Intermediate Code Generation

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

Walchand Institute of Technology, Solapur
(www.witsolapur.org)

Intermediate Code Generation

- Introduction
- Intermediate languages
- Declarations
- Assignment statements
- Boolean expressions
- Case statements
- Back patching
- Procedure calls

Introduction

- The front end translates a source program into an intermediate representation from which the back end generates target code.
- 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.

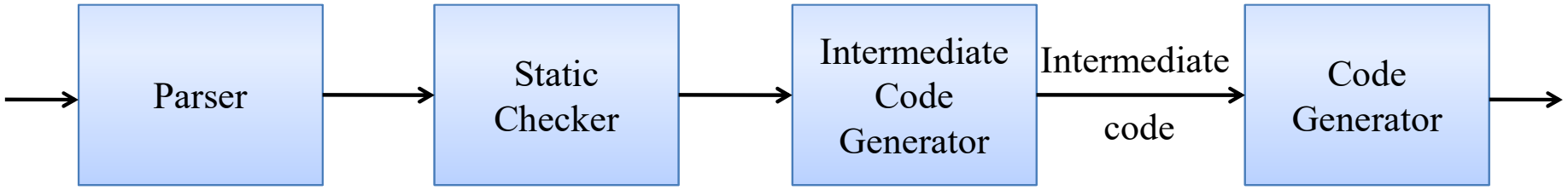


Fig: Position of intermediate code generator

Intermediate Language

- Three ways of intermediate representation:
  - ✓ Syntax tree
  - ✓ Postfix notation
  - ✓ Three address code
- The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

Intermediate Language

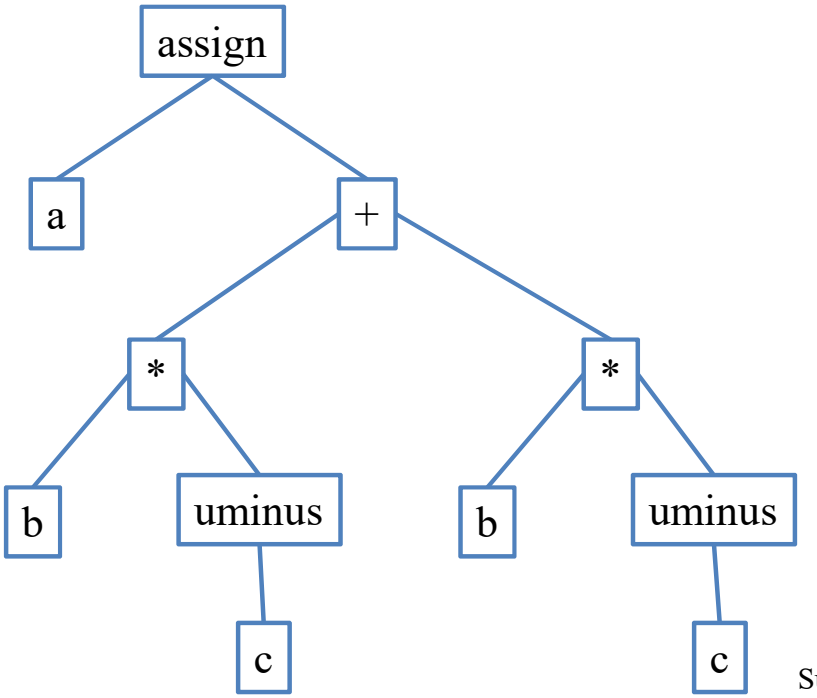
- **Graphical Representation**
  - ✓ **Syntax tree:** A syntax tree depicts the natural hierarchical structure of a source program.

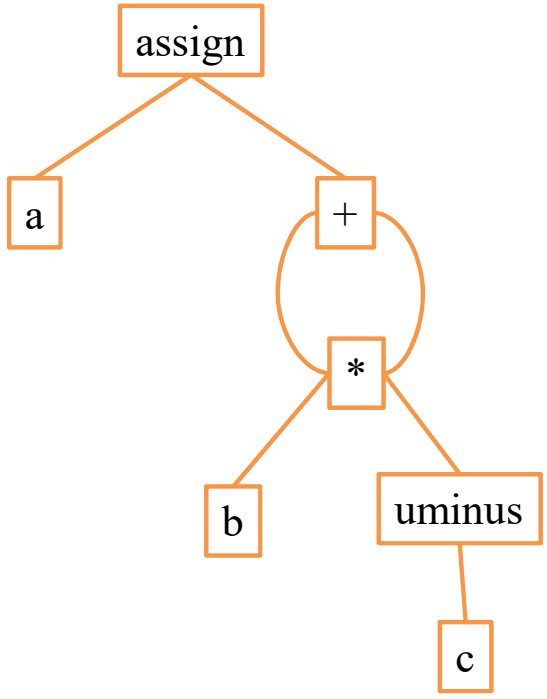
    A dag (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpression are identified.

    A syntax tree and dag for the assignment statement a : = b * - c + b * - c are as follows:

Intermediate Language

- **Graphical Representation**
  - ✓ **Syntax tree:** A syntax tree and dag for the assignment statement a : = b * - c + b * - c are as follows:





Intermediate Language

- **Graphical Representation**
  - ✓ **Postfix notation:** Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree (a : = b * - c + b * - c) given above is

    a b c uminus * b c uminus * + assign

Intermediate Language

- **Graphical Representation**
  - ✓ **Syntax-directed definition:** Syntax trees for assignment statements are produced by the syntax-directed definition.
    - Non-terminal S generates an assignment statement.
    - The two binary operators + and * are examples of the full operator set in a typical language.
    - Operator associativities and precedences are the usual ones, even though they have not been put into the grammar.
    - This definition constructs the tree from the input a : = b * - c + b* - c.

Intermediate Language

- **Graphical Representation**
  - ✓ **Syntax-directed definition:**

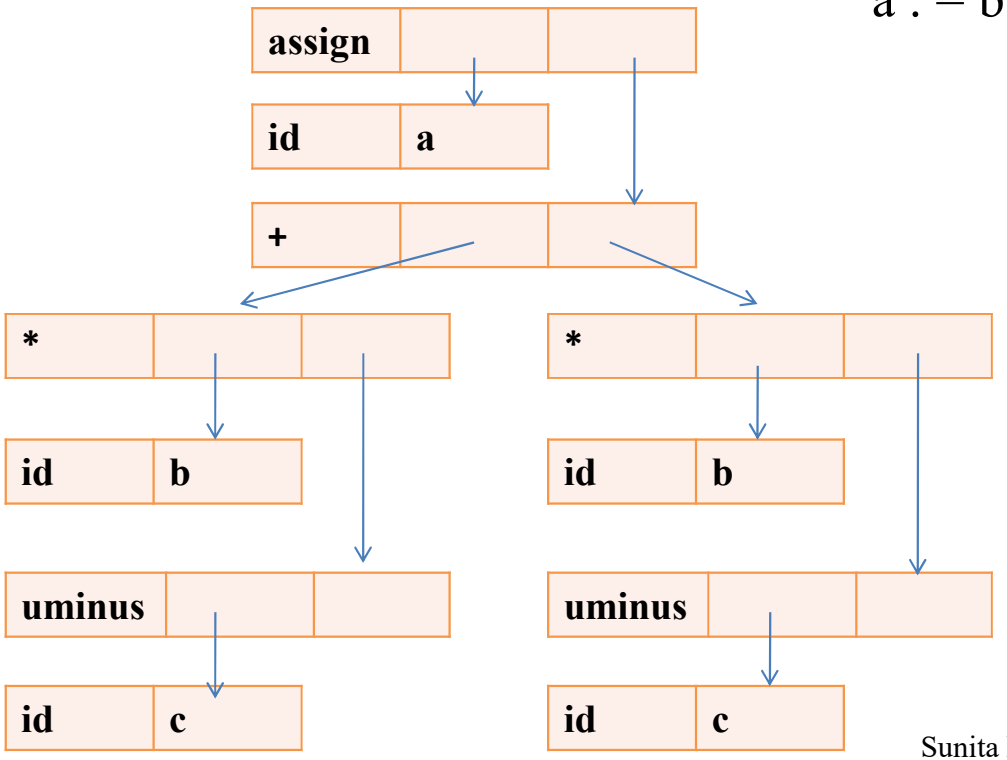
Syntax-directed definition to produce syntax trees for assignment statements

Production	Semantic Rules
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E1 + E2	E.nptr : = mknode('+', E1.nptr, E2.nptr)
E → E1 * E2	E.nptr : = mknode('*', E1.nptr, E2.nptr)
E → - E1	E.nptr : = mknode('uminus', E1.nptr)
E → (E1)	E.nptr : = E1.nptr
E → id	E.nptr : = mkleaf(id, id.place)

The token id has an attribute place that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute id.name, representing the lexeme associated with that occurrence of id.

