Intermediate Code Generation

Semantic Analysis of Functions and Calls

$FUNC\_DECL \rightarrow FUNC\_HEAD \; \{ \; VAR\_DECL \; BODY \; \}$

$FUNC\_HEAD \rightarrow RES\_ID \; (\; DECL\_PLIST \;)$

$RES\_ID \rightarrow RESULT \;\; id$

$RESULT \rightarrow int \mid float \mid void$

$DECL\_PLIST \rightarrow DECL\_PL \mid \epsilon$

$DECL\_PL \rightarrow DECL\_PL \; , \; DECL\_PARAM \mid DECL\_PARAM$

$DECL\_PARAM \rightarrow T \; id$

$VAR\_DECL \rightarrow DLIST \mid \epsilon$

$DLIST \rightarrow D \mid DLIST \; ; \; D$

$D \rightarrow T \; L$

$T \rightarrow int \mid float$

$L \rightarrow id \mid L \; , \; id$

Semantic Analysis of Functions and Calls

$BODY \rightarrow \{ \; VAR\_DECL \; STMT\_LIST \; \}$

$STMT\_LIST \rightarrow STMT\_LIST \; ; \; STMT \mid STMT$

$STMT \rightarrow BODY \mid FUNC\_CALL \mid ASG \mid$ /* others */

/* BODY may be regarded as a *compound statement* */

/* Assignment statement is being singled out */

/* to show how function calls can be handled */

$ASG \rightarrow LHS \; := \; E$

$LHS \rightarrow id$ /* array expression for exercises */

$E \rightarrow LHS \mid FUNC\_CALL \mid$/* other expressions */

$FUNC\_CALL \rightarrow id \; (\; PARAMLIST \;)$

$PARAMLIST \rightarrow PLIST \mid \epsilon$

$PLIST \rightarrow PLIST \; , \; E \mid E$

A Very Simple Symbol Table

A very simple symbol table (quite restricted and not really fast) is presented for use in the semantic analysis of functions

An array, *func_name_table* stores the function name records, assuming no nested function definitions

Each function name record has fields: name, result type, parameter list pointer, and variable list pointer

Parameter and variable names are stored as lists

Each parameter and variable name record has fields: name, type, parameter-or-variable tag, and level of declaration (1 for parameters, and 2 or more for variables)

A Very Simple Symbol Table

func_name_table

name	result type	parameter list pointer	local variable list pointer	number of parameters

Parameter/Variable name record

name	type	parameter or variable tag	level of declaration

A Very Simple Symbol Table

Two variables in the same function, with the same name but different declaration levels, are treated as different variables (in their respective scopes)

