Compiler Construction

Intermediate Representation I

Ran Shaham and Ohad Shacham
School of Computer Science
Tel-Aviv University

PAs

- PA2
  - Ones who perform error recovery should document
  - Cup 10 does not support java generics
- PA3
  - Online submission due January 12, 2008

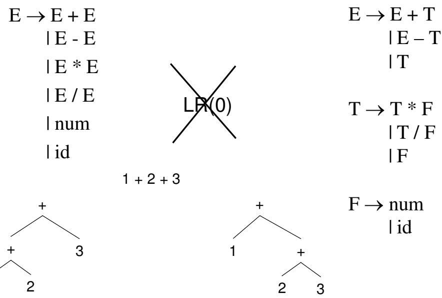
2

TA

- Question 1
  - Look at Left factoring for aid
  - Look on the web
- Question 3 was fixed
  - The grammar cannot be translated to LR(0)
  - Build a nonambigious grammar
  - Show why it is not LR(0)

3

Correction

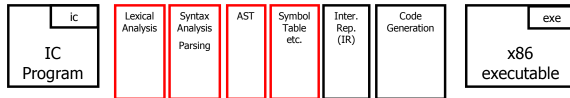


E → E + T
| E – T
| T

T → T * F
| T / F
| F

F → num
| id

IC compiler



We saw:
- Semantic analysis
- Scope checking
- Type checking
- Other semantic checks

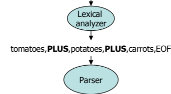
Today:
- Intermediate Representation

tomatoes + potatoes + carrots

Lexical analyzer

tomatoes,**PLUS**,potatoes,**PLUS**,carrots,EOF

Parser

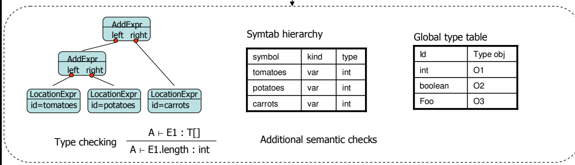


Symtab hierarchy

symbol	kind	type
tomatoes	var	int
potatoes	var	int
carrots	var	int

Global type table

Id	Type obj
int	O1
boolean	O2
Foo	O3



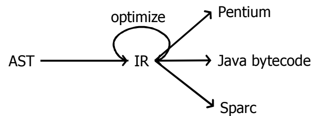
Additional semantic checks

```
Move tomatoes,R1
Move potatoes,R2
Add R2,R1
...
```

LIR

Intermediate representation

- Allows language-independent, machine independent optimizations and transformations
- Easy to translate from AST
- Easy to translate to assembly

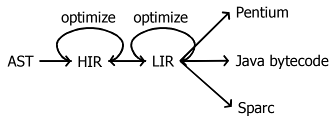


7

Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code

- Solution: use multiple IR stages



8

Optimizations

- LIR
  - Register allocation

- HIR
  - Constant propagation
  - Constant folding
  - Dead code elimination
  - Liveness analysis

9

What's in an AST?

- Administration
  - Declarations
  - For example, class declarations
  - Many nodes do not generate code
- Expressions
  - Data manipulation
- Flow of control
  - If-then, while, switch
  - Target language (usually) more limited
    - Usually only jumps and conditional jumps

10

High-level IR (HIR)

- Close to AST representation
  - High-level language constructs
- Statement and expression nodes
  - Method bodies
  - Only program's computation
- Statement nodes
  - if nodes
  - while nodes
  - statement blocks
  - assignments
  - break, continue
  - method call and return

11

High-level IR (HIR)

- Expression nodes
  - unary and binary expressions
  - Array accesses
  - field accesses
  - variables
  - method calls
  - New constructor expressions
  - length-of expressions
  - Constants

In this project we can make do with HIR=AST

12

Low-level IR (LIR)

- An abstract machine language
  - Not specific to a particular machine
- Low-level language constructs
  - No looping structures, only jumps/conditional jumps
- We will use – two-operand instructions
  - OP a b
    - b = b OP a
- Other alternatives
  - Three-address code: a = b OP c
    - Has at most three addresses (or fewer)
    - Also named quadruples: (a,b,c,OP)

LIR instructions

Instruction	Meaning
Move c,Rn	Rn = c
Move x,Rn	Rn = x
Move Rn,x	x = Rn
Add Rm,Rn	Rn = Rn + Rm
Sub Rm,Rn	Rn = Rn – Rm
Mul Rm,Rn	Rn = Rn * Rm
…	

Immediate (constant)

Memory (variable)

Note 1: rightmost operand = operation destination
Note 2: two register instr - second operand doubles as source and destination

Example

```
x = 42;
while (x > 0) {
  x = x - 1;
}
```

```
Move 42,R1
Move R1,x
_test_label:
Move x,R1
Compare 0,R1
JumpLE _end_label
Move x,R1
Move 1,R2
Sub R2,R1
Move R1,x
Jump _test_label
_end_label:
```

Condition depends on compare register

(warning: code shown is a naïve translation)

Translation (IR lowering)

- How to translate HIR to LIR?
- Assuming HIR has AST form (ignore non-computation nodes)
  - Define how each HIR node is translated
  - Recursively translate HIR (HIR tree traversal)
- TR[e] = LIR translation of HIR construct e
  - A sequence of LIR instructions
  - Temporary variables = new locations
    - Use temporary variables (LIR registers) to store intermediate values during translation

16

Translating Expressions - Example

```
x = x + 42
```

```
Move x, R1
Move 42, R2
Add R2, R1
Move R1, x
```

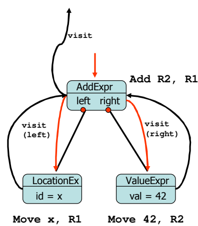
```
TR[x + 42]
```

```
Move x, R1
Move 42, R2
Add R2, R1
```

17

Translating expressions – example

```
TR[x + 42]
```



Move x, R1

Move 42, R2

Add R2, R1

Very inefficient translation – can we do better?

