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 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.
* The procedure enter(name, type, offset) creates a symbol-table entry for name, gives its type type and relative address offset in its dataarea.
* Attribute type represents a type expression constructed from the basic types integer and real by applying the type constructors pointer andarray. If type expressions are represented by graphs, then attribute type might be a pointer to the node representing a 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 each pointer is assumed to be4.

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]ofT1 { T.type : = array(num.val,T1.type);
T.width : = num.val X T1.width }

T→↑T1 { T.type : = pointer (T1.type);
T.width : = 4}

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:D

D:D; D |id: T |proc id; D ; S

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

For example, consider the symbol tables for procedures read array, 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.

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.
3. Add width(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 thisprocedure name.

Syntax directed translation scheme for nested procedures

P→MD	{ addwidth (top(tblptr) , top (offset)); pop (tblptr); pop (offset) }
M→ε	{ t : = mktable(nil); push (t,tblptr); push (0,offset) }
D→D1 ; D2	
D→proc id ; ND1; 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)}

* 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 sub trees for B and Cin

  A→BC{actionA}

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

The action for non terminal M initializes stack tblptr with a outermost scope, created by operation mktable(nil). The action also pushes relative address 0 onto stack offset. Similarly, the non terminal 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. 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 add width. Stacks tbl ptr 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□M D
M□ε
D□D ; D | id: T | proc id; N D ; S N□ε

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

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 }

INTERMEDIATE LANGUAGES

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.

Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A dag (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement a : = b * - c + b * - c are shown in Fig.3.2:

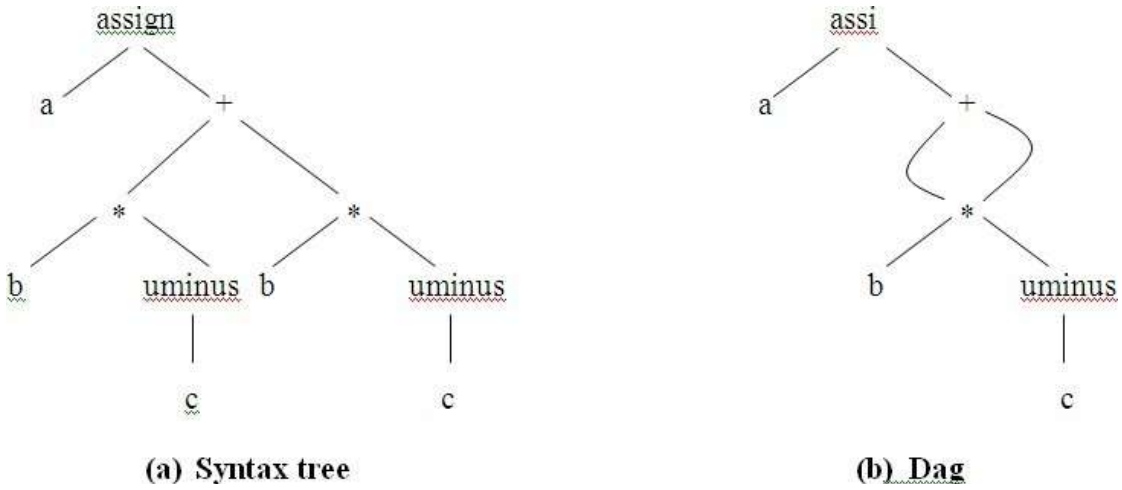
Postfix notation:

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 given above is

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.



(a) Syntax tree **(b) Dag**

Fig. 3.2 Graphical representation of a:=b*- c+b*- c

PRODUCTION	SEMANTIC RULE
$S \rightarrow id := E$	S.nptr := mknode('assign', mkleaf(id, id.place), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr := mknode('+', E_1.nptr, E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr := mknode('*', E_1.nptr, E_2.nptr)
$E \rightarrow -E_1$	E.nptr := mknode('uminus', E_1.nptr)
$E \rightarrow (E_1)$	E.nptr := E_1.nptr
$E \rightarrow id$	E.nptr := mkleaf(id, id.place)

Fig. Syntax-directed definition to produce syntax trees for assignment statements

The token id has an attribute place that points to the symbol-table entry for the to an identifier. A symbol-table entry can be found from an attribute id.name, representing the lexeme associated with that occurrence of id. If the lexical analyzer holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes

are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position10.

