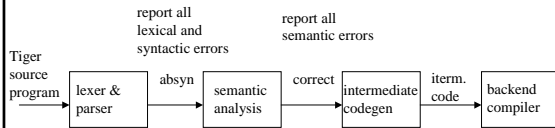


## Intermediate Code Generation

- Translating the abstract syntax into the intermediate representation.



- M source languages to N machines: without IR –  $M \cdot N$  compilers, with  $M+N$
- What should Intermediate Representation (IR) be like ?
  - not too low-level (machine independent) but also not too high-level (so that we can do optimizations)
- How to convert abstract syntax into IR ?

## Intermediate Representations (IR)

- What makes a good IR ? --- easy to convert from the absyn; easy to convert into the machine code; must be clear and simple; must support various machine-independent optimizing transformations;
- Some modern compilers use several IRs ( e.g.,  $k=3$  in SML/NJ ) --- each IR in later phase is a little closer (to the machine code) than the previous phase.
- Absyn  $\implies$  IR1  $\implies$  IR2 ...  $\implies$  IRk  $\implies$  machine code
  - pros : make the compiler cleaner, simpler, and easier to maintain
  - cons : multiple passes of code-traversal --- compilation may be slow
- The Tiger compiler uses one IR only --- the Intermediate Tree (itree) Absyn  $\implies$  itree frags  $\implies$  assembly  $\implies$  machine code
- How to design itree? stay in the middle of absyn and assembly!

## Case Study : itree

- a new language! – should describe simple things “fetch”, “store”, “move”, “jump”... but not at ass.level
- tree structured: convenient for 2-target conditional branches, easy for tree walk...
- datatype: word... wordSize will be specified in machine-dependent modules, e.g. Frame

```

structure Tree : TREE =
struct
...
datatype stm = SEQ of stm * stm | .....
and exp = BINOP of binop * exp * exp | .....
and binop = FPLUS | FMINUS | FDIV | FMUL
  | PLUS | MINUS | MUL | DIV
  | AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR
and relop = EQ | NE | LT | GT | LE | GE .....
end
  
```

## itree Statements and Expressions

- Here is the detail of itree statements **stm** and itree expressions **exp**

```

datatype stm = SEQ of stm * stm
  | LABEL of label
  | JUMP of exp * label list
  | CJUMP of relop * exp * exp * label * label
  | MOVE of exp * exp
  | EXP of exp
and exp = BINOP of binop * exp * exp
  | MEM of exp
  | TEMP of Temp.temp
  | ESEQ of stm * exp
  | NAME of label
  | CONST of int
  | CALL of exp * exp list
  
```

## itree Expressions

- itree expressions stand for the computation of some value, possibly with side-effects:
- CONST(*i*) the integer constant *i*
- NAME(*n*) the symbolic constant *n* (i.e., the assembly lang. label)
- TEMP(*t*) content of temporary *t* ; like registers (unlimited number)
- BINOP(*o*,*e1*,*e2*) apply binary operator *o* to operands *e1* and *e2* , here *e1* must be evaluated before *e2*

## itree Expressions (cont'd)

- MEM(*e*) the contents of *wordSize* bytes of memory starting at address *e*.
  - if used as the left child of a MOVE, it means “store”;
  - otherwise, it means “fetch”.
- CALL(*f*,*l*) a procedure call: the application of function *f* : the expression *f* is evaluated first, then the expression list (for arguments) *l* are evaluated from left to right.
- ESEQ(*s*,*e*) the statement *s* is evaluated for side effects, then *e* is evaluated for a result.

## itree Statements

- itree statements performs side-effects and control flow - no return value!
- SEQ(*s1*,*s2*) statement *s1* followed by *s2*
- EXP(*e*) evaluate expression *e* and discard the result
- LABEL(*n*) define *n* as the current code address (just like a label definition in the assembly language)
- MOVE(TEMP *t*, *e*) evaluate *e* and move it into temporary *t*
- MOVE(MEM(*e*), *e2*) evaluate *e1* to address *adr*, then evaluate *e2*, and store its result into MEM[*adr*]
- JUMP(*e*, labels) jump to the address *e*; the common case is jumping to a known label 1 JUMP(NAME(*l*)); labels – possible jump addresses based on value of expression *e*, e.g. switch
- CJUMP(*o*,*e1*,*e2*,*t*,*f*) conditional jump, first evaluate *e1* and then *e2*, do comparison *o*, if the result is true, jump to label *t*, otherwise jump the label *f*

