Chapter 6

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 Ideally details of the source language are con ned to the front end and details of the target machine to the back end With a suitably de ned intermediate representation a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j This approach to creating suite of compilers can save a considerable amount of e ort m n compilers can be built by writing just m front ends and n back ends

This chapter deals with intermediate representations static type checking and intermediate code generation For simplicity we assume that a com piler front end is organized as in Fig 6 1 where parsing static checking and intermediate code generation are done sequentially sometimes they can be com bined and folded into parsing We shall use the syntax directed formalisms of Chapters 2 and 5 to specify checking and translation Many of the translation schemes can be implemented during either bottom up or top down parsing us ing the techniques of Chapter 5 All schemes can be implemented by creating a syntax tree and then walking the tree

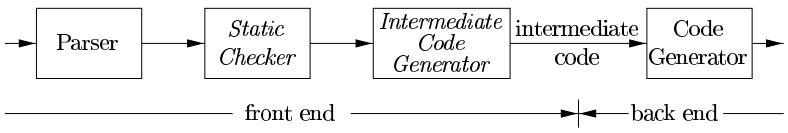


Figure 6 1 Logical structure of a compiler front end

Static checking includes *type checking* which ensures that operators are ap plied to compatible operands It also includes any syntactic checks that remain

From Chapter 6 of *Compilers: Principles, Techniques, and Tools*, Second Edition. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Copyright © 2007 by Pearson Education, Inc. All rights reserved.

after parsing For example static checking assures that a break statement in C is enclosed within a while for or switch statement an error is reported if such an enclosing statement does not exist

The approach in this chapter can be used for a wide range of intermediate representations including syntax trees and three address code both of which were introduced in Section 2 8 The term three address code comes from instructions of the general form $x \quad y \ op \ z$ with three addresses two for the operands y and z and one for the result x

In the process of translating a program in a given source language into code for a given target machine a compiler may construct a sequence of intermediate representations as in Fig 6 2 High level representations are close to the source language and low level representations are close to the target machine Syntax trees are high level they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking

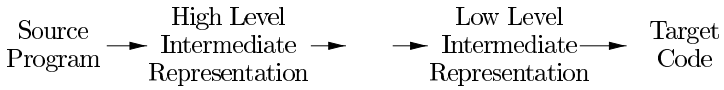


Figure 6 2 A compiler might use a sequence of intermediate representations

A low level representation is suitable for machine dependent tasks like reg ister allocation and instruction selection Three address code can range from high to low level depending on the choice of operators For expressions the di erences between syntax trees and three address code are super cial as we shall see in Section 6 2 3 For looping statements for example a syntax tree represents the components of a statement whereas three address code contains labels and jump instructions to represent the ow of control as in machine language

The choice or design of an intermediate representation varies from compiler to compiler An intermediate representation may either be an actual language or it may consist of internal data structures that are shared by phases of the compiler C is a programming language yet it is often used as an intermediate form because it is exible it compiles into e cient machine code and its com pilers are widely available The original C compiler consisted of a front end that generated C treating a C compiler as a back end

6 1 Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program the children of a node represent the meaningful components of a construct A directed acyclic graph hereafter called a *DAG* for an expression identi es the *common subexpressions* subexpressions that occur more than once of the expression As we shall see in this section DAG s can be constructed by using the same techniques that construct syntax trees

6 1 1 Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators The di erence is that a node N in a DAG has more than one parent if N represents a com mon subexpression in a syntax tree the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression Thus a DAG not only represents expressions more succinctly it gives the compiler important clues regarding the generation of e cient code to evaluate the expressions

Example 6 1 Figure 6 3 shows the DAG for the expression

```
a  a   b  c    b  c   d
```

The leaf for `a` has two parents because `a` appears twice in the expression More interestingly the two occurrences of the common subexpression `b c` are represented by one node the node labeled That node has two parents representing its two uses in the subexpressions `a b c` and `b c d` Even though `b` and `c` appear twice in the complete expression their nodes each have one parent since both uses are in the common subexpression `b c` □

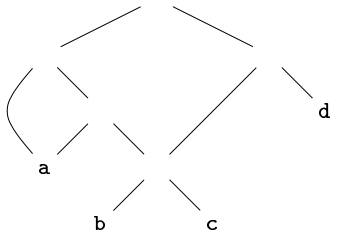


Figure 6 3 Dag for the expression `a a b c b c d`

The SDD of Fig 6 4 can construct either syntax trees or DAG s It was used to construct syntax trees in Example 5 11 where functions *Leaf* and *Node* created a fresh node each time they were called It will construct a DAG if before creating a new node these functions rst check whether an identical node already exists If a previously created identical node exists the existing node is returned For instance before constructing a new node *Node op left right* we check whether there is already a node with label *op* and children *left* and *right* in that order If so *Node* returns the existing node otherwise it creates a new node

Example 6 2 The sequence of steps shown in Fig 6 5 constructs the DAG in Fig 6 3 provided *Node* and *Leaf* return an existing node if possible as

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

Figure 6.4 Syntax-directed definition to produce syntax trees or DAG's

1	p_1	*Leaf*(**id**, *entry a*)	
2	p_2	*Leaf*(**id**, *entry a*)	p_1
3	p_3	*Leaf*(**id**, *entry b*)	
4	p_4	*Leaf*(**id**, *entry 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*(**id**, *entry b*)	p_3
9	p_9	*Leaf*(**id**, *entry c*)	p_4
10	p_{10}	*Node*'+', p_3, p_4	p_5
11	p_{11}	*Leaf*(**id**, *entry d*)	
12	p_{12}	*Node*'−', p_5, p_{11}	
13	p_{13}	*Node*'+', p_7, p_{12}	

Figure 6.5 Steps for constructing the DAG of Fig. 6.3

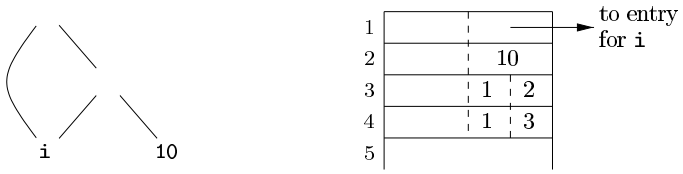
discussed above. We assume that *entry a* points to the symbol-table entry for `a`, and similarly for the other identifiers.

