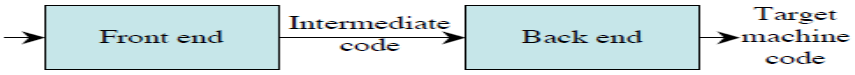
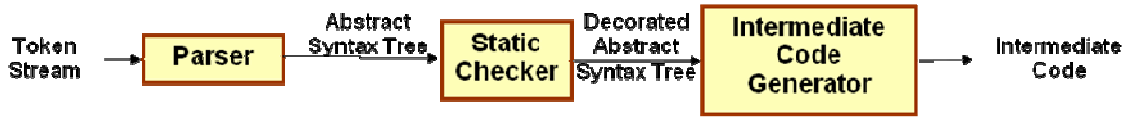
Unit 6 : Intermediate Code Generation



In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end.

Logical Stucture of a Compiler Front End



A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

Static Checking

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

- flow–of-control checks
  - Ex: Break statement within a loop construct
- Uniqueness checks
  - Labels in case statements
- Name-related checks

Intermediate Representations

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

- A clear distinction between the machine-independent and machine-dependent parts of the compiler
- Retargeting is facilitated; the implementation of language processors for new machines will require replacing only the back-end
- We could apply machine independent code optimisation techniques

Intermediate representations span the gap between the source and target languages.

- *High Level Representations*
  - closer to the source language
  - easy to generate from an input program
  - code optimizations may not be straightforward
- *Low Level Representations*
  - closer to the target machine
  - Suitable for register allocation and instruction selection

– easier for optimizations, final code generation

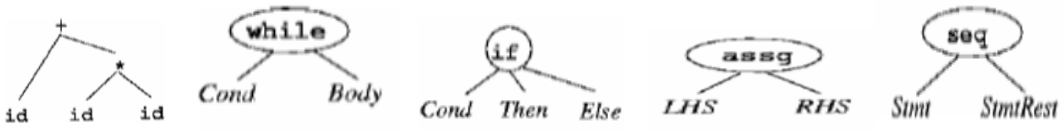
There are several options for intermediate code. They can be either

- Specific to the language being implemented
  - P-code for Pascal
  - Bytecode for Java
- Language independent:
  - 3-address code

IR can be either an actual language or a group of internal data structures that are shared by the phases of the compiler. C used as intermediate language as it is flexible, compiles into efficient machine code and its compilers are widely available.

In all cases, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions. Machine code can then be generated (access might be required to symbol tables etc).

Syntax Trees



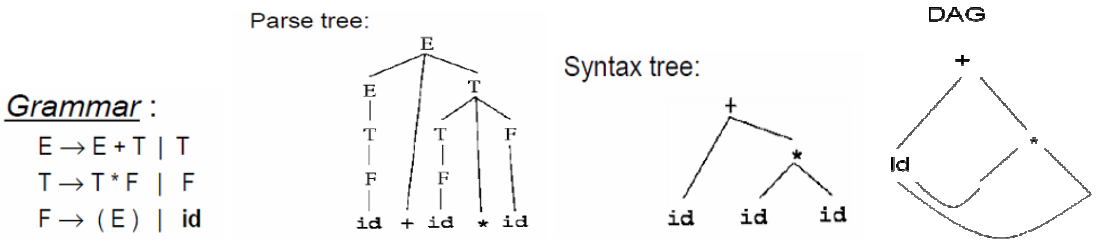
Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. Nodes represent constructs in source program and the children of a node represent meaningful components of the construct. Syntax trees are suited for static type checking.

Variants of Syntax Trees: DAG

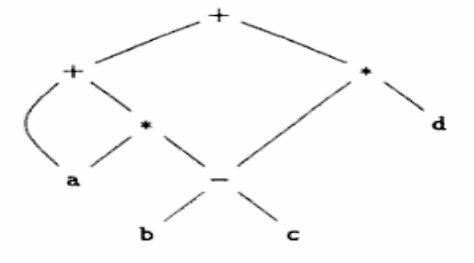
A directed acyclic graph (*DAG)* for an expression identifies the *common sub expressions* (sub expressions that occur more than once) of the expression. DAG's can be constructed by using the same techniques that construct syntax trees.

A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely. It gives clues to compiler about the generating efficient code to evaluate expressions.

Example 1: Given the grammar below, for the input string **id + id * id** , the parse tree, syntax tree and the DAG are as shown.



Example 2: DAG for the expression a + a * (b - c) + (b - c) * d is shown below.



Using the SDD to draw syntax tree or DAG for a given expression:-

- Draw the parse tree
- Perform a post order traversal of the parse tree
- Perform the semantic actions at every node during the traversal
  - Creates a syntax tree if a new node is created each time functions Leaf and Node are called
  - Constructs a DAG if before creating a new node, these functions check whether an identical node already exists. If yes, the existing node is returned.

SDD to produce Syntax trees or DAG is shown below.

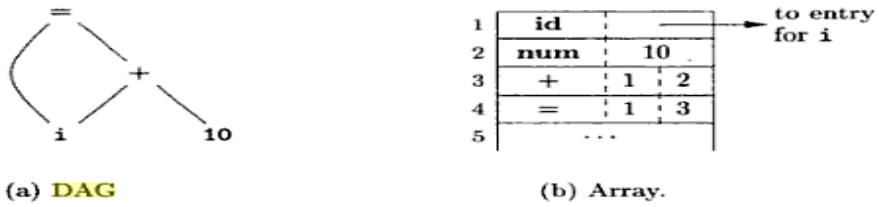
	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf(\textbf{num}, \textbf{num}.val)$

For the expression a + a * (b – c) + (b - c) * d, steps for constructing the DAG is as below.

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Value-Number Method for Constructing DAGs

Nodes of a syntax tree or DAG are stored in an array of records. The integer index of the record for a node in the array is known as the **value number** of that node.



(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

The signature of a node is a triple < op, l, r> where op is the label, l the value number of its left child, and r the value number of its right child. The value-number method for constructing the nodes of a DAG uses the signature of a node to check if a node with the same signature already exists in the array. If yes, returns the value number. Otherwise, creates a new node with the given signature.

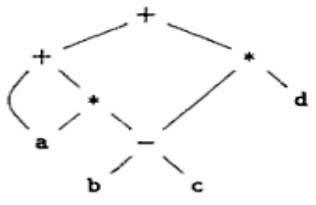
Since searching an unordered array is slow, there are many better data structures to use. Hash tables are a good choice.

