

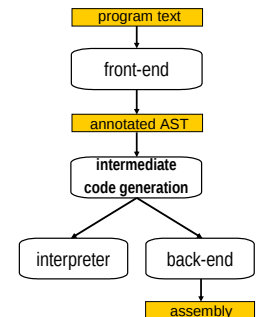
Compiler construction in4303 – lecture 8

Interpretation
Simple code generation

Chapter 4 – 4.2.4

Overview

- intermediate code
- interpretation
- code generation
 - code selection
 - register allocation
 - instruction ordering



Intermediate code

- language independent
 - no structured types, only basic types (char, int, float)
 - no structured control flow, only (un)conditional jumps
- linear format
 - three-address code $z := x \oplus y$
 - Java byte code

Interpretation

- **recursive** interpretation
 - operates directly on the AST [attribute grammar]
 - simple to write
 - thorough error checks
 - very slow: 1000x execution time of compiled code
- **iterative** interpretation
 - operates on intermediate code
 - good error checking
 - slow: 100x

Recursive interpretation

- function per node type
 - implement semantics
 - visit children
- status indicator
 - normal mode
 - return mode (value)
 - jump mode (label)
 - exception mode (name)

Recursive interpretation

```

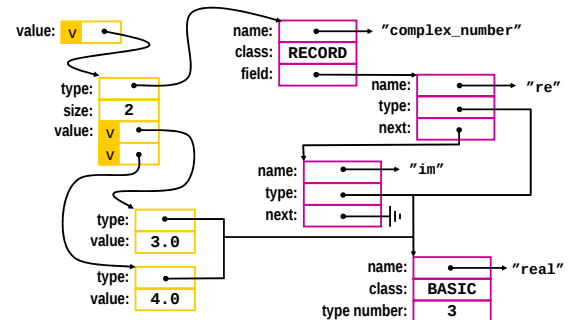
PROCEDURE Elaborate if statement (If node):
    SET Result TO Evaluate condition (If node .condition);
    IF Status .mode != Normal mode: RETURN;
    IF Result .type != Boolean:
        ERROR "Condition in if-statement is not of type Boolean";
        RETURN;
    IF Result .boolean .value = True:
        Elaborate statement (If node .then part);
    ELSE
        // Is there an else-part at all?
        IF If node .else part != No node:
            Elaborate statement (If node .else part);
    
```

Self-identifying data

- must handle user-defined data types
- value = pointer to type descriptor + array of subvalues
- example: complex number

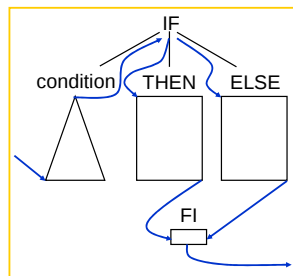
re: 3.0
im: 4.0

Complex numbers



Iterative interpretation

- operates on threaded AST
- active node pointer (similar to PC)
- flat loop over a case statement



```
WHILE Active node .type != End of program type:
  SELECT Active node .type:
  CASE ...
  CASE If type:
    // We arrive here after the condition has been evaluated;
    // the Boolean result is on the working stack.
    SET Value TO Pop working stack ();
    IF Value .boolean .value = True:
      SET Active node TO Active node .true successor;
    ELSE Value .boolean .value = False:
      IF Active node .false successor != No node:
        SET Active node TO Active node .false successor;
      ELSE Active node .false successor = No node:
        SET Active node TO Active node .successor;
  CASE ...
```

Iterative interpretation

- data structures implemented as arrays
 - global data of the source program
 - stack for storing local variables
- shadow memory to store properties
 - status: (un)initialized data
 - access: read-only / read-write data
 - type: data / code pointer / ...

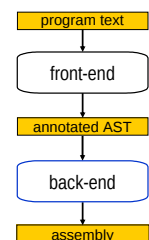
Code generation

tree rewriting

- replace nodes and subtrees of the AST by target code segments
- produce a linear sequence of instructions from the rewritten AST

example

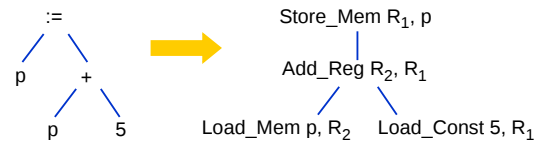
- code: $p := p + 5$
- target: RISC machine



Register machine instructions

Instruction	Action	Tree pattern
Load_Const c, R_n	$R_n := c$	$\begin{array}{c} R_n \\ \\ c \end{array}$
Load_Mem x, R_n	$R_n := x$	$\begin{array}{c} R_n \\ \\ x \end{array}$
Store_Mem R_n, x	$x := R_n$	$\begin{array}{c} := \\ \\ x \\ \\ R_n \end{array}$
Add_Reg R_m, R_n	$R_n := R_n + R_m$	$\begin{array}{c} R_n \\ \\ + \\ / \quad \backslash \\ R_m \quad R_n \end{array}$
Sub_Reg R_m, R_n	$R_n := R_n - R_m$	$\begin{array}{c} R_n \\ \\ - \\ / \quad \backslash \\ R_m \quad R_n \end{array}$
Mul_Reg R_m, R_n	$R_n := R_n * R_m$	$\begin{array}{c} R_n \\ \\ * \\ / \quad \backslash \\ R_m \quad R_n \end{array}$

Tree rewriting for $p := p + 5$



- linearize instructions: depth-first traversal

Load_Mem p, R_2
Load_Const $5, R_1$
Add_Reg R_2, R_1
Store_Mem R_1, p

Code generation

main issues:

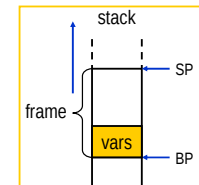
- code selection – which template?
- register allocation – too few!
- instruction ordering

optimal code generation is NP-complete

- consider small parts of the AST
- simplify target machine
- use conventions

Simple code generation

- consider one AST node at a time
- two simplistic target machines
 - pure register machine
 - pure stack machine

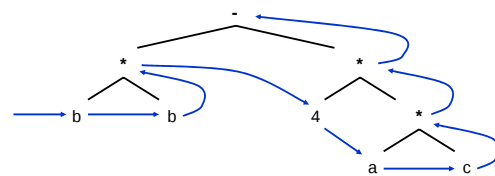


Stack machine instructions

Instruction	Action
Push_Const c	$SP++$; $stack[SP] = c$;
Push_Local i	$SP++$; $stack[SP] = stack[BP+i]$;
Store_Local i	$stack[BP+i] = stack[SP]$; $SP--$;
Add_Top2	$stack[SP-1] = stack[SP-1] + stack[SP]$; $SP--$;
Sub_Top2	$stack[SP-1] = stack[SP-1] - stack[SP]$; $SP--$;
Mul_Top2	$stack[SP-1] = stack[SP-1] * stack[SP]$; $SP--$;

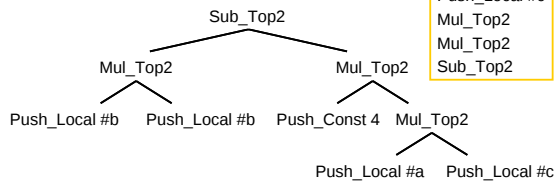
Simple code generation for a stack machine

- example: $b*b - 4*a*c$
- threaded AST



Simple code generation for a stack machine

- example: $b*b - 4*a*c$
- rewritten AST



Push_Local #b
Push_Local #b
Mul_Top2
Push_Const 4
Push_Local #a
Push_Local #c
Mul_Top2
Mul_Top2
Sub_Top2

Depth-first code generation for a stack machine

PROCEDURE Generate code (Node):

SELECT Node .type:

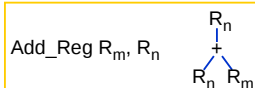
CASE Constant type: Emit ("Push_Const" Node .value);
CASE LocalVar type: Emit ("Push_Local" Node .number);
CASE StoreLocal type: Emit ("Store_Local" Node .number);
CASE Add type:
Generate code (Node .left); Generate code (Node .right);
Emit ("Add_Top2");
CASE Subtract type:
Generate code (Node .left); Generate code (Node .right);
Emit ("Sub_Top2");
CASE Multiply type:
...

Simple code generation for a register machine

- consider one AST node at a time
- similar to stack machine: depth-first

- register allocation

- each AST node leaves its result in a register
- specify target register when processing a subtree AND the set of "free" registers
- free registers: $R_{\text{target}+1} \dots R_{32}$



Depth-first code generation for a register machine

PROCEDURE Generate code (Node, a register number Target):

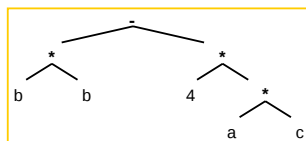
SELECT Node .type:

CASE Constant type:
Emit ("Load_Const " Node .value ",R" Target);
CASE Variable type:
Emit ("Load_Mem " Node .address ",R" Target);
CASE ...
CASE Add type:
Generate code (Node .left, Target);
Generate code (Node .right, Target+1);
Emit ("Add_Reg R" Target+1 ",R" Target);
CASE ...

Exercise (5 min.)

- generate code for the expression

$b*b - 4*a*c$

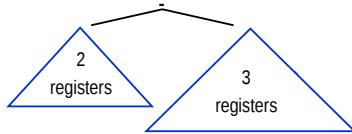


on a register machine with 32 registers
numbered R1 .. R32

Answers

Weighted register allocation

- registers are scarce, depth-first traversal is not optimal



- evaluate heaviest subtree first

```
Load_Mem b, R1
Load_Mem b, R2
Mul_Reg R2, R1
Load_Const 4, R2
Load_Mem a, R3
Load_Mem c, R4
Mul_Reg R4, R3
Mul_Reg R3, R2
Sub_Reg R2, R1
```

Computing weights

FUNCTION Weight of (Node) RETURNING an integer:

SELECT Node .type:

CASE Constant type: RETURN 1;

CASE Variable type: RETURN 1;

CASE ...

CASE Add type:

SET Required left TO Weight of (Node .left);

SET Required right TO Weight of (Node .right);

IF Required left > Required right: RETURN Required left;

IF Required left < Required right: RETURN Required right;

// Required left = Required right

RETURN Required left + 1;

CASE ...

Exercise (4 min.)

- determine the number of registers needed to evaluate the expression

$a + a*2 + a*(b+3)$

Answers

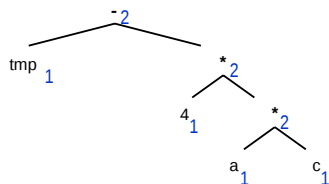
Register spilling

too few registers?

- spill** registers in memory, to be retrieved later
- heuristic: select subtree that uses all registers, and replace it by a temporary

example:

- $b*b - 4*a*c$
- 2 registers



Register spilling

```
Load_Mem b, R1
Load_Mem b, R2
Mul_Reg R2, R1
Store_Mem R1, tmp
Load_Mem a, R1
Load_Mem c, R2
Mul_Reg R2, R1
Load_Const 4, R2
Mul_Reg R1, R2
Load_Mem tmp, R1
Sub_Reg R2, R1
```