Intermediate Language

- **Graphical Representation**
  - ✓ **Syntax-directed definition:** Two representations of the syntax tree for a : = b * - c + b* - c.



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8

Intermediate Language

- **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.

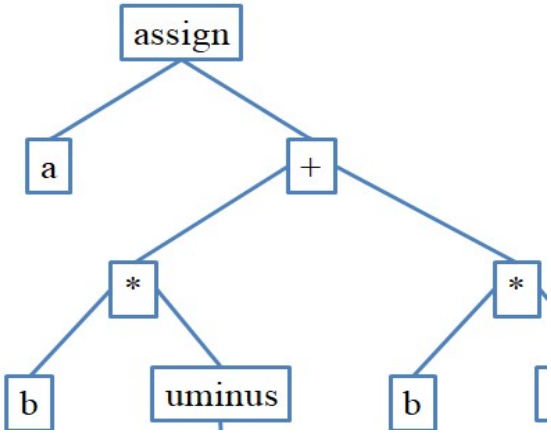
Intermediate Language

- **Three-Address Code:**
  - 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 is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three address statements.

Intermediate Language

- **Three-Address Code:**
  - Three-address code corresponding to the syntax tree and dag for a : = b * - c + b* - c

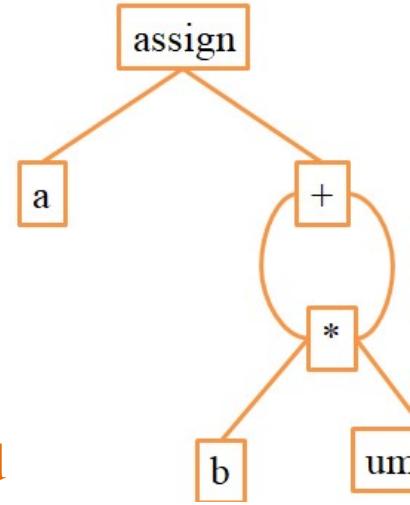


t1 : = - c
t2 : = b * t1
t3 : = - c
t4 : = b * t3
t5 : = t2 + t4
a : = t5

(a) Code for the syntax tree

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

(b) Code for the d



  - The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result

Intermediate Language

- **Types of Three-Address Statements:**

The common three-address statements are:

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.

Intermediate Language

- **Types of Three-Address Statements:**

The common three-address statements are:

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.

Intermediate Language

- **Types of Three-Address Statements:**

The common three-address statements are:

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).

Intermediate Language

- **Types of Three-Address Statements:**

The common three-address statements are:

7. Indexed assignments of the form ***x : = y[i]*** and ***x[i] : = y***. The statement ***x : = y[i]*** sets x to the value in the location i memory units beyond location y. The statement ***x[i] : = y*** sets the content of location i units beyond the x to the value y.

8. Address and pointer assignments of the form ***x : = &y , x : = *y,*** and ****x : = y.*** The statement ***x : = &y*** sets the value of x to be the location of y. In the statement ***x : = *y,*** presumably y is a pointer or a temporary whose r-value is a location. The statement ****x : = y*** sets the r-value of the oject pointed to be by x to the r-value of y.

Intermediate Language

- **Syntax-Directed Translation into Three-Address Code:**
  - When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree.
  - For example, ***id : = E*** consists of code to evaluate E into some temporary t, followed by the assignment ***id.place : = t.***
  - The synthesized attribute S.code represents the three-address code for the assignment S.
  - The nonterminal E has two attributes :
    1. 1. E.place, the name that will hold the value of E , and
    2. 2. E.code, the sequence of three-address statements evaluating E.

Intermediate Language

- **Syntax-Directed Translation into Three-Address Code:**

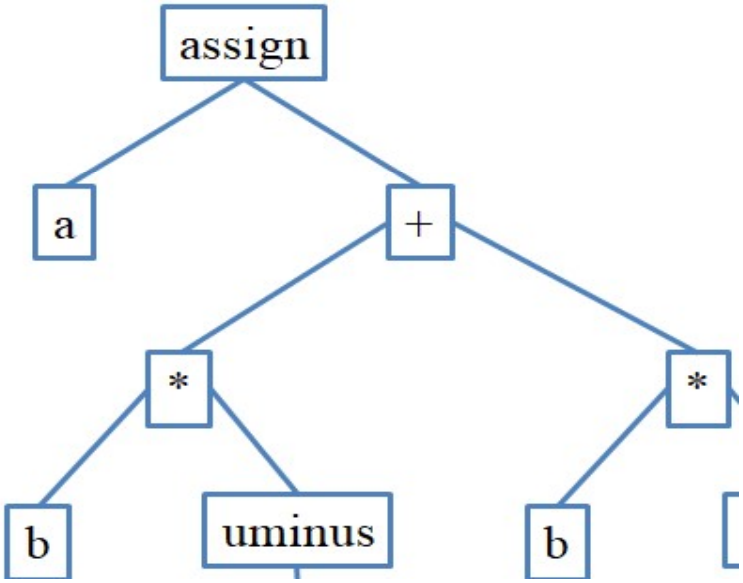
Syntax-directed definition to produce three-address code for assignments

Production	Semantic Rules
S → id : = E	S.code : = E.code || gen(id.place ':=' E.place)
E → E1 + E2	E.place := newtemp;
	E.code := E1.code || E2.code || gen(E.place ':=' E1.place '+' E2.place)
E → E1 * E2	E.place := newtemp;
	E.code := E1.code || E2.code || gen(E.place ':=' E1.place '*' E2.place)
E → - E1	E.place := newtemp;
	E.code := E1.code || gen(E.place ':=' 'uminus' E1.place)
E → (E1)	E.place : = E1.place;
	E.code : = E1.code
E → id	E.place : = id.place;
	E.code : = ''

Intermediate Language

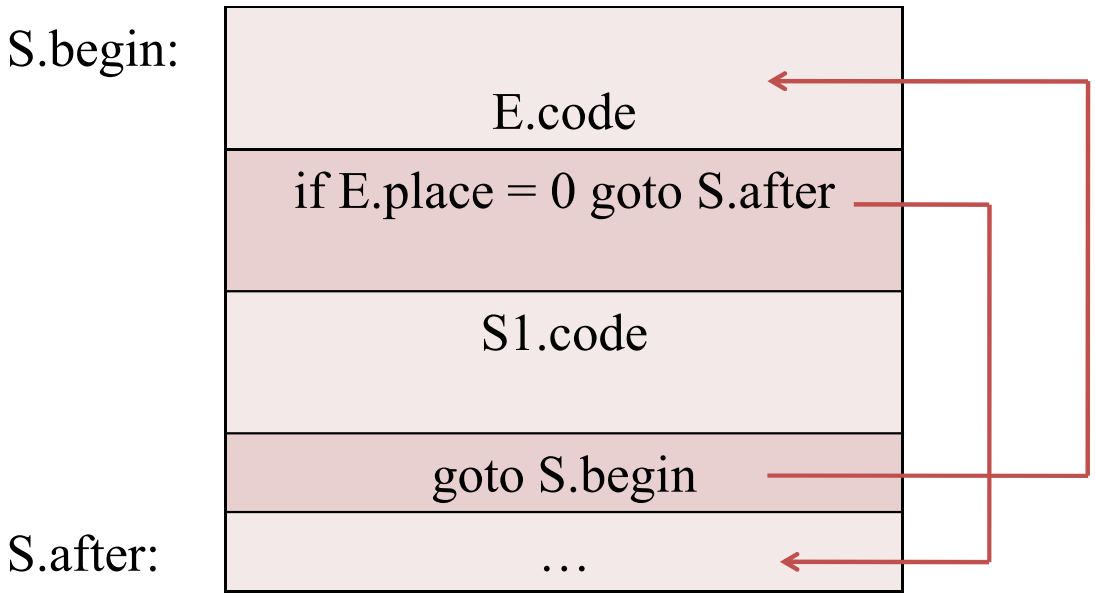
- **Syntax-Directed Translation into Three-Address Code:**
  - Given input a : = b * - c + b * - c, the three-address code is as shown below

