Compiler Principle

Prof. Dongming LU

Apr. 8th, 2024

Content

1. INTRODUCTION

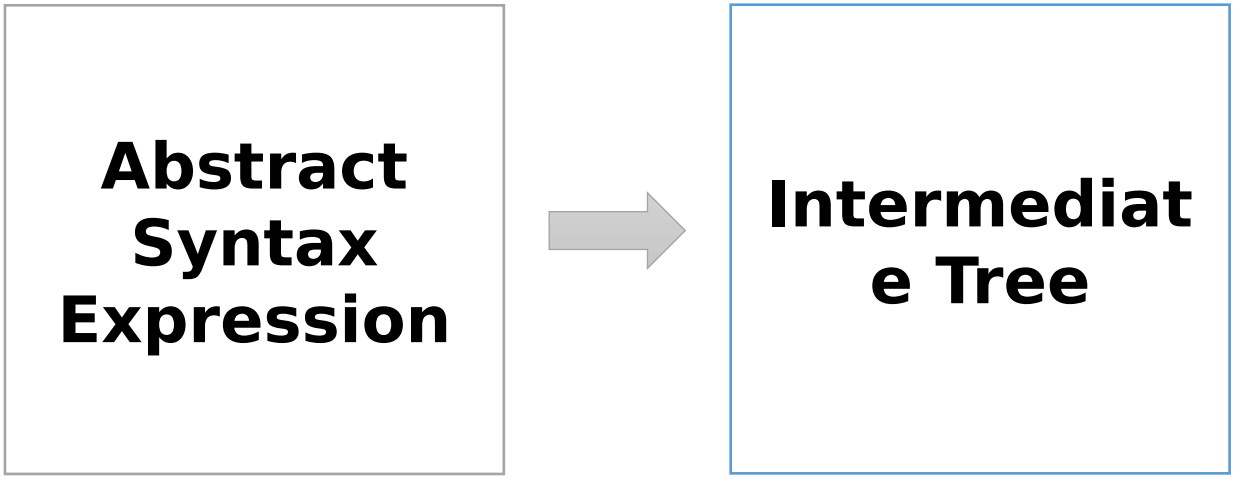
2. LEXICAL ANALYSIS
3. PARSING
4. ABSTRACT SYNTAX
5. SEMANTIC ANALYSIS

6. ACTIVATION RECORD
7. **TRANSLATING INTO INTERMEDIATE CODE**

8. OTHERS

7 Translation to Intermediate Code

7.2 Translation into Trees


Abstract Syntax Expression
Intermediate Tree

Kinds of Expressions

- The expression **A_exp** with return values
  - ✓The intermediate tree **T_exp**

- The expression **A_exp** (such as **procedure call**, or **while**) that returns no value
  - ✓Represented by the intermediate tree: **T_stm**

- The expression **A_exp**(such as **a > b**) with Boolean value
  - ✓Represented by the intermediate tree : **A conditional jump ——a combination of T_stm and a pair of destinations represented by Temp_labels**

Kinds of Expressions

```
/* in translate.h */
   typedef struct Tr_exp_ *Tr_exp;

/* in translate.c */
  struct Cx { patchList trues;  patchList falses; T_stm stm;};

  struct Tr_exp_
       { enum { Tr_exp,Tr_nx,Tr_cx } kind;
        union { T_exp ex; T_stm nx; struct Cx cx; } u; };

static Tr_exp Tr_Ex ( T_exp  ex);
static Tr_exp Tr_Nx ( T_stm  nx);
static Tr_exp Tr_Cx ( patchList trues, patchList falses,T_stm stm);
```

An example

a>b | c<d

```
Temp_label z = Temp_newlabel ( );
T_stm  s1 = T_Seq(T_Cjump(T_gt,a,b,
NULL_t,z),
            T_Seq (T_Label (z),
              T_Cjump (T_lt,c,d, NULL_t, NULL_f  )));
```

- The true-destination and false-destination are unknown!!!
- To represent “a list of places where a label must be filled in” by using patchList

```
typedef struct patchList_  * patchList;

struct patchList_   {  Temp_label  *head;  patchList  tail; };
static patchList patchList(Temp_label  *head, patchList tail);
```