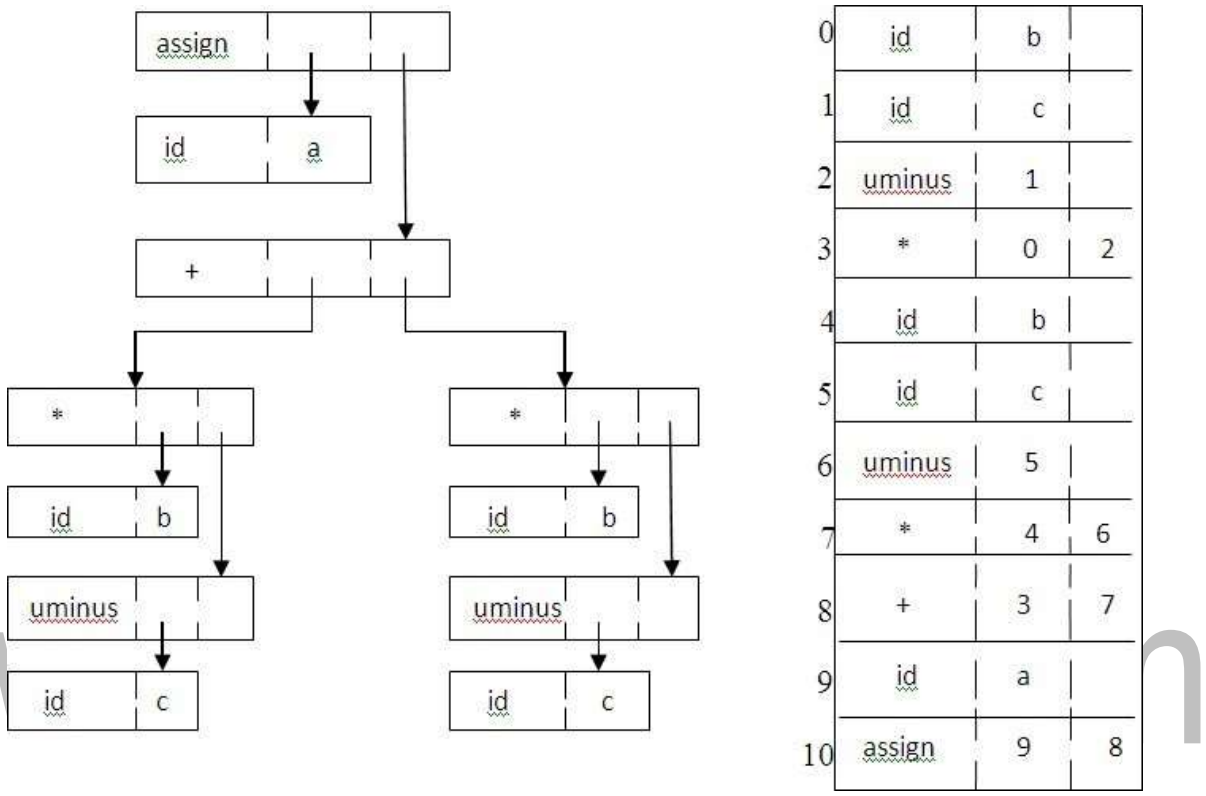


Fig. Two representations of the syntax tree

Three-address code

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

= y opz

wherex,yandzarenames,constants,orcompiler-generatedtemporaries;opstandsforanyoperator, such as a fixed-orfloating-point arithmeticoperator, or a logical operator on boolean-valued data.

