Intermediate Code generation
4th Phase of Compiler Construction

Section 5.1: Intermediate Code Generation

INTERMEDIATE CODE GENERATION

Intermediate codes are machine independent codes, but they are close to machine instructions.

- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.

Benefits of using a machine-independent intermediate form are:

- Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
- A machine-independent code optimizer can be applied to the intermediate representation.

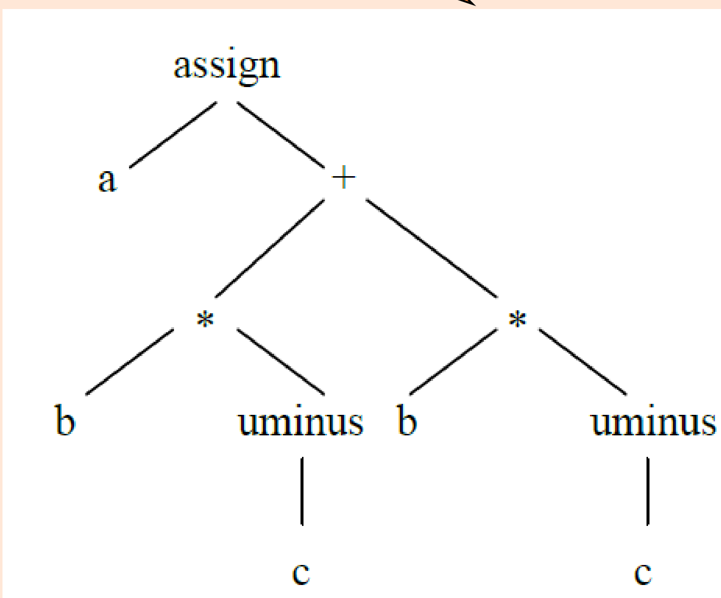
Three ways of intermediate representation:

- Graphical representations(syntax trees)
- Postfix notation (operations on values stored on operand stack; similar to JVM bytecode)
- Three-address code (triples or Quadruples)

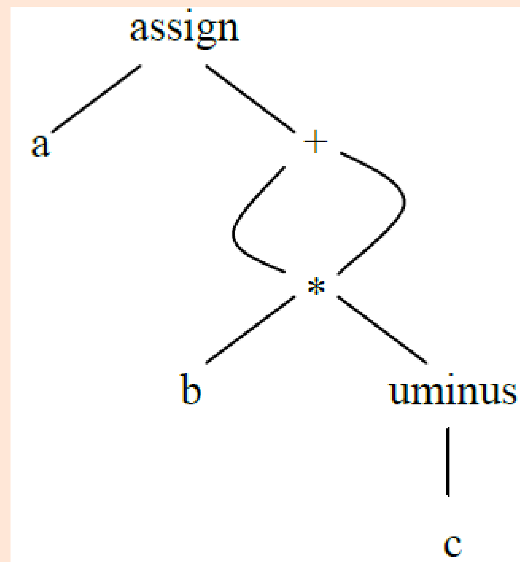
ABSTRACT SYNTAX TREE

- A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified.

a := b * -c + b * -c



Tree



DAG

ABSTRACT SYNTAX TREE

- ASTs don't show the whole syntactic clutter, but represent the parsed string in a structured way, discarding all information that may be important for parsing the string, but isn't needed for analyzing it.
- Abstract syntax trees, or simply *syntax trees*, differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees.