Three Address Code(TAC)

TAC can range from high- to low-level, depending on the choice of operators. In general, it is a statement containing at most 3 addresses or operands.

The general form is x := y op z, where "op" is an operator, x is the result, and y and z are operands. **x**, **y**, **z** are variables, constants, or "temporaries". A three-address instruction consists of at most 3 addresses for each statement.

It is a linearized representation of a binary syntax tree. Explicit names correspond to interior nodes of the graph. E.g. for a looping statement , syntax tree represents components of the statement, whereas three-address code contains labels and jump instructions to represent the flow-of-control as in machine language.



(a) DAG

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

A TAC instruction has at most one operator on the RHS of an instruction; no built-up arithmetic expressions are permitted.

e.g. x + y * z can be translated as

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

where t_1 & t_2 are compiler–generated temporary names.

Since it unravels multi-operator arithmetic expressions and nested control-flow statements, it is useful for target code generation and optimization.

Addresses and Instructions

- TAC consists of a sequence of instructions, each instruction may have up to three addresses, prototypically t1 = t2 op t3
- Addresses may be one of:
  - A name. Each name is a symbol table index. For convenience, we write the names as the identifier.
  - A constant.
  - A compiler-generated temporary. Each time a temporary address is needed, the compiler generates another name from the stream t1, t2, t3, etc.
    - Temporary names allow for code optimization to easily move instructions
    - At target-code generation time, these names will be allocated to registers or to memory.
- TAC Instructions
  - Symbolic labels will be used by instructions that alter the flow of control. The instruction addresses of labels will be filled in later.
    L: t1 = t2 op t3
  - Assignment instructions: x = y op z
    - Includes binary arithmetic and logical operations
  - Unary assignments: x = op y
    - Includes unary arithmetic op (-) and logical op (!) and type conversion
  - Copy instructions: x = y
  - Unconditional jump: goto L
    - L is a symbolic label of an instruction
  - Conditional jumps:
    if x goto L If x is true, execute instruction L next
    ifFalse x goto L If x is false, execute instruction L next
  - Conditional jumps:
    if x relop y goto L
  - Procedure calls. For a procedure call $p(x_1, \ldots, x_n)$
    param x_1
    …
    param x_n
    call p, n
  - Function calls : y= $p(x_1, \ldots, x_n)$ y = call p,n , return y

- Indexed copy instructions: x = y[i] and x[i] = y
  - Left: sets x to the value in the location i memory units beyond y
  - Right: sets the contents of the location i memory units beyond x to y
- Address and pointer instructions:
  - x = &y sets the value of x to be the location (address) of y.
  - x = *y, presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.
  - *x = y sets the value of the object pointed to by x to the value of y.

Example: Given the statement **do i = i+1; while (a[i] < v); ,** the TAC can be written as below in two ways, using either symbolic labels or position number of instructions for labels.

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

(a) Symbolic labels.

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

(b) Position numbers.

Three Address Code Representations

Data structures for representation of TAC can be objects or records with fields for operator and operands. Representations include quadruples, triples and indirect triples.

Quadruples

- In the quadruple representation, there are four fields for each instruction: *op, arg1, arg2, result*
  - Binary ops have the obvious representation
  - Unary ops don't use arg2
  - Operators like param don't use either arg2 or result
  - Jumps put the target label into result
- The quadruples in Fig (b) implement the three-address code in (a) for the expression a = b * - c + b * - c

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

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

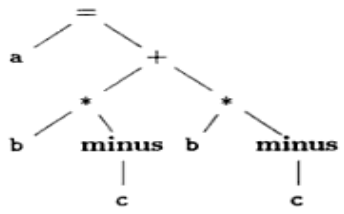
(b) Quadruples

Triples

- A triple has only three fields for each instruction: *op, arg1, arg2*
- The *result* of an operation *x op y* is referred to by its position.

- Triples are equivalent to signatures of nodes in DAG or syntax trees.
- Triples and DAGs are equivalent representations only for expressions; they are not equivalent for control flow.
- Ternary operations like ***x[i]*** = *y* requires two entries in the triple structure, similarly for x = y[i].
- Moving around an instruction during optimization is a problem

Example: Representations of a =b *– c + b * – c



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Indirect Triples

These consist of a listing of pointers to triples, rather than a listing of the triples themselves. An optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code.

- All assignments in SSA are to variables with distinct names; hence *static single-assignment.*
- *Φ*-FUNCTION

  Same variable may be defined in two different control-flow paths. For example,

  if (flag) x = -1; else x = 1;
  y = x * a;

  using Φ-function it can be written as

  if (flag) x_1 = -1; else x_2 = 1;
  $x_3 = \Phi(x_1,x_2)$;
  y = x_3 * a;

The Φ -function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment statement containing the Φ – function.

Types

A **type** typically denotes a **set of values** and a **set of operations** allowed on those values. Applications of types include type checking and translation.

Certain operations are legal for each type. For example, it doesn't make sense to add a function pointer and an integer in C. But it does make sense to add two integers. But both have the same assembly language implementation!

A language's **Type System** specifies which operations are valid for which types.

Type Checking is the process of verifying fully typed programs. Given an operation and an operand of some type, it determines whether the operation is allowed. The goal of type checking is to ensure that operations are used with the correct types. It uses logical rules to reason about the behavior of a program and enforces intended interpretation of values.

Type Inference is the process of filling in missing type information. Given the type of operands, determine the meaning of the operation and the type of the operation; or, without variable declarations, infer type from the way the variable is used.

Components of a Type System

- Built-in types
- Rules for constructing new types
- Rules for determining if two types are equivalent
- Rules for inferring the types of expressions

Type Expressions

Types have structure, represented using type expressions. Type expressions can be either basic type or formed by applying type constructors.

Example: an array type int[2][3] has the type expression is array(2, array(3, integer)) where array is the operator and takes 2 parameters, a number and a type.

Definition of Type Expressions

- A basic type is a type expression. Typical basic types for a language include *boolean, char, integer, float,* and *void.*
- A type name is a type expression.
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- 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*→ *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. Products can be used to represent a list or tuple of types (e.g., for function parameters).
- Type expressions may contain variables whose values are type expressions.