Thus a source language expression like x+ y*z might be translated into a sequence

t1 : = y*z

t2 : = x+t1

where t1 and t2 are compiler-generated temporary names.

Advantages of three-address code:

* The unraveling of complicated arithmetic expressions and of 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.

```
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 dag

Fig.3.5 Three-address code corresponding to the syntax tree and dag

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.

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.
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 as in the usual sequence.

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
paramx2
.
.
param xn
callp,n
```

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

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

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

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.

Given input a : = b * - c + b * - c, the three-address code is as shown in Fig. 8.3a. The synthesized attribute S.code represents the three-address code for the assignment S.

The nonterminal E has two attributes :

1. E.place, the name that will hold the value of E ,and
2. E.code, the sequence of three-address statements evaluating E.

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 := ''

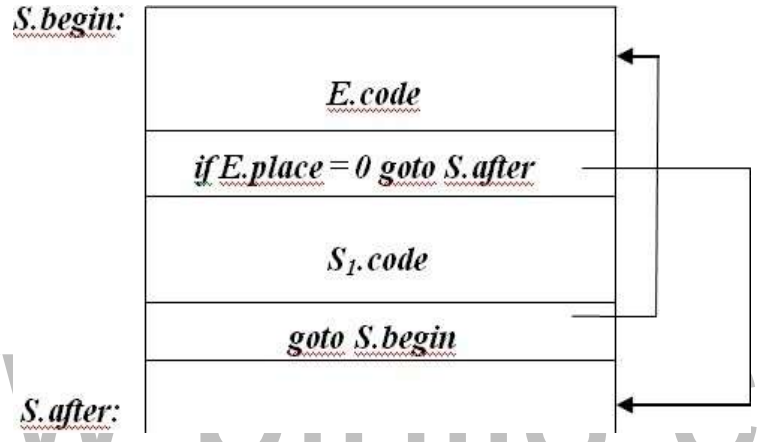


Fig. Semantic rules generating code for a while statement

PRODUCTION	SEMANTICRULES
S→whileEdoS1	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 ':')

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.
- 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.

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

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

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

Fig.(a)Quadruples **(b)Triples**

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.

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.

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

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

Fig. 3.8 (a) x[i] := y

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

(b) x := y[i]

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:

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

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

Fig. Indirect triples representation of three-address statements

RUN-TIME ENVIRONMENTS - SOURCE LANGUAGE ISSUES

Procedures:

A procedure definition is a declaration that associates an identifier with a statement. The identifier is the procedure name, and the statement is the procedure body. For example, the following is the definition of procedure named read array:

```
1. procedure read array; vari : integer;
2. begin
                for i : = 1 to 9 do read(a[i])
        end;
```

When a procedure name appears within an executable statement, the procedure is said to be called at that point.

Activation trees:

An activation tree is used to depict the way control enters and leaves activations. In an activation tree,

3. Each node represents an activation of aprocedure.
4. The root represents the activation of themain program.
5. The node for a is the parent of the node for b if and only if control flows from activation a to b.
6. The node for a is to the left of the node for b if and only if the lifetime of aoccurs before the lifetime ofb.

Control stack:

A control stack is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends. The contents of the control stack are related to paths to the root of the activation tree. When node n is at the top of control stack, the

stack contains the nodes along the path from n to the root.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a n Declarations may be explicit, such as:

vari : integer ;

or they may be implicit. Example, any variable name starting with I is as summed to denote an integer. The portion of the program to which a declaration applies is called the scope of that declaration.

Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. "Data object" corresponds to a storage location that holds values. The term environment refers to a function that maps a name to a storage location. The term state refers to a function that maps a storage location to the value held there. When an environment associates storage location s with a name x, we say that x is bound to s. This association is referred to as a binding of x.

