Semantic Analysis - 6

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)

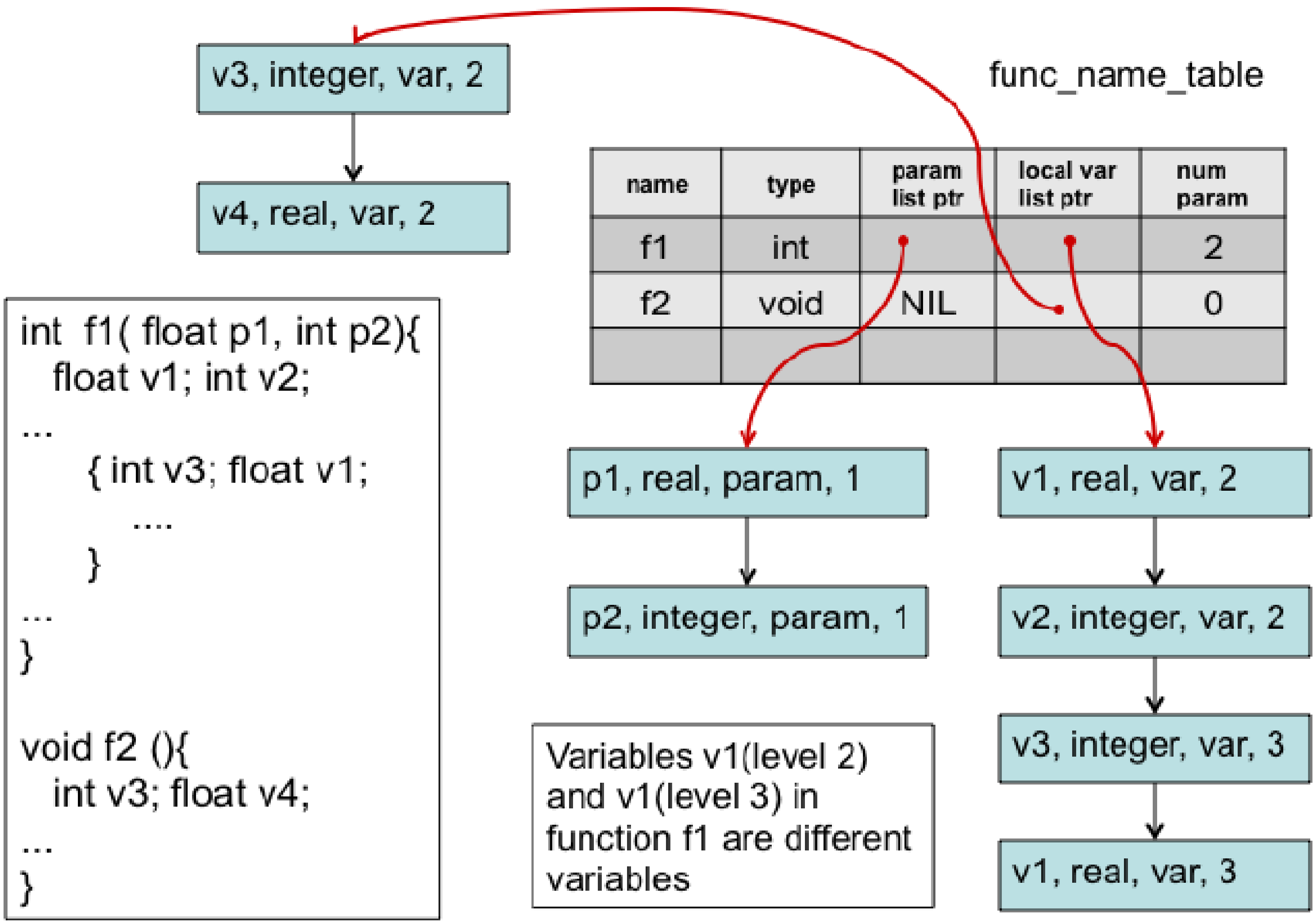
If a variable (at level > 2) and a parameter have the same name, then the variable name overrides the parameter name (only within the corresponding scope)