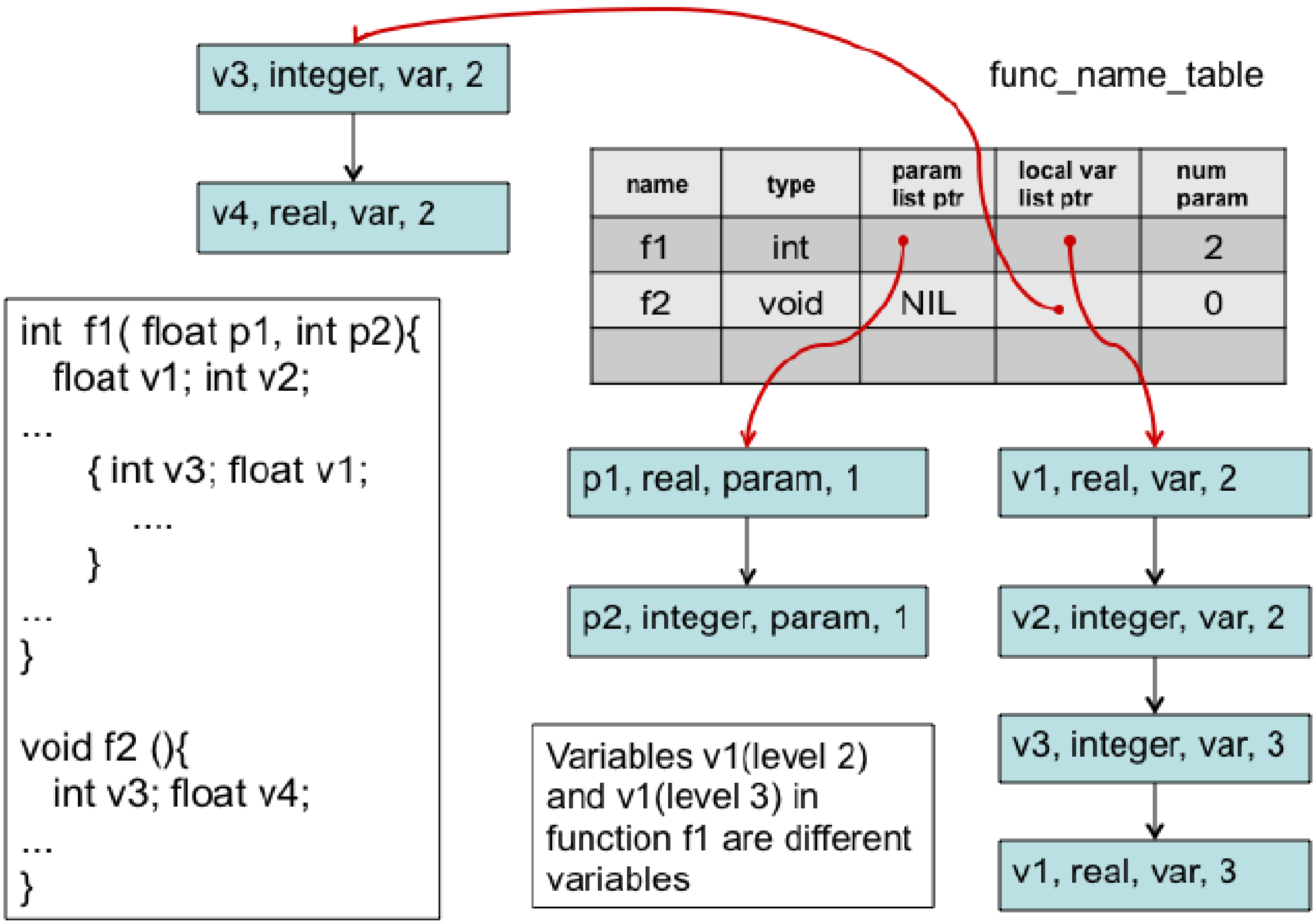
If a variable (at level > 2) and a parameter have the same name, then the variable name overrides the parameter name (only within the corresponding scope)

However, a declaration of a variable at level 2, with the same name as a parameter, is flagged as an error

The above two cases must be checked carefully

A search in the symbol table for a given name must always consider the names with the declaration levels *l*, *l-1*, ... , *2*, in that order, where *l* is the current level

A Very Simple Symbol Table



A Very Simple Symbol Table

The global variable, *active_func_ptr*, stores a pointer to the function name entry in *func_name_table* of the function that is currently being compiled

The global variable, *level*, stores the current nesting level of a statement block

The global variable, *call_name_ptr*, stores a pointer to the function name entry in *func_name_table* of the function whose call is being currently processed

The function *search_func*(*n*, *found*, *fnptr*) searches the function name table for the name *n* and returns *found* as T or F; if found, it returns a pointer to that entry in *fnptr*

A Very Simple Symbol Table

The function *search_param*(*p*, *fnptr*, *found*, *pnptr*) searches the parameter list of the function at *fnptr* for the name *p*, and returns *found* as T or F; if found, it returns a pointer to that entry in the parameter list, in *pnptr*

The function *search_var*(*v*, *fnptr*, *l*, *found*, *vnptr*) searches the variable list of the function at *fnptr* for the name *v* at level *l* or lower, and returns *found* as T or F; if found, it returns a pointer to that entry in the variable list, in *vnptr*. Higher levels are preferred

The other symbol table routines will be explained during semantic analysis

Semantic Analysis of Functions and Calls

FUNC_DECL $\rightarrow$ *FUNC_HEAD* { *VAR_DECL BODY* }
{delete_var_list(active_func_ptr, level);
active_func_ptr := NULL; level := 0;}

FUNC_HEAD $\rightarrow$ *RES_ID* (*DECL_PLIST*) {level := 2}

RES_ID $\rightarrow$ *RESULT id*
{ search_func(id.name, found, namptr);
if (found) error('function already declared');
else enter_func(id.name, RESULT.type, namptr);
active_func_ptr := namptr; level := 1}

RESULT $\rightarrow$ *int* {**action1**} | *float* {**action2**}
| *void* {**action3**}

{action 1:} {RESULT.type := integer}
{action 2:} {RESULT.type := real}
{action 3:} {RESULT.type := void}