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



Intermediate Language

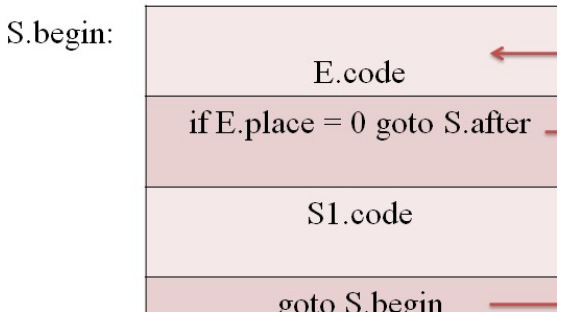
- **Semantic rules generating code for a while statement**



Intermediate Language

- **Semantic rules generating code for a while statement**

Production	Semantic Rules
S → while E do S1	S.begin := newlabel; S.after := newlabel; S.code := gen(S.begin ':') || E.code || gen ('if' E.place '=' '0' 'goto' S.after)|| S1.code || gen ('goto' S.begin) || gen (S.after ':')



Intermediate Language

- **Semantic rules generating code for a while statement**
  - The function ***newtemp*** returns a sequence of distinct names t1,t2,….. in response to successive calls.
  - Notation ***gen(x ':=' y '+' z)*** is used to represent three-address statement x := y + z. Expressions appearing instead of variables like x, y and z are evaluated when passed to gen, and quoted operators or operand, like '+' are taken literally.
  - Flow-of–control statements can be added to the language of assignments. The code for ***S → while E do S1*** is generated using new attributes ***S.begin*** and ***S.after*** to mark the first statement in the code for E and the statement following the code for S, respectively.

Intermediate Language

- **Semantic rules generating code for a while statement**
  - The function ***newlabel*** returns a new label every time it is called.
  - We assume that a ***non-zero*** expression represents ***true***; that is when the value of E becomes zero, control leaves the while statement.

Intermediate Language

- **Implementation of Three-Address Statements:**
  - A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands.
  - Three such representations are:
    - ✓ **Quadruples**
    - ✓ **Triples**
    - ✓ **Indirect triples**

Intermediate Language

- **Quadruples:**
  - 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.
  - The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Intermediate Language

- **Quadruples:**

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

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	: =	t3		a

Intermediate Language

- **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.
  - If we do so, 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.

Intermediate Language

- **Triples:**

  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)

Intermediate Language

- **Triples:**
  - A ternary operation like x[i] : = y requires two entries in the triple structure as shown as belo while x : = y[i] is naturally represented as two operations.

	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

(a) x[i] : = y

	op	arg1	arg2
(0)	= [ ]	y	i
(1)	assign	x	(0)

(b) x : = y[i]

Intermediate Language

- **Indirect Triples:**
  - Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
  - For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

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

	statement		op	arg1	arg2
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

Declarations

- As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure.
- For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name.
- The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Declarations

- **Declarations in a Procedure:**
  - The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group.
  - In this case, a global variable, say offset, can keep track of the next available relative address.
  - In the translation scheme shown below:
    - ✓ Non-terminal P generates a sequence of declarations of the form ***id : T***.
    - ✓ Before the first declaration is considered, ***offset*** is set to 0. As each new name is seen, that name is entered in the symbol table with offset equal to the current value of ***offset***, and ***offset*** is incremented by the width of the data object denoted by that name.

Declarations

- **Declarations in a Procedure:**
  - In the translation scheme shown below:
    - ✓ The procedure ***enter(name, type, offset)*** creates a symbol-table entry for ***name***, gives its type ***type*** and relative address ***offset*** in its data area.
    - ✓ Attribute ***type*** represents a type expression constructed from the basic types ***integer*** and ***real*** by applying the type constructors ***pointer*** and ***array***. If type expressions are represented by graphs, then attribute ***type*** might be a pointer to the node representing type expression.
    - ✓ The width of an array is obtained by multiplying the width of each element by the number of elements in the array.
    - ✓ The width of integer, real and each pointer is assumed to be 4, 8 and 4 respectively.

Declarations

- **Declarations in a Procedure:**
  - **Computing the types and relative addresses of declared names**

P → D	{ offset : = 0 }
D → D ; D	
D → id : T	{ enter(id.name, T.type, offset); offset : = offset + T.width }
T → integer	{ T.type : = integer; T.width : = 4 }
T → real	{ T.type : = real; T.width : = 8 }
T → array [num] of T1	{ T.type : = array(num.val, T1.type); T.width : = num.val X T1.width }
T → ↑ T1	{ T.type : = pointer (T1.type); T.width : = 4 }

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

      $P \rightarrow D$

      $D \rightarrow D \ ; \ D \mid id : T \mid proc \ id \ ; \ D \ ; \ S$

      One possible implementation of a symbol table is a linked list of entries for names.

Declarations

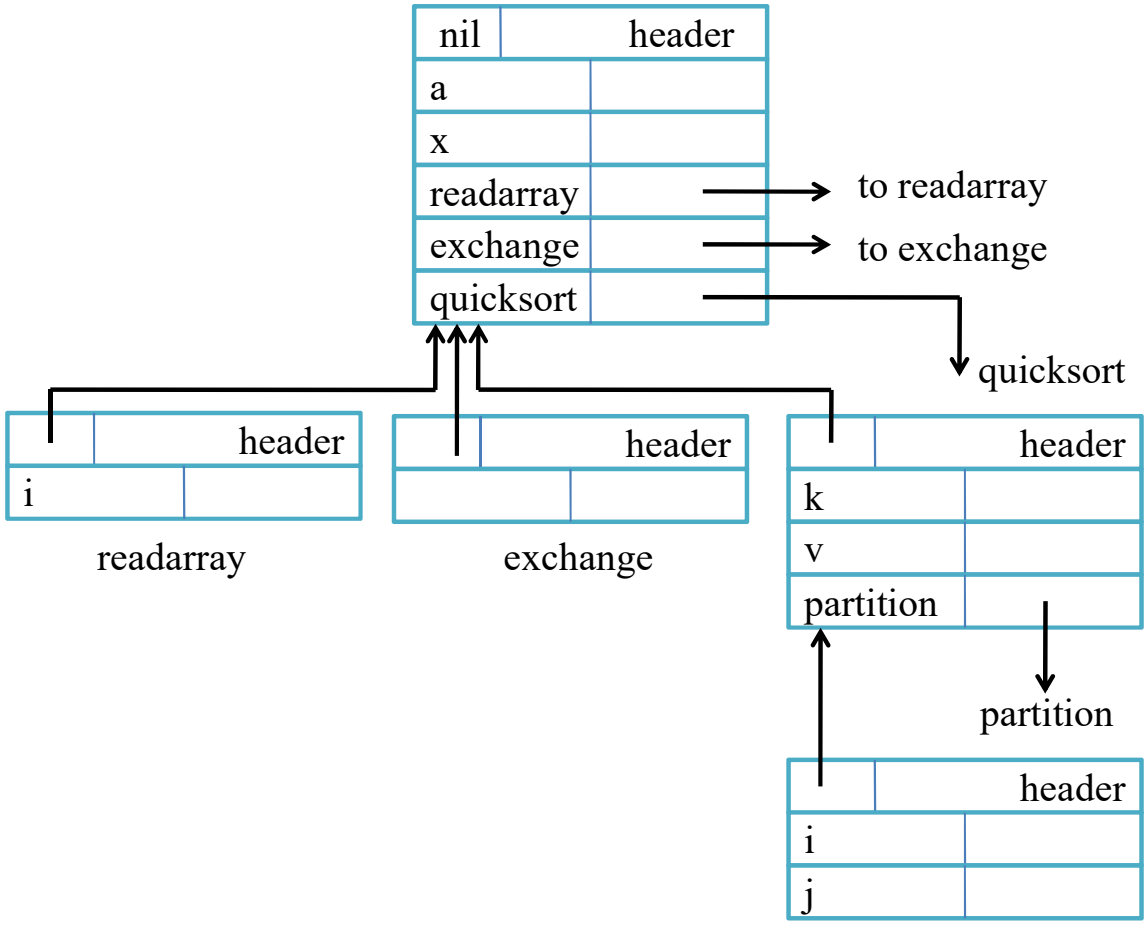
- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ A new symbol table is created when a procedure declaration ***D → proc id; D1;S*** is seen, and entries for the declarations in D1 are created in the new table.
    - ✓ The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure.
    - ✓ The only change from the treatment of variable declarations is that the procedure ***enter*** is told which symbol table to make an entry in.