An example

- **To complete the translation of a>b | c<d**

```
Temp_label z = Temp_newlabel ( );
T_stm  s1 = T_Seq(T_Cjump(T_gt,a,b,  NULL_t,z),
            T_Seq (T_Label (z),
              T_Cjump (T_lt,c,d, NULL_t, NULL_f  )));
```

```
patchList  trues =PatchList(&s1->u.SEQ.left-> u.CJUMP.true,
           PatchList(&s1->u.SEQ.right-> u.SEQ.right-> u.CJUMP.true,
           NULL));

patchList  falses  = PatchList(&s1->u.SEQ.right-> u.SEQ.right ->
           U.CJUMP.false, NULL);

Tr_exp  e1 = Tr_Cx (trues, falses, s1);
```

Conversion between equivalent expressions

- Three conversion functions: stripping off the corresponding constructor(Ex, Nx, or Cx)

```
static T_exp unEx(Tr_exp  e);
static T_stm unNx(Tr_exp e);
static struct Cx unCx(Tr_exp e);
```

- An example:

flag := (a>b | c<d)

e = Tr_Cx (trues, falses, stm)

MOVE ($TEMP_{flag}$, unEx(e))

Conversion between equivalent expressions

```
static T_exp unEx(Tr_exp e)  {
switch (e->kind)  {
   case Tr_ex:
       return e->u.ex;
   case Tr_cx:  {
     Temp_temp  r= Temp_newtemp(  );
     Temp_label t = Temp_newlabel( ), f=
Temp_newlabel( );
     doPatch(e->u.cx.trues, t);
     doPatch(e->u.cx.falses, f);
     return T_Eseq(T_move(T_Temp(r),T_Const(1)),
                T_Eseq(e->u.cx.stm, T_Eseq(T_Label(f),
                  T_Eseq(T_Move(T_Temp( r),T_Const(0)),
                    T_Eseq(T_Label(t), T_Temp(r ))))));
     }
      case Tr_nx:
          return T_Eseq(e->u.nx, T_Const(0));
     }
     assert(0);
  }
```

Conversion between equivalent expressions

```
case Tr_cx:  {
    Temp_temp  r= Temp_newtemp(  );
    Temp_label t = Temp_newlabel( ), f= Temp_newlabel( );
    doPatch(e->u.cx.trues, t);
    doPatch(e->u.cx.falses, f);
    return T_Eseq(T_move(T_Temp(r),T_Const(1)),
                  T_Eseq(e->u.cx.stm, T_Eseq(T_Label(f),
                     T_Eseq(T_Move(T_Temp( r),T_Const(0)),
                        T_Eseq(T_Label(t), T_Temp(r ))))));
}
```

- An example:

  flag := (a>b | c<d)

```
r=1
  GT,a,b,t,z
z:
  LT,c,d,t,f
f:
  r=0
t:
```

Conversion between equivalent expressions

```
void doPatch ( patchList tList, Temp_label label)  {
   for ( ; tList; tList = tList->tail)
       *(tList->head) = label;
}

patchList joinPatch (patchList first, patchList  second)
{
   if  (!first) return second;
  for (; first->tail; first = first->tail);
  first->tail = second;
  return first;
}
```

The interface between Semant and Translate

- The semant module should not contain any direct reference to the Tree or Frame module.
- Any manipulation of IR trees should be done by Translate.

```
Tr_Exp Tr_simpleVar (Tr_Access,
Tr_Level);
```