Semantic Analysis of Functions and Calls

$DECL\_PLIST \rightarrow DECL\_PL \mid \epsilon$

$DECL\_PL \rightarrow DECL\_PL\ ,\ DECL\_PARAM \mid DECL\_PARAM$

$DECL\_PARAM \rightarrow T\ id$

{search_param(id.name, active_func_ptr, found, pnptr);
if (found) {error('parameter already declared')}
else {enter_param(id.name, T.type, active_func_ptr)}

$T \rightarrow int$ {T.type := integer} | $float$ {T.type := real}

$VAR\_DECL \rightarrow DLIST \mid \epsilon$

$DLIST \rightarrow D \mid DLIST\ ;\ D$

/* We show the analysis of simple variable declarations. Arrays can be handled using methods desribed earlier. Extension of the symbol table and SATG to handle arrays is left as an exercise. */

Semantic Analysis of Functions and Calls

$D \rightarrow T\ L$ {patch_var_type(T.type, L.list, level)}
/* Patch all names on L.list with declaration level, *level*, with T.type */

$L \rightarrow id$

```
{search_var(id.name, active_func_ptr, level, found, vn);
  if (found && vn -> level == level)
      {error('variable already declared at the same level');
         L.list := makelist(NULL);}
  else if (level==2)
{search_param(id.name, active_func_ptr, found, pn);
  if (found) {error('redeclaration of parameter as variable');
                      L.list := makelist(NULL);}
} /* end of if (level == 2) */
  else {enter_var(id.name, level, active_func_ptr, vnptr);
           L.list := makelist(vnptr);}}
```

Semantic Analysis of Functions and Calls

$L_1 \rightarrow L_2$, id

```
{search_var(id.name, active_func_ptr, level, found, vn);
 if (found && vn -> level == level)
    {error('variable already declared at the same level');
       L1.list := L2.list;}
 else if (level==2)
{search_param(id.name, active_func_ptr, found, pn);
 if (found) {error('redclaration of parameter as variable');
                    L1.list := L2.list;}
} /* end of if (level == 2) */
 else {enter_var(id.name, level, active_func_ptr, vnptr);
           L1.list := append(L2.list, vnptr);}}
```

$BODY \rightarrow$ '{'{level++;} VAR_DECL $STMT\_LIST$

{delete_var_list(active_func_ptr, level); level- -;}'}'

$STMT\_LIST \rightarrow STMT\_LIST$; $STMT \mid STMT$

$STMT \rightarrow BODY \mid FUNC\_CALL \mid ASG \mid$ /* others */

Semantic Analysis of Functions and Calls

$ASG \rightarrow LHS\ :=\ E$

{if (LHS.type $\neq$ *errortype* && E.type $\neq$ *errortype*)
  if (LHS.type $\neq$ E.type) error('type mismatch of
      operands in assignment statement')}

$LHS \rightarrow id$

{search_var(id.name, active_func_ptr, level, found, vn);
 if ($\sim$found)
  {search_param(id.name, active_func_ptr, found, pn);
   if ($\sim$found){ error('identifier not declared');
             LHS.type := *errortype*}
   else LHS.type := pn -> type}
 else LHS.type := vn -> type}

$E \rightarrow LHS$ {E.type := LHS.type}

$E \rightarrow FUNC\_CALL$ {E.type := FUNC_CALL.type}

Semantic Analysis of Functions and Calls

$FUNC\_CALL \rightarrow id$ ($PARAMLIST$)

```
{ search_func(id.name, found, fnptr);
  if (~found) {error('function not declared');
                call_name_ptr := NULL;
                FUNC_CALL.type := errortype;}
  else {FUNC_CALL.type := get_result_type(fnptr);
        call_name_ptr := fnptr;
  if (call_name_ptr.numparam ≠ PARAMLIST.pno)
     error('mismatch in mumber of parameters
           in declaration and call');}
```

$PARAMLIST \rightarrow PLIST$ {PARAMLIST.pno := PLIST.pno }

$\quad\quad\quad\quad | \; \epsilon$ {PARAMLIST.pno := 0 }

Semantic Analysis of Functions and Calls

$PLIST \rightarrow E$ {PLIST.pno := 1;
check_param_type(call_name_ptr, 1, E.type, ok);
if ($\sim$ok) error('parameter type mismatch
in declaration and call');}

$PLIST_1 \rightarrow PLIST_2$, E {$PLIST_1$.pno := $PLIST_2$.pno + 1;
check_param_type(call_name_ptr, $PLIST_2$.pno + 1,
E.type, ok);
if ($\sim$ok) error('parameter type mismatch
in declaration and call');}

Semantic Analysis of Functions and Calls

$PLIST \rightarrow E$ {PLIST.pno := 1;
check_param_type(call_name_ptr, 1, E.type, ok);
if ($\sim$ok) error('parameter type mismatch
in declaration and call');}