When the call to *Leaf*(**id**, *entry a*) is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4, i.e., $p_8 = p_3$ and $p_9 = p_4$. Hence, the node returned at step 10 must be the same as that returned at step 5, i.e., $p_{10} = p_5$. □

6.1.2 The Value-Number Method for Constructing DAG's

Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value, either a symbol-table pointer or a constant, in this case, and

interior nodes have two additional fields indicating the left and right children



a DAG b Array

Figure 6 6 Nodes of a DAG for i i 10 allocated in an array

In this array we refer to nodes by giving the integer index of the record for that node within the array This integer historically has been called the *value number* for the node or for the expression represented by the node For instance in Fig 6 6 the node labeled has value number 3 and its left and right children have value numbers 1 and 2 respectively In practice we could use pointers to records or references to objects instead of integer indexes but we shall still refer to the reference to a node as its value number If stored in an appropriate data structure value numbers help us construct expression DAG s e ciently the next algorithm shows how

Suppose that nodes are stored in an array as in Fig 6 6 and each node is referred to by its value number Let the *signature* of an interior node be the triple $\langle op\ l\ r \rangle$ where op is the label l its left child s value number and r its right child s value number A unary operator may be assumed to have r 0

Algorithm 6 3 The value number method for constructing the nodes of a DAG

INPUT Label op node l and node r

OUTPUT The value number of a node in the array with signature $\langle op\ l\ r \rangle$

METHOD Search the array for a node M with label op left child l and right child r If there is such a node return the value number of M If not create in the array a new node N with label op left child l and right child r and return its value number □

While Algorithm 6 3 yields the desired output searching the entire array every time we are asked to locate one node is expensive especially if the array holds expressions from an entire program A more e cient approach is to use a hash table in which the nodes are put into buckets each of which typically will have only a few nodes The hash table is one of several data structures that support *dictionaries* e ciently [1] A dictionary is an abstract data type that

[1] See Aho A V J E Hopcroft and J D Ullman *Data Structures and Algorithms* Addison Wesley 1983 for a discussion of data structures supporting dictionaries

allows us to insert and delete elements of a set and to determine whether a given element is currently in the set A good data structure for dictionaries such as a hash table performs each of these operations in time that is constant or close to constant independent of the size of the set

To construct a hash table for the nodes of a DAG we need a *hash function* h that computes the index of the bucket for a signature $\langle op\ l\ r\rangle$ in a way that distributes the signatures across buckets so that it is unlikely that any one bucket will get much more than a fair share of the nodes The bucket index $h\ op\ l\ r$ is computed deterministically from $op\ l$ and r so that we may repeat the calculation and always get to the same bucket index for node $\langle op\ l\ r\rangle$

The buckets can be implemented as linked lists as in Fig 6 7 An array indexed by hash value holds the *bucket headers* each of which points to the rst cell of a list Within the linked list for a bucket each cell holds the value number of one of the nodes that hash to that bucket That is node $\langle op\ l\ r\rangle$ can be found on the list whose header is at index $h\ op\ l\ r$ of the array

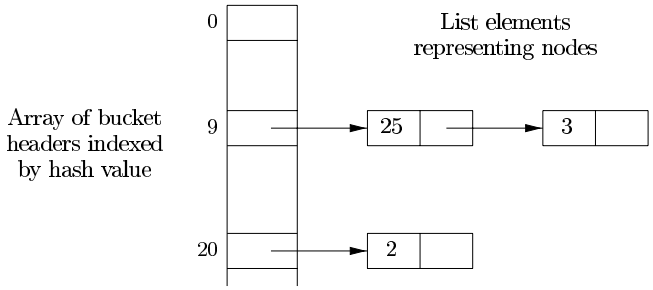


Figure 6 7 Data structure for searching buckets

Thus given the input node $op\ \ l$ and r we compute the bucket index $h\ op\ l\ r$ and search the list of cells in this bucket for the given input node Typically there are enough buckets so that no list has more than a few cells We may need to look at all the cells within a bucket however and for each value number v found in a cell we must check whether the signature $\langle op\ l\ r\rangle$ of the input node matches the node with value number v in the list of cells as in Fig 6 7 If we nd a match we return v If we nd no match we know no such node can exist in any other bucket so we create a new cell add it to the list of cells for bucket index $h\ op\ l\ r$ and return the value number in that new cell

6 1 3 Exercises for Section 6 1

Exercise 6 1 1 Construct the DAG for the expression

$$x\quad y\qquad x\quad y\qquad x\quad y\qquad\qquad x\quad y\qquad x\quad y$$

Exercise 6 1 2 Construct the DAG and identify the value numbers for the subexpressions of the following expressions assuming associates from the left

a $a \quad b \quad a \quad b$

b $a \quad b \quad a \quad b$

c $a \quad a \quad a \quad a \quad a \quad a \quad a \quad a \quad a$

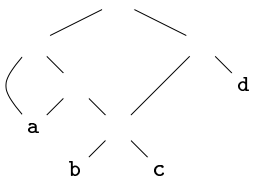
6 2 Three Address Code

In three address code there is at most one operator on the right side of an instruction that is no built up arithmetic expressions are permitted Thus a source language expression like `x y z` might be translated into the sequence of three address instructions

```
t1   y   z
t2   x   t1
```

where $\texttt{t}_1$ and $\texttt{t}_2$ are compiler generated temporary names This unraveling of multi operator arithmetic expressions and of nested ow of control statements makes three address code desirable for target code generation and optimization as discussed in Chapters 8 and 9 The use of names for the intermediate values computed by a program allows three address code to be rearranged easily

Example 6 4 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 DAG in Fig 6 3 is repeated in Fig 6 8 together with a correspond ing three address code sequence □



a DAG

```
t1   b   c
t2   a   t1
t3   a   t2
t4   t1   d
t5   t3   t4
```

b Three address code

Figure 6 8 A DAG and its corresponding three address code

6 2 1 Addresses and Instructions

Three address code is built from two concepts addresses and instructions In object oriented terms these concepts correspond to classes and the various kinds of addresses and instructions correspond to appropriate subclasses Al ternatively three address code can be implemented using records with elds for the addresses records called quadruples and triples are discussed brie y in Section 6 2 2

An address can be one of the following

A name For convenience we allow source program names to appear as addresses in three address code In an implementation a source name is replaced by a pointer to its symbol table entry where all information about the name is kept

A constant In practice a compiler must deal with many di erent types of constants and variables Type conversions within expressions are con sidered in Section 6 5 2

A compiler generated temporary It is useful especially in optimizing com pilers to create a distinct name each time a temporary is needed These temporaries can be combined if possible when registers are allocated to variables

We now consider the common three address instructions used in the rest of this book Symbolic labels will be used by instructions that alter the ow of control A symbolic label represents the index of a three address instruction in the sequence of instructions Actual indexes can be substituted for the labels either by making a separate pass or by backpatching discussed in Section 6 7 Here is a list of the common three address instruction forms

1 Assignment instructions of the form x y *op* z where *op* is a binary arithmetic or logical operation and x y and z are addresses