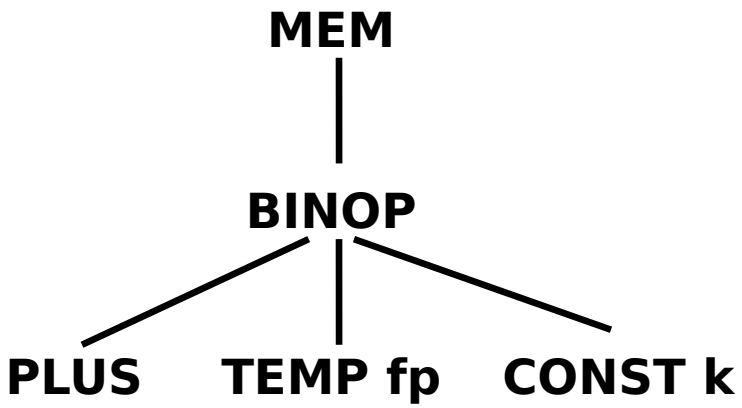
- Sement pass the access of x and the level of the function in which x is used.
- Get back a Tr_exp

```
/* frame.h*/

Temp_temp F_FP(void);
Extern const int F_wordSize;
T_exp F_Exp(F_access acc, T_exp framePtr)
```

Simple variables

- **A simple variable *v* declared in the current procedure's stack frame**
- **Translated as follows:**

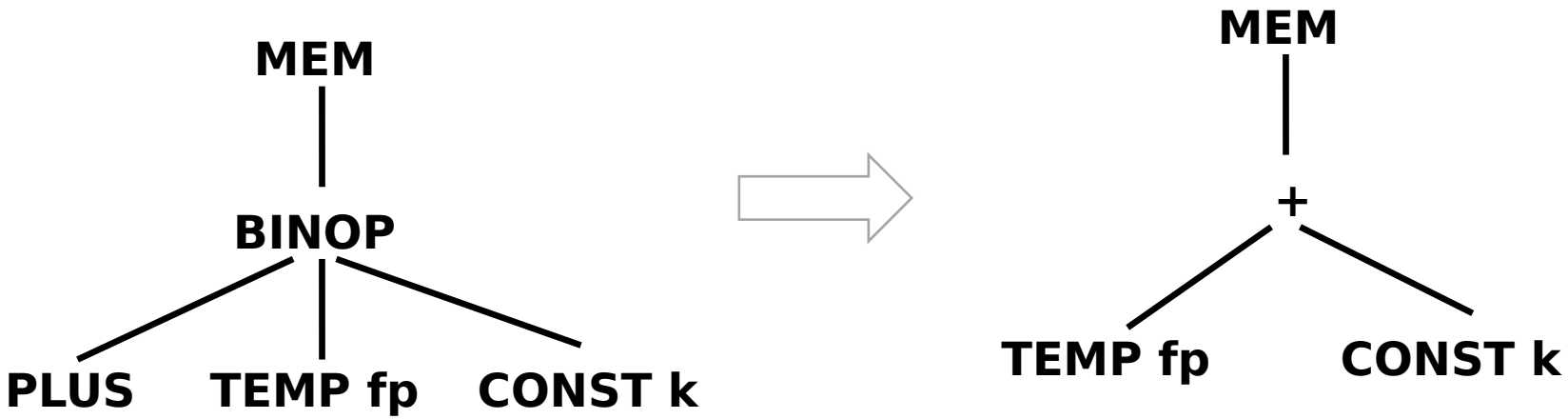


MEM(BINOP(PLUS,TEMP fp, CONST k))

- ***k* is the offset of *v* within the frame and**
- **TEMP fp is the frame-pointer register.**
- **All variables are the same size —— the natural word size of the machine.**