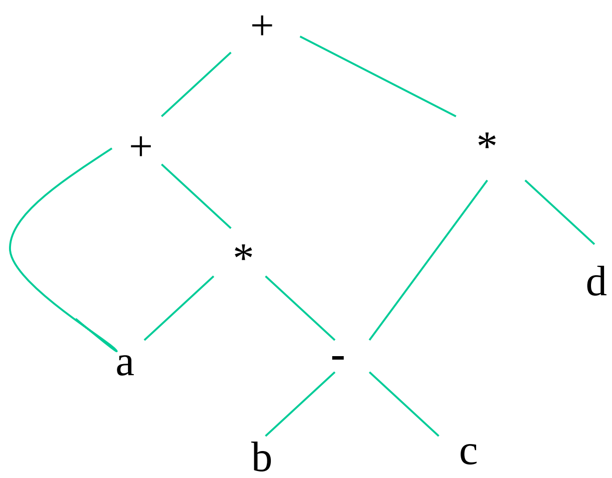
$PLIST_1 \rightarrow PLIST_2$, E {$PLIST_1$.pno := $PLIST_2$.pno + 1;
check_param_type(call_name_ptr, $PLIST_2$.pno + 1,
E.type, ok);
if ($\sim$ok) error('parameter type mismatch
in declaration and call');}

Intermediate Code Generation

- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)
- The type of intermediate code deployed is based on the application
  - Quadruples, triples, abstract syntax trees, DAGs, are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
  - Conditional constant propagation and global value numbering are more effective on SSA

Three Address Code

- In a three address code there is at most one operator at the right side of an instruction
- Example:



t1 = b – c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4

- *Linearised presentation of AST or DAG*
- *Explicit names given to interior nodes of the graph*

Implementations of Three Address Code

3-address code

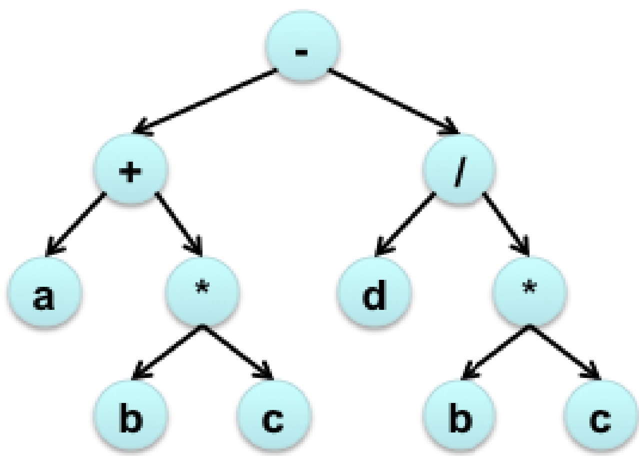
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

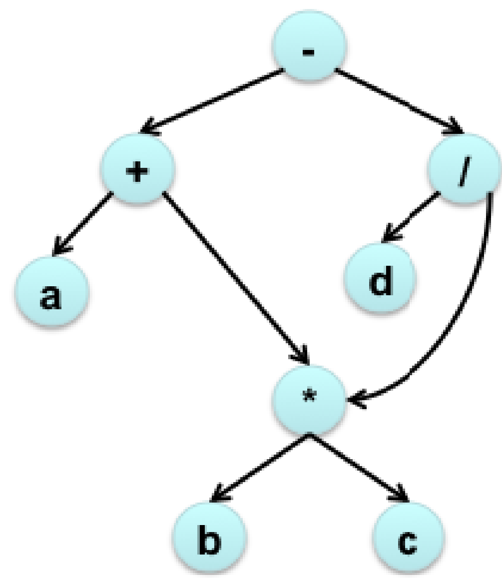
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)



Syntax tree



DAG

Three Address Instruction Forms

Assignments using:

- x = y op z
- x = op y
- x = y

Jumps using:

- goto L
- if x goto L
- if x relop y goto L

Procedure calls using:

- func begin A
- func end
- param x
- refparam x
- call f,n
- return
- return a

Arrays using:

- x = y[i] and x[i] = y

Pointers using:

- x = &y and x = *y and *x =y

Three Address Code – Example 1

Assignments using:
- x = y op z
- x = op y
- x = y

Jumps using:
- goto L
- if x goto L
- if x relop y goto L

Procedure calls using:
- func begin A
- func end
- param x
- refparam x
- call f,n
- return
- return a