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ For example, consider the symbol tables for procedures ***readarray***, ***exchange***, and ***quicksort*** pointing back to that for the containing procedure sort, consisting of the entire program. Since ***partition*** is declared within ***quicksort***, its table points to that of ***quicksort***.

```
program sort( input, output );
  var a: array [ 0..10 ] of integer;
      x: integer

  procedure readarray;
        var i: integer;
        begin…a….end {readarray};

  procedure exchange (i, j: integer);
        begin
              x := a[i]; a[i] := a[j]; a[j] := x;
        end{exchange};

  procedure quicksort( m, n: integer );
        var k, v: integer;

        function partition( y, z: integer ) : integer;
           var i, j: integer;
            begin  …a…
                  …v…
                  …exchange(i, j);…
            end {partition};

        begin …  end {quicksort};

  begin  … end {sort}
```

Declarations



Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ The semantic rules are defined in terms of the following operations:
    1. ***mktable(previous)*** creates a new symbol table and returns a pointer to the new table. The argument ***previous*** points to a previously created symbol table, presumably that for the enclosing procedure.
    2. ***enter(table, name, type, offset)*** creates a new entry for name ***name*** in the symbol table pointed to by ***table***. Again, enter places type ***type*** and relative address ***offset*** in fields within the entry.

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ The semantic rules are defined in terms of the following operations:

    3. ***addwidth(table, width)*** records the cumulative width of all the entries in ***table*** in the header associated with this symbol table.
    4. ***enterproc(table, name, newtable)*** creates a new entry for procedure name in the symbol table pointed to by ***table***. The argument ***newtable*** points to the symbol table for this procedure name.

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**

Syntax directed translation scheme for nested procedures

P →M D	{ addwidth (top(tblptr) , top (offset)); pop (tblptr); pop (offset) }
M → ε	{ t : = mktable (nil); push (t,tblptr); push (0,offset) }
D → D1 ; D2	

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**

Syntax directed translation scheme for nested procedures

D → proc id ; N D1 ; S	{ t : = top (tblptr); addwidth (t, top (offset)); pop (tblptr); pop (offset); enterproc (top (tblptr), id.name, t) }
D → id : T	{ enter (top (tblptr), id.name, T.type, top (offset)); top (offset) := top (offset) + T.width }
N → ε	{ t := mktable (top (tblptr)); push (t, tblptr); push (0,offset) }

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ The stack ***tblptr*** is used to contain pointers to the tables for ***sort***, ***quicksort***, and ***partition*** when the declarations in partition are considered.
    - ✓ The top element of stack ***offset*** is the next available relative address for a local of the current procedure.
    - ✓ All semantic actions in the subtrees for B and C in

      $A \rightarrow BC \{actionA\}$

      are done before ***actionA*** at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ The action for nonterminal M initializes stack ***tblptr*** with a symbol table for the outermost scope, created by operation ***mktable(nil)***. The action also pushes relative address 0 onto stack offset.
    - ✓ Similarly, the nonterminal N uses the operation ***mktable(top(tblptr))*** to create a new symbol table. The argument ***top(tblptr)*** gives the enclosing scope for the new table.
    - ✓ For each variable declaration ***id: T***, an entry is created for ***id*** in the current symbol table. The top of stack ***offset*** is incremented by ***T.width***.

Declarations

- **Declarations in a Procedure:**
  - **Keeping Track of Scope Information:**
    - ✓ When the action on the right side of ***D → proc id; ND1; S*** occurs, the width of all declarations generated by D1 is on the top of stack offset; it is recorded using ***addwidth***. Stacks ***tblptr*** and ***offset*** are then popped.
      At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

Assignment Statements

- Suppose that the context in which an assignment appears is given by the following grammar.

  $P \rightarrow M\ D$

  $M \rightarrow \varepsilon$

  $D \rightarrow D\ ;\ D \mid id : T \mid proc\ id\ ;\ N\ D\ ;\ S$

  $N \rightarrow \varepsilon$

- Non-terminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Assignment Statements

Translation scheme to produce three-address code for assignments

S → id : = E	{ p : = lookup (id.name); if p ≠ nil then emit(p ‘ : =’ E.place) else error }
E → E1 + E2	{ E.place : = newtemp; emit(E.place ‘: =’ E1.place ‘ + ‘ E2.place) }
E → E1 * E2	{ E.place : = newtemp; emit(E.place ‘: =’ E1.place ‘ * ‘ E2.place) }
E → - E1	{ E.place : = newtemp; emit (E.place ‘: =’ ‘uminus’ E1.place) }
E → (E1)	{ E.place : = E1.place }
E → id	{ p : = lookup (id.name); if p ≠ nil then E.place : = p else error }

Assignment Statements

- The lexeme for the name represented by ***id*** is given by attribute ***id.name***
- Operation ***lookup(id.name)*** checks if there is an entry for this occurrence of name in symbol table
  - If so pointer to entry is returned
  - Otherwise ***lookup*** returns ***nil*** to indicate that no entry was found
- The semantic actions used procedure ***emit*** to emit three address statements to output file rather than building up ***code*** for attribute for nonterminal.

Assignment Statements

- **Reusing Temporary Names**
  - The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.
  - Temporaries can be reused by changing newtemp. The code generated by the rules for E → E1 + E2 has the general form:

    evaluate E1 into t1
    evaluate E2 into t2
    t : = t1 + t2
  - The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
  - Keep a count c , initialized to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use $c and increase c by 1.

Assignment Statements

- **Reusing Temporary Names**
  - For example, consider the assignment x := a * b + c * d – e * f

statement	value of c
	0
$0 := a * b	1
$1 := c * d	2
$0 := $0 + $1	1
$1 := e * f	2
$0 := $0 - $1	1
x := $0	0

Assignment Statements

- **Addressing Array Elements:**
  - Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the ith element of array A begins in location

    base + (i – low) x w

    where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array. That is, base is the relative address of A[low].
  - The expression can be partially evaluated at compile time if it is rewritten as

    i x w + (base – low x w)

    The subexpression c = base – low x w can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding i x w to c.

Assignment Statements

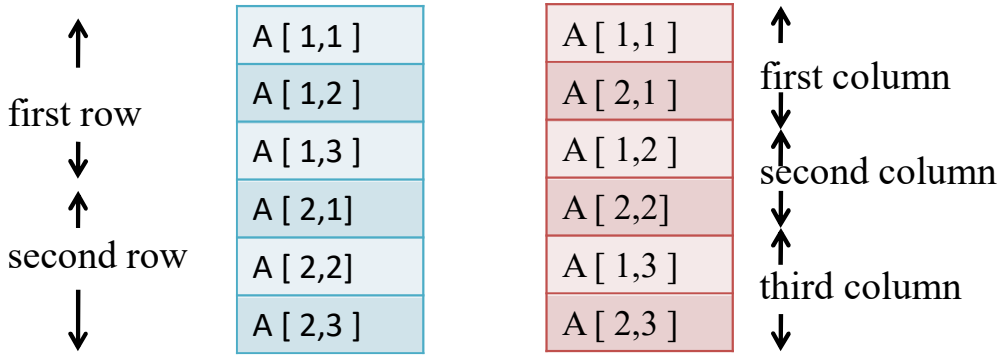
- **Addressing Array Elements:**
  - **Address calculation of multi-dimensional arrays:**

    A two-dimensional array is stored in of the two forms :

    ✓ **Row-major (row-by-row)**

    ✓ **Column-major (column-by-column)**

Layouts for a 2 x 3 array



(a) Row-major (b) Column major

Assignment Statements

- **Addressing Array Elements:**
  - **Address calculation of multi-dimensional arrays:**

    In the case of row-major form, the relative address of A[i1 , i2] can be calculated by the formula

    base + ((i1 – low1) x n2 + i2 – low2) x w

    where, low1 and low2 are the lower bounds on the values of i1 and i2 and n2 is the number of values that i2 can take. That is, if high2 is the upper bound on the value of i2, then n2 = high2 – low2 + 1.

