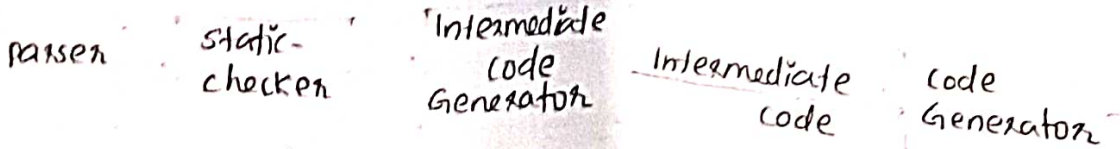
Module-4 : Intermediate Code Generation Refregue (1)

In the analysis-Synthesis model of a compiler, the front end analyses a source program and creates an intermediate representation, from which the backend generates target code.



Different forms of Intermediate representation in use are:

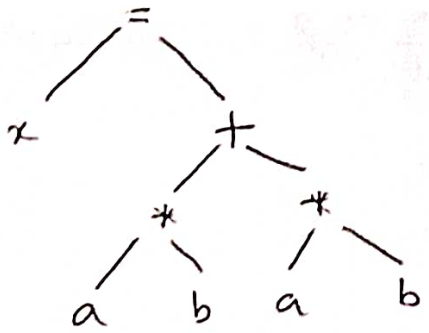
- Syntax tree - DAG: Directed Acyclic Graph
- Three address code.

Directed Acyclic Graph (DAG)

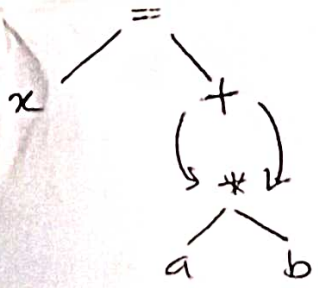
DAG is a special kind of Abstract Syntax Tree. Syntax tree is a tree representation of the Abstract Syntax tree Syntactic structure of source code.

DAG identifies the common subexpressions (subexpressions that occur more than once) of the expression.

$x = a*b + a*b$



[Abstract Syntax Tree]



[DAG]

DAG each node contains a unique value. It does not contain any cycles in it, nce called acyclic. A DAG is usually constructed using Three address code.

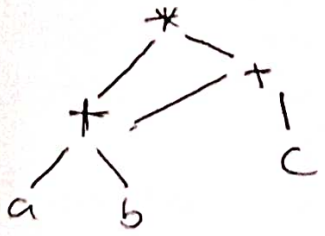
Eg: $(a+b) * (a+b+c)$

three address code :-

$T_1 = a+b$

$T_2 = T_1 + c$

$T_3 = T_1 * T_2$.



Three address code

three address code is a sequence of statements of the general form:

$$x = y \text{ op } z$$

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 logical operator on boolean valued data. No built-up arithmetic expressions are permitted, as there is only one on the right side of a statement.

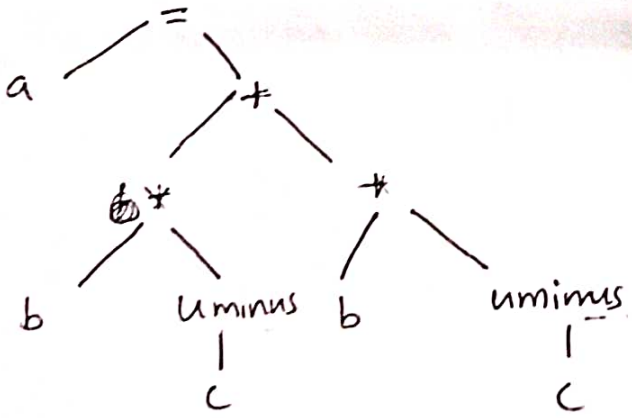
Eg: Expression $x+y*z$ might be translated into a sequence

$$t_1 = y*z$$
$$t_2 = x+t_1$$

where t_1 and t_2 are compiler generated temporary names.

Three address code is a linearized representation of a syntax tree or a DAG, in which explicit names correspond to the interior nodes of the graph. variable names can appear directly in three-address statements.