Arrays using:
- x = y[i] and x[i] = y

Pointers using:
- x = &y and x = *y
- *x =y

C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

```
    dot_prod = 0;          |     T6 = T4[T5]
    i = 0;                 |     T7 = T3*T6
L1: if(i >= 10)goto L2     |     T8 = dot_prod+T7
    T1 = addr(a)           |     dot_prod = T8
    T2 = i*4               |     T9 = i+1
    T3 = T1[T2]            |     i = T9
    T4 = addr(b)           |     goto L1
    T5 = i*4               |L2:
```

Three Address Code – Example 2

Assignments using:
- x = y op z
- x = op y
- x = y

Jumps using:
- goto L
- if x goto L
- if x relop y goto L

Procedure calls using:
- func begin A
- func end
- param x
- refparam x
- call f,n
- return
- return a

Arrays using:
- x = y[i] and x[i] = y

Pointers using:
- x = &y and x = *y
- *x =y

C-Program

```
int a[10], b[10], dot_prod, i; int* a1; int* b1;
dot_prod = 0; a1 = a; b1 = b;
for (i=0; i<10; i++) dot_prod += *a1++ * *b1++;
```

Intermediate code

```
    dot_prod = 0;          |    b1 = T6
    a1 = &a                |    T7 = T3*T5
    b1 = &b                |    T8 = dot_prod+T7
    i = 0                  |    dot_prod = T8
L1: if(i>=10)goto L2       |    T9 = i+1
    T3 = *a1               |    i = T9
    T4 = a1+1              |    goto L1
    a1 = T4                |L2:
    T5 = *b1
    T6 = b1+1
```

Three Address Code – Example 3

Assignments using:
- x = y op z
- x = op y
- x = y

Jumps using:
- goto L
- if x goto L
- if x relop y goto L

Procedure calls using:
  - func begin A
  - func end
  - param x
  - refparam x
  - call f,n
  - return
  - return a

Arrays using:
- x = y[i] and x[i] = y

Pointers using:
- x = &y and x = *y
- *x =y

C-Program (function)

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
```

Intermediate code

```
     func begin dot_prod |     T6 = T4[T5]
     d = 0;              |     T7 = T3*T6
     i = 0;              |     T8 = d+T7
L1:  if(i >= 10)goto L2  |     d = T8
     T1 = addr(x)        |     T9 = i+1
     T2 = i*4            |     i = T9
     T3 = T1[T2]         |     goto L1
     T4 = addr(y)        |L2:  return d
     T5 = i*4            |     func end
```

Three Address Code – Example 3

Assignments using:
- x = y op z
- x = op y
- x = y

Jumps using:
- goto L
- if x goto L
- if x relop y goto L

Procedure calls using:
- func begin A
- func end
- param x
- refparam x
- call f,n
- return
- return a

Arrays using:
- x = y[i] and x[i] = y

Pointers using:
- x = &y and x = *y
- *x =y

C-Program (main)

```
main(){
   int p; int a[10], b[10];
   p = dot_prod(a,b);
}
```

Intermediate code

```
func begin main
refparam a
refparam b
refparam result
call dot_prod, 3
p = result
func end
```

Code Template for If-Then-Else

```
If  (E)  S1  else  S2
        code for E (result in T)
        if  T≤ 0  goto  L1 /* if T is false, jump to else part */
        code for S1 /* all exits from within S1 also jump to L2 */
        goto L2 /* jump to exit */
L1:     code for S2 /* all exits from within S2 also jump to L2 */
L2:     /* exit */
```

```
If  (E)  S
        code for E (result in T)
        if  T≤ 0  goto  L1 /* if T is false, jump to exit */
        code for S /* all exits from within S also jump to L1 */
L1:     /* exit */
```

Code Template for While-Do

while (E) do S

```
L1:     code for E (result in T)
        if  T≤ 0  goto  L2 /* if T is false, jump to exit */
        code for S /* all exits from within S also jump to L1 */
        goto L1 /* loop back */
L2:     /* exit */
```

If-Then-Else Translation

Let us see the code generated for the following code fragment.
A_i are all assignments, and E_i are all expressions
if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

```
1           code for E1 /* result in T1 */
10          if (T1 <= 0), goto L1 (61)
            /* if T1 is false jump to else part */
11          code for E2 /* result in T2 */
35          if (T2 <= 0), goto L2 (43)
            /* if T2 is false jump to else part */
36          code for A1
42          goto L3 (82)
43    L2:   code for A2
60          goto L3 (82)
61    L1:   code for A3
82    L3:   code for A4
```

While-Do Translation

Code fragment:

while (E_1) do {if (E_2) then A_1; else A_2;} A_3;

