CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 18-19: Intermediate Code
4, 11 March 05

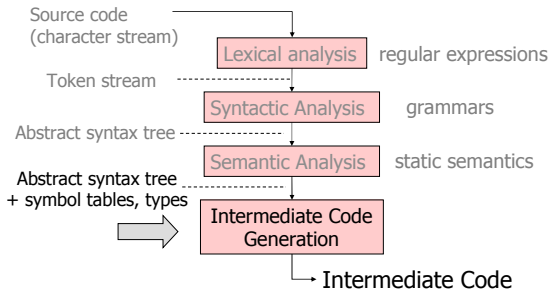
Summary: Semantic Analysis

- Check errors not detected by lexical or syntax analysis
- Scope errors:
  - Variables not defined
  - Multiple declarations
- Type errors:
  - Assignment of values of different types
  - Invocation of functions with different number of parameters or parameters of incorrect type
  - Incorrect use of return statements

Semantic Analysis

- Type checking
  - Use type checking rules
  - Static semantics = formal framework to specify type-checking rules
- There are also control flow errors:
  - Must verify that a break or continue statement is always enclosed by a while (or for) statement
  - Java: must verify that a break X statement is enclosed by a for loop with label X
  - Can easily check control-flow errors by recursively traversing the AST

Where We Are

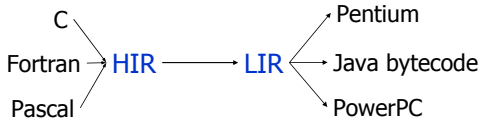


Intermediate Code

- Usually two IRs:

High-level IR
Language-independent
(but closer to language)

Low-level IR
Machine independent
(but closer to machine)



High-level IR

- Tree node structure very similar to the AST
- Contains high-level constructs common to many languages
  - Expression nodes
  - Statement nodes
- Expression nodes for:
  - Integers and program variables
  - Binary operations: e1 OP e2
    - Arithmetic operations
    - Logic operations
    - Comparisons
  - Unary operations: OP e
  - Array accesses: e1[e2]

High-level IR

- Statement nodes:
  - Block statements (statement sequences): (s1, …, sN)
  - Variable assignments: v = e
  - Array assignments: e1[e2] = e3
  - If-then-else statements: if c then s1 else s2
  - If-then statements: if c then s
  - While loops: while (c) s
  - Function call statements: f(e1, …, eN)
  - Return statements: return or return e
- May also contain:
  - For loop statements: for(v = e1 to e2) s
  - Break and continue statements
  - Switch statements: switch(e) { v1: s1, …, vN: sN }

Low-Level IR

- Low-level representation is essentially an instruction set for an abstract machine
- Alternatives for low-level IR:
  - Three-address code or quadruples (Dragon Book):
    a = b OP c
  - Tree representation (Tiger Book)
  - Stack machine (like Java bytecode)

Three-Address Code

- In this class: three-address code
  a = b OP c
- Has at most three addresses (may have fewer)
- Also named quadruples because can be represented as: (a, b, c, OP)
- Example:

```
a = (b+c)*(-e);        t1 = b + c
                       t2 = – e
                       a = t1 * t2
```


Low IR Instructions

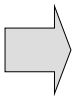
- Assignment instructions:
  - Binary operations: a = b OP c
    - arithmetic: ADD, SUB, MUL, DIV, MOD
    - logic: AND, OR, XOR
    - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - Unary operation a = OP b
    - Arithmetic MINUS or logic NEG
  - Copy instruction: a = b
  - Load /store: a = *b, *a = b
  - Other data movement instructions

Low IR Instructions, cont.

- Flow of control instructions:
  - label L : label instruction
  - jump L : Unconditional jump
  - cjump a L : conditional jump
- Function call
  - call $f(a_1, ..., a_n)$
  - a = call $f(a_1, ..., a_n)$
  - Is an extension to quads
- … IR describes the Instruction Set of an abstract machine

Example

```
m = 0;
if (c  == 0) {
   m = m+ n*n;
} else {
   m = m + n;
}
```



```
m = 0
t1 = c == 0
fjump t1 falseb
t2 = n * n
m = m + t2
jump end
label falseb
m = m+n
label end
```

How To Translate?