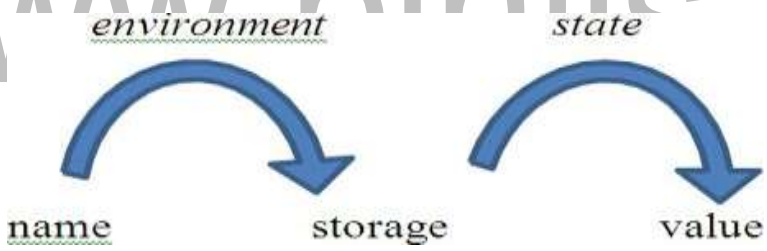


Fig. 2.8 Two-stage mapping from names to values

SPECIFICATION OF A SIMPLE TYPECHECKER

A type checker for a simple language checks the type of each identifier. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E
D → D ; D
D → id:T { addtype(id.entry ,T.type) }
T→char { T.type: = char}
T→integer { T.type: = integer}
T → ↑T1 { T.type: = pointer(T1.type)}
T → array [num] of T1 { T.type : = array (1… num.val , T1.type) }

In the above language,

→ There are two basic types : char and integer ; → type_error is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example , ↑ integer leads to the type expression pointer (integer).

Type checking of expressions

In the following rules, the attribute type for E gives the type expression assigned to the expression generated by E.

1. E → literal { E.type: = char }
E→num { E.type: = integer}

Here, constants represented by the tokens literal and num have type char and integer.

2. E→id { E.type: = lookup (id.entry)}

lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E1 mod E2 {E.type: = if E1. type = integer and E2. type = integer then

integer else
type_error}

Theexpressionformedbyapplyingthemodoperatortotwosubexpressionsoftypeintegerhast
ype integer; otherwise, its type istype_error.

4. E → E1 [E2] { E.type: = if E2.type = integerand
E1.type
=
array(s,t)
then t
else
type_err
or }

In an array reference E1 [E2] , the index expression E2 must have type integer. The result is the element type t obtained from the type array(s,t) of E1.

5. E → E1 ↑ {E.type:=ifE1.type=
pointer(t)thent else
type_error}

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

Type checking of statements

Statements do not have values; hence the basic type void can be assigned to them. If an error is detected within a statement, then type_erroris assigned.

Translation scheme for checking the type of statements:

1. Assignmentstatement:

S→id:=E { S.type: = if id.type= E.typethenvoid
else type_error}

2. Conditionalstatement:

S→if EthenS1 { S.type: = if E.type= booleanthenS1.type
else type_error }

3. Whilestatement:

S → while EdoS1 { S.type: = if E.type= booleanthenS1.type
else type_error }

4. Sequence ofstatements:

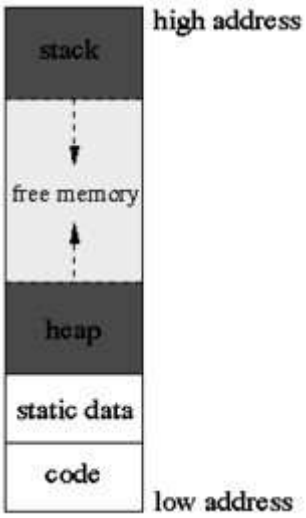
S → S1;S2 { S.type: = if S1.type = voidand
S1.type = void then void else type_error }

Type checking of functions

The rule for checking the type of a function application is : E → E1 (E2) { E.type: = if E2.type = s and
E1.type = s → t then t
else type_error}

STORAGE ORGANIZATION

- An executable program generated by a compiler will have the following organization in memory on a typical architecture (such as on MIPS):



- This is the layout in memory of an executable program.
- Note that in a virtual memory architecture (which is the case for any modern operating system), some parts of the memory layout may in fact be located on disk blocks and they are retrieved in memory by demand (lazily).
- The machine code of the program is typically located at the lowest part of the layout.
- Then, after the code, there is a section to keep all the fixed size static data in the program.
- The dynamically allocated data (ie. the data created using malloc in C) as well as the static data without a fixed size (such as arrays of variable size) are created and kept in the heap. The heap grows from low to high addresses.
- When you call malloc in C to create a dynamically allocated structure, the program tries to find an empty place in the heap with sufficient space to insert the new data; if it can't do that, it puts the data at the end of the heap and increases the heap size.
- The focus of this section is the stack in the memory layout. It is called the run-time stack.
- The stack, in contrast to the heap, grows in the opposite direction (upside-down): from high to low addresses, which is a bit counterintuitive. The stack is not only used to push the return

address when a function is called, but it is also used for allocating some of the local variables of a function during the function call, as well as for some bookkeeping.