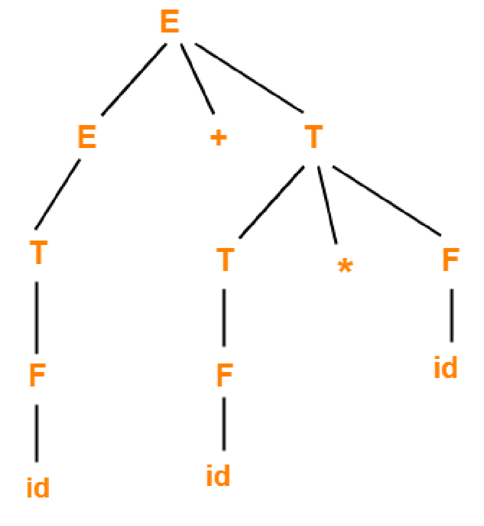
For the Grammar

$E \rightarrow E + T \mid T$

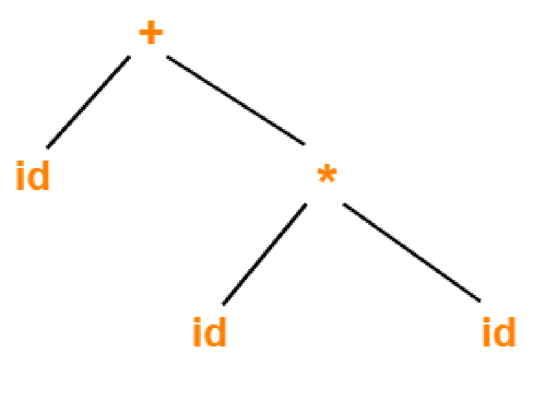
$T \rightarrow T \times F \mid F$

$F \rightarrow (E) \mid id$

Parse the string 'id + id x id'