2 Assignments of the form x *op* y where *op* is a unary operation Essential unary operations include unary minus logical negation and conversion operators that for example convert an integer to a oating point number

3 *Copy instructions* of the form x y where x is assigned the value of y

4 An unconditional jump `goto` L The three address instruction with label L is the next to be executed

5 Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L These instructions execute the instruction with label L next if x is true and false respectively Otherwise the following three address instruction in sequence is executed next as usual

6 Conditional jumps such as `if` x *relop* y `goto` L which apply a relational operator etc to x and y and execute the instruction with label L next if x stands in relation *relop* to y If not the three address instruction following `if` x *relop* y `goto` L is executed next in sequence

7 Procedure calls and returns are implemented using the following instruc tions `param` x for parameters `call` p n and y `call` p n for procedure and function calls respectively and `return` y where y representing a returned value is optional Their typical use is as the sequence of three address instructions

```
param x1
param x2

param xn
call p n
```

generated as part of a call of the procedure p x_1 x_2 x_n The in teger n indicating the number of actual parameters in `call` p n is not redundant because calls can be nested That is some of the rst `param` statements could be parameters of a call that comes after p returns its value that value becomes another parameter of the later call The implementation of procedure calls is outlined in Section 6 9

8 Indexed copy instructions of the form x y i and x i y The instruc tion x y i sets x to the value in the location i memory units beyond location y The instruction x i y sets the contents of the location i units beyond x to the value of y

9 Address and pointer assignments of the form x y x y and x y The instruction x y sets the r value of x to be the location l value of y [2] Presumably y is a name perhaps a temporary that denotes an expression with an l value such as `A i j` and x is a pointer name or temporary In the instruction x y presumably y is a pointer or a temporary whose r value is a location The r value of x is made equal to the contents of that location Finally x y sets the r value of the object pointed to by x to the r value of y

Example 6 5 Consider the statement

```
do i   i 1  while  a i    v
```

Two possible translations of this statement are shown in Fig 6 9 The trans lation in Fig 6 9 a uses a symbolic label `L` attached to the rst instruction

[2] From Section 2 8 3 and values are appropriate on the left and right sides of assign ments respectively

The translation in b shows position numbers for the instructions starting arbitrarily at position 100 In both translations the last instruction is a conditional jump to the first instruction The multiplication `i 8` is appropriate for an array of elements that each take 8 units of space □

```
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

Figure 6 9 Two ways of assigning labels to three address statements

The choice of allowable operators is an important issue in the design of an intermediate form The operator set clearly must be rich enough to implement the operations in the source language Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine However if the front end must generate long sequences of instructions for some source language operations then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations

6 2 2 Quadruples

The description of three address instructions specifies the components of each type of instruction but it does not specify the representation of these instructions in a data structure In a compiler these instructions can be implemented as objects or as records with fields for the operator and the operands Three such representations are called quadruples triples and indirect triples

A *quadruple* or just *quad* has four fields which we call *op* arg_1 arg_2 and *result* The *op* field contains an internal code for the operator For instance the three address instruction x y z is represented by placing in *op* y in arg_1 z in arg_2 and x in *result* The following are some exceptions to this rule

1 Instructions with unary operators like x `minus` y or x y do not use arg_2 Note that for a copy statement like x y *op* is while for most other operations the assignment operator is implied

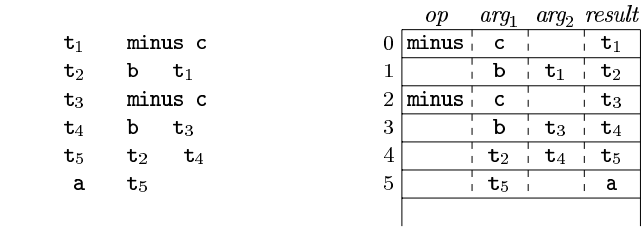
2 Operators like `param` use neither arg_2 nor *result*

3 Conditional and unconditional jumps put the target label in *result*

Example 6 6 Three address code for the assignment `a b c b c` appears in Fig 6 10 a The special operator `minus` is used to distinguish the

unary minus operator as in c from the binary minus operator as in b c Note that the unary minus three address statement has only two addresses as does the copy statement a t_5

The quadruples in Fig 6 10 b implement the three address code in a □



a Three address code b Quadruples

Figure 6 10 Three address code and its quadruple representation

For readability we use actual identi ers like a b and c in the elds arg_1 arg_2 and *result* in Fig 6 10 b instead of pointers to their symbol table entries Temporary names can either by entered into the symbol table like programmer de ned names or they can be implemented as objects of a class *Temp* with its own methods

6 2 3 Triples

A *triple* has only three elds which we call *op* arg_1 and arg_2 Note that the *result* eld in Fig 6 10 b is used primarily for temporary names Using triples we refer to the result of an operation $x\ op\ y$ by its position rather than by an explicit temporary name Thus instead of the temporary t_1 in Fig 6 10 b a triple representation would refer to position 0 Parenthesized numbers represent pointers into the triple structure itself In Section 6 1 2 positions or pointers to positions were called value numbers

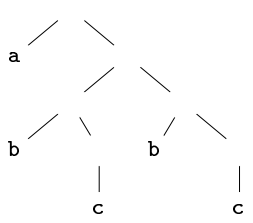
Triples are equivalent to signatures in Algorithm 6 3 Hence the DAG and triple representations of expressions are equivalent The equivalence ends with expressions since syntax tree variants and three address code represent control ow quite di erently

Example 6 7 The syntax tree and triples in Fig 6 11 correspond to the three address code and quadruples in Fig 6 10 In the triple representation in Fig 6 11 b the copy statement a t_5 is encoded in the triple representation by placing a in the arg_1 eld and 4 in the arg_2 eld □

A ternary operation like $x\ i\ y$ requires two entries in the triple structure for example we can put x and i in one triple and y in the next Similarly $x\ y\ i$ can implemented by treating it as if it were the two instructions

Why Do We Need Copy Instructions

A simple algorithm for translating expressions generates copy instructions for assignments as in Fig 6 10 a where we copy t_5 into a rather than assigning t_2 t_4 to a directly Each subexpression typically gets its own new temporary to hold its result and only when the assignment operator is processed do we learn where to put the value of the complete expression A code optimization pass perhaps using the DAG of Section 6 1 1 as an intermediate form can discover that t_5 can be replaced by a



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

Figure 6 11 Representations of a b c b c

t y i and x t where t is a compiler generated temporary Note that the temporary t does not actually appear in a triple since temporary values are referred to by their position in the triple structure

A bene t of quadruples over triples can be seen in an optimizing compiler where instructions are often moved around With quadruples if we move an instruction that computes a temporary t then the instructions that use t require no change With triples the result of an operation is referred to by its position so moving an instruction may require us to change all references to that result This problem does not occur with indirect triples which we consider next

Indirect triples consist of a listing of pointers to triples rather than a listing of triples themselves For example let us use an array *instruction* to list pointers to triples in the desired order Then the triples in Fig 6 11 b might be represented as in Fig 6 12

With indirect triples an optimizing compiler can move an instruction by reordering the *instruction* list without a ecting the triples themselves When implemented in Java an array of instruction objects is analogous to an indi rect triple representation since Java treats the array elements as references to objects

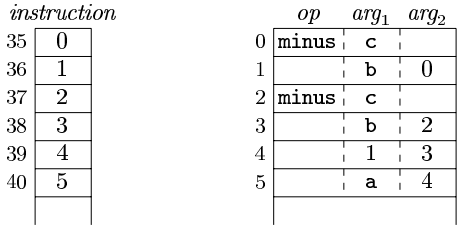


Figure 6 12 Indirect triples representation of three address code

6 2 4 Static Single Assignment Form

Static single assignment form SSA is an intermediate representation that fa cilitates certain code optimizations Two distinctive aspects distinguish SSA from three address code The rst is that all assignments in SSA are to vari ables with distinct names hence the term *static single assigment* Figure 6 13 shows the same intermediate program in three address code and in static single assignment form Note that subscripts distinguish each de nition of variables p and q in the SSA representation

