STATIC SEMANTICS
&
INTERMEDIATE CODE GENERATION

Supratim Biswas

Course Material

March 2024

Department of Computer Science & Engineering
BIT Mesra

Principles of Compiler Design : Semantic Analysis and Intermediate Code Generation

Theme 1 : Semantic analysis - concepts and examples; Syntax Directed Translation Scheme (SDTS); semantic analysis of declarations; semantic analysis of expressions in C/C++. Intermediate code forms. Illustration through examples.

Theme 2 : Semantic analysis of assignment statement; translation of boolean expressions - partial and complete evaluation; translation of control flow statements. Illustration through examples.

Theme 3 : Translation of procedure calls; runtime environments and activation records. Illustration through examples.

Pre-requisite : This module assumes familiarity with scanning and parsing, particularly LR parsers.

Text Book :

Compilers – Principles, Techniques and Tools, Alfred V Aho, Monica S Lam, Ravi Sethi and Jeffrey D Ullman, Pearson Education Inc, 2nd Edition, 2007

Material of this course is covered in Chapters 5, 6 and 7, pages 303 to 502 of this book.

Syntax Analysis vs Semantic Analysis

Which of the following situation falls under the scope of semantic analysis ?

- The compiler reports errors but how to know whether the error was detected during lexical phase or syntax phase or beyond these two phases.

- A program is syntactically correct but the compiler reports errors. It implies that the parser alone would have passed the program as a valid string in L(G). However the string has errors in the sense that it can not be assigned unique meaning. It should be possible to create many instances of such errors.

- A program compiles correctly : implies that there are no compilation errors reported. However when the program is executed, it is terminated abnormally with run-time errors. The behavior of the program at run-time is not as expected by the programmer.

- A program compiles, executes and produces output. But the output is not the desired one.

We would examine several programs in C++ to get some insight into the issues mentioned above and also to motivate us to learn semantics.

The programs will be compiled through g++ and our intuitions will be matched with the actions performed by g++.

Example Program 1	**Example Program 2**	**Example Program 3**

int main()
{
  int a = b;
  int b = 5;
  return 0;
}
	int main()
	{
	  int a = b, b = 5;
	  return 0;
	}
		int main()
		{
		  float b;
		  int b = 5;
		  return 0;
		}

Program 1 : An error message is given by g++.

```
p1.C: In function :
p1.C:6:11: error:  was not declared in this scope
   6 | { int a = b;
     |          ^
```

Program 2 : Error message

```
p2.C: In function :
p2.C:6:11: error:  was not declared in this scope
   6 | { int a = b; b = 5;
     |          ^
```

Program 3 : Error message

```
p3.C: In function :
p3.C:7:7: error: conflicting declaration
   7 |   int b = 5;
     |       ^
p3.C:6:9: note: previous declaration as
   6 | { float b;
     |         ^
```

Example Program 4	Example Program 5	Example Program 6

int main()
{ int &i;
  cout << i << endl;
  return 0;
}
	int main()
	{ short s = 1234567890;
	  cout << s << endl;
	  return 0;
	}
		int main()
		{int i = 40;
		 if (1 <= i <= 5)
		   cout << " In range \n" ;
		 else
		  cout << " Out of Range \n";
		  return 0;
		}

Program 4 : Error message

```
p4.C: In function :
p4.C:6:8: error:  declared as reference but not initialized
   6 | { int &i;
     |        ^
```

Program 5 : Warning Message

```
p5.C: In function :
p5.C:6:13: warning: overflow in conversion from  to  changes value from  to  [-Woverflow]
   6 | { short s = 1234567890;
     |             ^~~~~~~~~~
```

Program 6 : No compilation error.

Produces the following output on execution :

```
In range
```

Example Program 7	**Example Program 8**	**Example Program 9**

int main()
{
int a[5] = {1, 2, 3, 4, 5,
              6, 7, 8};
return 0;
}
	int main()
	{ int a[5] = {1, 2, 3};
	  int sum;
	  for (int i = 0; i < 10000;
	        i++)
	      sum = sum +a[i];
	  cout << sum << endl;
	 return 0;
	}
		int f(int x)
		{ if (x > 10) return x;
		  else if (x > 5)
		     return x+5;
		}
		
		int main()
		{ int i = -5;
		   cout << f(i) << endl;
		   return 0;
		}

Program 7 : Error Message

p7.C: In function :

```
p7.C:6:37: error: too many initializers for
   6 | { int a[5] = {1, 2, 3, 4, 5, 6, 7, 8};
     |                                     ^
```

Program 8 : No Compilation error.