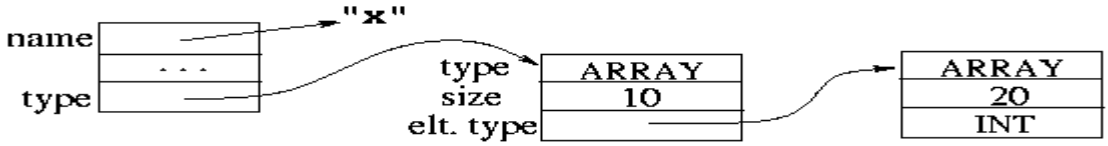
Representing Type Expressions

- Construct a DAG for type expression adapting the value-number method.
- Interior nodes represent type constructors.
- Leaves represent basic types, type names, and type variables.

***Type graphs*:** are a graph-structured representation of type expressions:

- Basic types are given predefined "internal values";
- Named types can be represented via pointers into a hash table.
- A composite type expression $f\ (T_1,\ldots,T_n)$ is represented as a node identifying the constructor *f* and with pointers to the nodes for $T_1, \ldots, T_n$.

E.g.: type graph for the type expression of **int x[10][20]** is shown below**.**



Type Equivalence

The two types of type equivalence are structural equivalence and name equivalence.

Structural equivalence: When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:

1. They are the same basic type
2. They are formed by applying the same constructor to structurally equivalent types.
3. One is a type name that denotes the other

Name equivalence: If type names are treated as standing for themselves, the first two conditions above lead to name equivalence of type expressions.

Example 1: in the Pascal fragment

```
type p = ↑node;
     q = ↑node;
var x : p;
    y : q;
```

x and y are structurally equivalent, but not name-equivalent**.**

Example 2: Given the declarations

Type t1 = Array [1..10] of integer;
Type t2 = Array [1..10] of integer;

– are they name equivalent? No because they have different type names.

Example 3: Given the declarations

type vector = array [1..10] of real
type weight = array [1..10] of real
var x, y: vector; z: weight

Name Equivalence: When they have the same name.

– x, y have the same type; z has a different type.

Structural Equivalence: When they have the same structure.

– x, y, z have the same type.

Translation Applications of Types

From the type of a name, we can determine the storage needed for the name at run time, calculate the address denoted by an array reference, insert explicit type conversions, and choose the right version of an arithmetic operator.

Types and storage layout for names are declared within a procedure or class. Actual storage is allocated at run time. Type of a name can used to determine the amount of storage needed for the name at run time. At compile time these amounts are used to assign each name a relative address. Local declarations are examined and relative addresses are laid out with respect to an offset from the start of a data area. The type and relative address are saved in the symbol-table entry for the name. Pointer is assigned for data for varying length like strings and dynamic arrays.

Declarations

The grammar

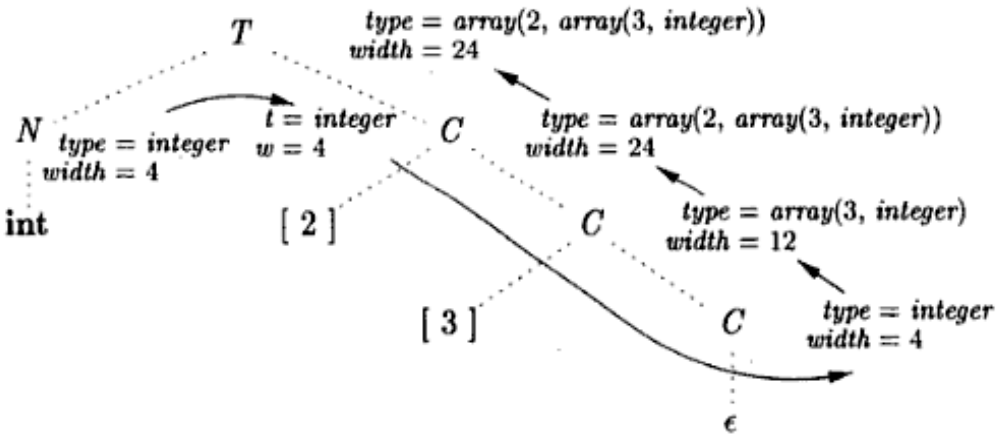
$D \rightarrow T$ id ; D | **ε**
$T \rightarrow B\ C$ | record ' { ' D '}'
$B \rightarrow$ int | float
$C \rightarrow$ **ε** | [n u m] C

declares just one name at a time; D- generates a sequence of declarations, T – generates basic, array or record types, C – for 'component', generates strings of zero or more integers , each integer surrounded by brackets.

SDT for computing types and their widths is as below.

$T \rightarrow B\ C$	{ $t = B.type$; $w = B.width$; }
$B \rightarrow$ **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	{ $array(\mathbf{num}.value, C_1.type)$; $C.width = \mathbf{num}.value \times C_1.width$; }

Syntax-directed translation of array types is shown in the dependency graph below for the type declaration **int [2][3].**



Sequences of Declarations

The translation scheme below deals with a sequence of declarations of the form *T* **id,** where *T* generates a type. Before the first declaration is considered, *offset* is set to 0. As each new name *x* is seen, *x* is entered into the symbol table with its relative address set to the current value of *offset,* which is then incremented by the width of the type of *x.*

$P \rightarrow$ { *offset* = 0; }
D

$D \rightarrow T$ **id** ; { *top.put*(**id**.*lexeme*, *T.type*, *offset*);
offset = *offset* + *T.width*; }
D_1

$D \rightarrow \epsilon$

The semantic action within the production *D → T id ; D1* creates a symbol table entry by executing *top.put(id.lexeme, T.type, offset).* Here *top* denotes the current symbol table. The method *top.put* creates a symbol-table entry for *id.lexeme*, with type *T.type* and relative address *offset* in its data area.

Fields in Records and Classes

Since records can essentially have a bunch of declarations inside, we only need add T → RECORD { D } to get the syntax right. For the semantics we need to push the environment and offset onto stacks since the namespace inside a record is distinct from that on the outside. The width of the record itself is the final value of (the inner) offset.

```
T → record {        { Env.push(top);  top = new Env();
                      Stack.push(offset); offset = 0; }
D }                 { T.type = record(top); T.width = offset;
                      top = Env.pop(); offset = Stack.pop(); }
```

Translation of Expressions

The goal is to generate 3-address code for expressions. Assume there is a function gen() that given the pieces needed does the proper formatting so gen(x = y + z) will output the

corresponding 3-address code. gen() is often called with addresses rather than lexemes like x. The constructor Temp() produces a new address in whatever format gen needs.

