

# 50006 - Compilers - (Prof Kelly) Lecture 3

Oliver Killane

10/01/22

## Simple Programming Language

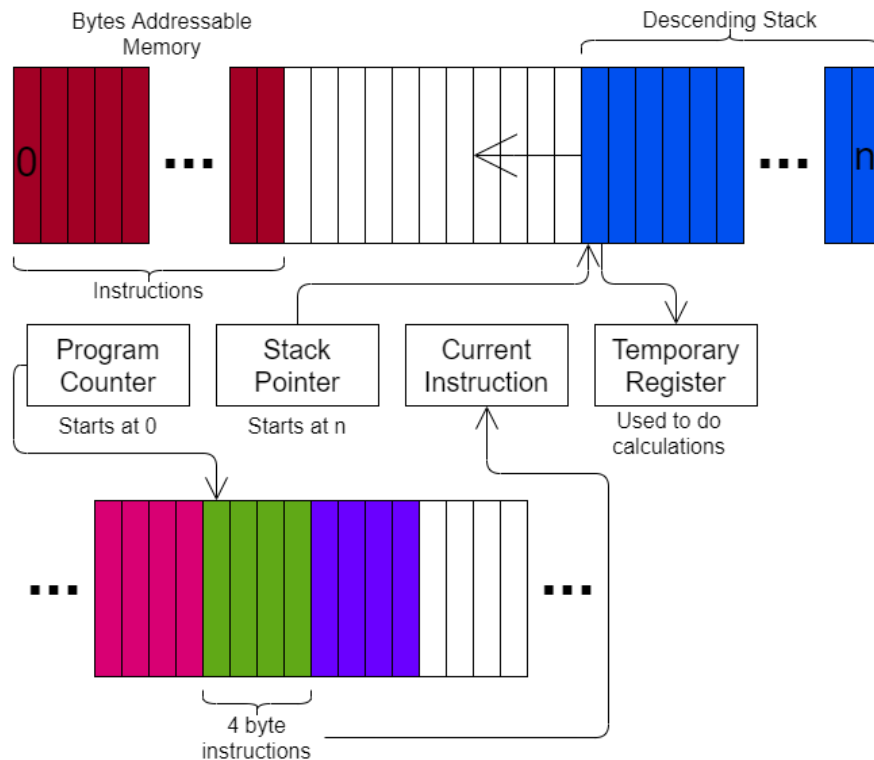
The grammar is expressed as:

$$\begin{aligned} stat &\rightarrow ident \text{ ':=' } exp \mid stat \text{ ';' } stat \mid \text{ 'for' } ident \text{ 'from' } exp \text{ 'to' } exp \text{ 'do' } stat \text{ 'od' } \\ exp &\rightarrow exp \text{ binop } exp \mid unop \text{ } exp \mid ident \mid num \\ binop &\rightarrow \text{ '+' } \mid \text{ '-' } \mid \text{ '*' } \mid \text{ '/' } \mid \\ unop &\rightarrow \text{ '-' } \end{aligned}$$

And abstract syntax tree as:

```
1 data Stat =
2   Assign Name Exp |
3   Seq Stat Stat |
4   ForLoop Name Exp Exp Stat
5   deriving (Show)
6
7 data Exp =
8   BinOp Op Exp Exp |
9   Unop Op Exp |
10  Ident Name |
11  Const Int
12  deriving (Show)
13
14 data Op =
15   Plus |
16   Minus |
17   Times |
18   Divide
19   deriving (Show)
20
21 type Name = [Char]
```

# Target Stack Machine



Assembly instructions (Some are directives/pseudoinstructions):

```

1  data Instruction
2    = Add | Sub | Mul | Div
3    | PushImm Int    — Push an immediate value
4    | PushAbs Name   — push variable at given location on the Stack
5    | Pop Name       — remove the top of the stack and store at location name
6    | CompEq         — Subtract top two elements of the stack, replace with a
7                      — 1 is the result was zero, zero otherwise
8    | Jump Label     — Jump to the label
9    | JTrue Label    — Remove top item from stack, if 1 jump to label
10   | JFalse Label   — Remove top item from stack, if 0 jump to label
11   | Define Label   — Set destination for jump (An assembler directive,
12                      — not instruction).
```