- May have nested language constructs
  - Nested if and while statements
- Need an algorithmic way to translate
- Solution:
  - Start from the AST representation
  - Define translation for each node in the AST
  - Recursively translate nodes in the AST

Notation

- Use the following notation:
  T[e] = the low-level IR representation of high-level IR construct e
- T[e] is a sequence of Low-level IR instructions
- If e is an expression (or a statement expression), it represents a value
- Denote by t = T[e] the low-level IR representation of e, whose result value is stored in t
- For variable v: t = T[v] is the copy instruction t = v

Translating Expressions

- Binary operations: t = T[e1 OP e2]
  (arithmetic operations and comparisons)

```
t1 = T[ e1 ]
t2 = T[ e2 ]
t = t1 OP t2
```

- Unary operations: t = T[OP e]

```
t1 = T[ e ]
t = OP t1
```

Translating Boolean Expressions

- t = T[e1 OR e2]

```
t1 = T[ e1 ]
t2 = T[ e2 ]
t = t1 OR t2
```

- … how about short-circuit OR?
- Should compute e2 only if e1 evaluates to false

Translating Short-Circuit OR

- Short-circuit OR: t = T[e1 SC-OR e2]

```
t = T[ e1 ]
tjump t Lend
t = T[ e2 ]
label Lend
```

- … how about short-circuit AND?

Translating Short-Circuit AND

- Short-circuit AND: t = T[e1 SC-AND e2]

```
t = T[ e1 ]
fjump t Lend
t = T[ e2 ]
label Lend
```

Array and Field Accesses

- Array access: t = T[v[e]]

  t1 = T[e]
  t = v[t1]

- Field access: t = T[e1.f]

  t1 = T[e1]
  t = t1.f

Nested Expressions

- In these translations, expressions may be nested;
- Translation recurses on the expression structure

- Example: t = T[(a - b) * (c + d)]

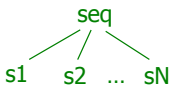
  t1 = a
  t2 = b
  t3 = t1 – t2 } T[(a - b)]
  t4 = b
  t5 = c
  t5 = t4 + t5 } T[(c + d)]
  t = t3 * t5

  T[(a - b) * (c + d)]

Translating Statements

- Statement sequence: T[s1; s2; …; sN]

  T[s1]
  T[s2]
  …
  T[sN]

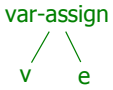


- IR instructions of a statement sequence = concatenation of IR instructions of statements

Assignment Statements

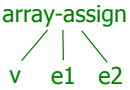
- Variable assignment: T[v = e]

  t = T[e]
  v = t
  [alternatively]
  v = T[e]



- Array assignment: T[v[e1] = e2]

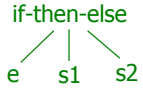
  t1 = T[e1]
  t2 = T[e2]
  v[t1] = t2



Translating If-Then-Else

- T[if (e) then s1 else s2]

  t1 = T[e]
  fjump t1 Lfalse
  T[s1]
  jump Lend
  label Lfalse
  T[s2]
  label Lend



Translating If-Then

- T[if (e) then s]

  t1 = T[e]
  fjump t1 Lend
  T[s]
  label Lend

While Statements

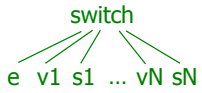
- T[while (e) { s }]

```
label Ltest
t1 = T[ e ]
fjump t1 Lend
T[ s ]
jump Ltest
label Lend
```

Switch Statements

- T[switch (e) { case v1: s1, …, case vN: sN }]

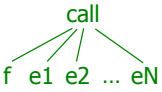
```
t = T[ e ]
c = t != v1
tjump c L2
T[ s1 ]
jump Lend
label L2
c = t != v2
tjump c L3
T[ s2 ]
jump Lend
…
label LN
c = t != vN
tjump c Lend
T[ sN ]
label Lend
```