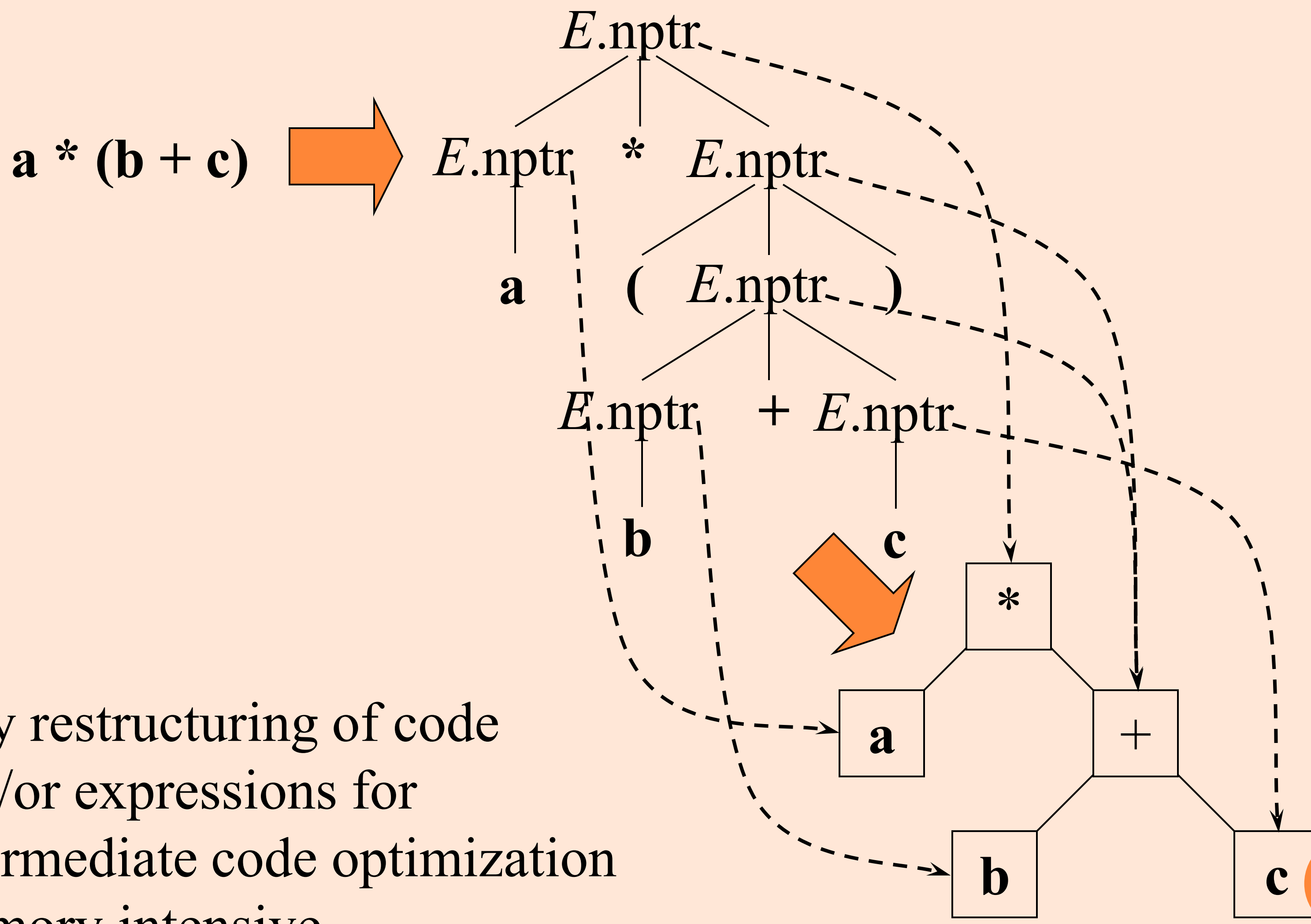


Parse Tree



Syntax Tree

ABSTRACT SYNTAX TREES



Pro: easy restructuring of code and/or expressions for intermediate code optimization

Cons: memory intensive

Postfix Notation

a := b * -c + b * -c

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

Pro: easy to generate
Cons: stack operations are more difficult to optimize

```
iload 2 // push b
iload 3 // push c
ineg // uminus
imul // *
iload 2 // push b
iload 3 // push c
ineg // uminus
imul // *
iadd // +
istore 1 // store a
```

THREE-ADDRESS CODE

Three-address code is a sequence of statements of the general form

x : = y op z

where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data. Thus a source language expression like

x+ y*z might be translated into a sequence

t1 : = y * z

t2 : = x + t1

where t1 and t2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three address code to be easily rearranged – unlike postfix notation

Three-Address Code (Quadruples)

a := b * -c + b * -c

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Linearized representation of a syntax tree

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a := t5
```

Linearized representation of a syntax DAG

COMMON THREE-ADDRESS STATEMENTS

1. Assignment statements of the form x : = y op z, where op is a binary arithmetic or logical operation.

2. Assignment instructions of the form x : = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3. Copy statements of the form x : = y where the value of y is assigned to x.

4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

5. Conditional jumps such as if x relop y goto L. This instruction applies a relational operator (<, =, >=, etc.) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.

COMMON THREE-ADDRESS STATEMENTS

6. param x and call p, n for procedure calls and return y, where y representing a returned value is optional. For example,

param x1

param x2

. . .

param xn

call p,n

generated as part of a call of the procedure p(x1, x2, …. ,xn).

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Three-Address Statements

Binary Operator: `op y,z,result` or `result := y op z`

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex:
```
add  a,b,c
gt   a,b,c
addr a,b,c
addi a,b,c
```

Unary Operator: `op y,,result` or `result := op y`

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex:
```
uminus    a,,c
not       a,,c
inttoreal a,,c
```

THREE-ADDRESS STATEMENTS (CONT.)

Move Operator: `mov y,,result` or `result := y`

where the content of `y` is copied into `result`.

Ex:

```
mov   a,,c
movi  a,,c
movr  a,,c
```

Unconditional Jumps: `jmp ,,L` or `goto L`

Jump to the three-address code with the label `L`, and the execution continues from that statement.

Ex:

```
jmp   ,,L1     // jump to L1
jmp   ,,7      // jump to the statement 7
```

Three-Address Statements (cont.)

Conditional Jumps: `jmp`***`relop`***` y,z,L`

or

`if y `***`relop`***` z goto L`

Jump to the three-address code with the label `L` if the result of `y relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Ex:

```
jmpgt    y,z,L1       // jump to L1 if y>z
jmpgte   y,z,L1       // jump to L1 if y>=z
jmpe     y,z,L1       // jump to L1 if y==z
jmpne    y,z,L1       // jump to L1 if y!=z
```

Relational operator can also be a unary operator.