Activate Record

- It is used to store the current record and the record is been stored in the stack.
- It contains return value .After the execution the value is been return.
- It can be called as return value.

Parameter

- It specifies the number of parameters used in functions.

Local Data

- The data that is been used inside the function is called as local address

Temporary Data

- It is used to store the data in temporary variables.

Links

- It specifies the additional links that are required by the program.

Status

- It specifies the status of program that is the flag used.

TYPECHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking, called static checking, detects and reports programming errors.

Some examples of static checks:

1. Type checks - A compiler should report an error if an operator is applied to an incompatible operand.Example: Ifanarrayvariableandfunctionvariableareaddedtogether.

2. Flow-of-control checks - Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An enclosing statement, such as break, does not exist in switchstatement.

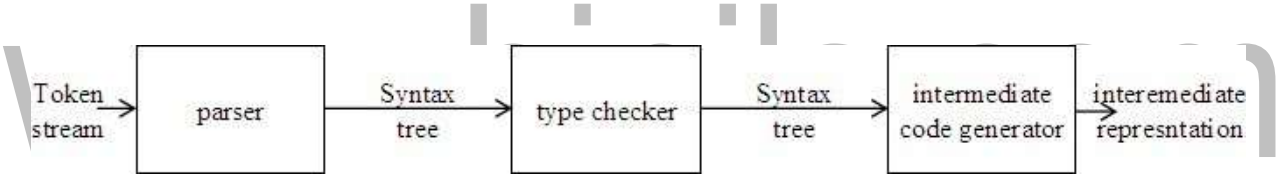


Fig. Position of type checker

A typechecker verifies that the type of a construct matches that expected by its context. For example :arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that the operands of mod have type integer. Type information gathered by a type checker may be needed when code is generated.

Type Systems

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

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

The type of a language construct will be denoted by a "type expression." A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are typeexpressions.
   A special basic type, type_error , will signal an error during type checking; void denoting "the absence of a value" allows statements to be checked.
2. Sincetypeexpressionsmaybenamed,atypenameisatypeexpression.
3. A type constructor applied to type expressions is a typeexpression.

Constructors include:

Arrays : If T is a type expression then array (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products : If T1 and T2 are type expressions, then their Cartesian product T1 X T2 is a typeexpression.

Records : The difference between a record and a product is that the names. The record type constructor will be applied to a tuple formed from field names and field types.

For example:

```
type row = record
        address: integer;
        lexeme:
        array[1..15
        ] of char
        end;
```

var table: array[1...101] of row;

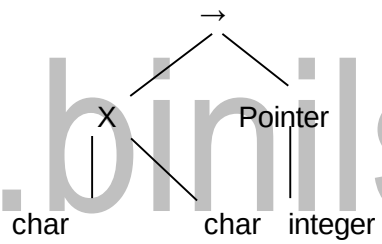
declares the type name row representing the type expression record((address X integer) X (lexeme X array(1..15,char))) and the variable table to be an array of records of this type.

Pointers : If T is a type expression, then pointer(T) is a type expression denoting the type "pointer to an object of type T".

For example, var p: ↑ row declares variable p to have type pointer(row).

Functions : A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.



Fg. 5.7 Tree representation for char x char → pointer (integer)

Type systems

A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. It is specified in a syntax-directed manner. Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A sound type system eliminates the need for dynamic checking follows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than type_error to a program part, then type errors cannot occur when the target code for the program part isrun.

Strongly typed language

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

Error Recovery

Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input. Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.