calculate i+1
    i = f3                              // Put that back into i
    goto factLoop                       // Go back to start of loop
aftFactLoop:                            // After while loop
    return result                       // Return the result
```

Listing 1.2: TAC for calculating the n^{th} catalan number.

2 Symbol Table Creation

See attached code for full overview.

PSA: I noticed some bugs in the previous problem set. My approach to how to simplify the tree was in some cases too agressive. I have now changed it to be more flexible and added a resolve relations experimental function.

2.1 Initialization

The `void create_symbol_table ( void )` function initializes all the ”global” variables used in the `ir.c` file. Then it finds all the globals and then traverses each of the functions on the lookout for local variables, and local lookup.

2.2 Finding globals

Go trough the first level in the root node, and find all global declarations and functions. If there is a function, find all the parameters and give them a sequence number.

2.3 Finding local symbols and strings

For each of the functions found in the previous step, find all the remaining variables. Continue the sequence number from the parameters.

When finding a identifier that is not a declaration, look it up. First in the local scopes, then in the paramteters, then the global variables.

2.4 Printing and cleanup

When printing, the output contains the type, the identifier, the sequence number and the node that it is connected to. The list of found strings will also be printed. A simple example symbol table from the `globals.vsl` is shown in listing 2.1 (This looks a lot better in terminal/file). Everything is tested with Valgrind, all heaps are freed, no memory leaks possible.

```
1  -FUNCTION: main                  [nparams= 0, seq= 1, node=0x56225ccc10c0]
2   *-[LOCAL_VAR]: a                            [seq= 0, node=0x56225ccc0d80]
3
4  -FUNCTION: my_func               [nparams= 2, seq= 0, node=0x56225ccc0b10]
5   |-[PARAMETER]: param0                       [seq= 0, node=0x56225ccc0630]
6   |-[PARAMETER]: param1                       [seq= 1, node=0x56225ccc06f0]
7   *-[LOCAL_VAR]: a                            [seq= 2, node=0x56225ccc0800]
8
9  -GLOBAL_VAR: global_var1         [nparams= 0, seq= 0, node=0x56225ccc0410]
10
11 -GLOBAL_VAR: global_var0         [nparams= 0, seq= 0, node=0x56225ccc0350]
12
13 -STRINGS [1]
14  *-[0]: "a string"
```

Listing 2.1: Symbol table for the `globals.vsl` program.

A Custom lexer for `minted`

Use the python code in listing A.1 to lex and highlight your VSL code in minted. Just add the `.py` file in the root directory of the main LaTeX file. Then when choosing language in minted, use `vsl.py:VSLLexer -x` as language.

```python
from pygments.lexer import RegexLexer, bygroups
from pygments.token import *

class VSLLexer(RegexLexer):
    name        = "VSL"
    aliases     = ["vsl"]
    filenames   = ["*.vsl"]

    tokens = {
        "root": [
            (r"[\ \t\v\r\n]+", Whitespace),
            (r"\/\/[^\n]+", Comment.Single),
            (r"var", Keyword.Declaration),
            (r"func|print|return|continue|if|then|else|while|do|begin|end", Keyword),
            (r"\^|\||:|=|\+|-|\*|\/|<|>|&", Operator),
            (r"[0-9]+", Number.Integer),
            (r"([A-Za-z_][0-9A-Za-z_]*)([\ \t\v\r\n]*)(\()", bygroups(Name.Function,
            ↪  Whitespace, Punctuation)),
            (r"\(|\)|\[|\]|{|}", Punctuation),
            (r"[A-Za-z_][0-9A-Za-z_]*", Name.Variable),
            (r"\"([^\"\n]|\\\")*\"", String),
            (r".", Text),
        ]
    }
```

Listing A.1: Pygments lexer class for highlighting VSL.

This can also be found at `https://gist.github.com/oyvindskaaden/9ca83c1a0a972f45b6a349faccbd5f74`. There is a usage example in appendix A.1 (next page).

A.1 Example usage

With `minted` enivironment, you can use the code in listing A.2 as an example.

```
\begin{minted}{vsl.py:VSLLexer -x}
func add(a, b) begin
    return a + b
end

func main()
begin
    print add(40, 2)
end
\end{minted}
```

Listing A.2: Example usage of lexer and highlighter.

This will generate the output in listing A.3.

```
func add(a, b) begin
    return a + b
end

func main()
begin
    print add(40, 2)
end
```

Listing A.3: Example output of lexer and highlighter.