```
jmpnz    y,,L1        // jump to L1 if y is not zero
jmpz     y,,L1        // jump to L1 if y is zero
jmpt     y,,L1        // jump to L1 if y is true
jmpf     y,,L1        // jump to L1 if y is false
```

THREE-ADDRESS STATEMENTS (CONT.)

Procedure Parameters: `param x,,` or `param x`

Procedure Calls: `call p,n,` or `call p,n`

where `x` is an actual parameter, we invoke the procedure `p` with `n` parameters.

Ex:

$$\begin{array}{l} \texttt{param } x_1\texttt{,,} \\ \texttt{param } x_2\texttt{,,} \\ \qquad\qquad\qquad\rightarrow\ p(x_1,\ldots,x_n) \\ \texttt{param } x_n\texttt{,,} \\ \texttt{call p,n,} \end{array}$$

`f(x+1,y)` ➔

```
add   x,1,t1
param t1,,
param y,,
call  f,2,
```

Three-Address Statements (cont.)

Indexed Assignments:

```
move y[i],,x   or   x := y[i]
move x,,y[i]   or   y[i] := x
```

Address and Pointer Assignments:

```
moveaddr y,,x   or   x := &y
movecont y,,x   or   x := *y
```

SYNTAX-DIRECTED TRANSLATION INTO THREE ADDRESS CODE

- Use attributes
  - E.*place:* the name that will hold the value of E
    - Identifier will be assumed to already have the place attribute defined.
  - E.*code:* hold the three address code statements that evaluate E (this is the `translation' attribute).

- Use function ***newtemp*** that returns a new temporary variable which can be used.

- Use function ***gen*** to generate a single three address statement given the necessary information (variable names and operations).

SYNTAX-DIRECTED TRANSLATION INTO THREE ADDRESS CODE

$S \rightarrow$ **id** := E	S.code = E.code || **gen**('mov' E.place ',,' id.place)
$E \rightarrow E_1 + E_2$	E.place = **newtemp**();
	E.code = E_1.code || E_2.code || **gen**('add' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow E_1 * E_2$	E.place = **newtemp**();
	E.code = E_1.code || E_2.code || **gen**('mult' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow - E_1$	E.place = **newtemp**();
	E.code = E_1.code || **gen**('uminus' E_1.place ',,' E.place)
$E \rightarrow (E_1)$	E.place = E_1.place;
	E.code = E_1.code
$E \rightarrow$ **id**	E.place = **id**.place;
	E.code = null

SYNTAX-DIRECTED TRANSLATION (CONT.)

$S \rightarrow$ while E do S_1	S.begin = newlabel();
	S.after = newlabel();
	S.code = gen(S.begin ":") ‖ E.code ‖
	gen('jmpf' E.place ',,' S.after) ‖ S_1.code ‖
	gen('jmp' ',,' S.begin) ‖
	gen(S.after ':")
$S \rightarrow$ if E then S_1 else S_2	S.else = newlabel();
	S.after = newlabel();
	S.code = E.code ‖
	gen('jmpf' E.place ',,' S.else) ‖ S_1.code ‖
	gen('jmp' ',,' S.after) ‖
	gen(S.else ':") ‖ S_2.code ‖
	gen(S.after ':")

TRANSLATION SCHEME TO PRODUCE THREE-ADDRESS CODE

$S \rightarrow$ **id** := E	{ p= lookup(id.name);
	if (p is not nil) then emit('**mov**' E.place ',,' p)
	else error("undefined-variable") }
$E \rightarrow E_1 + E_2$	{ E.place = newtemp();