```
p   a   b          p1  a   b
q   p   c          q1  p1  c
p   q   d          p2  q1  d
p   e   p          p3  e   p2
q   p   q          q2  p3  q1
```

a Three address code b Static single assignment form

Figure 6 13 Intermediate program in three address code and SSA

The same variable may be de ned in two di erent control ow paths in a program For example the source program

```
if  flag  x  1  else x  1
y  x  a
```

has two control ow paths in which the variable x gets de ned If we use di erent names for x in the true part and the false part of the conditional statement then which name should we use in the assignment y x a Here is where the second distinctive aspect of SSA comes into play SSA uses a notational convention called the function to combine the two de nitions of x

```
if  flag  x1  1  else x2  1
x3    x1 x2
```

Here x1 x2 has the value x1 if the control ow passes through the true part of the conditional and the value x2 if the control ow passes through the false part That is to say the function returns the value of its argument that corresponds to the control ow path that was taken to get to the assignment statement containing the function

6 2 5 Exercises for Section 6 2

Exercise 6 2 1 Translate the arithmetic expression *a* *b* *c* into

a A syntax tree

b Quadruples

c Triples

d Indirect triples

Exercise 6 2 2 Repeat Exercise 6 2 1 for the following assignment state ments

i `a b i c j`

ii `a i b c b d`

iii `x f y 1 2`

iv `x p y`

Exercise 6 2 3 Show how to transform a three address code sequence into one in which each de ned variable gets a unique variable name

6 3 Types and Declarations

The applications of types can be grouped under checking and translation

> *Type checking* uses logical rules to reason about the behavior of a program at run time Speci cally it ensures that the types of the operands match the type expected by an operator For example the operator in Java expects its two operands to be booleans the result is also of type boolean
>
> *Translation Applications* From the type of a name a compiler can de termine the storage that will be needed for that name at run time Type information is also needed to calculate the address denoted by an array reference to insert explicit type conversions and to choose the right ver sion of an arithmetic operator among other things

In this section we examine types and storage layout for names declared within a procedure or a class The actual storage for a procedure call or an object is allocated at run time when the procedure is called or the object is created As we examine local declarations at compile time we can however lay out *relative addresses* where the relative address of a name or a component of a data structure is an o set from the start of a data area

6 3 1 Type Expressions

Types have structure which we shall represent using *type expressions* a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression The sets of basic types and constructors depend on the language to be checked

Example 6 8 The array type `int 2 3` can be read as array of 2 arrays of 3 integers each and written as a type expression *array* 2 *array* 3 *integer* This type is represented by the tree in Fig 6 14 The operator *array* takes two parameters a number and a type □

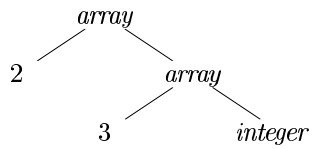


Figure 6 14 Type expression for `int 2 3`

We shall use the following de nition of type expressions

A basic type is a type expression Typical basic types for a language include *boolean char integer oat* and *void* the latter denotes the absence of a value

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 elds A type expression can be formed by applying the *record* type constructor to the eld names and their types Record types will be implemented in Section 6 3 6 by applying the constructor *record* to a symbol table containing entries for the elds

A type expression can be formed by using the type constructor for func tion types We write s t for function from type s to type t Function types will be useful when type checking is discussed in Section 6 5

Type Names and Recursive Types

Once a class is de ned its name can be used as a type name in C or Java for example consider `Node` in the program fragment

```
public class Node

public Node n
```

Names can be used to de ne recursive types which are needed for data structures such as linked lists The pseudocode for a list element