Call and Return Statements

- T[call f(e1, e2, …, eN)]

```
t1 = T[ e1 ]
t2 = T[ e2 ]
…
tN = T[ eN ]
call f(t1, t2, …, tN)
```



- T[return e]

```
t = T[ e ]
return t
```

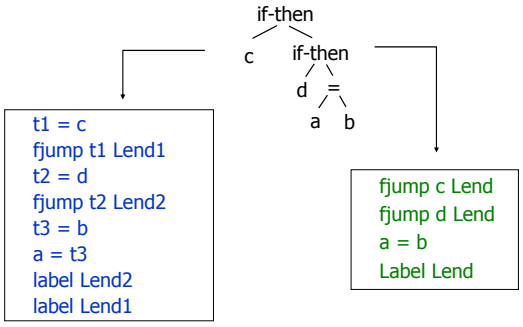
Nested Statements

- Same for statements as expressions: recursive translation
- Example: T[if c then if d then a = b]

```
t1 = c
fjump t1 Lend1
t2 = d
fjump t2 Lend2
t3 = b
a = t3
label Lend2
label Lend1
```

T[a = b] } T[if d …] } T[if c then …]

IR Lowering Efficiency



Efficient Lowering Techniques

- How to generate efficient Low IR:

1. Reduce number of temporaries
   1. Don't use temporaries that duplicate variables
   2. Use "accumulator" temporaries
   3. Reuse temporaries in Low IR
2. Don't generate multiple adjacent label instructions
3. Encode conditional expressions in control flow

No Duplicated Variables

- Basic algorithm:
  - Translation rules recursively traverse expressions until they reach terminals (variables and numbers)
  - Then translate t = T[v] into t = v for variables
  - And translate t = T[n] into t = n for constants
- Better:
  - terminate recursion one level before terminals
  - Need to check at each step if expressions are terminals
  - Recursively generate code for children only if they are non-terminal expressions

No Duplicated Variables

- t = T[e1 OP e2]

  t1 = T[e1], if e1 is not terminal
  t2 = T[e2], if e2 is not terminal
  t = x1 OP x2

  where:

  x1 = t1, if e1 is not terminal
  x1 = e1, if e1 is terminal
  x2 = t2, if e2 is not terminal
  x2 = e2, if e2 is terminal
- Similar translation for statements with conditional expressions: if, while, switch

Example

- t = T[(a+b)*c]
- Operand e1 = a+b, is not terminal
- Operand e2 = c, is terminal
- Translation: t1 = T[e1]
  t = t1 * c
- Recursively generate code for t1 = T[e1]
- For e1 = a+b, both operands are terminals
- Code for t1 = T[e1] is t1 = a+b
- Final result: t1 = a + b
  t = t1 * c

Accumulator Temporaries

- Use the same temporary variables for operands and result
- Translate t = T[e1 OP e2] as:

  t = T[e1]
  t1 = T[e2]
  t = t OP t1
- Example: t = T[(a+b)*c]

  t = a + b
  t = t * c

Reuse Temporaries

- Idea: in the translation of t = T[e1 OP e2] as:

  t = T[e1], t' = T[e2], t = t OP t'

  temporary variables from the translation of e1 can be reused in the translation of e2
- Observation: temporary variables compute intermediate values, so they have limited lifetime
- Algorithm:
  - Use a stack of temporaries
  - This corresponds to the stack of the recursive invocations of the translation functions t = T[e]
  - All the temporaries on the stack are alive

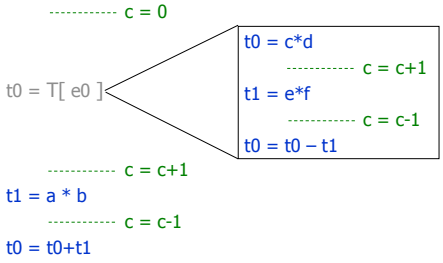
Reuse Temporaries

- Implementation: use counter c to implement the stack
  - Temporaries t(0), …,t(c) are alive
  - Temporaries t(c+1), t(c+2), … can be reused
  - Push means increment c, pop means decrement c
- In the translation of t(c) = T[e1 OP e2]

  t(c) = T[e1]
  ---------- c = c+1
  t(c) = T[e2]
  ---------- c = c-1
  t(c) = t(c) OP t(c+1)