    Assuming that i1 and i2 are the only values that are known at compile time, we can rewrite the above expression as

    ((i1 x n2) + i2) x w + (base – ((low1 x n2) + low2) x w)

Assignment Statements

- **Addressing Array Elements:**
  - **Generalized formula:**

    The expression generalizes to the following expression for the relative address of A[i1,i2,…,ik]

    ((. . . ((i1n2 + i2) n3 + i3) . . .) nk + ik) x w + base – ((. . .((low1n2 + low2)n3 + low3) . . .) nk + lowk) x w

    for all j, nj = highj – lowj + 1

Assignment Statements

- **Addressing Array Elements:**
  - **The Translation Scheme for Addressing Array Elements :**

    Semantic actions will be added to the grammar :

    (1) S → L : = E
    (2) E → E + E
    (3) E → (E)
    (4) E → L
    (5) L → Elist]
    (6) L → id
    (7) Elist → Elist , E
    (8) Elist → id [E

Assignment Statements

- **Addressing Array Elements:**
  - **The Translation Scheme for Addressing Array Elements :**

    We generate a normal assignment if L is a simple name, and an indexed assignment into the location denoted by L otherwise :

    (1) S → L : = E { if L.offset = null then / * L is a simple id */
    emit (L.place ‘: =’ E.place) ;
    else
    emit (L.place ‘ [‘ L.offset ‘]’ ‘: =’ E.place) }

    (2) E → E1 + E2 { E.place : = newtemp;
    emit (E.place ‘: =’ E1.place ‘ +’ E2.place) }

    (3) E → (E1) { E.place : = E1.place }

Assignment Statements

- **Addressing Array Elements:**
  - **The Translation Scheme for Addressing Array Elements :**

    When an array reference L is reduced to E , we want the r-value of L. Therefore we use indexing to obtain the contents of the location L.place [L.offset] :

    (4) E → L { if L.offset = null then /* L is a simple id* /
    E.place : = L.place
    else begin
    E.place : = newtemp;
    emit (E.place ‘: =’ L.place ‘ [‘ L.offset ‘]’)
    end }

Assignment Statements

- **Addressing Array Elements:**
  - **The Translation Scheme for Addressing Array Elements :**

    When an array reference L is reduced to E , we want the r-value of L. Therefore we use indexing to obtain the contents of the location L.place [L.offset] :

    (5) L → Elist] { L.place : = newtemp;
    L.offset : = newtemp;
    emit (L.place ‘: =’ c(Elist.array));
    emit (L.offset ‘: =’ Elist.place ‘*’ width (Elist.array)) }

    (6) L → id { L.place := id.place;
    L.offset := null }

Assignment Statements

- **Addressing Array Elements:**
  - **The Translation Scheme for Addressing Array Elements :**

    When an array reference L is reduced to E , we want the r-value of L. Therefore we use indexing to obtain the contents of the location L.place [L.offset] :

    (7) Elist → Elist1 , E { t := newtemp;
    m : = Elist1.ndim + 1;
    emit (t ‘: =’ Elist1.place ‘*’ limit (Elist1.array,m));
    emit (t ‘: =’ t ‘+’ E.place);
    Elist.array : = Elist1.array;
    Elist.place : = t;
    Elist.ndim : = m }

    (8) Elist → id [E { Elist.array : = id.place;
    Elist.place : = E.place;
    Elist.ndim : = 1 }

Assignment Statements

- **Type conversion within Assignments :**

  Consider the grammar for assignment statements as above, but suppose there are two types – real and integer, with integers converted to real when necessary.

  We have another attribute E.type, whose value is either real or integer. The semantic rule for E.type associated with the production E → E + E is :

  E → E + E { E.type : =
  if E1.type = integer **and**
  E2.type = integer **then** integer
  else real }

Assignment Statements

- **Type conversion within Assignments :**
  The entire semantic rule for E → E + E and most of the other productions must be modified to generate, when necessary, three-address statements of the form x : = inttoreal y, whose effect is to convert integer y to a real of equal value, called x.

Assignment Statements

- **Type conversion within Assignments :**

Semantic action for E → E1 + E2

```
E.place := newtemp;
if E1.type = integer and E2.type = integer then
begin
        emit( E.place ‘: =’ E1.place ‘int +’ E2.place);
        E.type : = integer
end
else if E1.type = real and E2.type = real then
begin
        emit( E.place ‘: =’ E1.place ‘real +’ E2.place);
        E.type : = real
end
```

```
else if E1.type = integer and E2.type = real then
begin
        u : = newtemp;
        emit( u ‘: =’ ‘inttoreal’ E1.place);
        emit( E.place ‘: =’ u ‘ real +’ E2.place);
        E.type : = real
end
else if E1.type = real and E2.type =integer then
begin
    u : = newtemp;
    emit( u ‘: =’ ‘inttoreal’ E2.place);
    emit( E.place ‘: =’ E1.place ‘ real +’ u);
    E.type : = real
end
else
        E.type : = type_error;
```

Boolean Expressions

- Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.
- Boolean expressions are composed of the boolean operators (**and**, **or**, and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E1 **relop** E2, where E1 and E2 are arithmetic expressions.
- Here we consider boolean expressions generated by the following grammar :

  E → E **or** E | E **and** E | **not** E | (E) | id **relop** id | **true** | **false**

Boolean Expressions

- **Methods of Translating Boolean Expressions:**

  There are two principal methods of representing the value of a boolean expression. They are :

  - To encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
  - To implement boolean expressions by flow of control, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Boolean Expressions

- **Numerical Representation**

  Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

  For example :

  - The translation for

    a **or** b **and not** c

    is the three-address sequence

    t1 : = **not** c

    t2 : = b **and** t1

    t3 : = a **or** t2

Boolean Expressions

- **Numerical Representation**

  Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

  For example :

  - A relational expression such as a < b is equivalent to the conditional statement

    if a < b then 1 else 0

    which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

    100 : **if** a < b **goto** 103
    101 : t : = 0
    102 : **goto** 104
    103 : t : = 1
    104 :

Boolean Expressions

- **Numerical Representation**

Translation scheme using a numerical representation for Booleans

E → E1 **or** E2	{ E.place : = newtemp; emit(E.place ‘: =’ E1.place ‘**or**’ E2.place) }
E → E1 **and** E2	{ E.place : = newtemp; emit(E.place ‘: =’ E1.place ‘**and**’ E2.place) }
E → **not** E1	{ E.place : = newtemp; emit(E.place ‘: =’ ‘**not**’ E1.place) }
E → (E1)	{ E.place : = E1.place }

Boolean Expressions

- **Numerical Representation**

Translation scheme using a numerical representation for Booleans

E → id1 **relop** id2 { E.place : = newtemp;
emit('**if**' id1.place **relop.op** id2.place '**goto**' nextstat + 3);
emit(E.place ': =' '**0**');
emit('**goto**' nextstat +2);
emit(E.place ': =' '**1**') }

E → **true** { E.place : = newtemp;
emit(E.place ': =' '**1**') }

E → **false** { E.place : = newtemp;
emit(E.place ': =' '**0**') }

Boolean Expressions

- **Short-Circuit Code:**
  - We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “short-circuit” or “jumping” code.

  - It is possible to evaluate boolean expressions without generating code for the boolean operators and, or, and not if we represent the value of an expression by a position in the code sequence.