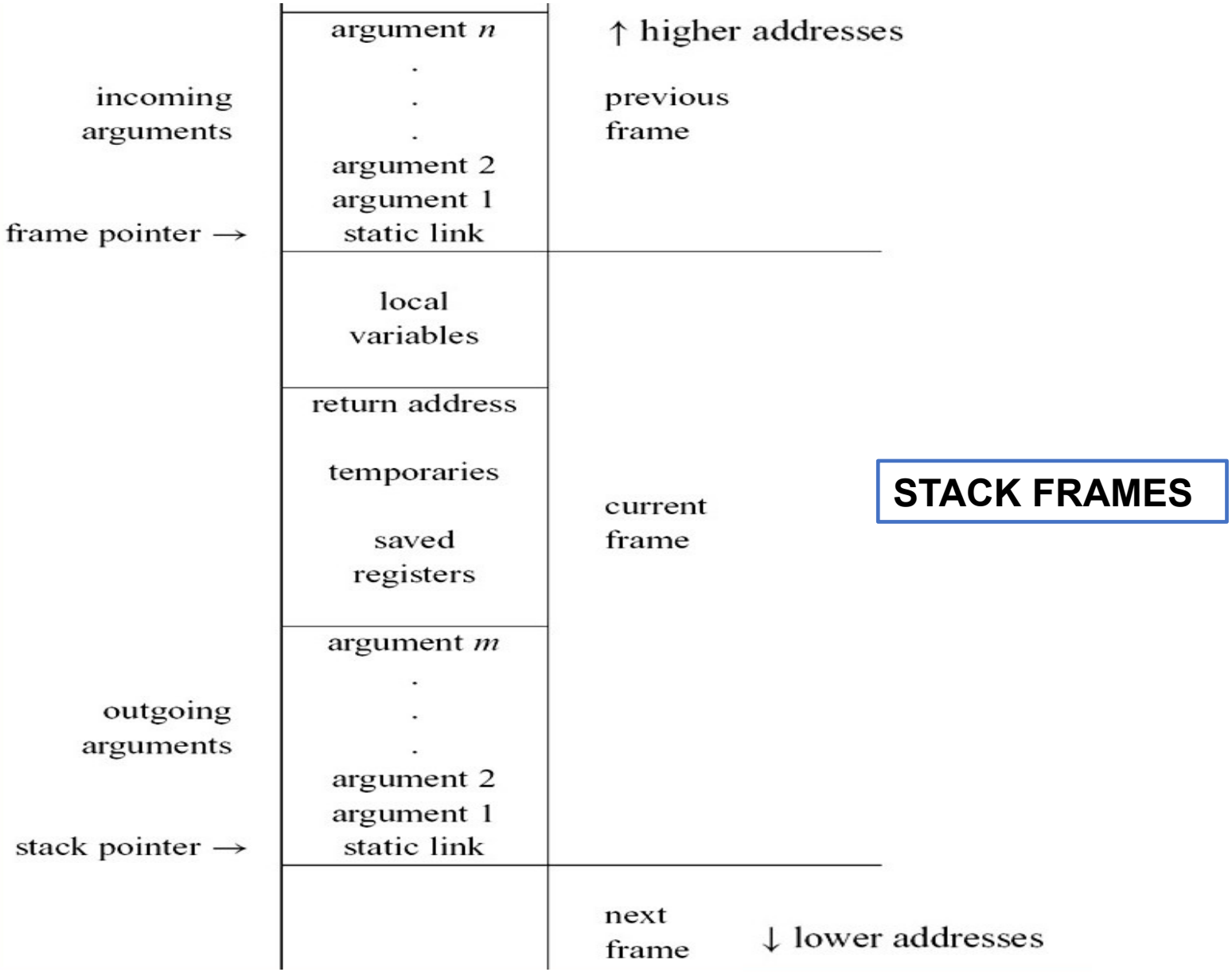
Simple variables

- **Abbreviate BINOP(PLUS, *e*1, *e*2) as + (*e*1, *e*2)**



MEM(BINOP(PLUS,TEMP fp, CONST k)) **+(TEMP fp,CONST k)**

Simple variable



Simple variables

- **The function F_Exp is used by Translate to turn a F_access into the Tree expression**
- **The T_exp argument is the address of the stack frame that the F_access lives in**

```
T_exp F_exp(F_access acc, T_exp framePtr);
```

- **An access *a* such as InFrame(*k*):**

F_Exp(a,T_Temp(F_FP( )))

Returns

MEM(BINOP(PLUS,TEMP(FP),CONST(*k*)))

Following Static Links

- A variable x is declared at an outer level of static scope, static links will be used.

```
MEM(+(CONST Kn, MEM(+(CONST Kn-1,…
                        MEM(+(CONST K1, TEMP
FP))…))))
```

- To fetch static links from all frames between the level of use and the level of definition

Array Variables

- **In Pascal**

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

copies the contents of array *a* into array *b*.