Example

- t0 = T[((c*d) – (e*f)) + (a*b)]



Trade-offs

- Benefits of fewer temporaries:
  - Smaller symbol tables
  - Smaller analysis information propagated during dataflow analysis
- Drawbacks:
  - Same temporaries store multiple values
  - Some analysis results may be less precise
  - Also harder to reconstruct expression trees (more convenient for instruction selection)
- Possible compromise:
  - Different temporaries for intermediate values in each statement
  - Reuse temporaries for different statements

No Adjacent Labels

- Translation of control flow constructs (if, while, switch) and short-circuit conditionals generates label instructions
- Nested if/while/switch statements and nested short-circuit AND/OR expressions may generate adjacent labels
- Simple solution: have a second pass that merges adjacent labels
  - And a third pass to adjust the branch instructions
- More efficient: backpatching
  - Directly generate code without adjacent label instructions
  - Code has placeholders for jump labels, fill in labels later

Encode Booleans in Control-Flow

- Consider T[if (a<b SC-AND c<d) then x = y;]

```
t = a<b
fjump t L1
t = c<d
label L1
fjump t L2
x = y
label L2
```

Condition: t = a<b SC-AND c<d

Control flow: if (t) x = y

- … can we do better?

Encode Booleans in Control-Flow

- Consider T[if (a<b SC-AND c<d) then x = y;]

```
t = a<b                t = a<b
fjump t L1             fjump t L2
t = c<d                t = c<d
label L1               fjump t L2
fjump t L2             x = y
x = y                  label L2
label L2
```

Condition and control flow

- If t = a<b is false, program branches to label L2

How It Works

- For each boolean expression e, and b either true or false:

  T[e, L, b]

  is the code that computes e and branches to L if e evaluates to b, and falls through to the next sequential instruction on !b
- Must redefine define:

  T[s] for if, while statements

Define New Translations

- T[if(e) then s1 else s2]

  T[e, L, false]
  T[s1]
  jump Lend
  label L
  T[s2]
  label Lend

- T[if(e) then s]

  T[e, L, false]
  T[s]
  label L

While Statement

- T[while (e) s]

  label Ltest
  T[e, L, false]
  label L
  T[s]
  jump Ltest
  label L

SC-Boolean Expression Translations

- T[v, L, b] : if b then tjump v, L else fjump v, L
- T[!e, L, b] : T[e, L, !b]
- T[e1 SC-OR e2, L, true]
  T[e1, L, true]
  T[e2, L, true]
- T[e1 SC-AND e2, L, false]
  T[e1, L, false]
  T[e2, L, false]
- T[e1 SC-OR e2, L, false]
  T[e1, Lnext, true]
  T[e2, L, false]
  label Lnext
- T[e1 SC-AND e2, L, true]
  T[e1, Lnext, false]
  T[e2, L, true]
  label Lnext

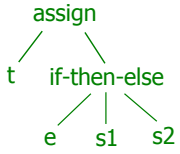
Statement Expressions

- So far: statements that do not return values
- Easy extensions for statement expressions:
  - Block statements
  - If-then-else
  - Assignment statements
- t = T[s] is the sequence of low IR code for statement s, whose result is stored in t

Statement Expressions

- t = T[if (e) then s1 else s2]

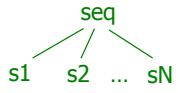
  t1 = T[e]
  cjump t1 Ltrue
  t = T[s2]
  jump Lend
  label Ltrue
  t = T[s1]
  label Lend



Block Statements

- t = T[s1; s2; …; sN]

  T[s1]
  T[s2]
  …
  t = T[sN]



- Result value of a block statement = value of last statement in the sequence

Assignment Statements

- t = T[v = e]

v = T[e]
t = v

- Result value of an assignment statement = value of the assigned expression