```
1    L1:   code for E1 /* result in T1 */
15         if (T1 <= 0), goto L2 (79)
           /* if T1 is false jump to loop exit */
16         code for E2 /* result in T2 */
30         if (T2 <= 0), goto L3 (55)
           /* if T2 is false jump to else part */
31         code for A1
54         goto L1 (1)/* loop back */
55   L3:   code for A2
78         goto L1 (1)/* loop back */
79   L2:   code for A3
```

SATG Attributes

S.next, N.next: list of quads indicating where to jump; target of jump is still undefined

IFEXP.falselist: quad indicating where to jump if the expression is false; target of jump is still undefined

E.result: pointer to symbol table entry

- All temporaries generated during intermediate code generation are inserted into the symbol table
- In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names
- However, textual examples will use names

SATG Attributes

nextquad: global variable containing the number of the next quadruple to be generated

backpatch(list, quad_number): patches target of all 'goto' quads on the 'list' to 'quad_number'

merge(list-1, list-2,...,list-n): merges all the lists supplied as parameters

gen('quadruple'): generates 'quadruple' at position 'nextquad' and increments 'nextquad'

- In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names
  However, textual examples will use names

newtemp(temp-type): generates a temporary name of type *temp-type*, inserts it into the symbol table, and returns the pointer to that entry in the symbol table

SATG If-Then-Else

$IFEXP \rightarrow if \;\; E$

{ IFEXP.falselist := makelist(nextquad);
gen('if E.result $\leq$ 0 goto __'); }

$S \rightarrow IFEXP \;\; S_1; \;\; N \;\; else \;\; M \;\; S_2$

{ backpatch(IFEXP.falselist, M.quad);
S.next := merge(S_1.next, S_2.next, N.next); }

$S \rightarrow IFEXP \;\; S_1;$

{ S.next := merge(S_1.next, IFEXP.falselist); }

$N \rightarrow \epsilon$

{ N.next := makelist(nextquad);
gen('goto __'); }

$M \rightarrow \epsilon$

{ M.quad := nextquad; }

SATG of Other Statements

$S \rightarrow$ '{' L '}'

{ S.next := L.next; }

$S \rightarrow A$

{ S.next := makelist(nil); }

$S \rightarrow$ *return E*

{ gen('return E.result'); S.next := makelist(nil); }

$L \rightarrow L_1$ ';' M S

{ backpatch(L_1.next, M.quad);
L.next := S.next; }

$L \rightarrow S$

{ L.next := S.next; }

When the body of a procedure ends, we perform the following actions in addition to other actions:

{ backpatch(S.next, nextquad); gen('func end'); }

Sample If-Then-Else Trace

A_i are all assignments, and E_i are all expressions

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow IFEXP\ \ S_1;\ \ N_1\ \ else\ \ M_1\ \ S_2$

$\Rightarrow^* IFEXP_1\ \ IFEXP_2\ \ S_{21};\ \ N_2\ \ else\ \ M_2\ \ S_{22};\ \ N_1\ else\ \ M_1\ \ S_2$

Consider outer if-then-else

Code generation for E_1

gen('if E_1.result $\leq$ 0 goto __')
on reduction by $IFEXP_1 \rightarrow if\ \ E_1$
Remember the above quad address in $IFEXP_1$.falselist

Consider inner if-then-else

Code generation for E_2

gen('if E_2.result $\leq$ 0 goto __')
on reduction by $IFEXP_2 \rightarrow if\ \ E_2$
Remember the above quad address in $IFEXP_2$.falselist

Sample If-Then-Else Trace

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow^* IFEXP_1\ IFEXP_2\ S_{21};\ N_2\ else\ M_2\ S_{22};\ N_1\ else\ M_1\ S_2$

Code generated so far:

Code for E_1; if E_1.result ≤ 0 goto __ (on $IFEXP_1$.falselist);

Code for E_2; if E_2.result ≤ 0 goto __ (on $IFEXP_2$.falselist);

Code generation for S_{21}

gen('goto __'), on reduction by $N_2 \rightarrow \epsilon$
(remember in N_2.next)

L1: remember in M_2.quad, on reduction by $M_2 \rightarrow \epsilon$

Code generation for S_{22}

backpatch($IFEXP_2$.falselist, L1) (processing E_2 == false)
on reduction by $S_1 \rightarrow IFEXP_2\ S_{21}\ N_2\ else\ M_2\ S_{22}$
N_2.next is not yet patched; put on S_1.next

Sample If-Then-Else Trace

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow IFEXP \;\; S_1; \;\; N_1 \;\; else \;\; M_1 \;\; S_2$

$S \Rightarrow^* IFEXP_1 \; IFEXP_2 \; S_{21}; \; N_2 \; else \; M_2 \; S_{22}; \; N_1 \; else \; M_1 \; S_2$

Code generated so far:

Code for E_1; if E_1.result $\leq$ 0 goto __ (on $IFEXP_1$.falselist)

Code for E_2; if E_2.result $\leq$ 0 goto L1

Code for S_{21}; goto __ (on S_1.next)

L1: Code for S_{22}

gen('goto __'), on reduction by $N_1 \rightarrow \epsilon$ (remember in N_1.next)

L2: remember in M_1.quad, on reduction by $M_1 \rightarrow \epsilon$

Code generation for S_2

backpatch(IFEXP.falselist, L2) (processing E_1 == false)

on reduction by $S \rightarrow IFEXP \;\; S_1 \;\; N_1 \;\; else \;\; M_1 \;\; S_2$

N_1.next is merged with S_1.next, and put on S.next

Sample If-Then-Else Trace

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow^* IFEXP_1\ IFEXP_2\ S_{21};\ N_2\ else\ M_2\ S_{22};\ N_1\ else\ M_1\ S_2$

$L \Rightarrow^* L_1$ ';' $M_3\ \ S_4\ \ \Rightarrow^* S_3$ ';' $M_3\ \ S_4$

Code generated so far (for S_3/L_1 above):

Code for E_1; if E_1.result $\leq$ 0 goto L2

Code for E_2; if E_2.result $\leq$ 0 goto L1

Code for S_{21}; goto __ (on S_3.next/L_1.next)

L1: Code for S_{22}

goto __ (on S_3.next/L_1.next)

L2: Code for S_2

L3: remember in M_3.quad, on reduction by $M_3 \to \epsilon$

Code generation for S_4

backpatch(L_1.next, L3), on reduction by $L \to L_1$ ';' $M_3\ \ S_4$

L.next is empty

Sample If-Then-Else Trace

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow^* \mathit{IFEXP}_1\ \mathit{IFEXP}_2\ S_{21};\ N_2\ \mathit{else}\ M_2\ S_{22};\ N_1\ \mathit{else}\ M_1\ S_2$

$L \Rightarrow^* L_1$ ‘;’ $M_3\ \ S_4\ \Rightarrow^* S_3$ ‘;’ $M_3\ \ S_4$

Final generated code

Code for E_1; if E_1.result $\leq$ 0 goto L2
Code for E_2; if E_2.result $\leq$ 0 goto L1
Code for S_{21}; goto L3
L1: Code for S_{22}
goto L3
L2: Code for S_2
L3: Code for S_4

SATG for While-Do

$WHILEXEP \rightarrow while \ \ M \ \ E$

{ WHILEEXP.falselist := makelist(nextquad);

gen('if E.result $\leq$ 0 goto ___');

WHILEEXP.begin := M.quad; }

$S \rightarrow WHILEXEP \ \ do \ \ S_1$

{ gen('goto WHILEEXP.begin');

backpatch(S_1.next, WHILEEXP.begin);

S.next := WHILEEXP.falselist; }

$M \rightarrow \epsilon$ (repeated here for convenience)

{ M.quad := nextquad; }

Code Template Function Calls

Assumtion: No nesting of functions