18

Translating expressions

- (HIR) AST Visitor
  - Generate LIR sequence for each visited node
  - Propagating visitor – register information
- When visiting an expression node
  - A single Target register designated for storing result
  - A set of available auxiliary registers
  - TR[node, target, available set]
- Leaf nodes
  - Emit code using target register
  - No auxiliaries required
- What about internal nodes?

19

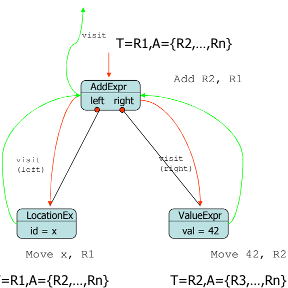
Translating expressions

- Internal nodes
  - Process first child, store result in target register
  - Process second child
    - Target is now occupied by first result
    - Allocate a new register Target2 from available set for result of second child
  - Apply node operation on Target and Target2
  - Store result in Target
  - All initially available register now available again
  - Result of internal node stored in Target (as expected)

20

Translating Expressions – Example (Cont'd)

`TR[x + 42,T,A]`




```
Move x, R1
Move 42, R2
Add R2, R1
```

21

Translating expressions

Binary operations (arithmetic and comparisons)

TR[e1 OP e2]

```
R1 := TR[e1]
R2 := TR[e2]
R3 := R1 OP R2
```

Fresh virtual (LIR) register generated by translation

Shortcut notation to indicate target register NOT LIR instruction

Unary operations

TR[OP e]

```
R1 := TR[e]
R2 := OP R1
```

22

Translating (short-circuit) OR

TR[e1 OR e2]

```
R1 := TR[e1]
Compare 1,R1
JumpTrue _end_label
R2 := T[e2]
Or R2,R1
_end_label:
```

(OR can be replaced by Move operation since R1 is 0)

23

Translating (short-circuit) AND

TR[e1 AND e2]

```
R1 := TR[e1]
Compare 0,R1
JumpTrue _end_label
R2 := T[e2]
And R2,R1
_end_label:
```

(AND can be replaced by Move operation since R1 is 1)

24

Translating array and field access

TR[e1[e2]]

```
R1 := TR[e1]
R2 := TR[e2]
MoveArray R1[R2], R3
```

Give class type of e1 need to compute offset of field f

TR[e1.f]

```
R1 := TR[e1]
MoveField R1.cf, R3
```

Need to identify class type of e1 from semantic analysis phase

Constant representing offset of field f in objects of class type of e1

25

Translating statement block

TR[s1; s2; … ; sN]

```
TR[s1]
TR[s2]
TR[s3]
…
TR[sN]
```

26

Translating if-then-else

TR[if (e)
then s1
else s2]

```
R1 := TR[e]
Compare 0,R1
JumpTrue _false_label
TR[s1]
Jump _end_label
_false_label:
TR[s2]
_end_label:
```

Fresh labels generated during translation

27

Translating if-then

TR[if (e) then s]

```
R1 := TR[e]
Compare 0,R1
JumpTrue _end_label
TR[s]
_end_label:
```

28

Translating while

TR[while (e) s]

```
_test_label:
R1 := TR[e]
Compare 0,R1
JumpTrue _end_label
TR[s]
Jump _test_label
_end_label
```

29

Translating call/return

TR[C.foo(e1,…,en)]

```
R1 := TR[e1]
…
Rn := TR[en]
StaticCall C_foo(x1=R1,…,xn=Rn),R
```


TR[e1.foo(e2)]

```
R1 := TR[e1]
R2 := TR[e2]
VirtualCall R1.cfoo(x=R2),R
```


TR[return e]

```
R1 := TR[e]
Return R1
```

30

Translation implementation

- Define classes for LIR instructions
- Define **`LoweringVisitor`**
  - Apply visitor to translate each method
  - Mechanism to generate new LIR registers
  - Mechanism to generate new labels
  - For each node type translation returns
    - List of LIR instructions TR[e]
    - Target register
  - Splice lists: **`instructions.addAll`**(TR[e])
    (input is a tree, output is a flat list)

31

Example revisited

```
TR[
   x = 42;
   while (x > 0) {
   x=x-1;
   }
]
```

```
TR[x = 42]
TR[
   while (x > 0) {
   x=x-1;
   }
]
```

```
Move 42,R1
Move R1,x
TR[
   while (x > 0) {
   x=x-1;
   }
]
```

```
Move 42,R1
Move R1,x
_test_label:
Move x,R1
Compare 0,R1
JumpLE _end_label
TR[x=x-1]
Jump _test_label
_end_label:
```

```
Move 42,R1
Move R1,x
_test_label:
Move x,R1
Compare 0,R1
JumpLE _end_label
Move x,R1
Move 1,R2
Sub R2,R1
Move R1,x
Jump _test_label
_end_label:
```

spliced list for TR[x=x-1] (Move x,R1 … Move R1,x)

32