	emit('**add**' E_1.place ',' E_2.place ',' E.place) }
$E \rightarrow E_1 * E_2$	{ E.place = newtemp();
	emit('**mult**' E_1.place ',' E_2.place ',' E.place) }
$E \rightarrow - E_1$	{ E.place = newtemp();
	emit('**uminus**' E_1.place ',,' E.place) }
$E \rightarrow (E_1)$	{ E.place = E_1.place; }
$E \rightarrow$ **id**	{ p= lookup(id.name);
	if (p is not nil) then E.place = **id**.place
	else error("undefined-variable") }

Translation Scheme with Locations

$S \rightarrow$ **id** := { E.inloc = S.inloc } E
{ p = lookup(id.name);
if (p is not nil) then { emit(E.outloc 'mov' E.place ',,' p); S.outloc=E.outloc+1 }
else { error("undefined-variable"); S.outloc=E.outloc } }

$E \rightarrow$ { E_1.inloc = E.inloc } E_1 + { E_2.inloc = E_1.outloc } E_2
{ E.place = newtemp(); emit(E_2.outloc 'add' E_1.place ',' E_2.place ',' E.place); E.outloc=E_2.outloc+1 }

$E \rightarrow$ { E_1.inloc = E.inloc } E_1 + { E_2.inloc = E_1.outloc } E_2
{ E.place = newtemp(); emit(E_2.outloc 'mult' E_1.place ',' E_2.place ',' E.place); E.outloc=E_2.outloc+1 }

$E \rightarrow$ - { E_1.inloc = E.inloc } E_1
{ E.place = newtemp(); emit(E_1.outloc 'uminus' E_1.place ',,' E.place); E.outloc=E_1.outloc+1 }

$E \rightarrow$ (E_1) { E.place = E_1.place; E.outloc=E_1.outloc+1 }

$E \rightarrow$ **id** { E.outloc = E.inloc; p= lookup(id.name);
if (p is not nil) then E.place = **id**.place
else error("undefined-variable") }

BOOLEAN EXPRESSIONS

$E \rightarrow$ { E_1.inloc = E.inloc } E_1 and { E_2.inloc = E_1.outloc } E_2

{ E.place = newtemp(); emit(E_2.outloc 'and' E_1.place ',' E_2.place ',' E.place); E.outloc=E_2.outloc+1 }

$E \rightarrow$ { E_1.inloc = E.inloc } E_1 or { E_2.inloc = E_1.outloc } E_2

{ E.place = newtemp(); emit(E_2.outloc 'and' E_1.place ',' E_2.place ',' E.place); E.outloc=E_2.outloc+1 }

$E \rightarrow$ not { E_1.inloc = E.inloc } E_1

{ E.place = newtemp(); emit(E_1.outloc 'not' E_1.place ',,' E.place); E.outloc=E_1.outloc+1 }

$E \rightarrow$ { E_1.inloc = E.inloc } E_1 **relop** { E_2.inloc = E_1.outloc } E_2

{ E.place = newtemp();

emit(E_2.outloc **relop**.code E_1.place ',' E_2.place ',' E.place); E.outloc=E_2.outloc+1 }

TRANSLATION SCHEME(CONT.)

$S \rightarrow$ while { E.inloc = S.inloc } E do

{ emit(E.outloc 'jmpf' E.place ',,' '**NOTKNOWN**');

S_1.inloc=E.outloc+1; } S_1

{ emit(S_1.outloc 'jmp' ',,' S.inloc);

S.outloc=S_1.outloc+1;

backpatch(E.outloc,S.outloc); }

$S \rightarrow$ if { E.inloc = S.inloc } E then

{ emit(E.outloc 'jmpf' E.place ',,' '**NOTKNOWN**');

S_1.inloc=E.outloc+1; } S_1 else

{ emit(S_1.outloc 'jmp' ',,' '**NOTKNOWN**');

S_2.inloc=S_1.outloc+1;

backpatch(E.outloc,S_2.inloc); } S_2

{ S.outloc=S_2.outloc;

backpatch(S_1.outloc,S.outloc); }

THREE ADDRESS CODES - EXAMPLE

```
x:=1;
y:=x+10;
while (x<y) {
    x:=x+1;
    if (x%2==1) then y:=y+1;
    else y:=y-2;
}
```

➔