```
result foo(parameter list){ variable declarations; Statement list; }
func begin foo
/* creates activation record for foo - */
/* - space for local variables and temporaries */
code for Statement list
func end /* releases activation record and return */

x = bar(p1,p2,p3);
code for evaluation of p1, p2, p3 (result in T1, T2, T3)
/* result is supposed to be returned in T4 */
param T1;  param T2;  param T3; refparam T4;
call bar, 4
/* creates appropriate access links, pushes return address */
/* and jumps to code for bar */
x = T4
```

SATG Function Calls

$FUNC\_CALL \rightarrow id$ **{action 1}** ($PARAMLIST$) **{action 2}**

{action 1:} {search_func(id.name, found, fnptr);
call_name_ptr := fnptr }

{action 2:}

{ result_var := newtemp(get_result_type(call_name_ptr));
gen('refparam result_var');
/* Machine code for return a places a in result_var */
gen('call call_name_ptr, PARAMLIST.pno+1'); }

$PARAMLIST \rightarrow PLIST$ { PARAMLIST.pno := PLIST.pno }

$PARAMLIST \rightarrow \epsilon$ {PARAMLIST.pno := 0 }

$PLIST \rightarrow E$ { PLIST.pno := 1; gen('param E.result'); }

$PLIST_1 \rightarrow PLIST_2 \text{ , } E$

{ $PLIST_1$.pno := $PLIST_2$.pno + 1; gen('param E.result'); }

SATG Function Calls

$FUNC\_DECL \rightarrow FUNC\_HEAD \; \{ \; VAR\_DECL \;\; BODY \; \}$

{ backpatch(BODY.next, nextquad);
 gen('func end');}

$FUNC\_HEAD \rightarrow RESULT \; id \; (\; DECL\_PLIST \;)$

{ search_func(id.name, found, namptr);
 active_func_ptr := namptr;
 gen('func begin active_func_ptr'); }

Code Template – Expressions and Assignments