Boolean Expressions

- **Short-Circuit Code:**

Translation of a < b or c < d and e < f	
100 : if a < b goto 103	107 : t2 : = 1
101 : t1 : = 0	108 : if e < f goto 111
102 : goto 104	109 : t3 : = 0
103 : t1 : = 1	110 : goto 112
104 : if c < d goto 107	111 : t3 : = 1
105 : t2 : = 0	112 : t4 : = t2 and t3
106 : goto 108	113 : t5 : = t1 or t4

Boolean Expressions

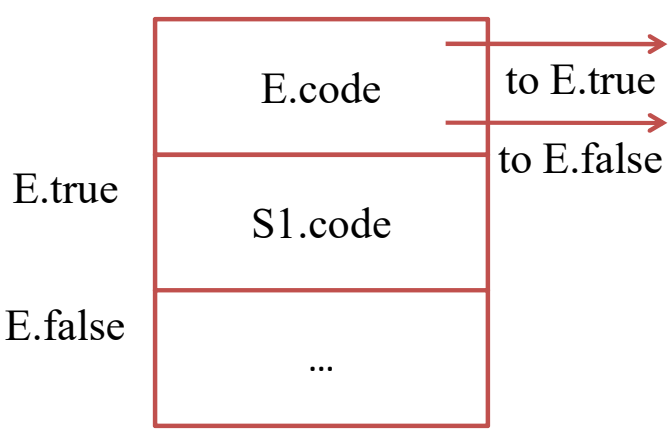
- **Flow-of-Control Statements**
  - We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

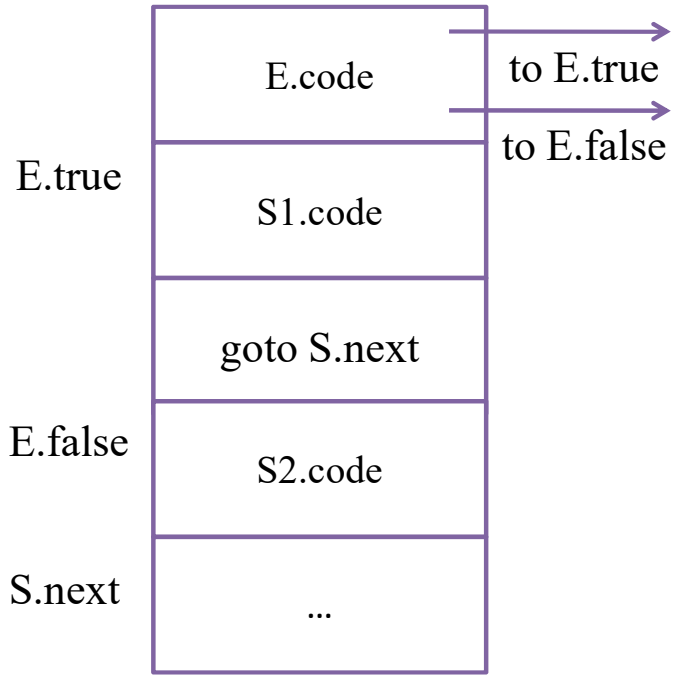
    S → if E then S1
    | if E then S1 else S2
    | while E do S1

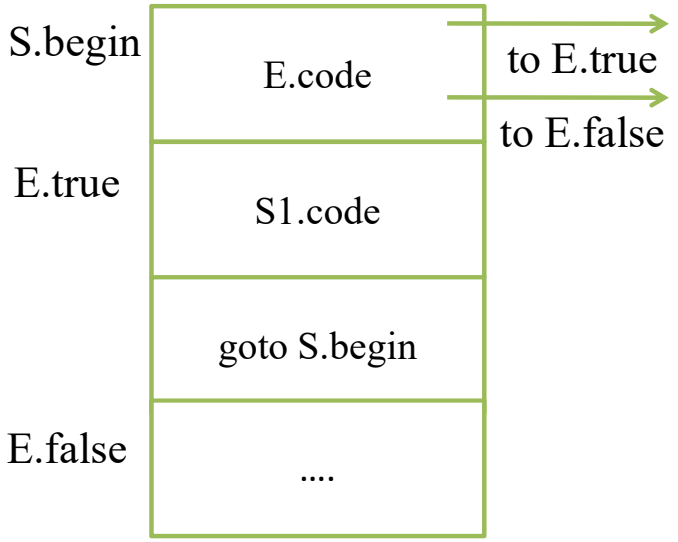
  - In each of these productions, E is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function newlabel returns a new symbolic label each time it is called.

Boolean Expressions

- **Flow-of-Control Statements**
  - **Code for if-then , if-then-else, and while-do statements**

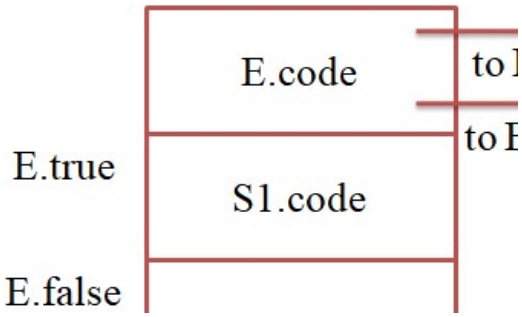






Boolean Expressions

- **Flow-of-Control Statements**

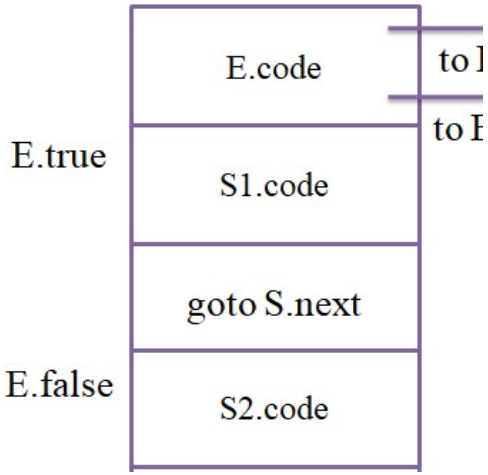


Syntax-directed definition for flow-of-control statements

Production	Semantic Rules
S → if E then S1	E.true : = newlabel; E.false : = S.next; S1.next : = S.next; S.code : = E.code || gen(E.true ':') || S1.code

Boolean Expressions

- **Flow-of-Control Statements**

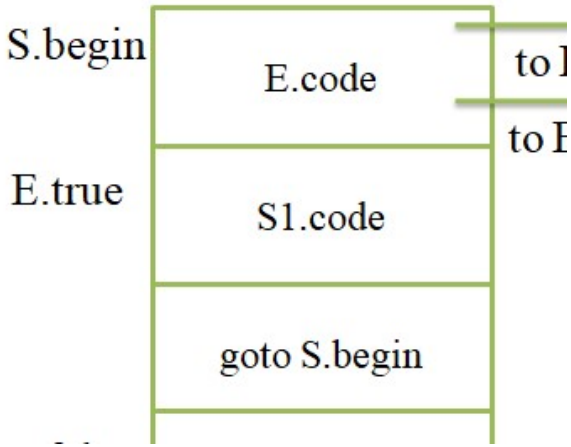


Syntax-directed definition for flow-of-control statements

Production	Semantic Rules
S → if E then S1 else S2	E.true : = newlabel; E.false : = newlabel; S1.next : = S.next; S2.next : = S.next; S.code : = E.code || gen(E.true ':') || S1.code || gen('goto' S.next) || gen(E.false ':') || S2.code

Boolean Expressions

- **Flow-of-Control Statements**



Syntax-directed definition for flow-of-control statements

Production	Semantic Rules
S → while E do S1	S.begin : = newlabel; E.true : = newlabel; E.false : = S.next; S1.next : = S.begin; S.code : = gen(S.begin ':')|| E.code || gen(E.true ':') || S1.code || gen('goto' S.begin)

Boolean Expressions

- **Control-Flow Translation of Boolean Expressions:**

Syntax-directed definition to produce three-address code for booleans

Production	Semantic Rules
E → E1 or E2	E1.true : = E.true;
	E1.false : = newlabel;
	E2.true : = E.true;
	E2.false : = E.false;
	E.code : = E1.code || gen(E1.false ':') || E2.code
E → E1 and E2	E1.true : = newlabel;
	E1.false : = E.false;
	E2.true : = E.true;
	E2.false : = E.false;
	E.code : = E1.code || gen(E1.true ':') || E2.code

