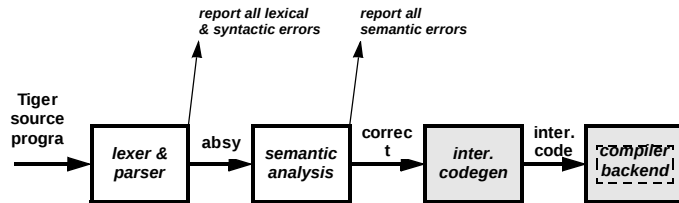


Intermediate Code Generation

- Translating the **abstract syntax** into the **intermediate representation**.



- 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 $\Rightarrow$ IR₁ $\Rightarrow$ IR₂ ... $\Rightarrow$ IR_k $\Rightarrow$ 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 $\Rightarrow$ itree frags $\Rightarrow$ assembly $\Rightarrow$ machine code

How to design itree? stay in the middle of absyn and assembly!

Case Study : itree

- Here is one example, defined using ML datatype definition:

```

structure Tree : TREE =
struct

  type label = string
  type size = int
  type temp = int

  datatype stm = SEQ of stm * stm | .....
    and exp = BINOP of binop * exp * exp | .....

    and test = TEST of relop * 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
      | ULT | ULE | UGT | UGE
      | FEQ | FNE | FLT | FLE | FGT | FGE

    and cvtop = CVTSU | CVTSS | CVTSF | CVTUU
      | CVTUS | CVTFS | CVTFF

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
  | CJUMP of test * label * label
  | MOVE of exp * exp
  | EXP of exp

and exp = BINOP of binop * exp * exp
  | CVTOP of cvtop * exp * size * size
  | MEM of exp * size
  | TEMP of temp
  | ESEQ of stm * exp
  | NAME of label
  | CONST of int
  | CONSTF of real
  | 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

CONSTF(x) the real constant x

NAME(n) the symbolic constant n (i.e., the assembly lang. label)

TEMP(t) content of temporary t ; like registers (unlimited number)

BINOP(o, e_1, e_2) apply binary operator o to operands e_1 and e_2 , here e_1 must be evaluated before e_2

itree Expressions (cont'd)

- more *itree* expressions:

CVTOP(o, e, j, k) converting j -byte operand e to a k -byte value using operator o .

MEM(e, k) the contents of k bytes of memory starting at address e . if used as the left child of a **MOVE**, it means “store”; otherwise, it means “fetch”. (k is often a word)

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(s_1, s_2) statement s_1 followed by s_2

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_1, k), e_2) evaluate e_1 to address adr , then evaluate e_2 , and store its result into $MEM[adr]$

JUMP(e) jump to the address e ; the common case is jumping to a known label l **JUMP**(**NAME**(l))

CJUMP(**TEST**(o, e_1, e_2), t, f) conditional jump, first evaluate e_1 and then e_2 , 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* ?

representing it as a *itree PROC fragment* :

```
datatype frag
= PROC of {name : Tree.label,      function name
           body : Tree.stm,        function body itree
           frame : Frame.frame}    static 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*

Example: Absyn => itree Frags

```

function tigermain () : resType = (* added for uniformity !*)
let type intArray = array of int
var a := intArray [9] of 0
function readarray () = ...
function writearray () = ...
function exchange(x : int, y : int) =
  let var z := a[x] in a[x] := a[y]; a[y] := z end
function quicksort(m : int, n : int) =
  let function partition(y : int, z : int) : int =
    let var i := y      var j := z + 1
    in (while (i < j) do
      (i := i+1; while a[i] < a[y] do i := i+1;
       j := j-1; while a[j] > a[y] do j := j-1;
       if i < j then exchange(i,j));
      exchange(y,j); j)
    end
  in if n > m then (let var i := partition(m,n)
    in quicksort(m, i-1);
    quicksort(i+1, n)
    end)
  end
in readarray(); quicksort(0,8); writearray()
end

```

Example: Absyn => itree frags

- The **quicksort** program (in absyn) is translated to a list of **itree frags** :

```

PROC(label Tigermain, itree code for Tigermain's body,
      Tigermain's frame layout info)

PROC(label readarray, itree code for readarray's body,
      readarray's frame layout info)

PROC(label writearray, itree code for writearray's body,
      writearray's frame layout info)

PROC(label exchange, itree code for exchange's body,
      exchange's frame layout info)

PROC(label partition, itree code for partition's body,
      partition's frame layout info)

PROC(label quicksort, itree code for quicksort's body,
      quicksort's frame layout info)

DATA("... assembly code for string literal #1 ...")
DATA("... assembly code for string literal #2 ...")
.....