Array Variables

- In C

```
{  int a[12], b[12];
  a=b;
 }
```

This is illegal

```
{  int a[12],
*b;
   b=a;
 }
```

This is quite legal

b now points to the beginning of the array a.

Array Variables

In Tiger (as in Java and ML), array variables behave like pointers.

- **Has no named array constants as in C,**
- **New array values are created (and initialized) by the construct t_a[n] of i;**
  - ✓**t_a is the name of an array type,**
  - ✓***n* is the number of elements**
  - ✓**i is the initial value of each element.**

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

- ***a* ends up pointing to the same 12 sevens as the variable *b*;**
- **the original 12 zeros allocated for *a* are discarded.**

Structured *L*-Values

- **R-value: appear on the *right* of an assignment,**
  - ✓**a+3 or f(x).**
  - ✓***r*-value does not denote a assignable *location***
- **L-value: the result of an expression that can occur on the *left* of an assignment statement, such as**
  - ✓**x, p, y or a[i+2]**
  - ✓**denotes a *location* that can be assigned to**
  - ✓**can occur on the right of an assignment statement**

Structured *L*-Values

- **An integer or pointer value is a "scalar": it has only one component.**
  - ✓ **All the variables and *l*-values in Tiger are scalar.**
  - ✓ **a Tiger array or record variable is really a pointer (a kind of scalar).**

- **In C or Pascal there are structured *l*-values - structs in C, arrays and records in Pascal - that are not scalar.**

```
T_exp T_Mem(T_exp, int size);
 Mem(+(TEMP fp,CONST k ), S)
```

- ***S*** indicates the size of the object to be fetched or stored

Subscripting and Field Selection

- **To subscript an array in Pascal or C(to compute a[i]) by computing the address of *a*[*i*]:**

$$(i - l) \times s + a$$

  - ✓**l: lower bound; s: size of the element; a: base address**

- **If a is global, the constant *a* − *s* × *l can be computed at compile-time***
- **To calculate address of field of a record, *a. f***

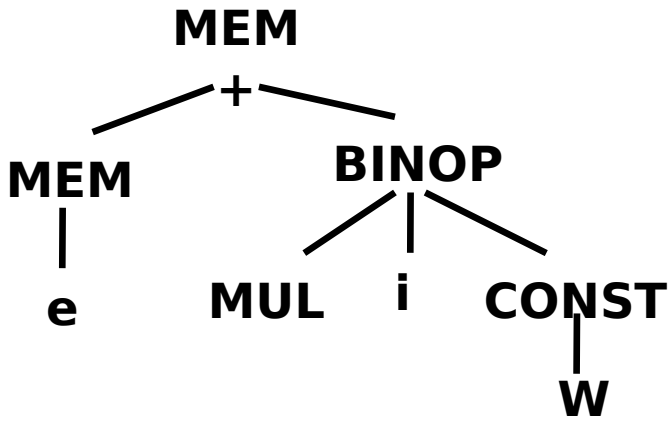
  Offset(f)+a

Subscripting and Field Selection

- **An array variable *a* is an *l*-value; so is an array subscript expression *a*[*i*].**
- **To calculate the *l*-value *a*[*i*] from *a*, we do arithmetic on the address of *a*.**
  - ✓ **A "+" node would add the index times the element size to the *l*-value for the base of the array.**
  - ✓ **The *l*-value (representing the entire array) might be used in a context where an *r*-value is required (e.g., passed as a by-value parameter, or assigned to another array variable).**
  - ✓ **Then the *l*-value is *coerced* into an *r*-value by applying the MEM operator to it.**

Subscripting and Field Selection

- IR tree of *a*[*i*] is



MEM(+(MEM(e),BINOP(MUL, i, CONST W)))

- MEM means
  - store (when used as the left child of a MOVE)
  - fetch (when used elsewhere).

Arithmetic

- **Each arithmetic operator corresponds to a Tree operator.**

- The Tree language has no unary arithmetic operators.
  - Unary negation of integers: implemented as subtraction from zero;
  - Unary complement: implemented as XOR with all ones.
- Unary floating-point negation cannot be implemented as subtraction from zero.
  - Many floating-point representations allow a negative zero.
  - The negation of negative zero is positive zero, and vice versa.