```
int a[10][20][35], b;
b = exp1;
code for evaluation of exp1 (result in T1)
b = T1
/* Assuming the array access to be, a[i][j][k] */
/* base address = addr(a), offset = (((i*n2)+j)*n3)+k)*ele_size */
a[exp2][exp3][exp4] = exp5;

10: code for exp2 (result in T2)  || 141: T8 = T7+T6
70: code for exp3 (result in T3)  || 142: T9 = T8*intsize
105: T4 = T2*20                   || 143: T10 = addr(a)
106: T5 = T4+T3                   || 144: code for exp5 (result in T11)
107: code for exp4 (result in T6) || 186: T10[T9] = T11
140: T7 = T5*35
```

SATG – Expressions and Assignments

$S \to L \ := \ E$

/* L has two attributes, L.place, pointing to the name of the variable or temporary in the symbol table, and L.offset, pointing to the temporary holding the offset into the array (NULL in the case of a simple variable) */

{ if (L.offset == NULL) gen('L.place = E.result');
else gen('L.place[L.offset] = E.result');}

$E \to (\ E_1 \ )$ {E.result := E_1.result; }

$E \to L$ { if (L.offset == NULL) E.result := L.place;
else { E.result := newtemp(L.type);
gen('E.result = L.place[L.offset]'); }

$E \to num$ { E.result := newtemp(num.type);
gen('E.result = num.value'); }

SATG – Expressions and Assignments

$E \rightarrow E_1 + E_2$

{ result_type := compatible_type(E_1.type, E_2.type);
 E.result := newtemp(result_type);
 if (E_1.type == result_type) operand_1 := E_1.result;
 else if (E_1.type == integer && result_type == real)
 { operand_1 := newtemp(real);
 gen('operand_1 = cnvrt_float(E_1.result); };
 if (E_2.type == result_type) operand_2 := E_2.result;
 else if (E_2.type == integer && result_type == real)
 { operand_2 := newtemp(real);
 gen('operand_2 = cnvrt_float(E_2.result); };
 gen('E.result = operand_1 + operand_2');
}

SATG – Expressions and Assignments

$E \rightarrow E_1 || E_2$

{ E.result := newtemp(integer);

gen('E.result = E_1.result || E_2.result');

$E \rightarrow E_1 < E_2$

{ E.result := newtemp(integer);

gen('E.result = 1');

gen('if E_1.result < E_2.result goto nextquad+2');

gen('E.result = 0');

}

$L \rightarrow id$ { search_var_param(id.name, active_func_ptr,
level, found, vn); L.place := vn; L.offset := NULL; }

Note: *search_var_param*() searches for *id.name* in the variable list first, and if not found, in the parameter list next.

SATG – Expressions and Assignments

$ELIST \rightarrow id\ [\ E$
{ search_var_param(id.name, active_func_ptr,
level, found, vn); ELIST.dim := 1;
ELIST.arrayptr := vn; ELIST.result := E.result; }

$L \rightarrow ELIST\ ]$ { L.place := ELIST.arrayptr;
temp := newtemp(int); L.offset := temp;
ele_size := ELIST.arrayptr -> ele_size;
gen('temp = ELIST.result * ele_size'); }

$ELIST \rightarrow ELIST_1\ ,\ E$
{ ELIST.dim := $ELIST_1$.dim + 1;
ELIST.arrayptr := $ELIST_1$.arrayptr
num_elem := get_dim($ELIST_1$.arrayptr, $ELIST_1$.dim + 1);
temp1 := newtemp(int); temp2 := newtemp(int);
gen('temp1 = $ELIST_1$.result * num_elem');
ELIST.result := temp2; gen('temp2 = temp1 + E.result'); }