```

Summary: Absyn => itree Frags

- Each **absyn function** declaration is translated into an **itree PROC** frag
 - TODO: 1. functions are no longer nested -- must figure out the stack frame layout information and the runtime access information for local and non-local variables !
 - 2. must convert function body (**Absyn.exp**) into **itree stm**
 - 3. 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** ---- To reference the constant, just use this **label**.
- Future work:** translate **itree-Frags** into the assembly code of your favourite machine (PowerPC, or SPARC)

Review: Tiger Abstract Syntax

```

exp = VarExp of var
      NilExp
      IntExp of int
      StringExp of string * pos
      AppExp of {func: Symbol.symbol, args: exp list, pos: pos}
      OpExp of {left: exp, oper: oper, right: exp, pos: pos}
      RecordExp of {typ: Symbol.symbol, pos: pos,
                    fields: (Symbol.symbol * exp * pos) list}
      SeqExp of (exp * pos) list
      AssignExp of {var: var, exp: exp, pos: pos}
      IfExp of {test: exp, then': exp, else': exp option, pos: pos}
      WhileExp of {test: exp, body: exp, pos: pos}
      ForExp of {var: Symbol.symbol, lo: exp, hi: exp,
                body: exp, pos: pos}
      BreakExp of pos
      LetExp of {decs: dec list, body: exp, pos: pos}
      ArrayExp of {typ: Symbol.symbol, size: exp,
                  init: exp, pos: pos}

dec = VarDec of {var: Symbol.symbol, init: exp, pos : pos,
                typ: (Symbol.symbol * pos) option}
      FunctionDec of fundec list
      TypeDec of {name: Symbol.symbol, ty: ty, pos: pos} list

```

Mapping Absyn Exp into itree

- We define the following new **generic** expression type **gexp**

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

this introduce three new constructors:

```
Ex : Tree.exp -> gexp
Nx : Tree.stm -> gexp
Cx : (Tree.label * Tree.label -> Tree.stm) -> gexp
```

- 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 | c<d` would be translated into

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

Mapping Absyn Exp into itree

- 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)
```

Examples:

```
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
```

Simple Variables

- Define the **frame** and **level** type for each function definition:

```
type frame = {formals: int, offlst : int list,
  locals : int ref, maxargs : int ref}
```

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

```
type access = level * offset
```

- The **access information** for a variable **v** is a pair (l, k) where l is the level in which **v** is defined and k is the frame offset.
- The **frame offset** can be calculated by the `allocLocal` function in the Frame structure (which is architecture-dependant). The **access** information will be put in the **env** in the typechecking phase.

Simple Variables (cont'd)

- To access a **local variable v** at offset k , assuming the frame pointer is fp , just do `MEM(BINOP(PLUS, TEMP fp, CONST k),w)`

- To access a **non-local variable v** inside function **f** at level l_f , assuming **v**'s access is (l_g, k) ; we do the following:

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

Strip levels from l_f , we use the static link offsets $k_1, k_2, \dots$ from these levels to construct the tree. When we reach l_g , we stop.

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

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

Array and Record Variables

- In **Pascal**, an array variable stands for the contents of the array --- the assignment will do the copying :

```
var a, b : array [1..12] of integer;
begin
  a := b
end;
```

- In **Tiger** and **ML**, an array or record variable just stands for the pointer to the object, not the actual object. The assignment just assigns the pointer.

```
let type intArray = array of int
var a := intArray[12] of 0
var b := intArray[12] of 7
in a := b
end
```

- In **C**, assignment on array variables are illegal !

```
int a[12], b[12], *c;      a = b; is illegal!      c = a; is legal!
```

Array Subscription

- If a is an array variable represented as $\text{MEM}(e)$, then **array subscription** $a[i]$ would be (ws is the word size)

$\text{MEM}(\text{BINOP}(\text{PLUS}, \text{MEM}(e), \text{BINOP}(\text{MUL}, i, \text{CONST } ws)))$

- To ensure **safety**, we must do the **array-bounds-checking**: if the array bounds are $L..H$, and the subscript is i ; then report runtime **errors** when either $i < L$ or $i > H$ happens.
- Array subscription can be either **l-values** or **r-values** --- use it properly.
- **Record field selection** can be translated in the same way. We calculate the offset of each field at compile time.