Pseudocode for execution behaviour:

```

1  ADD / MINUS / MUL / DIV /:
2    T := store[SP]
3    SP := SP + 4
4    T := store[SP] [+*=/] T
5    store[SP] := T
6
7  PUSHIMM:
8    SP := SP - 4
9    store[SP] := operand(IR)
```

```

11 PUSHABS:
12     T := store[operand(IR)]
13     SP := SP - 4
14     store[SP] := T
15
16 POP:
17     T := store[SP]
18     SP := SP + 4
19     store[operand(IR)] := T
20
21 COMPEQ:
22     T:= store[SP]
23     SP := SP + 4
24     T := store[SP] - T
25     store[SP] = T=0 ? 1 : 0
26
27 JTRUE:
28     T := store[SP]
29     SP := SP + 4
30     PC := T=1 ? operand(IR) : PC
31
32 JFALSE:
33     T := store[SP]
34     SP := SP + 4
35     PC := T=0 ? operand(IR) : PC

```

#### Typical Assembly

```

1  start:
2      PushAbs i
3      PushImm 1
4      Sub
5      Pop i
6      PushAbs i
7      PushImm 100
8      CompEq
9      JTrue start

```

#### Compiler Code Generated

The

```

1  [
2      Define "start",
3      PushAbs "i",
4      PushImm 1,
5      Sub,
6      Pop "i",
7      PushAbs "i",
8      CompEq,
9      Jtrue "Start"
10 ]

```

define directive (and assembly label) are directives to make the linker/assembler convert jumps to the label into jumps to the memory address of the instruction immediately after the label.

## Translation (Naive Implementation)

```

1  data Stat =
2      Assign Name Exp |
3      Seq Stat Stat |
4      ForLoop Name Exp Exp Stat
5      deriving (Show)
6
7  data Exp =
8      BinOp Op Exp Exp |
9      Unop Op Exp |
10     Ident Name |
11     Const Int
12     deriving (Show)
13
14  data Op =

```

```

15   Plus |
16   Minus |
17   Times |
18   Divide
19   deriving (Show)
20
21   type Name = [Char]
22
23   data Instruction
24     = Add | Sub | Mul | Div
25     | PushImm Int    -- Push an immediate value
26     | PushAbs Name   -- push variable at given location on the Stack
27     | Pop Name       -- remove the top of the stack and store at location name
28     | CompEq         -- Subtract top two elements of the stack, replace with a
29                     -- 1 is the result was zero, zero otherwise
30     | Jump Label     -- Jump to the label
31     | JTrue Label    -- Remove top item from stack, if 1 jump to label
32     | JFalse Label   -- Remove top item from stack, if 0 jump to label
33     | Define Label   -- Set destination for jump (An assembler directive,
34                     -- not instruction).
35
36   type Label = [Char]
37
38   transExp :: Exp -> [Instruction]
39   transExp (BinOp op e1 e2)
40     = transExp e1 ++ transExp e2 ++ [case op of
41       Plus -> Add
42       Minus -> Sub
43       Times -> Mul
44       Divide -> Div]
45   transExp (Unop Minus e)
46     = transExp e ++ [PushImm (-1), Mul]
47   transExp (Unop _ e)
48     = error "(transExp) Only '-' unary operator supported"
49   transExp (Ident id) = [PushAbs id]
50   transExp (Const n) = [PushImm n]
51
52   transStat :: Stat -> [Instruction]
53   transStat (Assign id exp) = transExp exp ++ [Pop id]
54   transStat (Seq s1 s2) = transStat s1 ++ transStat s2
55
56
57   {-
58   for x:e1 to e2 do
59     body
60   od
61
62   x := <e1>
63   loop:
64     if <e2> then goto break
65     <body>
66     x := x + 1
67     goto loop
68   break:
69
70
71   <transExp e1>
72   Pop x
73   define "loop"
74   <tranEval e2>

```

```

75  CompEq
76  JTrue "break"
77  <transStat body>
78  PushImm 1
79  Add
80  Pop x
81  Jump "loop"
82  Define "break"
83  -}
84
85  — assumes the labels will be unique, this almost always not the case
86  transStat (ForLoop x e1 e2 body)
87  = transExp e1 ++ Pop x:Define "loop":transExp e2 ++ CompEq:JTrue "break":transStat
    ↪ body ++ [PushImm 1, Add, Pop x, Jump "loop", Define "break"]

```

## Intermediate Representations

- **Abstract Syntax Tree** Usually the first intermediate representation. Can include statements, operations and expressions in a uniform way (simple data structure)  
Useful for sophisticated instruction selections and register allocation.
- **Flattened Control Flow Graph** Represents assembler-level code  
Order of operations defines control flow, useful for loop-invariant code motion.
- **Dependency Based Graphs** More complex, used by most modern compilers.  
Used for optimisations, can create 'static single assignment' graphs to deal with dependencies on mutable data.