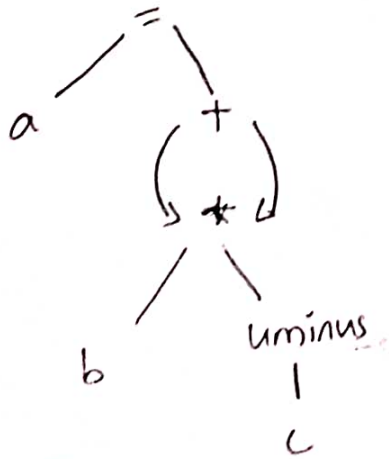
$$a = b*(\text{uminus } c) + b*(\text{uminus } c)$$



[syntax tree]

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

[three address code]



[DAG]

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

[three address code]

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operand and one for the result. In the implementation of three address code A programmer defined name is replaced by a pointer to a symbol table entry for that name.

Types of three address code:

1) Assignment statement of the form $x = y$ op z, where op is a binary arithmetic or logical operation
Assignment instructions of the form $x = $ op y, where op is a unary operation.
copy statements of the form $x = y$, value of y is assigned to x
unconditional jump goto L - three address statement with label L is the next to be executed
conditional jumps such as if x relop y goto L
param x and call p, n for procedure calls and return y, where y representing a returned value is optional.

```
param x1
param x2
 .
 .
call p,n
```

generated as part of a call of the procedure $p(x_1, x_2, \cdots, x_n)$
Indexed assignments of the form $x = y[i]$ and $x[i] = y$
Address and pointer assignments of the form $x = \&y$, $x = *y$

Implementation of three address statements

A compiler these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

1) Quadruples
2) Triples
3) Indirect triple

Quadruples

- A quadruple is a record structure with four fields which we call op, arg1, arg2, and result.
- The op field contains an internal code for the operator
- $x = y$ op z is represented by placing y in arg1, z in arg2 and x in result

statements with unary operators like $x = y$ or $x = -y$ do not use arg2 operators like param use neither arg2 nor result.
conditional and unconditional jumps put the target label in result.

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

quadruple representation

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

Triples

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- 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.

Triple representation for $a = b * -c + b * -c$.

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Indirect Triples

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples

Indirect triples representation: $a = b * -c + b * -c$.

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Single-Static Assignment (SSA)

It is an intermediate representation. It facilitates certain code optimization. All assignments are to variables with distinct name.

Three address code.

$p = a + b$

$q = p - c$

$p = q * d$

$p = e - p$

$q = p + q$

Static single assignment form

$p_1 = a + b$

$q_1 = p_1 - c$

$p_2 = q_1 * d$

$p_3 = e - p_2$

$q_2 = p_3 + q_1$

Why SSA? (advantage)

Optimization algorithms becomes simpler, if each variable has only one definition

Unrelated uses of same variable become independent.

More values become available at each program point.

Types and declarations

1) Type Expressions
2) Type equivalence
3) Declarations
4) storage layout for local names

Type Expressions

It will denotes the type of language construction. Type expressions can be divided into two:

(1) Basic type, and
(2) Type name

Basic type expression is also called as primitive type expression, such as integer, real, boolean, character.

A type name is also called as a type constructor. It can be used to denote type expression such as arrays, procedure, pointer, function.

A record is a data structure with named fields.

A type expression can be formed by applying the record type constructor to the field names and their types.

A type expression can be formed by using the type constructor → for function types. We write $S \to t$ for "function from type S to type t".

If S and t are type expressions, then their cartesian product $S \times t$ is a type expression.

type expressions may contain variables whose values are type expressions

Type Equivalence

Two expressions are structurally equivalent if there are two expressions same basic type or are formed by applying same constructor.

Structural Equivalence algorithm (sequiv):

```
if(s and t are same basic types) then return true
else if(s=array(s1,s2) and t=array(t1,t2) then return (sequiv(s1,t1) and
                                                       sequiv(s2,t2))
else if(s=s1×s2 and t=t1×t2) then return(sequiv(s1,t1) and sequiv(s2,t2))
else if(s=pointer(s1) and t=pointer(t1)) then return (sequiv(s1,t1))
else if(s=s1→s2 and t=t1→t2) then return (sequiv(s1,t1) and sequiv(s2,t2)
else return false
```

Declarations

Type and declarations using a simplified grammar that declares just one name at a time declarations with list of names can also be handled. The grammar is

~~D → T id | T~~

$D \rightarrow T\ id\ ;\ D \mid \epsilon$

$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$

$B \rightarrow int \mid float$

$C \rightarrow \epsilon \mid [num]\ C$

Non terminal D generates a sequence of declarations
Non terminal T generates basic, array, or record types
Non terminal B generates one of the basic types int and float.
Non terminal C for "component", generates strings of zero or more integers, each integer surrounded by brackets.
An array type consist of a basic type, specified by B, followed by an array component specified by non terminal C
A record type is a sequence of declarations for the fields of the record, all surrounded by curley braces

Storage layout for local names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time, we can use these amounts to assign each name a relative address, The type and relative address are saved in the symbol table entry for the name.