```
type point intlist = {hd : int, tl : intlist}
the offset for "hd" and "tl" is 0 and 4
```

Record and Array Creation

- **Tiger record creation**: $\text{var } z = \text{foo } \{f_1 = e_1, \dots, f_n = e_n\}$

we can implement this by calling the C malloc function, and then **move** each e_i to the corresponding field of foo . (see Appel pp164)

In real compilers, calling malloc is expensive; we often inline the malloc code.

- **Tiger array creation**: $\text{var } z = \text{foo } n \text{ of } \text{initv}$

by calling a C `initArray(size, initv)` function, which allocates an array of size `size` with initial value `initv`.

to support array-bounds-checking, we can put the array **length** in the **0-th field**. $z[i]$ is accessed at offset $(i+1)*\text{word_sz}$

- **Requirement**: a way to call external C functions inside Tiger.

Integer and String

- **Integer**: $\text{absyn IntExp}(i) \Rightarrow \text{itree CONST}(i)$
- **Arithmetic**: $\text{absyn OpExp}(i) \Rightarrow \text{itree BINOP}(i)$
- **Strings**: every string literal in Tiger or C is the constant address of a segment of memory initialized to the proper characters.

During translation from **Absyn** to **itree**, we associate a label l for each string literal s :
to refer to s , just use `NAME l`

Later, we'll generate assembly instructions that define and initialize this label l and string literal s .

String representations:

1. a word containing the length followed by characters (in Tiger)
2. a pointer to a sequence of characters followed by `\000` (in C)
3. a fixed length array of characters (in Pascal)

Conditionals

- Each **comparison** expression $a < b$ will be translated to a **Cx** generic expression $\text{fn } (t, f) \Rightarrow (\text{TEST}(\text{LT}, a, b), t, f)$
- Given a conditional expression (in absyn) **if** e_1 **then** e_2 **else** e_3
 - translate e_1, e_2, e_3 into itree generic expressions e_1, e_2, e_3
 - apply **unCx** to e_1 , and **unEx** to e_2 and e_3
 - make three labels, **then** case: t and **else** case: f and **join** : j
 - allocate a temporary r , after label t , move e_2 to r , then jump to j ; after label f , move e_3 to r , then jump to j
 - apply **unCx-ed** version of e_1 to label t and f
- Need to recognize certain **special case**: $(x < 5) \ \& \ (a > b)$ it is converted to "**if** $x < 5$ **then** $a > b$ **else** 0" in absyn ----- too many labels if using the above algorithm --- **inefficient**. (read Appel page 162)

Loops

- Translating **while** loops:

```
test:
  if not (condition) goto done
  ... the loop body ...
  goto test
done:
```

each round executes one conditional branch plus one jump

```
      goto test
top:
  ... the loop body ...
test:
  if (condition) goto top
done:
```

each round executes one conditional branch only

- Translating **break** statements: just **JUMP** to done
need to pass down the label *done* when translating the loop body!
- Translating **For** loops: (exercise, or see Appel pp 166)
for $i := lo$ **to** hi **do** body

Function Calls

- Inside a function g , the function call $f(e_1, e_2, \dots, e_n)$ is translated into **CALL**(NAME l_f , [$s1, e_1, e_2, \dots, e_n$])
 $s1$ is the static link --- it is just a pointer to f 's **parent level**, but how can we find it when we are inside g ?
stripping the **level** of g one by one, generate the code that follow g 's chains of static links until we reach f 's **parent level**.
- When calling external C functions, what kind of static link do we pass ?
- In the future, we need to decide what is the **calling convention** ----- where the callee is expecting the formal parameters and the static link?

Declarations

- Variable** declaration: need to figure out the **offset** in the frame, then **move** the expression on the r.h.s. to the proper **slot** in the **frame**.
- Type** declaration: no need to generate any **itree** code !
- Function** declaration: build the **PROC itree** fragment
Later we translate **PROC**(name : label, body : stm, frame)
to assembly:

```

name:      _global name
          .....
          assembly code for body
          .....
          prologue
          epilogue

```

The **prologue** and **epilogue** captures the **calling sequence**, and can be figured out from the frame layout information in frame. **Prologue** and **epilogue** are often machine-dependant.

Function Declarations

- Generating *prologue* :
 1. *psuedo-instructions* to announce the beginning of a function
 2. a *label* definiton fo the function name
 3. an instruction to adjust the stack pointer (allocating a new frame)
 4. **store** instructions to save callee-save registers and return address
 5. **store** instructions to save arguments and static links
- Generating *epilogue* :
 1. an instruction to move the return result to a special register
 2. **load** instructions to restore callee-save registers
 3. an instruction to reset the stack pointer (pop the frame)
 4. a **return** instruction (jump to the return address)
 5. *psuedo-instructions* to announce the end of a function