```
class Cell   int info  Cell next
```

de nes the recursive type `Cell` as a class that contains a eld `info` and a eld `next` of type `Cell` Similar recursive types can be de ned in C using records and pointers The techniques in this chapter carry over to recursive types

If s and t are type expressions then their Cartesian product $s \times t$ is a type expression Products are introduced for completeness they can be used to represent a list or tuple of types e g for function parameters We assume that $\times$ associates to the left and that it has higher precedence than $\rightarrow$

Type expressions may contain variables whose values are type expressions Compiler generated type variables will be used in Section 6 5 4

A convenient way to represent a type expression is to use a graph The value number method of Section 6 1 2 can be adapted to construct a dag for a type expression with interior nodes for type constructors and leaves for basic types type names and type variables for example see the tree in Fig 6 14 [3]

6 3 2 Type Equivalence

When are two type expressions equivalent Many type checking rules have the form **if** two type expressions are equal **then** return a certain type **else** error Potential ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression

[3] Since type names denote type expressions they can set up implicit cycles see the box on Type Names and Recursive Types If edges to type names are redirected to the type expressions denoted by the names then the resulting graph can have cycles due to recursive types

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

They are the same basic type.

They are formed by applying the same constructor to structurally equivalent types.

One is a type name that denotes the other.

If type names are treated as standing for themselves, then the first two conditions in the above definition lead to *name equivalence* of type expressions.

Name equivalent expressions are assigned the same value number, if we use Algorithm 6.3. Structural equivalence can be tested using the unification algorithm in Section 6.5.5.

6.3.3 Declarations

We shall study types and declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can be handled as discussed in Example 5.10. The grammar is

$$
\begin{array}{lll}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

The fragment of the above grammar that deals with basic and array types was used to illustrate inherited attributes in Section 5.3.2. The difference in this section is that we consider storage layout as well as types.

Nonterminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types. Nonterminal B generates one of the basic types **int** and **float**. Nonterminal C, for "component", generates strings of zero or more integers, each integer surrounded by brackets. An array type consists of a basic type specified by B, followed by array components specified by nonterminal C. A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

6.3.4 Storage Layout for Local Names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name. Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known, fixed amount of storage for a pointer to the data. Run-time storage management is discussed in Chapter 7.

Address Alignment

The storage layout for data objects is strongly in uenced by the address ing constraints of the target machine For example instructions to add integers may expect integers to be *aligned* that is placed at certain posi tions in memory such as an address divisible by 4 Although an array of ten characters needs only enough bytes to hold ten characters a compiler may therefore allocate 12 bytes the next multiple of 4 leaving 2 bytes unused Space left unused due to alignment considerations is referred to as *padding* When space is at a premium a compiler may *pack* data so that no padding is left additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned

Suppose that storage comes in blocks of contiguous bytes where a byte is the smallest unit of addressable memory Typically a byte is eight bits and some number of bytes form a machine word Multibyte objects are stored in consecutive bytes and given the address of the rst byte

The *width* of a type is the number of storage units needed for objects of that type A basic type such as a character integer or oat requires an integral number of bytes For easy access storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes [4]

The translation scheme SDT in Fig 6 15 computes types and their widths for basic and array types record types will be discussed in Section 6 3 6 The SDT uses synthesized attributes *type* and *width* for each nonterminal and two variables t and w to pass type and width information from a B node in a parse tree to the node for the production C In a syntax directed de nition t and w would be inherited attributes for C

The body of the T production consists of nonterminal B an action and nonterminal C which appears on the next line The action between B and C sets t to B *type* and w to B *width* If B **int** then B *type* is set to *integer* and B *width* is set to 4 the width of an integer Similarly if B **oat** then B *type* is *oat* and B *width* is 8 the width of a oat

The productions for C determine whether T generates a basic type or an array type If C then t becomes C *type* and w becomes C *width*

Otherwise C speci es an array component The action for C **num** C_1 forms C *type* by applying the type constructor *array* to the operands **num** *value* and C_1 *type* For instance the result of applying *array* might be a tree structure such as Fig 6 14

[4] Storage allocation for pointers in C and C is simpler if all pointers have the same width The reason is that the storage for a pointer may need to be allocated before we learn the type of the objects it can point to

T	B	{ t = 0; $B.type$ = w = $B.width$ }
	C	{ $T.type$ = $C.type$; $T.width$ = $C.width$ }
B	**int**	{ $B.type$ = *integer*; $B.width$ = 4 }
B	**oat**	{ $B.type$ = *oat*; $B.width$ = 8 }
C		{ $C.type$ = t; $C.width$ = w }
C	**num** C_1	{ $C.type$ = *array*(**num**.*value*, $C_1.type$); $C.width$ = **num**.*value* × $C_1.width$ }

Figure 6.15 Computing types and their widths

The width of an array is obtained by multiplying the width of an element by the number of elements in the array. If addresses of consecutive integers differ by 4, then address calculations for an array of integers will include multiplications by 4. Such multiplications provide opportunities for optimization, so it is helpful for the front end to make them explicit. In this chapter, we ignore other machine dependencies such as the alignment of data objects on word boundaries.

Example 6.9 The parse tree for the type `int[2][3]` is shown by dotted lines in Fig. 6.16. The solid lines show how the type and width are passed from B down the chain of C's through variables t and w, and then back up the chain as synthesized attributes *type* and *width*. The variables t and w are assigned the values of $B.type$ and $B.width$, respectively, before the subtree with the C nodes is examined. The values of t and w are used at the node for C to start the evaluation of the synthesized attributes up the chain of C nodes. □

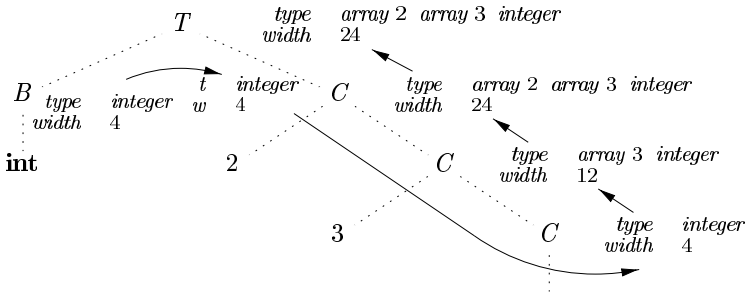


Figure 6.16 Syntax-directed translation of array types

6 3 5 Sequences of Declarations

Languages such as C and Java allow all the declarations in a single procedure to be processed as a group The declarations may be distributed within a Java procedure but they can still be processed when the procedure is analyzed Therefore we can use a variable say *o set* to keep track of the next available relative address

The translation scheme of Fig 6 17 deals with a sequence of declarations of the form T **id** where T generates a type as in Fig 6 15 Before the rst declaration is considered *o set* 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 *o set* which is then incremented by the width of the type of x

P		{ *o set* 0 }
	D	
D	T **id**	{ *top put* **id** *lexeme* T *type* *o set*
		o set *o set* T *width* }
	D_1	
D		

Figure 6 17 Computing the relative addresses of declared names

The semantic action within the production D T **id** D_1 creates a symbol table entry by executing *top put* **id** *lexeme* T *type* *o set* 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 *o set* in its data area

The initialization of *o set* in Fig 6 17 is more evident if the rst production appears on one line as

$$P \quad \{ o\ set \quad 0 \}\ D \qquad 6\,1$$

Nonterminals generating called marker nonterminals can be used to rewrite productions so that all actions appear at the ends of right sides see Sec tion 5 5 4 Using a marker nonterminal M 6 1 can be restated as

P	$M\ D$	
M		{ *o set* 0 }

6 3 6 Fields in Records and Classes

The translation of declarations in Fig 6 17 carries over to elds in records and classes Record types can be added to the grammar in Fig 6 15 by adding the following production

$$T \quad \textbf{record}\ ''\ D\ ''$$

The fields in this record type are specified by the sequence of declarations generated by D. The approach of Fig. 6.17 can be used to determine the types and relative addresses of fields, provided we are careful about two things:

- The field names within a record must be distinct; that is, a name may appear at most once in the declarations generated by D.
- The offset or relative address for a field name is relative to the data area for that record.

Example 6.10 The use of a name x for a field within a record does not conflict with other uses of the name outside the record. Thus, the three uses of `x` in the following declarations are distinct and do not conflict with each other:

```
float x
record   float x  float y    p
record   int tag  float x  float y    q
```

A subsequent assignment `x = p.x + q.x` sets variable `x` to the sum of the fields named `x` in the records `p` and `q`. Note that the relative address of `x` in `p` differs from the relative address of `x` in `q`. □

For convenience, record types will encode both the types and relative addresses of their fields, using a symbol table for the record type. A record type has the form *record*(t), where *record* is the type constructor, and t is a symbol table object that holds information about the fields of this record type.

The translation scheme in Fig. 6.18 consists of a single production to be added to the productions for T in Fig. 6.15. This production has two semantic actions. The embedded action before D saves the existing symbol table, denoted by *top*, and sets *top* to a fresh symbol table. It also saves the current *offset* and sets *offset* to 0. The declarations generated by D will result in types and relative addresses being put in the fresh symbol table. The action after D creates a record type using *top*, before restoring the saved symbol table and offset.

$T \rightarrow$ **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*(); }

Figure 6.18: Handling of field names in records

For concreteness, the actions in Fig. 6.18 give pseudocode for a specific implementation. Let class *Env* implement symbol tables. The call *Env.push*(*top*) pushes the current symbol table, denoted by *top*, onto a stack. Variable *top* is then set to a new symbol table. Similarly, *offset* is pushed onto a stack called *Stack*. Variable *offset* is then set to 0.

After the declarations in D have been translated the symbol table *top* holds the types and relative addresses of the elds in this record Further o *set* gives the storage needed for all the elds The second action sets T *type* to *record top* and T *width* to o *set* Variables *top* and o *set* are then restored to their pushed values to complete the translation of this record type

This discussion of storage for record types carries over to classes since no storage is reserved for methods See Exercise 6 3 2

6 3 7 Exercises for Section 6 3

Exercise 6 3 1 Determine the types and relative addresses for the identi ers in the following sequence of declarations

```
float x
record   float x  float y    p
record   int tag  float x  float y    q
```

Exercise 6 3 2 Extend the handling of eld names in Fig 6 18 to classes and single inheritance class hierarchies

a Give an implementation of class *Env* that allows linked symbol tables so that a subclass can either rede ne a eld name or refer directly to a eld name in a superclass

b Give a translation scheme that allocates a contiguous data area for the elds in a class including inherited elds Inherited elds must maintain the relative addresses they were assigned in the layout for the superclass

6 4 Translation of Expressions

The rest of this chapter explores issues that arise during the translation of ex pressions and statements We begin in this section with the translation of ex pressions into three address code An expression with more than one operator like a b c will translate into instructions with at most one operator per in struction An array reference $A\,i\,j$ will expand into a sequence of three address instructions that calculate an address for the reference We shall consider type checking of expressions in Section 6 5 and the use of boolean expressions to direct the ow of control through a program in Section 6 6

6 4 1 Operations Within Expressions

The syntax directed de nition in Fig 6 19 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

PRODUCTION	SEMANTIC RULES
$S \to E$	$S.code = E.code \,\|\|$ *gen top get lexeme* $E.addr$
$E \to E_1 + E_2$	$E.addr =$ *Temp* $E.code = E_1.code \,\|\| E_2.code \,\|\|$ *gen* $E.addr$ ' = ' $E_1.addr$ ' + ' $E_2.addr$
$\mid\ (E_1)$	$E.addr =$ *Temp* $E.code = E_1.code \,\|\|$ *gen* $E.addr$ ' = ' $E_1.addr$
$\mid\ -E_1$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid$ **id**	$E.addr =$ *top get lexeme* **id**.*lexeme* $E.code = ''$

Figure 6 19 Three address code for expressions

hold the value of E Recall from Section 6 2 1 that an address can be a name a constant or a compiler generated temporary

Consider the last production $E \to$ **id** in the syntax directed de nition in Fig 6 19 When an expression is a single identi er say x then x itself holds the value of the expression The semantic rules for this production de ne $E.addr$ to point to the symbol table entry for this instance of **id** Let *top* denote the current symbol table Function *top get* retrieves the entry when it is applied to the string representation **id**.*lexeme* of this instance of **id** $E.code$ is set to the empty string

When $E \to (E_1)$ the translation of E is the same as that of the subex pression E_1 Hence $E.addr$ equals $E_1.addr$ and $E.code$ equals $E_1.code$

The operators and unary in Fig 6 19 are representative of the operators in a typical language The semantic rules for $E \to E_1 + E_2$ generate code to compute the value of E from the values of E_1 and E_2 Values are computed into newly generated temporary names If E_1 is computed into $E_1.addr$ and E_2 into $E_2.addr$ then $E_1 + E_2$ translates into $t = E_1.addr + E_2.addr$ where t is a new temporary name $E.addr$ is set to t A sequence of distinct temporary names $t_1, t_2, \ldots$ is created by successively executing **new** *Temp*

For convenience we use the notation *gen* x ' = ' y ' + ' z to represent the three address instruction $x = y + z$ Expressions appearing in place of variables like x, y, and z are evaluated when passed to *gen* and quoted strings like ' = ' are taken literally [5] Other three address instructions will be built up similarly

[5] In syntax directed de nitions *gen* builds an instruction and returns it In translation schemes *gen* builds an instruction and incrementally emits it by putting it into the stream

by applying *gen* to a combination of expressions and strings

When we translate the production $E \quad E_1 \quad E_2$ the semantic rules in Fig 6 19 build up E *code* by concatenating E_1 *code* E_2 *code* and an instruc tion that adds the values of E_1 and E_2 The instruction puts the result of the addition into a new temporary name for E denoted by E *addr*

The translation of $E \quad E_1$ is similar The rules create a new temporary for E and generate an instruction to perform the unary minus operation

Finally the production $S \quad$ **id** E generates instructions that assign the value of expression E to the identi er **id** The semantic rule for this production uses function *top get* to determine the address of the identi er represented by **id** as in the rules for $E \quad$ **id** S *code* consists of the instructions to compute the value of E into an address given by E *addr* followed by an assignment to the address *top get* **id** *lexeme* for this instance of **id**

Example 6 11 The syntax directed de nition in Fig 6 19 translates the as signment statement `a b c` into the three address code sequence