```
01: mov   1,,x
02: add   x,10,t1
03: mov   t1,,y
04: lt    x,y,t2
05: jmpf  t2,,17
06: add   x,1,t3
07: mov   t3,,x
08: mod   x,2,t4
09: eq    t4,1,t5
10: jmpf  t5,,14
11: add   y,1,t6
12: mov   t6,,y
13: jmp   ,,16
14: sub   y,2,t7
15: mov   t7,,y
16: jmp   ,,4
17:
```

Implementation of Three-Address Statements: Quads

a:=(-c*b)+(-c * b)

A quadruple is a record structure with four fields, which are, op, arg1, arg2 and result.

- The op field contains an internal code for the operator. The three-address statement x : = y op z is represented by placing y in arg1, z in arg2 and x in result (Res).
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. Temporary names must be entered into the symbol table as they are created.

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Quads (quadruples)

Pro: easy to rearrange code for global optimization

Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

a:=(-c*b)+(-c * b)

Three-address statements can be represented by records with only three fields: op, arg1 and arg2.

- The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as triples

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Triples

Pro: temporaries are implicit
Cons: difficult to rearrange code

Indirect Triples

Listing pointers to triples, rather than listing the triples themselves is called indirect triples.

a:=(-c*b)+(-c * b)

#	*Stmt*
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Triple container

Pro: temporaries are implicit & easier to rearrange code

EXAMPLE

Three address code for a+b*c –d/(b*c)

3-address code

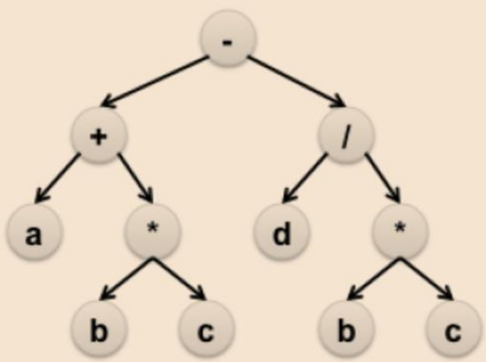
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

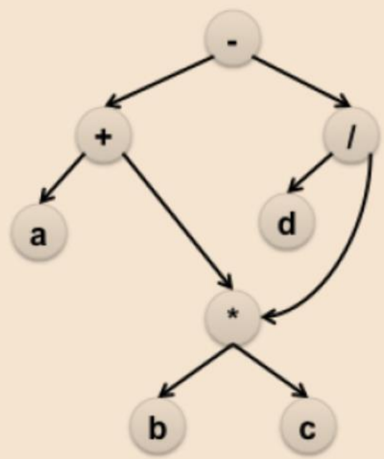
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

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



Syntax tree



DAG

Section 5.2: Type Checking

Type Checking

- A compiler has to do both syntactic and semantic check of the source program
- Semantic Checks can be of two types:
  - Static – done during compilation
  - Dynamic – done during run-time
- ***Type checking*** is one of these static checking operations.
  - we may not do all type checking at compile-time.
  - Some systems also use dynamic type checking too.
- A ***type system*** is a collection of rules for assigning type expressions to the parts of a program.
- A ***type checker*** implements a type system.
- A ***sound*** type system eliminates run-time type checking for type errors because it allow us to determine statically that these errors cannot occur when the target program runs.
- A programming language is ***strongly-typed***, if every program its compiler accepts will execute without type errors.
  - In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
  - Ex: int x[100]; … x[i] ➔ most of the compilers cannot guarantee that i will be between 0 and 99

TYPE EXPRESSION

- The type of a language construct is denoted by a *type expression*.
- A *type expression* can be:
  - **A basic type**
    - a primitive data type such as *integer, real, char, boolean*, …
    - *type-error* to signal a type error
    - *void* : no type
  - **A type name**
    - a name can be used to denote a type expression.
  - **A type constructor applies to other type expressions.**
    - **arrays**: If T is a type expression, then *array(I,T)* is a type expression where I denotes index range. Ex: array(0..99,int)
    - **products**: If T_1 and T_2 are type expressions, then their cartesian product $T_1 \times T_2$ is a type expression. Ex: int x int
    - **pointers**: If T is a type expression, then *pointer(T)* is a type expression. Ex: pointer(int)
    - **functions**: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression $D \rightarrow R$ where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

