50006 - Compilers - (Prof Kelly) Lecture 3

Oliver Killane

10/01/22

Lecture Recording

Lecture recording is available here

Simple Programming Language

The grammar is expressed as:

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

And abstract syntax tree as:

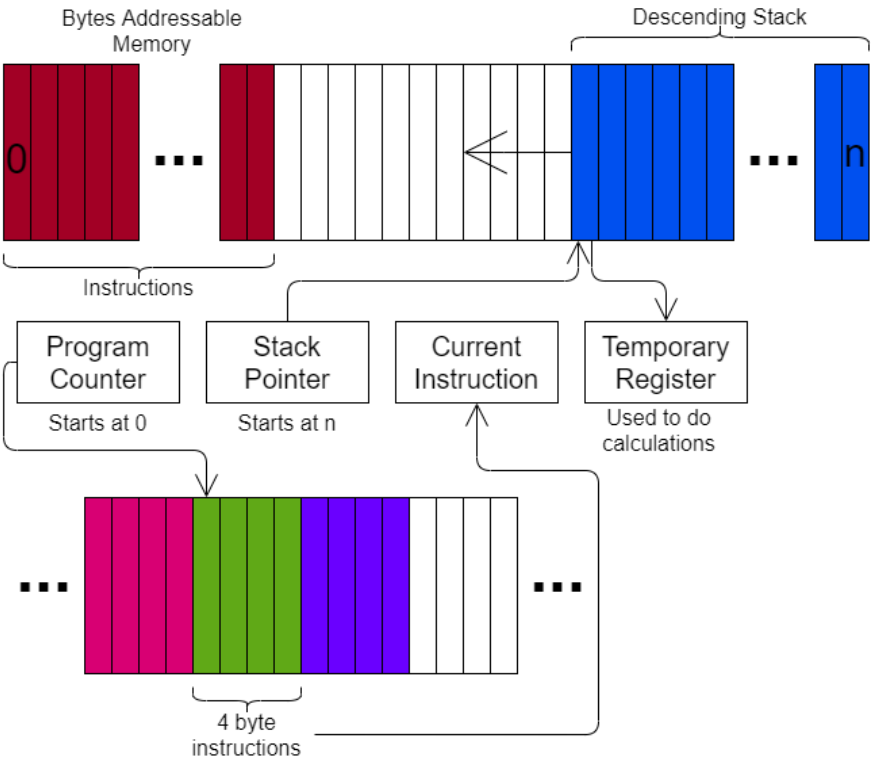
```haskell
data Stat =
    Assign Name Exp |
    Seq Stat Stat |
    ForLoop Name Exp Exp Stat
    deriving (Show)

data Exp =
    BinOp Op Exp Exp |
    Unop Op Exp |
    Ident Name |
    Const Int
    deriving (Show)

data Op =
    Plus |
    Minus |
    Times |
    Divide
    deriving (Show)

type Name = [Char]
```

Target Stack Machine



Assembly instructions (Some are directives/pesudoinstructions):

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

Pesudocode for execution behaviour:

```
ADD / MINUS / MUL / DIV /:
    T := store[SP]
    SP := SP + 4
    T := store[SP] [+=*/] T
    store[SP] := T

PUSHIMM:
    SP := SP - 4
    store[SP] := operand(IR)

```

```
PUSHABS:
    T := store[operand(IR)]
    SP := SP - 4
    store[SP] := T

POP:
    T := store[SP]
    SP := SP + 4
    store[operand(IR)] := T

COMPEQ:
    T:= store[SP]
    SP := SP + 4
    T := store[SP] - T
    store[SP] = T=0 ? 1 : 0

JTRUE:
    T := store[SP]
    SP := SP + 4
    PC := T=1 ? operand(IR) : PC

JFALSE:
    T := store[SP]
    SP := SP + 4
    PC := T=0 ? operand(IR) : PC
```

Typical Assembly

```
start:
    PushAbs i
    PushImm 1
    Sub
    Pop i
    PushAbs i
    PushImm 100
    CompEq
    JTrue start
```

Compiler Code Generated

```
[
    Define "start",
    PushAbs "i",
    PushImm 1,
    Sub,
    Pop "i",
    PushAbs "i",
    CompEq,
    Jtrue "Start"
]
```

The 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)

```
data Stat =
    Assign Name Exp |
    Seq Stat Stat |
    ForLoop Name Exp Exp Stat
    deriving (Show)

data Exp =
    BinOp Op Exp Exp |
    Unop Op Exp |
    Ident Name |
    Const Int
    deriving (Show)

data Op =
```

```
    Plus |
    Minus |
    Times |
    Divide
    deriving (Show)

type Name = [Char]

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

type Label = [Char]

transExp :: Exp -> [Instruction]
transExp (BinOp op e1 e2)
    = transExp e1 ++ transExp e2 ++ [case op of
        Plus -> Add
        Minus -> Sub
        Times -> Mul
        Divide -> Div]
transExp (Unop Minus e)
    = transExp e ++ [PushImm (-1), Mul]
transExp (Unop _ _)
    = error "(transExp) Only '-' unary operator supported"
transExp (Ident id) = [PushAbs id]
transExp (Const n) = [PushImm n]

transStat :: Stat -> [Instruction]
transStat (Assign id exp) = transExp exp ++ [Pop id]
transStat (Seq s1 s2) = transStat s1 ++ transStat s2


{-
for x:e1 to e2 do
    body
od

x := <e1>
loop:
    if <e2> then goto break
    <body>
    x := x + 1
    goto loop
break:


<transExp e1>
Pop x
define "loop"
<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 opeerations 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.