```
t1   minus c
t2   b   t1
a   t2
```

□

6 4 2 Incremental Translation

Code attributes can be long strings so they are usually generated incremen tally as discussed in Section 5 5 2 Thus instead of building up E *code* as in Fig 6 19 we can arrange to generate only the new three address instructions as in the translation scheme of Fig 6 20 In the incremental approach *gen* not only constructs a three address instruction it appends the instruction to the sequence of instructions generated so far The sequence may either be retained in memory for further processing or it may be output incrementally

The translation scheme in Fig 6 20 generates the same code as the syntax directed de nition in Fig 6 19 With the incremental approach the *code* at tribute is not used since there is a single sequence of instructions that is created by successive calls to *gen* For example the semantic rule for $E \quad E_1 \quad E_2$ in Fig 6 20 simply calls *gen* to generate an add instruction the instructions to compute E_1 into E_1 *addr* and E_2 into E_2 *addr* have already been generated

The approach of Fig 6 20 can also be used to build a syntax tree The new semantic action for $E \quad E_1 \quad E_2$ creates a node by using a constructor as in

$$E \quad E_1 \quad E_2 \quad \{ \; E \; addr \quad \textbf{new} \; Node' \; ' \; E_1 \; addr \; E_2 \; addr \; \}$$

Here attribute *addr* represents the address of a node rather than a variable or constant

---

of generated instructions

S **id** E { *gen top get* **id** *lexeme* ' ' E *addr* }

E E_1 E_2 { E *addr* **new** *Temp*
gen E *addr* ' ' E_1 *addr* ' ' E_2 *addr* }

| E_1 { E *addr* **new** *Temp*
gen E *addr* ' ' **minus**' E_1 *addr* }

| E_1 { E *addr* E_1 *addr* }

| **id** { E *addr* *top get* **id** *lexeme* }

Figure 6 20 Generating three address code for expressions incrementally

6 4 3 Addressing Array Elements

Array elements can be accessed quickly if they are stored in a block of consecu tive locations In C and Java array elements are numbered 0 1 n 1 for an array with n elements If the width of each array element is w then the ith element of array A begins in location

$$\textit{base} \quad i \quad w \qquad 6\,2$$

where *base* is the relative address of the storage allocated for the array That is *base* is the relative address of A 0

The formula 6 2 generalizes to two or more dimensions In two dimensions let us write $A\,i_1\ i_2$ as in C for element i_2 in row i_1 Let w_1 be the width of a row and let w_2 be the width of an element in a row The relative address of $A\,i_1\ i_2$ can then be calculated by the formula

$$\textit{base} \quad i_1 \quad w_1 \quad i_2 \quad w_2 \qquad 6\,3$$

In k dimensions the formula is

$$\textit{base} \quad i_1 \quad w_1 \quad i_2 \quad w_2 \qquad i_k \quad w_k \qquad 6\,4$$

where w_j for 1 j k is the generalization of w_1 and w_2 in 6 3

Alternatively the relative address of an array reference can be calculated in terms of the numbers of elements n_j along dimension j of the array and the width w w_k of a single element of the array In two dimensions i e k 2 and w w_2 the location for $A\,i_1\ i_2$ is given by

$$\textit{base} \quad i_1 \quad n_2 \quad i_2 \quad w \qquad 6\,5$$

In k dimensions the following formula calculates the same address as 6 4

$$\textit{base} \qquad i_1 \quad n_2 \quad i_2 \quad n_3 \quad i_3 \qquad n_k \quad i_k \quad w \qquad 6\,6$$

More generally array elements need not be numbered starting at 0 In a one dimensional array the array elements are numbered *low low* 1 *high* and *base* is the relative address of A *low* Formula 6 2 for the address of $A\,i$ is replaced by

$$base \quad i \quad low \quad w \qquad 6\,7$$

The expressions 6 2 and 6 7 can be both be rewritten as $i \quad w \quad c$ where the subexpression $c \quad base \quad low \quad w$ can be precalculated at compile time Note that $c \quad base$ when *low* is 0 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 $i \quad w$ to c

Compile time precalculation can also be applied to address calculations for elements of multidimensional arrays see Exercise 6 4 5 However there is one situation where we cannot use compile time precalculation when the array s size is dynamic If we do not know the values of *low* and *high* or their gen eralizations in many dimensions at compile time then we cannot compute constants such as c Then formulas like 6 7 must be evaluated as they are written when the program executes

The above address calculations are based on row major layout for arrays which is used in C for example A two dimensional array is normally stored in one of two forms either *row major* row by row or *column major* column by column Figure 6 21 shows the layout of a 2 3 array A in a row major form and b column major form Column major form is used in the Fortran family of languages

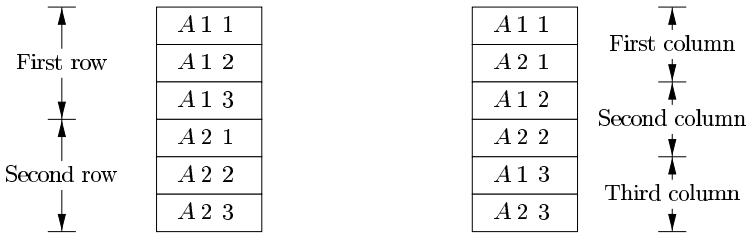


Figure 6 21 Layouts for a two dimensional array

We can generalize row or column major form to many dimensions The generalization of row major form is to store the elements in such a way that as we scan down a block of storage the rightmost subscripts appear to vary fastest like the numbers on an odometer Column major form generalizes to the opposite arrangement with the leftmost subscripts varying fastest

6 4 4 Translation of Array References

The chief problem in generating code for array references is to relate the address calculation formulas in Section 6 4 3 to a grammar for array references Let nonterminal L generate an array name followed by a sequence of index expressions

$$L \quad L \quad E \quad | \ \textbf{id} \quad E$$

As in C and Java assume that the lowest numbered array element is 0 Let us calculate addresses based on widths using the formula 6 4 rather than on numbers of elements as in 6 6 The translation scheme in Fig 6 22 generates three address code for expressions with array references It consists of the productions and semantic actions from Fig 6 20 together with productions involving nonterminal L

$S \quad \textbf{id} \quad E \quad \{ \ gen \ top.get(\textbf{id}.lexeme) \ ' \ ' \ E.addr \ \}$

$| \quad L \quad E \quad \{ \ gen \ L.array.base \ ' \ ' \ L.addr \ ' \ ' \ ' \ ' \ E.addr \ \}$

$E \quad E_1 \quad E_2 \quad \{ \ E.addr = \textbf{new} \ Temp()$
$gen(E.addr \ ' \ ' \ E_1.addr \ ' \ ' \ E_2.addr) \ \}$

$| \quad \textbf{id} \quad \{ \ E.addr = top.get(\textbf{id}.lexeme) \ \}$

$| \quad L \quad \{ \ E.addr = \textbf{new} \ Temp()$
$gen(E.addr \ ' \ ' \ L.array.base \ ' \ ' \ L.addr \ ' \ ' \ ) \ \}$

$L \quad \textbf{id} \quad E \quad \{ \ L.array = top.get(\textbf{id}.lexeme)$
$L.type = L.array.type.elem$
$L.addr = \textbf{new} \ Temp()$
$gen(L.addr \ ' \ ' \ E.addr \ ' \ ' \ L.type.width) \ \}$

$| \quad L_1 \quad E \quad \{ \ L.array = L_1.array$
$L.type = L_1.type.elem$
$t = \textbf{new} \ Temp()$
$L.addr = \textbf{new} \ Temp()$
$gen(t \ ' \ ' \ E.addr \ ' \ ' \ L.type.width)$
$gen(L.addr \ ' \ ' \ L_1.addr \ ' \ ' \ t) \ \}$

Figure 6 22 Semantic actions for array references

Nonterminal L has three synthesized attributes

1 $L.addr$ denotes a temporary that is used while computing the o set for the array reference by summing the terms $i_j \quad w_j$ in 6 4

2 L *array* is a pointer to the symbol table entry for the array name The base address of the array say L *array base* is used to determine the actual l value of an array reference after all the index expressions are analyzed

3 L *type* is the type of the subarray generated by L For any type t we assume that its width is given by t *width* We use types as attributes rather than widths since types are needed anyway for type checking For any array type t suppose that t *elem* gives the element type

The production S **id** E represents an assignment to a nonarray vari able which is handled as usual The semantic action for S L E generates an indexed copy instruction to assign the value denoted by expression E to the location denoted by the array reference L Recall that attribute L *array* gives the symbol table entry for the array The array s base address the address of its 0th element is given by L *array base* Attribute L *addr* denotes the temporary that holds the o set for the array reference generated by L The location for the array reference is therefore L *array base* L *addr* The generated instruction copies the r value from address E *addr* into the location for L

Productions E E_1 E_2 and E **id** are the same as before The se mantic action for the new production E L generates code to copy the value from the location denoted by L into a new temporary This location is L *array base* L *addr* as discussed above for the production S L E Again attribute L *array* gives the array name and L *array base* gives its base address Attribute L *addr* denotes the temporary that holds the o set The code for the array reference places the r value at the location designated by the base and o set into a new temporary denoted by E *addr*

Example 6 12 Let `a` denote a 2 3 array of integers and let `c i` and `j` all denote integers Then the type of `a` is *array* 2 *array* 3 *integer* Its width w is 24 assuming that the width of an integer is 4 The type of `a i` is *array* 3 *integer* of width w_1 12 The type of `a i j` is *integer*

An annotated parse tree for the expression `c a i j` is shown in Fig 6 23 The expression is translated into the sequence of three address instructions in Fig 6 24 As usual we have used the name of each identi er to refer to its symbol table entry □

6 4 5 Exercises for Section 6 4

Exercise 6 4 1 Add to the translation of Fig 6 19 rules for the following productions

a E E_1 E_2

b E E_1 unary plus

Exercise 6 4 2 Repeat Exercise 6 4 1 for the incremental translation of Fig 6 20

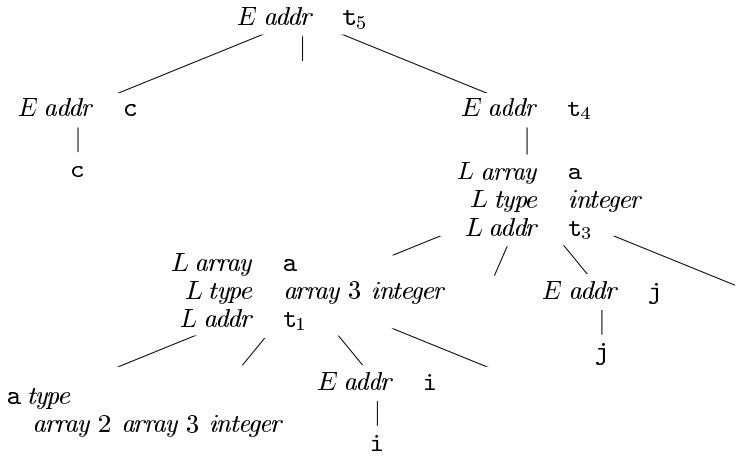


Figure 6 23 Annotated parse tree for `c a i j`