A Simple Type Checking System

$P \rightarrow D;E$

$D \rightarrow D;D$

$D \rightarrow$ **id**$:T$ { addtype(id.entry,T.type) }

$T \rightarrow$ char { T.type=char }

$T \rightarrow$ int { T.type=int }

$T \rightarrow$ real { T.type=real }

$T \rightarrow \uparrow T_1$ { T.type=pointer(T_1.type) }

$T \rightarrow$ array[**intnum**] of T_1 { T.type=array(1..intnum.val,T_1.type) }

The prefix operator ↑ builds a pointer type. Eg. ↑ Integer leads to the type expression pointer(integer)

Type Checking of Expressions

$E \rightarrow \textbf{id}$ { E.type=lookup(id.entry) }

*Lookup(E) is used to fetch the type saved in the symbol table entry pointed to by e

$E \rightarrow \textbf{literal}$ { E.type=char }

$E \rightarrow \textbf{num}_1$ { E.type=int }

$E \rightarrow \textbf{num}_2$ { E.type=real }

*Constants represented by the tokens **literal,** $\textbf{num}_1$ and $\textbf{num}_2$ have type char, int and real.

$E \rightarrow E_1 + E_2$ { if (E_1.type=int and E_2.type=int) then E.type=int
else if (E_1.type=int and E_2.type=real) then E.type=real
else if (E_1.type=real and E_2.type=int) then E.type=real
else if (E_1.type=real and E_2.type=real) then E.type=real
else E.type=type-error }

TYPE CHECKING OF EXPRESSIONS

$E \rightarrow E_1 [E_2]$ { if (E_2.type=int and E_1.type=array(s,t)) then E.type=t
else E.type=type-error }

*The index expression $E_{2,}$ the index expression E_2 must have type integer. The result is the element type t obtained from the type array(s,t) of E_1.

$E \rightarrow E_1 \uparrow$ { if (E_1.type=pointer(t)) then E.type=t
else E.type=type-error }

*The postfix operator yields the object pointed to by its operand. The type of E is the type t of the object pointed to by the pointer E.

Type Checking of Statements

Assignment Statement

$S \rightarrow \textbf{id} = E$ { if (id.type=E.type then S.type=void
else S.type=type-error }

Conditional Statement

$S \rightarrow$ if E then S_1 { if (E.type=boolean then S.type=S_1.type
else S.type=type-error }

While Statement

$S \rightarrow$ while E do S_1 { if (E.type=boolean then S.type=S_1.type
else S.type=type-error }

Type Checking of Functions

$E \to E_1 (E_2)$ { if (E_2.type=s and E_1.type=s$\to$t) then E.type=t
else E.type=type-error }

Ex: int f(double x, char y) { ... }

f: double x char $\to$ int

argument types return type

Structural Equivalence of Type Expressions

- How do we know that two type expressions are equal?
- As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

Structural Equivalence Algorithm (sequiv):

if (s and t are same basic types) then return true

else if (s=array(s_1,s_2) and t=array(t_1,t_2)) then return (sequiv(s_1,t_1) and sequiv(s_2,t_2))

else if ($s = s_1 \times s_2$ and $t = t_1 \times t_2$) then return (sequiv(s_1,t_1) and sequiv(s_2,t_2))

else if (s=pointer(s_1) and t=pointer(t_1)) then return (sequiv(s_1,t_1))

else if ($s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$) then return (sequiv(s_1,t_1) and sequiv(s_2,t_2))

else return false

NAMES FOR TYPE EXPRESSIONS

- In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

```
type link = ↑ cell;
var p,q : link;
var r,s : ↑ cell
```

p,q,r,s have same types ?

- How do we treat type names?
  - Get equivalent type expression for a type name (then use structural equivalence), or
  - Treat a type name as a basic type.

Cycles in Type Expressions

```
type link = ↑ cell;
type cell = record
               x : int,
               next : link
            end;
```

- We cannot use structural equivalence if there are cycles in type expressions.
- We have to treat type names as basic types.
  - ➔ but this means that the type expression `link` is different than the type expression ↑`cell`.