Boolean Expressions

- **Control-Flow Translation of Boolean Expressions:**

Syntax-directed definition to produce three-address code for booleans

Production	Semantic Rules
E → not E1	E1.true : = E.false; E1.false : = E.true; E.code : = E1.code
E → (E1)	E1.true : = E.true; E1.false : = E.false; E.code : = E1.code
E → id1 relop id2	E.code : = gen('if' id1.place relop.op id2.place 'goto' E.true) || gen('goto' E.false)
E → true	E.code : = gen('goto' E.true)
E → false	E.code : = gen('goto' E.false)

Boolean Expressions

- **Control-Flow Translation of Boolean Expressions:**
  - Translation of a < b or c < d and e < f using Syntax-directed definition to produce three-address code for booleans

Translation of a < b or c < d and e < f

```
        if a < b goto Ltrue
        goto L1
L1:     if c < d goto L2
        goto Lfalse
L2:     if e < f goto Ltrue
        goto Lfalse
```

Translation of a < b or c < d and e < f

```
100 : if a < b goto 103          107 : t2 : = 1
101 : t1 : = 0                   108 : if e < f goto 111
102 : goto 104                   109 : t3 : = 0
103 : t1 : = 1                   110 : goto 112
104 : if c < d goto 107          111 : t3 : = 1
105 : t2 : = 0                   112 : t4 : = t2 and t3
106 : goto 108                   113 : t5 : = t1 or t4
```

Boolean Expressions

- **Control-Flow Translation of Boolean Expressions:**
  - Translate the following statement using Syntax-directed definition to produce three-address code for booleans

```
while a < b do
        if c < d then
                x := y + z
        else
                x := y - z
```

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 := x + y
        x := t1
        goto L1
L4:     t2 := x - y
        x := t2
        goto L1
```

Boolean Expressions

- **Mixed Mode Boolean Expression**
  - The method of representing boolean expression by jumping code can be used even if arithmetic expression are represented by code to compute their value.
  - Example –

    E → E + E | E and E | E relop E | id

  - We assume that ***E + E*** produces an integer arithmetic result while expression ***E and E*** and ***E relop E*** produce boolean values represented by flow of control.
  - Expression ***E and E*** requires both arguments to be boolean but the operation + and ***relop*** take either type of argument including mixed one.

Boolean Expressions

- **Mixed Mode Boolean Expression**
  - A synthesized attribute ***E.type*** which will be either ***arith*** or ***bool*** depeneding on the type of E is used
  - E will have inherited attributes ***E.true*** and ***E.false*** for boolean expression and synthesized attribute ***E.place*** for arithmatic expression.

Boolean Expressions

- **Mixed Mode Boolean Expression**
  - The part of the semantic rule for $E \rightarrow E + E$ is shown below

Semantic action for E → E1 + E2

```
E.type := arith;
if E1.type = arith and E2.type = arith then
begin
        E.place : = newtemp;
        E.code = E1.code || E.code ||
        gen( E.place ‘: =’ E1.place ‘+’ E2.place);
end
```

Semantic action for E → E1 + E2

```
else if E1.type = arith and E2.type = bool then
begin
        E.place = newlabel;
        E2.true : = newlabel;
        E2.false : = newlabel;
         E.code = E1.code || E2.code ||
                gen(E2.true ‘:’ E.place ‘: =’ E1.place ‘+’ 1) ||
                gen(‘goto’ nextstat + 1) ||
                gen(E2.false ‘:’ E.place ‘: =’ E1.place)
end
else if …
```

Boolean Expressions

- **Mixed Mode Boolean Expression**
  - In the mixed mode case, we generate the code for E1 then E2 followed by three statements

```
E2.true : E.place := E1.place
          goto nextstat +1
E2.false : E.place = E1.place
```

Boolean Expressions

- **CASE Statement**
  - The “switch” or “case” statement is available in a variety of languages. The switch-statement syntax is as shown below :

Switch-statement syntax

```
switch expression
begin
case value : statement
case value : statement
. . .
case value : statement
default : statement
end
```

Boolean Expressions

- **CASE Statement**
  - There is a selector expression, which is to be evaluated, followed by n constant values that the expression might take, including a default "value" which always matches the expression if no other value does.
  - The intended translation of a switch is code to:
    1. Evaluate the expression.
    2. Find which value in the list of cases is the same as the value of the expression.
    3. Execute the statement associated with the value found.

Boolean Expressions

- **CASE Statement**

  Step (2) can be implemented in one of several ways :
  - By a sequence of conditional goto statements, if the number of cases is small.
  - By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
  - If the number of cases s large, it is efficient to construct a hash table.
  - There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say i_{min} to i_{max}, and the number of different values is a reasonable fraction of i_{max} - i_{min}, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset j - i_{min} and the label for the default in entries not filled otherwise. To perform switch, evaluate the expression to obtain the value of j , check the value is within range and transfer to the table entry at offset j-i_{min}.

Boolean Expressions

- **CASE Statement**

Syntax-Directed Translation of Case Statements:

```
switch E
begin
      case V1 : S1
      case V2 : S2
      . . .
      case Vn-1 : Sn-1
      default : Sn
End
```

Translation of a case statement
code to evaluate E into t