- The Tree language does not support unary negation very well.

Conditionals

- **The result of a comparison operator --- a Cx expression**
  - ✓ **a statement *s* that will jump to any true-destination and false-destination.**

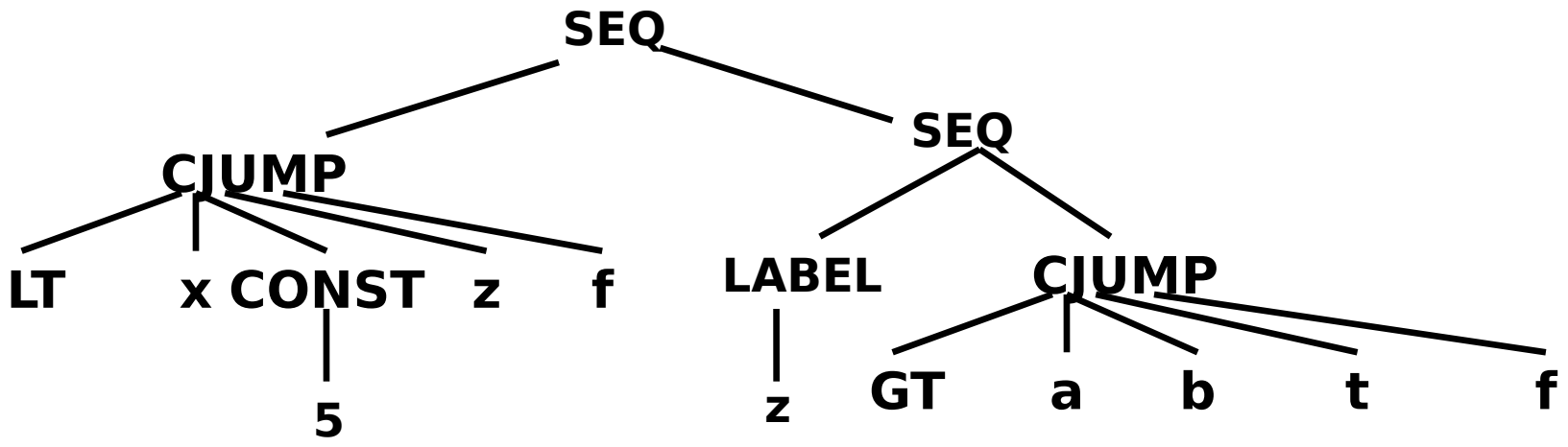
if e_1 then e_2 else e_3

- **e_1: Cx expression; e_2 and e_3: Ex expressions**
  - **use the unCx method of e_1**
  - **unEx of e_2 and e_3**
  - **Make two labels *t* and *f***
  - **Allocate a temporary *r***
  - **after label *t*, move e_2 to *r***
  - **after label *f*, move e_3 to *r***
  - **Both branches finish by jumping to a newly created "join" label.**

Conditionals

if *x* < 5 then *a* > *b* else 0

- ***x*** **< 5 translates into Cx(*s*1)**
- ***a*** **> *b* will be translated as Cx(*s*2)**



SEQ(S1(z,f),SEQ(LABEL Z,s2(t,f)))

While Loops

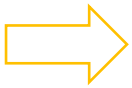
```
test:
     if not(condition) goto done
     body
     goto test
done:
```

- If a break statement occurs within the *body* (and not nested within any interior while statements), the translation is simply a JUMP to *done*.
- Translation of break statements needs to have a new formal parameter ***break*** that is the *done* label of the nearest enclosing loop.

For Loops

- **A straightforward approach:**
  - ✓**rewrite the *abstract syntax* into the abstract syntax of the while statement shown.**

```
for  i:= lo to hi
do body
```



```
let var i := lo
      var limit := hi
in while I<= limit
    do (body;  i:=
i+1)
end
```

Exception:

If *limit=maxint* Then *i* + 1 will overflow;

The end of Chapter 7(2)