Operations within Expressions

The syntax-directed definition below builds up the three-address code for an assignment statement *S* using attribute *code* for *S* and attributes *addr* and *code* for an expression *E.* Attributes *S.code* and *E.code* denote the three-address code for *S* and *E,* respectively. Attribute *E.addr* denotes the address that will hold the value of *E.*

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \ \|\| E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Incremental Translation

The method in the previous section generates long strings and we walk the tree. By using SDT instead of using SDD, you can output parts of the string as each node is processed.

$S \rightarrow \mathbf{id} = E\ ;$ { $gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ }

$E \rightarrow E_1 + E_2$ { $E.addr = \mathbf{new}\ Temp();$
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }

$\mid\ - E_1$ { $E.addr = \mathbf{new}\ Temp();$
$gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);$ }

$\mid\ (\ E_1\ )$ { $E.addr = E_1.addr;$ }

$\mid\ \mathbf{id}$ { $E.addr = top.get(\mathbf{id}.lexeme);$ }

Addressing Array Elements

The idea is that you associate the base address with the array name. That is, the offset stored in the identifier table is the address of the first element of the array. The indices and the array bounds are used to compute the amount, often called the offset by which the address of the referenced element differs from the base address.

One Dimensional Arrays

For one dimensional arrays, this is especially easy: The address increment is the width of each element times the index (assuming indexes start at 0). So the **address of A[i]** is the base address of A plus i times the width of each element of A.

The width of each element is the width of what we have called the base type. ***base*** is the relative address of the first location of array A, width is the width of each array element, low is the index of the first array element. Then,

location of A[i] ➔ base+(i-low)*width

Two Dimensional Arrays

Let us assume *row major ordering*. That is, the first element stored is A[0,0], then A[0,1], ... A[0,k-1], then A[1,0], ... Modern languages use row major ordering.

With the alternative *column major ordering*, after A[0,0] comes A[1,0], A[2,0], ...

For two dimensional arrays the address of A[i,j] is the sum of three terms

1. The base address of A.
2. The distance from A to the start of row i. This is i times the width of a row, which is i times the number of elements in a row times the width of an element. The number of elements in a *row* is the *column* array bound.
3. The distance from the start of row i to element A[i,j]. This is j times the width of an element.

Grammar for Expressions With Array References

$S \rightarrow$ **id** $= E;$ | $L = E;$

$E \rightarrow E_1 + E_2$ | **id** | L

$L \rightarrow$ **id** $[\ E\ ]$ | $L_1\ [\ E\ ]$

Generates expressions of the form a = b

a = b + c , a = b[i] , a[j]= c , a[j] = b[k] , a = b[c[i]], a[i]= b[i][j][k]

Nonterminal L has 3 synthesized attributes

- *L.addr* – temporary that is used while computing the offset for the array reference by summing the terms $i_j\, x\, w_j$
- *L.array* is a pointer to the symbol table entry for the array name.
  - *L.array.base* is used to determine the actual l-value of an array reference after all the index expressions are analyzed.
- *L.type* - type of the subarray generated by L .
  - For an array type *t* , *t.elem* gives the element type

The SDT for translating array references is as below.

$S \rightarrow \mathbf{id} = E\ ;$ { $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$; }

| $L = E\ ;$ { $gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr)$; }

$E \rightarrow E_1 + E_2$ { $E.addr = \mathbf{new}\ Temp()$;
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }

| $\mathbf{id}$ { $E.addr = top.get(\mathbf{id}.lexeme)$; }

| L { $E.addr = \mathbf{new}\ Temp()$;
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']')$; }

$L \rightarrow \mathbf{id}\ [\ E\ ]$ { $L.array = top.get(\mathbf{id}.lexeme)$;
$L.type = L.array.type.elem$;
$L.addr = \mathbf{new}\ Temp()$;
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width)$; }

| $L_1\ [\ E\ ]$ { $L.array = L_1.array$;
$L.type = L_1.type.elem$;
$t = \mathbf{new}\ Temp()$;
$L.addr = \mathbf{new}\ Temp()$;
$gen(t\ '='\ E.addr\ '*'\ L.type.width)$; }
$gen(L.addr\ '='\ L_1.addr\ '+'\ t)$; }

An annotated parse tree for the expression **c** + a [i] [j] is shown in figure below. The expression is translated into the sequence of three-address instructions as shown.

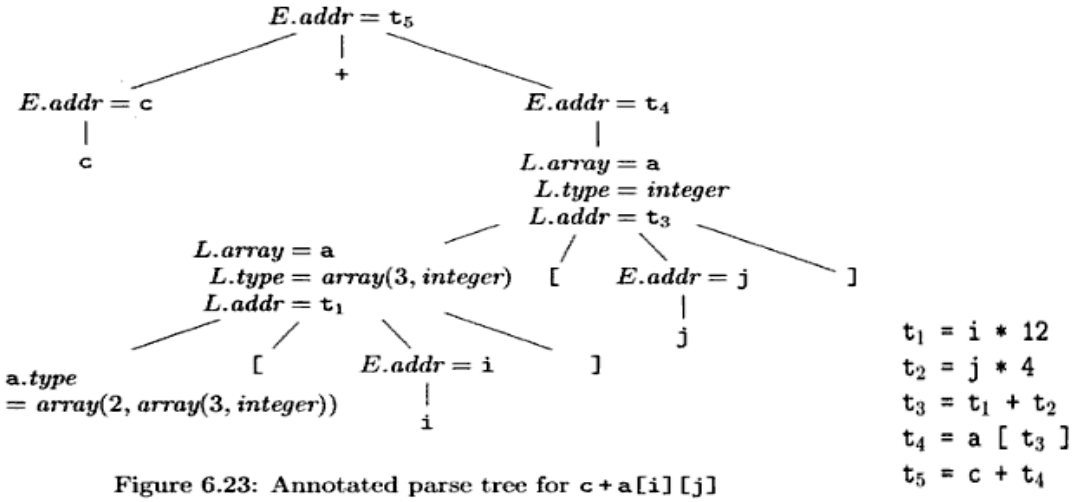


Figure 6.23: Annotated parse tree for c + a[i][j]

Type Checking

Type Checking includes several aspects.

1. The language comes with a *type system*, i.e., a set of rules saying what types can appear where.
2. The compiler assigns a type expression to parts of the source program.
3. The compiler checks that the type usage in the program conforms to the type system for the language.

A *sound* type system guarantees that all checks can be performed prior to execution. This does not mean that a given compiler will make all the necessary checks.

An implementation is *strongly typed* if compiled programs are guaranteed to run without type errors.