```
                goto test
L1 :            code for S1
                goto next
L2 :            code for S2
                goto next
                . . .
Ln-1 :          code for Sn-1
                goto next
Ln :            code for Sn
                goto next
test :          if t = V1 goto L1
                if t = V2 goto L2
                . . .
                if t = Vn-1 goto Ln-1
                goto Ln
                next :
```

Boolean Expressions

- **CASE Statement**

  To translate into above form :

  - When keyword switch is seen, two new labels test and next, and a new temporary t are generated.
  - As expression E is parsed, the code to evaluate E into t is generated. After processing E , the jump goto test is generated.
  - As each case keyword occurs, a new label Li is created and entered into the symbol table. A pointer to this symbol-table entry and the value Vi of case constant are placed on a stack (used only to store cases).
  - Each statement case Vi : Si is processed by emitting the newly created label Li, followed by the code for Si , followed by the jump goto next.

Boolean Expressions

- **CASE Statement**

  To translate into above form :

  - Then when the keyword end terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

    case V1 L1
    case V2 L2
    . . .
    case Vn-1 Ln-1
    case t Ln
    label next

    where t is the name holding the value of the selector expression E, and Ln is the label for the default statement.

Backpatching

- The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes.
- First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations.
- The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated.

Backpatching

- Hence, a series of branching statements with the targets of the jumps left unspecified is generated.
- Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

Backpatching

- To manipulate lists of labels, we use these functions :
  1. ***makelist(i)*** creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.
  2. ***merge(p1,p2)*** concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.
  3. ***backpatch(p,i)*** inserts i as the target label for each of the statements on the list pointed to by p.
  4. ***nextquad()***: returns the index of the next quadruple to be generated

Backpatching

- **Boolean Expressions:**
  - We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

    (1) E → E1 or M E2
    (2) | E1 and M E2
    (3) | not E1
    (4) | (E1)
    (5) | id1 relop id2
    (6) | true
    (7) | false
    (8) M → ε

Backpatching

- **Boolean Expressions:**
  - Synthesized attributes ***truelist*** and ***falselist*** of nonterminal ***E*** are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by ***E.truelist*** and ***E.falselist***.

  - Consider production ***E → E1 and M E2***. If ***E1*** is false, then ***E*** is also false, so the statements on ***E1.falselist*** become part of ***E.falselist***. If ***E1*** is true, then we must next test ***E2***, so the target for the statements ***E1.truelist*** must be the beginning of the code generated for ***E2***. This target is obtained using marker nonterminal ***M***.

Backpatching

- **Boolean Expressions:**
  - Attribute ***M.quad*** records the number of the first statement of ***E2.code***. With the production ***M → ε*** we associate the semantic action

    { M.quad : = nextquad }

    The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E1.truelist when we have seen the remainder of the production ***E → E1 and M E2***.

Backpatching

- **Boolean Expressions:**

(1) E → E1 or M E2	{ backpatch (E1.falselist, M.quad); E.truelist : = merge(E1.truelist, E2.truelist); E.falselist : = E2.falselist }
(2) E → E1 and M E2	{ backpatch (E1.truelist, M.quad); E.truelist : = E2.truelist; E.falselist : = merge(E1.falselist, E2.falselist) }
(3) E → not E1	{ E.truelist : = E1.falselist; E.falselist : = E1.truelist; }
(4) E → (E1)	{ E.truelist : = E1.truelist; E.falselist : = E1.falselist; }

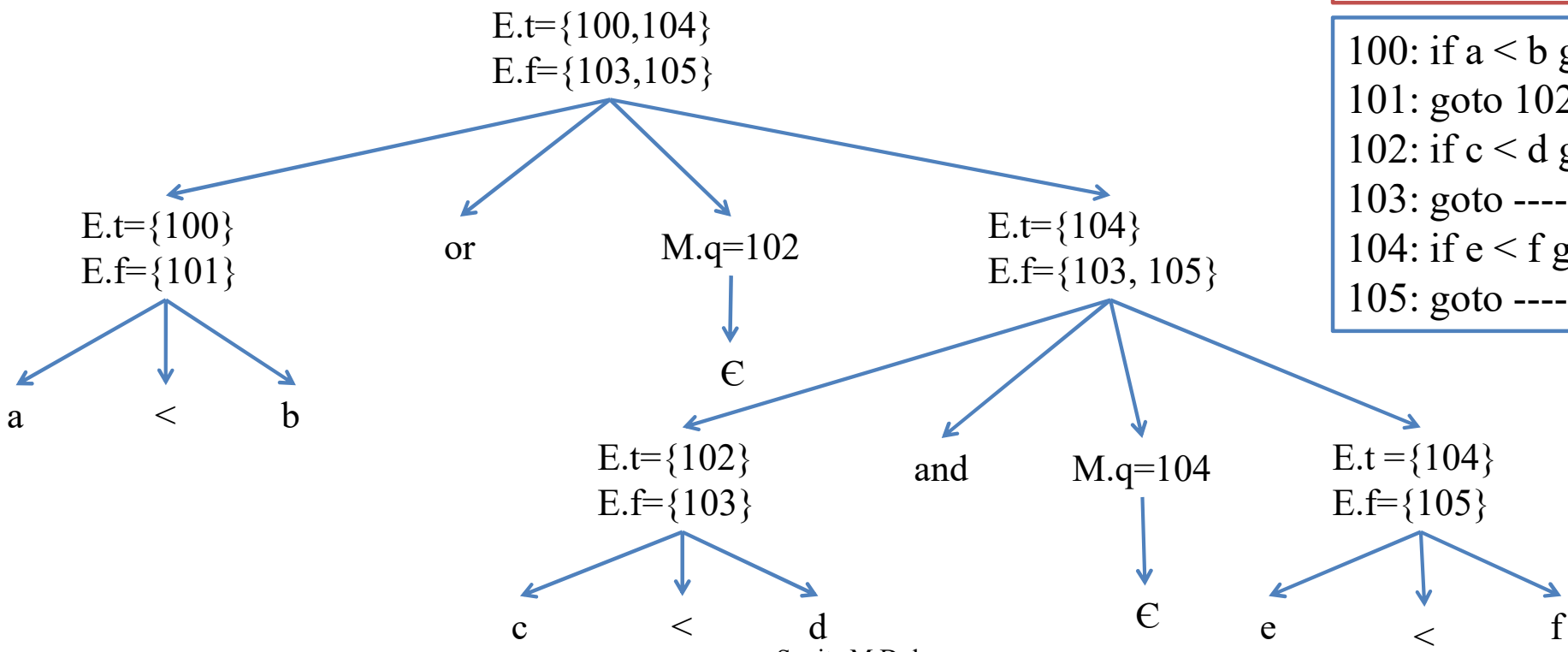
Backpatching

- **Boolean Expressions:**

(5) E → id1 relop id2	{ E.truelist : = makelist (nextquad); E.falselist : = makelist(nextquad + 1); emit('if' id1.place relop.op id2.place 'goto_') emit('goto_') }
(6) E → true	{ E.truelist : = makelist(nextquad); emit('goto_') }
(7) E → false	{ E.falselist : = makelist(nextquad); emit('goto_') }
(8) M → ε	{ M.quad : = nextquad }

Backpatching

- **Boolean Expressions:**
  - Example - Generate code for a < b or c < d and e < f



100: if a < b goto ----
101: goto ----
102: if c < d goto ----
103: goto ----
104: if e < f goto ----
105: goto ----

100: if a < b goto ----
101: goto 102
102: if c < d goto 104
103: goto ----
104: if e < f goto ----
105: goto ----

Backpatching

- **Flow-of-Control Statements:**
  - A translation scheme is developed for statements generated by the following grammar :

    (1) S → if E then S
    (2) | if E then S else S
    (3) | while E do S
    (4) | begin L end
    (5) | A
    (6) L → L ; S
    (7) | S
  - Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Backpatching

- **Scheme to implement the Translation:**
  - The nonterminal E has two attributes E.truelist and E.falselist.
  - L and S also need a list of unfilled quadruples that must eventually be completed by backpatching.
  - These lists are pointed to by the attributes L..nextlist and S.nextlist.
  - S.nextlist is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and L.nextlist is defined similarly.

Backpatching

- **Scheme to implement the Translation:**
  - The semantic rules for the revised grammar are as follows:

(1) S → if E then M1 S1N else M2 S2
{ backpatch (E.truelist, M1.quad);
backpatch (E.falselist, M2.quad);
S.nextlist : = merge (S1.nextlist, merge (N.nextlist, S2.nextlist)) }

  - We backpatch the jumps when E is true to the quadruple M1.quad, which is the beginning of the code for S1. Similarly, we backpatch jumps when E is false to go to the beginning of the code for S2. The list S.nextlist includes all jumps out of S1 and S2, as well as the jump generated by N.

Backpatching

- **Scheme to implement the Translation:**

(2) N → ε { N.nextlist : = makelist(nextquad);
emit('goto _') }

(3) M → ε { M.quad : = nextquad }

(4) S → if E then M S1 { backpatch(E.truelist, M.quad);
S.nextlist : = merge(E.falselist, S1.nextlist) }

(5) S → while M1 E do M2 S1 { backpatch(S1.nextlist, M1.quad);
backpatch(E.truelist, M2.quad);
S.nextlist : = E.falselist
emit('goto' M1.quad) }

Backpatching

- **Scheme to implement the Translation:**

(6) S → begin L end { S.nextlist : = L.nextlist }

(7) S → A { S.nextlist : = nil }

The assignment S.nextlist : = nil initializes S.nextlist to an empty list.

(8) L → L1 ; M S { backpatch(L1.nextlist, M.quad);
L.nextlist : = S.nextlist }

The statement following L1 in order of execution is the beginning of S. Thus the L1.nextlist list is backpatched to the beginning of the code for S, which is given by M.quad.

(9) L → S { L.nextlist : = S.nextlist }

Procedure Calls

- The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns.
- The run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.
- Let us consider a grammar for a simple procedure call statement

(1) S →call id (Elist)
(2) Elist → Elist , E
(3) Elist → E

Procedure Calls

- **Calling Sequences:**
  - The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure.
  - The following are the actions that take place in a calling sequence :
    - ✓ When a procedure call occurs, space must be allocated for the activation record of the called procedure.
    - ✓ The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
    - ✓ Environment pointers must be established to enable the called procedure to access data in enclosing blocks.

Procedure Calls

- **Calling Sequences:**
  - The following are the actions that take place in a calling sequence :
    - ✓ The state of the calling procedure must be saved so it can resume execution after the call.
    - ✓ Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.
    - ✓ Finally a jump to the beginning of the code for the called procedure must be generated.

Procedure Calls

- **Calling Sequences:**

For example, consider the following syntax-directed translation

(1) S → call id (Elist)
{ for each item p on queue do
emit (‘ param’ p);
emit (‘call’ id.place) }

(2) Elist → Elist , E
{ append E.place to the end of queue }

(3) Elist → E
{ initialize queue to contain only E.place }

Procedure Calls

- **Calling Sequences:**
  - Here, the code for S is the code for Elist, which evaluates the arguments, followed by a param p statement for each argument, followed by a call statement.
  - queue is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E

References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann “Compilers- Principles, Techniques and Tools”, Pearson Education.

Thank You !!!