```
t1   i   12
t2   j   4
t3   t1   t2
t4   a   t3
t5   c   t4
```

Figure 6 24 Three address code for expression `c a i j`

Exercise 6 4 3 Use the translation of Fig 6 22 to translate the following assignments

a `x a i b j`

b `x a i j b i j`

c `x a b i j c k`

Exercise 6 4 4 Revise the translation of Fig 6 22 for array references of the Fortran style that is **id** E_1 E_2 E_n for an n dimensional array

Exercise 6 4 5 Generalize formula 6 7 to multidimensional arrays and indicate what values can be stored in the symbol table and used to compute o sets Consider the following cases

a An array A of two dimensions in row major form The rst dimension has indexes running from l_1 to h_1 and the second dimension has indexes from l_2 to h_2 The width of a single array element is w

Symbolic Type Widths

The intermediate code should be relatively independent of the target ma chine so the optimizer does not have to change much if the code generator is replaced by one for a di erent machine However as we have described the calculation of type widths an assumption regarding basic types is built into the translation scheme For instance Example 6 12 assumes that each element of an integer array takes four bytes Some intermediate codes e g P code for Pascal leave it to the code generator to ll in the size of array elements so the intermediate code is independent of the size of a machine word We could have done the same in our translation scheme if we replaced 4 as the width of an integer by a symbolic constant

b The same as a but with the array stored in column major form

c An array A of k dimensions stored in row major form with elements of size w The jth dimension has indexes running from l_j to h_j

d The same as c but with the array stored in column major form

Exercise 6 4 6 An integer array $A\,i\;j$ stored row major has index i ranging from 1 to 10 and index j ranging from 1 to 20 Integers take 4 bytes each Suppose array A is stored starting at byte 0 Find the location of

a $A\,4\;5$ b $A\,10\;8$ c $A\,3\;17$

Exercise 6 4 7 Repeat Exercise 6 4 6 if A is stored in column major order

Exercise 6 4 8 A real array $A\,i\;j\;k$ has index i ranging from 1 to 4 j ranging from 0 to 4 and k ranging from 5 to 10 Reals take 8 bytes each If A is stored row major starting at byte 0 nd the location of

a $A\,3\;4\;5$ b $A\,1\;2\;7$ c $A\,4\;3\;9$

Exercise 6 4 9 Repeat Exercise 6 4 8 if A is stored in column major order

6 5 Type Checking

To do *type checking* a compiler needs to assign a type expression to each com ponent of the source program The compiler must then determine that these type expressions conform to a collection of logical rules that is called the *type system* for the source language

Type checking has the potential for catching errors in programs In principle any check can be done dynamically if the target code carries the type of an