## itree Fragments

- How to represent Tiger function declarations inside the itree ? – can only represent the body!  
representing it as a itree PROC fragment :  

```
datatype frag = PROC of {name : Tree.label,  function name
                        body : Tree.stm,     function body
                        frame : Frame.frame}  frame layout
                | DATA of string
```

  
each itree PROC fragment will be translated into a function definition in the final “assembly code”
- The itree DATA fragment is used to denote Tiger string literal. It will be placed as string constants in the final “assembly code”.
- Our job is to convert Absyn into a list of itree Fragments; function entry/exit added later/target dependent

## Absyn => itree Frags

- Each absyn function declaration is translated into an itree PROC frag
  - functions are no longer nested -- must figure out the stack frame layout information and the runtime access information for local and non-local variables !
  - must convert function body (Absyn.exp) into itree stm
  - calling conventions for Tiger functions and external C functions (which uses standard convention...)
- Each string literal or real constant is translated into an itree DATA frag, associated with a assembly code label.
- translate itree-Frags into the assembly code of your favourite machine (e.g., MIPS)
- *More later....*

## Kind of expressions

- compute a value (Ex) -> Tree.exp
- return no value (e.g. while...) (Nx) -> Tree.stm
- conditional -> Tree.stm + destination labels
- generic type of expression in the Translate module

```
datatype exp = Ex of Tree.exp
             | Nx of Tree.stm
             | Cx of Temp.label * Temp.label -> Tree.stm
```

## Mapping Absyn Exp into itree

- Each Absyn.exp that computes a value is translated into Tree.exp
- Each Absyn.exp that returns no value is translated into Tree.stm
- Each “conditional” Absyn.exp (which computes a boolean value) is translated into a function  
Tree.label \* Tree.label -> Tree.stm

Tiger Expression:  $a > b \mid c < d$  would be translated into

```
Cx(fn (t,f) => SEQ(CJUMP(GT,a,b,t,z),
                SEQ(LABEL z, CJUMP(LT,c,d,t,f))))
```

- but flag:=  $(a > b \mid c < d)$  – this is Ex... need ability to convert from one kind of expression to another  
MOVE(flag, unEx e)

- Utility functions for conversion among Ex, Nx, and Cx expressions:

```
unEx : gexp -> Tree.exp
unNx : gexp -> Tree.stm
unCx : gexp -> (Tree.label * Tree.label -> Tree.stm)
```

```
fun seq [] = error "..."  
  | seq [a] = a  
  | seq (a::r) = SEQ(a,seq r)  
fun unEx(Ex e) = e  
  | unEx(Nx s) = T.ESEQ(s, T.CONST 0)  
  | unEx(Cx genstm) =  
    let val r = T.newtemp()  
        val t = T.newlabel and f = T.newlabel()  
    in T.ESEQ(seq[T.MOVE(T.TEMP r, T.CONST 1),  
                genstm(t,f),  
                T.LABEL f,  
                T.MOVE(T.TEMP r, T.CONST 0)  
                T.LABEL t],  
              T.TEMP r)  
    end
```

(a>b) | (c<d)

if (a>b) then 1 else (c<d)

typically: if Cx then Ex else Ex

recongize case:

if Cx then t else newlabel\_z

newlabel\_z: if Cx then t else f

seq(stm(t, z), seq(label z, stm(t, f)))

## Simple Variables

- allocLocal allocated InReg(temp) or InFrame(offset) for variable
- access information for a variable is given by level\*offset (level where it's defined, offset in frame)

```
MEM(BINOP(PLUS, TEMP fp, CONST k))  
MEM(+ (CONST kn, MEM(+ (CONST kn-1, ...  
MEM(+ (CONST k1, TEMP fp) ...))
```

Strip levels from lf, we use the static link offsets k1, k2, ... from these levels to construct the tree. When we reach lg, we stop.

```
datatype level = LEVEL of {frame : frame,  
parent : level} * unit ref  
| TOP
```

use "unit ref" to test if two levels are the same one.

(if l\_value store, otherwise fetch)