Rules for Type Checking

There are two forms of type checking.

1. *type synthesis* where the types of parts are used to infer the type of the whole. For example, integer+real=real. Synthesis computes the type of an expression using its sub-expressions and requires names to be declared before they are used.

   Typical rule for type synthesis:

   if f has type s → t and x has type s,
   then expression f(x) has type t

2. *Type inference* where the type of a construct is determined from usage. This permits languages like ML to check types even though names need not be declared.

   *T*ypical rule for type synthesis

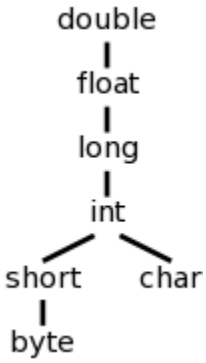
   if f(x) is an expression,
   then for some α and β, f has type has type α → β and x has type β

Type Conversions

A very strict type system would do no automatic conversion. Instead it would offer functions for the programmer to explicitly convert between selected types. Then either the program has compatible types or is in error.

However, we will consider a more liberal approach in which the language permits certain implicit conversions that the compiler is to supply. This is called *type coercion*. Explicit conversions supplied by the programmer are called *casts*.

Consider the more general case where there are multiple types some of which have coercions (often called widening). For example in C/Java, int can be widened to long, which in turn can be widened to float as shown in the figure to the right.



Semantic Action For Checking E →E1 + E2

Two functions, max and widen are used.

1. max(t1,t2)
   - takes two types t1 and t2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy.
   - declares an error if either t1 or t2 is not in the hierarchy; e.g., if either type is an array or a pointer type.
2. widen(a, t, w)
   - generates type conversions if needed to widen an address a of type t into a value of type w.
   - It returns a itself if t and w are the same type.
   - Otherwise, it generates an instruction to do the conversion and place the result in a temporary t, which is returned as the result.