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 → *FUNC_HEAD* { *VAR_DECL BODY* }
{delete_var_list(active_func_ptr, level);
active_func_ptr := NULL; level := 0;}

FUNC_HEAD → *RES_ID* (*DECL_PLIST*) {level := 2}

RES_ID → *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 → *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');}

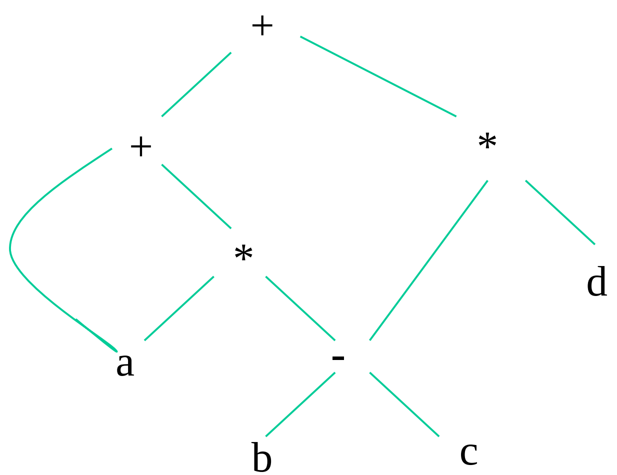
$PLIST_1 \rightarrow PLIST_2$, E {$PLIST_1$.pno := $PLIST_2$.pno + 1;
check_param_type(call_name_ptr, $PLIST_2$.pno + 1,
E.type, ok);
if ($\sim$ok) error('parameter type mismatch
in declaration and call');}

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

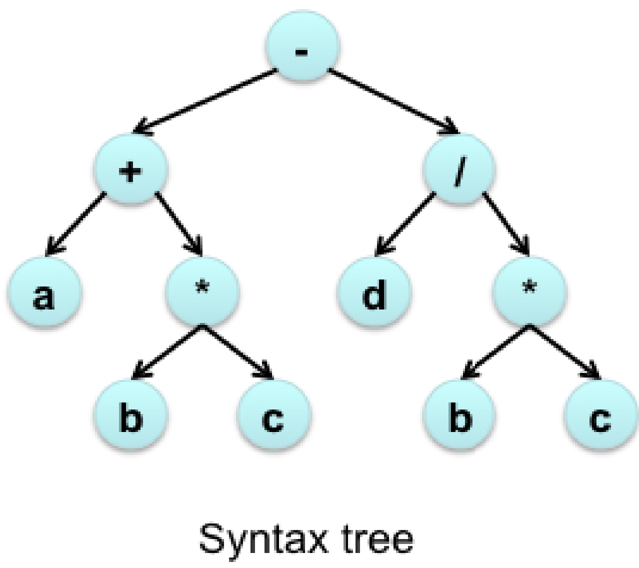
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

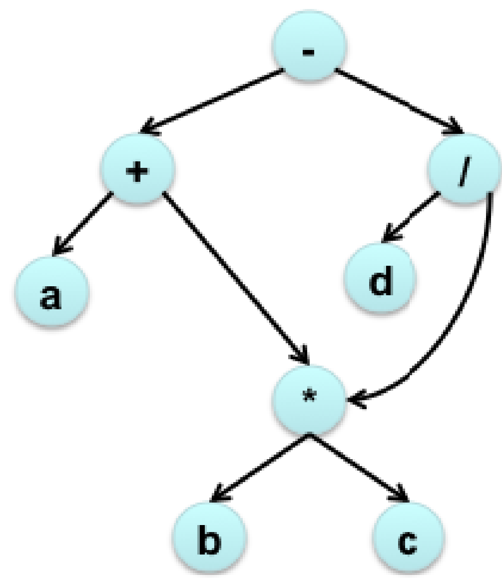
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

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
```