Data of varying length such as string, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to the data.

The width of a type is the number of storage units needed for objects of that type A basic type, such as character, integer, or float requires an integral number of bytes

For easy access, storage for aggregates such as arrays and classes is allocated one contiguous block of bytes

The translation scheme (SDT) below computes types and their widths for basic and array types. The SDT uses synthesized attributes type and width for each non terminal and two variables t and w to pass type and width information

$T \rightarrow B$ $\{t = B.type;\ w = B.width;\}$

C $\{T.type = C.type;\ T.width = C.width;\}$

$B \rightarrow int$ $\{B.type = integer;\ B.width = 4;\}$

$B \rightarrow float$ $\{B.type = float;\ B.width = 8;\}$

$C \rightarrow \epsilon$ $\{C.type = t;\ C.width = w;\}$

$C \rightarrow [num]C_1$ $\{C.type = array(num.value), C_1.type);$

$C.width = num.value \times C_1.width;\}$

Type checking

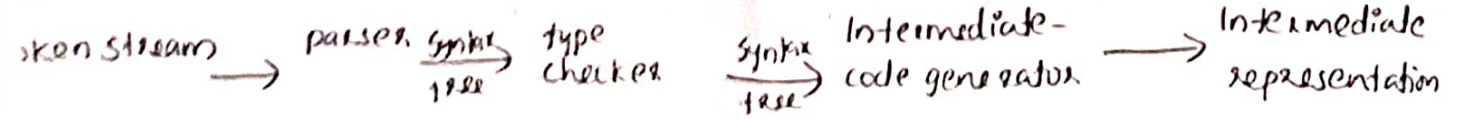
A compiler must check that the source program follows both the syntactic and semantic nventions of the source language. This checking called static checking, ensures that certain inds of programming errors will be detected and reported. Examples of static checks nclude:

1) Type checks: A compiler should report an error if an operator is applied to an incompatible operand. For eg: if an array variable and a function variable are added together

2) Flow-of-control checks: statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control.

Uniqueness checks: There are situations in which an object must be defined exactly once.

Name related checks: sometimes, the same name must appear two or more times. e compiler must check that the same name is used at both places.



[position of type-checker]

Type systems

The design of a type checker for a language is based on information about the ntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs

type expressions, type equivalence, declarations, storage layout for names]

A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system.

Static and dynamic checking of types

checking done by a compiler is a static, while if it done at run time it is dynamic

A sound type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs.

A language is strongly typed if its compiler can guarantee that the program it accepts will execute without type errors.

Error recovery

It is important for a type checker to do something reasonable when an error is discovered. At the very least, the compiler must report the nature and location of the error. It is desirable for the type checker to recover from errors, so that it can check the rest of the input

Type checking of Expressions

The synthesized attribute 'type' for E gives the type of the expression assigned by the type system for the expression generated by E. The function lookup returns the type of id.

Type checking of Statements

production	semantic rules
$S \rightarrow id = E$	S.type = if id.type = E.type then void else type_error
if E then S_1	S.type = if E.type = boolean then void else type_error
while E do S_1	S.type = if E.type = boolean then void else type_error
$S \rightarrow S_1 ; S_2$	S.type = if S_1.type = void and S_2.type = void then void else type_error

Statements don't have values, therefore a special type void can assign to them. If an error is detected within a statement, the type assigned to the statement is type_error

Rules for checking statements

The first rule checks that the left and right sides of an assignment statemen ave the same type.
The second and third rule specify that expressions in conditional and while statements must have type boolean.
Errors are propagated by the last rule, because a sequence of statement has type void only if each substatement has type void.
A mismatch of type produces the type, type_error.

Type conversion

since the representation of integers and real is different within a computer, an ifferent machine instructions are used for operations on integers and real, the mpiler may have to convert one of the operands to ensure that both operands re of the same type, when the operation takes place. The language definition pecifies what conversions are necessary. When an integer is assigned to a eal or vice versa, the conversion is to the type of the left side of the tatement.

conversions from one type to another is said to be implicit, if it is to be ne automatically by the compiler. Implicit type conversions also called coercions nversions are said to be explicit, if the programmer must write something , cause the conversion.