Pseudocode for Function *widen*

```
Addr  widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Introducing Type Conversions into Expression Evaluation

E → E1 + E2 { E.type = max(E1.type, E2.type);
a1= widen(E1. addr, E1. type, E. type);
a2 = widen(E2. addr, E2 .type, E. type);
E. Addr = new Temp ();
gen(E.addr '=' a1 '+' a2); }

Overloading of Functions and Operators

An overloaded symbol has different meanings depending on its context. Overloading is resolved when a unique meaning is determined for each occurrence of a name. Overloading in Java can be resolved by looking only at the arguments of a function. The + operator in Java denotes either string concatenation or addition, depending on the types of its operands.

Type-Synthesis Rule for Overloaded Functions

If f can have type $s_i \rightarrow t_i$, *for* $1 \leq i \leq n$, *where* $s_i \neq s_j$ *for* $i \neq j$

and *x has type* s_k, *for some* $1 \leq k \leq n$

then expression *f(x) has type* t_k

The value-number method can be applied to type expressions to resolve overloading based on argument types, efficiently. Since the signature for a function consists of the function name and the types of its arguments, overloading can be resolved based on signatures. However, it is not always possible to resolve overloading by looking only at the arguments of a function.

Type Inference and Polymorphic Functions

The term "polymorphic" refers to any code fragment that can be executed with arguments of different types. In *parametric polymorphism,* the polymorphism is characterized by parameters or type variables.

Example: ML program for the length of a list is defined as

fun *length(x)* =if *null(x)* then *0* else *length(tl(x)) +1;*

Here function *length* determines the length or number of elements of a list x, predefined function *null* tests whether a list is empty, and function *tl* (short for "tail") returns the remainder of a list after the first element is removed.

All elements of a list must have the same type, but *length* can be applied to lists whose elements are of any one type.

Example:

length(["sun", "mon", "tue"]) + length([10,9,8,7])

Type of length :

$\forall$ *α.list (α)⟶ integer*

A type expression with a $\forall$ symbol in it is referred to informally as a "polymorphic type." $\forall$ symbol is the *universal quantifier.* A type variable to which it is applied is said to be bound by it. Bound variables can be renamed at will, provided all occurrences of the variable are renamed. Each time a polymorphic function is applied, its bound type variables can denote a different type.

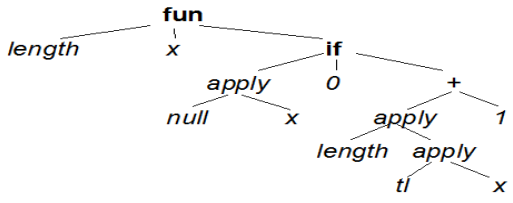
During type checking, at each use of a polymorphic type we replace the bound variables by fresh variables and remove the universal quantifiers.

We can infer a type for length using the type inference rule

if *f(x) is an expression,*

then for some α and β, f *has type* α →β *and x has type* α

Abstract syntax tree for the length function definition



Substitution

A *substitution* is a mapping from type variables to type expressions. The result of applying the substitution S to the variables in type expression t is denoted S(t). All occurrences of each type variable α in t is replaced by S(α). S(t) is called an instance of t.

Example: list(integer) is an instance of list(α).

Equivalence of Types

Two type expressions *t1* and *t2 unify* if there exists some substitution *S* such that *S(t1)* = *S(t2).*

Suppose *E1 of type s →s' is applied to E2 of* type *t.*

Unification: the problem of determining whether two expressions *s and t can be made identical by substituting expressions for the variables in s and t.*

Testing equality of expressions is a special case of unification

If *s* and *t* have constants but no variables, then s and t unify if and only if they are identical.

Most general unifier is a substitution that imposes the fewest constraints on the variables in the expressions.

Type Inference for Polymorphic Functions

INPUT: A program consisting of a sequence of function definitions followed by an expression to be evaluated. An expression is made up of function applications and names, where names can have predefined polymorphic types.

OUTPUT: Inferred types for the names in the program.

METHOD: The type of a function *f(x1,x2)* with two parameters can be represented by a type expression s_1 x $s_2 \rightarrow t$, where s_1 and s_2 are the types of *x1 and x2,* respectively, and *t* is the type of the result *f(x1,x2).*

An expression f (a, b) can be checked by matching the type of *a* with s_1 and the type of *b* with s_2. We first check the function definitions and the expression in the input sequence and then use the inferred type of a function if it is subsequently used in an expression.

For a function definition fun id1(id2) = *E*

- create fresh type variables *α and β.*
- Associate the type *α→β* with the function i*d1* and the type *α* with the parameter id2.
- Infer a type for expression *E.* Suppose *α* denotes type *s* and *β* denotes type *t* after type inference for *E.* The inferred type of function id1 is $s \rightarrow t$.
- Bind any type variables that remain unconstrained in $s \rightarrow t$ by universal quantifiers.

For a function application E1(E2)

- infer types for E1 and E2. Since E1 is used as a function, its type must have the form s → s'. Let t be the inferred type of E1.
- Unify s and t.
- If unification fails, the expression has a type error. Otherwise, the inferred type of E1(E2) is s'.

For each occurrence of a polymorphic function

- replace the bound variables in its type by distinct fresh variables and remove the quantifiers.
- The resulting type expression is the inferred type of this occurrence.

For a name that is encountered for the first time

- introduce a fresh variable for its type.

Using the above rules , a type for function **length** can be inferred as

∀ ***α.list (α)→ integer*** . The steps are as in the table below.

LINE	EXPRESSION : TYPE	UNIFY
1)	length: $\beta \rightarrow \gamma$	
2)	x : β	
3)	if : boolean × α_i × α_i → α_i	
4)	null : list (α_n) → boolean	
5)	null(x) : boolean	list (α_n) = β
6)	0: integer	α_i = integer
7)	+ : integer × integer → integer	
8)	tl : list(α_t) → list(α_t)	
9)	tl(x) : list(α_t)	list(α_t) = list(α_n)
10)	length(tl(x)) : γ	γ = integer

Graph-Theoretic Formulation Of Unification

In this scheme, types are represented by graphs. Type variables are represented by leaves and type constructors are represented by interior nodes.

Nodes are grouped into equivalence classes; if two nodes are in the same equivalence class, then the type expressions they represent must unify.

Thus, all interior nodes in the same class must be for the same type constructor, and their corresponding children must be equivalent.

ALGORITHM: Unification of a pair of nodes in a type graph.

INPUT: A graph representing a type and a pair of nodes *m and n to be unified.*

OUTPUT: Boolean value true if the expressions represented by the nodes *m* and *n unify; false, otherwise.*

METHOD: A node is implemented by a record with fields for a binary operator and pointers to the left and right children. The sets of equivalent nodes are maintained using the *set* field. One node in each equivalence class is chosen to be the unique representative of the equivalence class by making its *set* field contains a null pointer. The *set* fields of the remaining nodes in the equivalence class will point (possibly indirectly through other nodes in the set) to the representative. Initially, each node n is in an equivalence class by itself, with n as its own representative node.

Unification algorithm uses two operations *find(n)* and *union(m,n)* on nodes.

- *find(n)* returns the representative node of the equivalence class currently containing node n.
- *union(m, n)* merges the equivalence classes containing nodes m and n.
  - The *union operation on* sets is implemented by simply changing the *set* field of the representative of one equivalence class so that it points to the representative of the other.

To find the equivalence class that a node belongs to, we follow the *set* pointers of nodes until the representative (the node with a *null* pointer in the set field) is reached

Unification Algorithm

boolean unify(Node m, Node n) {

s = find(m); t = find(n);

if (*s — t*) return *true;*

else if (nodes *s and t represent the same basic type*) return *true;*

else if (s is an op-node with children *s1* and *s2* and

t is an op-node with children *t1* and *t2) {*

union(s, t);

return *unify(s1,t1)* and *unify(s2,t2);*

}

else if *s* or *t* represents a variable {

union(s,t);

return *true;*

}

else return *false;*

}

Control Flow

Control flow includes the study of Boolean expressions, which have two roles.

1. They can be computed and treated similar to integers or real. Once can declare Boolean variables, there are boolean constants and boolean operators. There are also relational operators that produce Boolean values from arithmetic operands.
2. They are used in certain statements that alter the normal flow of control.

Boolean Expressions

One question that comes up with Boolean expressions is whether both operands need be evaluated. If we need to evaluate A OR B and find that A is true, must we evaluate B? For example, consider evaluating A=0 OR 3/A < 1.2 when A is zero.

This comes up some times in arithmetic as well. Consider A*F(x). If the compiler knows that for this run A is zero must it evaluate F(x)? Functions can have side effects, do it could be a potential problem .

Short-Circuit Code

This is also called jumping code. Here the Boolean operators AND, OR, and NOT do not appear in the generated instruction stream. Instead we just generate jumps to either the true branch or the false branch.

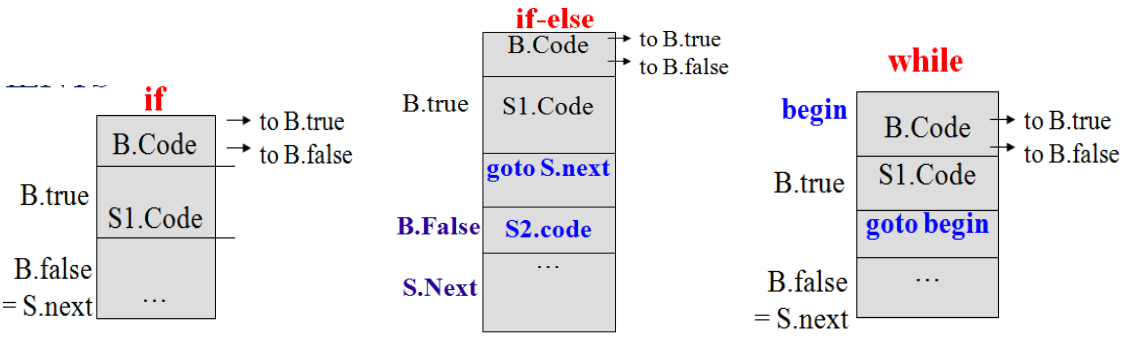
Example:

if (x **< 100** II x **> 200** && x != y) x = **0;**
if x **< 100 goto L2**
if False x **> 200 goto L1**
if False x != y **goto L1**
L2: x = **0**
L1:

Flow-of-Control Statements

The grammar is (S for statement, B for boolean expression)

$S \rightarrow$ if (B) S_1 $S \rightarrow$ if (B) S_1 else S_2 $S \rightarrow$ while (B) S_1



If and while SDDs

Production	Semantic Rules	Kind
$P \rightarrow S$	S.next = newlabel() P.code = S.code || label(S.next)	Inherited Synthesized
$S \rightarrow$ if (B) S_1	B.true = newlabel() B.false = S.next S_1.next = S.next S.code = B.code || label(B.true) || S_1.code	Inherited Inherited Inherited Synthesized
$S \rightarrow$ if (B) S_1 else S_2	B.true = newlabel() B.false = newlabel() S_1.next = S.next S_2.next = S.next S.code = B.code || label(B.true) || S_1.code || gen(goto S.next) || label(B.false) || S_2.code	Inherited Inherited Inherited Inherited Synthesized
$S \rightarrow$ while (B) S_1	begin = newlabel() B.true = newlabel() B.false = S.next S_1.next = begin S.code = label(begin) || B.code || label(B.true) || S_1.code || gen(goto begin)	Synthesized Inherited Inherited Inherited Synthesized
$S \rightarrow S_1 S_2$	S_1.next = newlabel() S_2.next = S.next S.code = S_1.code || label(S_1.next) || S_2.code	Inherited Inherited Synthesized

Control-Flow Translation of Boolean Expressions

SDD for Boolean Expressions is shown below.

Production	Semantic Rules	Kind
$B \rightarrow B_1 \,\|\|\, B_2$	B_1.true = B.true B_1.false = newlabel() B_2.true = B.true	Inherited Inherited Inherited

	B_2.false = B.false	Inherited
	B.code = B_1.code || label(B1.false) || B_2.code	Synthesized
B → B_1 && B_2	B_1.true = newlabel()	inherited
	B_1.false = B.false	inherited
	B_2.true = B.true	inherited
	B_2.false = B.false	inherited
	B.code = B_1.code || label(B1.true) || B_2.code	Synthesized
B → ! B_1	B_1.true = B.false	Inherited
	B_1.false = B.true	Inherited
	B.code = B_1.code	Synthesized
B → E_1 relop E_2	B.code = E_1.code || E_2.code || gen(if E_1.addr relop.lexeme E_2.addr goto B.true) || gen(goto B.false)	Synthesized
B → true	B.code = gen(goto B.true)	Synthesized
B → false	B.code = gen(goto B.false)	Synthesized
B → ID	B.code = gen(if get(ID.lexeme) goto B.true) || gen(goto B.false)	Synthesized

The translation of **if (x < 5 || x > 10 && x == y) x = 3 ;**

if x < 5 goto L_2
goto L_3
L_3: if x > 10 goto L_4
goto L_1
L_4: if x == y goto L_2
goto L_1
L_2: x = 3

Note that there are three extra gotos. One is a goto the next statement. Two others could be eliminated by using ifFalse.

Avoiding Redundant Gotos

x > 200 translates into the code fragment:

```
        if x > 200 goto L4
        goto L1
L4 : •••
```

Instead, consider the instruction:

```
        if Falsex > 200 goto L1
L4 : •••
```

ifFalse instruction takes advantage of the natural flow from one instruction to the next in sequence, so control simply "falls through" to label L4 if x > 200 is false, thereby avoiding a jump.

Rewriting Using *fall* Label

S →if (B) S1	B.True = fall B.False = S1.next = S.next S.Code = B.code || S1.code

Semantic rules for B → E1 rel E2

test = E1.addr **rel**.*op E2.addr*

s = **if** *B.true ≠ fall and B.false ≠ fall* **then**
gen('if' *test* 'goto' *B.true)* || *gen(*'goto' *B.false)*
else if *B.true ≠ fall* **then** *gen(*'if' *test* 'goto' *B.true)*
else if *B.false ≠ fall* **then** *gen(*'ifFalse' *test* 'goto' *B.false)*
else ''

B.code = E1.code || E2.code || s

Semantic rules for B → B1 || B2

B1.true = **if** *B.true ≠ fall* **then** *B.true* **else** *newlabel()*

B1. false = fall

B2.true = B.true

B2. false = B.false

B.code = **if** *B.true ≠ fall* **then** *B1.code || B2.code*
else *B1.code || B2.code || label(B1.true)*

Using the above rules **if (x < 100 || x > 200 && x != y) x = 0;** translates to

```
        if x < 100 goto L2
        ifFalse x > 200 goto L2
        ifFalse x != y goto L1
L2 : x = 0
L1 :
```

Boolean Values and Jumping Code

First build a syntax tree for expressions, using either of the following approaches:

1. ***Use two passes:*** Construct a complete syntax tree for the input, and then walk the tree in depth-first order, computing the translations specified by the semantic rules.

2. ***Use one pass for statements, but two passes for expressions:*** translate E in **while (E) S1** before S1 is examined.

The following grammar has a single nonterminal *E for expressions:*

S →id = E ; | if (E) S | while (E)S | S S

E → E || E | E&&E | E rel E | E + E | (E) | id | true | false

E.n represents syntax tree node for E. Two methods jump and rvalue are used in evaluating expressions.

jump – method to generate jumping code at an expression node

- When *E appears in S → while (E) S1* , call *E.n.jump(t, f),* where *t* is a new label for the first instruction of *S1.code* and *f* is the label *S.next*.

rvalue – method to generate code to compute the value of the node into a temporary.

- When *E* appears in ***S → id = E;*** , method *rvalue* is called at node *E.n*.
- 1. E has form of E → *E1 + E2* , call *E.n.rvalue()*
- *2.* E has form of E → *E1 && E2,* generate jumping code for E and then assign true or false to a new temporary t at the true and false exits, respectively, from the jumping code.

Backpatching

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example, the translation of the boolean expression *B* in if (*B*) *S* contains a jump, for when *B* is false, to the instruction following the code for *S*. In a one-pass translation, *B* must be translated before *S* is examined. Labels can be passed as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses.

In ***backpatching,*** lists of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified. Each such jump

is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

One-Pass Code Generation Using Backpatching

Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass.

Synthesized attributes *truelist* and *falselist* of nonterminal *B* are used to manage labels in jumping code for boolean expressions. *B. truelist* will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if *B* is true. *B.falselist* likewise is the list of instructions that eventually get the label to which control goes when *B* is false. As code is generated for *B,* jumps to the true and false exits are left incomplete, with the label field unfilled. These incomplete jumps are placed on lists pointed to by *B.truelist* and *B.falselist,* as appropriate.

A statement *S* has a synthesized attribute *S.nextlist,* denoting a list of jumps to the instruction immediately following the code for *S.*

Instructions are generated into an instruction array, and labels will be indices into this array.

To manipulate lists of jumps, we use three functions: *makelist(i), merge(p1,p2), backpatch(p,i)*

- *makelist(i)* creates a new list containing only *i,* an index into the array of instructions; *makelist* returns a pointer to the newly created list.
- *merge(p1,p2)* concatenates the lists pointed to by *p1* and *p2,* and returns a pointer to the concatenated list.
- *backpatch(p,i)* inserts *i* as the target label for each of the instructions on the list pointed to by *p.*

Backpatching for Boolean Expressions

$B \rightarrow B1 \mid\mid M\ B2 \mid B1\ \&\&\ M\ B2 \mid !\ B \mid (\ B\ ) \mid E_1$ rel $E_2 \mid$ true $\mid$ false

$M \rightarrow \varepsilon$

- A marker nonterminal *M* in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated.

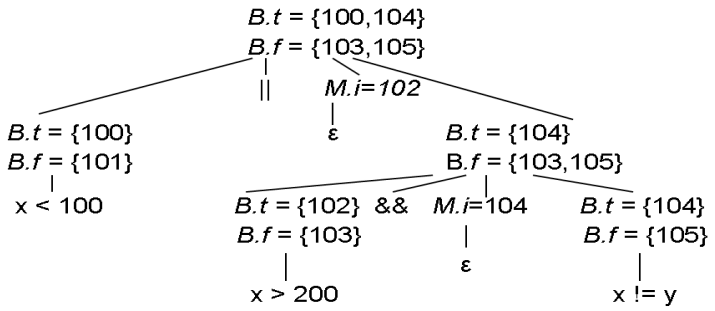
Translation Scheme for Boolean Expressions

$B \rightarrow B1 \mid\mid M\ B2$ { *backpatch(B1.falselist, M.instr);*
B.truelist = merge(B1. truelist, B2.truelist);
B.falselist = B2.falselist; }

$B \rightarrow B1\ \&\&\ M\ B2$ { *backpatch(B1.truelist, M.instr);*
B.truelist = B2.truelist;
B.falselist = merge(B1.falselist,B2.falselist); }

$B \rightarrow ! \ B1$	*{ B.truelist = B1.falselist;* *B.falselist = Bi.truelist; }*
$B \rightarrow (\ B1 \ )$	*{ B.truelist = B1.truelist;* *B.falselist = B1.falselist; }*
$B \rightarrow E1$ **rel** $E2$	*{ B.truelist = makelist(nextinstr);* *B.falselist = makelist(nextinstr* + 1); *emit(*'**if**' *E1.addr* **rel.***op E2.addr* 'goto _'); emit('goto _'); }
$B \rightarrow$ **true**	{ *B.truelist = makelist(nextinstr); emit('goto _'); }*
$B \rightarrow$ **false**	{ *B.falselist = makelist(nextinstr); emit('goto _'); }*
$M \rightarrow \varepsilon$	{ *M.instr = nextinstr, }*

Applying the above SDT to the statement **if (x < 100 II x > 200 && x != y) x = 0;** gives a annotated parse tree as below**.**



Backpatching for Flow-Of-Control Statements

The grammar is given by the following productions :

$S \rightarrow$ **if** *(B) S* | **if** *(B) S* **else** *S* | **while** *(B) S*
| *{ L }* | *A ;*
$L \rightarrow L \ S \ | \ S$

Where S – statement
L – list of statements
A- assignment statement
B – boolean expression

Translation of flow-of-control Statements Using Backpatching

S →if (B) M S1

{ backpatch{B.truelist, M.instr);
S.nextlist = merge(B.falselist, S1.nextlist); }

S → if (B) M1 S1 N else M2 S2

{ backpatch(B .truelist, M1.instr);
backpatch(B .falselist, M2.instr);
temp = merge(S1.nextlist, N.nextlist);
S.nextlist = merge(temp, S2.nextlist); }

S → while M1 (B) M2 S1

{ backpatch(S1.nextlist, M1.instr);
backpatch(B. truelist, M2.instr);
S.nextlist = B. falselist;
emit('goto' M1.instr); }

S → { L } *{ S.nextlist = L.nextlist; }*

S →A ; *{ S.nextlist =* ***null; }***
M →ε *{ M.instr = nextinstr; }*
S → { L }

{ N.nextlist = makelist(nextinstr);
emit('goto _'); }

L → L1 M S

{ backpatch(L1.nextlist, M.instr)
L. Nextlist = S. nextlist; }

L → S *{ L. nextlist — S. nextlist; }*

Switch Statements
Consider a mulit-way branching statement with C-like syntax:

```
switch (E) {
        case V [1]: S[1]
                • . . .
        case V [k]: S[k]
        default: S[d]
}
```

Translation sequence:
• Evaluate the expression.
• Find which value in the list matches the value of the expression, match default only if there is no match.
• Execute the statement associated with the matched value. The matched value can be found by Sequential test, Look-up table or Hash table.

Implementation of case statements: Use a table and a loop to find the address to jump.

When there are more than 10 entries, use a hash table to find the correct table entry.

Back-patching: We generate a series of branching statements with the targets of the jumps temporarily left unspecified. Use a to-be-determined label table, each entry of which contains a list of places that need to be back-patched. The same table can also be used to implement labels and goto's.

There are two possible ways to translate switch statements.
Scheme 1:

```
code to evaluate E into t
  goto test
L[1]: code for S[1]
  goto next
...
L[k]: code for S[k]
  goto next
L[d]: code for S[d]
  goto next
test:
if t = V[1] goto L[1]
...
if t = V[k] goto L[k]
goto L[d]
next:
...
```

Scheme 2:

```
code to evaluate E into t
  if t <> V[1] goto L[1]
  code for S[1]
  goto next
L[1]: if t <> V[2] goto L[2]
  code for S[2]
  goto next
  ...
L[k-1]: if t <> V[k] goto
                        L[k]
        code for S[k]
        goto next
L[k]: code for S[d]
next:
```

The first scheme is simpler to implement as it creates all the necessary labels first and then goes for test and branching.

Case three-address-code instructions used to translate a switch statement are as below.

```
case t V1 L1
```

case t *V2 L2*
...
case t *Vn-1 Ln-1*
case t t Ln
label next

Intermediate Code for Procedures

Assume that parameters are passed by value. Suppose that a is an array of integers, and that f is a function from integers to integers. Then, the assignment n = f (a [i]) ; might translate into the following three-address code:

1. **t 1 = i * 4**
2. **t 2 = a [t 1]**
3. **param t2**
4. **t3 = call f, 1**
5. **n = t 3**

The grammar below adds functions to the source language

D → define T id (F) { S }

F → ε | T id , F

S → return E ;

E → id (A)

A → ε | E , A