In execution, produces the following message.

```
Segmentation fault (core dumped)
```

Program 9 : Warning message.

p9.C: In function :

```
p9.C:8:1: warning: control reaches end of non-void function [-Wreturn-type]
   8 | }
     | ^
```

Example Program 10	**Example Program 11**	**Example Program 12**
int main() { float inc = 0.01;	/* * Test Program	short f(short a) { cout << " short \n";

Example Program 10	**Example Program 11**	**Example Program 12**

float sum = 0.0;
while (inc != 0.1)
{ cout << inc << endl;
  inc = inc + 0.01;
}
 cout << sum << endl;
return 0;
}
	*
	int main()
	{ int a = b;
	  int b = 5;
	  return 0;
	}
		return a;}
		long f(long x)
		{ cout << "long \n";
		  return x;}
		char f (char c)
		{ cout << "char \n";
		  return c;}
		
		int main() { f(100);}

Program 10 : No Compilation error.

On execution, no output is received.

Program 11 : Error message.

```
p11.C:4:1: error: unterminated comment
    4 | /*
      | ^
```

Program 12 : Error message.

```
p12.C: In function :
p12.C:10:7: error: call of overloaded  is ambiguous
  10 |  f(100);
     |      ^
p12.C:3:7: note: candidate:
   3 | short f(short a) { cout << " short \n"; return a;}
     |       ^
p12.C:4:6: note: candidate:
   4 | long f(long x) { cout << " long \n"; return x;}
     |      ^
p12.C:5:6: note: candidate:
   5 | char f (char c)
     |      ^
```

Example Program 13

long f(long a) { cout << " long \n"; return a;}
int f(int x) { cout << " int \n"; return x;}
char f (char c) { cout << " char \n"; return c;}
 	
int main()
{ short d = 25;

Example Program 13

```
   char ch = '$';
   f(1000000000000);      f(1234);    f(ch);      f(d);
}
```

Program 13 : No complation error.

On execution, the following output is produced.

```
long
int
char
int
```

Take Home Exercise : Write different instances of syntactically valid programs in C/C++ that are not semantically acceptable.

Summary :

- The programs reveal the behavior of the g++ compiler through different program examples.
- You have been exposed to a major part of the front-end of a compiler that comprises of Scanning (Lexical analysis) and Parsing (Suntax Analysis).
- The semantics analysis module of a compiler logically starts after parsing but in practice it is

interleaved with syntax analysis. Of course, this is not mandatory and semantic analysis could have been defined by a separate pass (or passes) post syntax analysis also but as we shall see the first approach turnrs out to be efficient and adequate.

- The semantic analysis that we shall discuss in this module is based on a bottom-up parser and we will assume an LALR(1) parser for our purpose. This does not preclude performing semantic analysis with top-down parsers but that is not under this course.

The first two phases that you have learnt so far can be used in applications, other than compilers, as well. The characteristic of such an application should involve recognizing strings from a language that is generated by a context free grammar.

The subsequent phases of the compiler that we shall study are closely tied to implementation of programming language features. Instead of dealing with all the semantic related issues for a specific PL, in this course we take the stand that interesting linguistic features across PL be examined.

The following approach would be adopted for the semantic analysis part of this course.

- Abstract a linguistic feature for examination and determine from language reference manual / compiler writers guidelines, the meaning specified for the feature.

- At the simplest level of abstraction, we shall often use English as our description mechanism. However it should be obvious that a formal notation for specifying semantics is more desirable for precise description and unambiguous interpretation.

- A Context Free Grammar (CFG), G that generates all the syntactically correct strings of the underlying language L(G), is to chosen among various equivalent alternaives. However the set of all syntactically correct strings may include several strings that are not necessarily semantically valid. WHY ?

- The choice of the grammar rules are dictated by the semantic actions that determine whether the syntactically valid w ε L(G) is also semantically correct. Knowing that the string w is syntactically valid is not enough for semantic analysis.

Review of Background in Compilers

The front-end of a compiler comprises of three phases :

- Lexical analysis or scanning
- Syntax analysis or parsing
- Semantic analysis and intermediate code generation

We assume familiarity with the first two phases. These two phases can be used in applications, other than compilers, as well. The characteristic of such an application should involve recognizing strings from a language that is generated by a context free grammar.

The subsequent phases of the compiler that we shall study are closely tied to implementation of programming language features. Instead of dealing with all the semantic related issues for a specific PL, this course advocates that interesting linguistic features across PL be examined.

The following approach would be adopted for the semantic analysis part of this course.

- Abstract a linguistic feature for examination and determine from language reference manual / compiler writers guidelines, the meaning specified for the feature.

- At the simplest level of abstraction, we shall often use English as our description mechanism. However it should be obvious that a formal notation for specifying semantics is more desirable for precise description and unambiguous interpretation.

- A CFG, G that generates all the semantically correct strings of the underlying language L(G), including others that are not, is chosen among various equivalent grammars.

- The choice of the grammar rules are dictated by the semantic actions that determine whether the syntactically valid $w \varepsilon L(G)$ is also semantically correct.

- Knowing that the string w is semantically valid is a necessary requirement for a compiler but not sufficient.

- Besides semantic validation, a simpler equivalent representation of the source is generated in a format known as intermediate code.

- In summary the purpose of semantic analysis is to find the semantics of the source code and express it in the intermediate code instructions such that the semantics of both remain the same.

- An important characteristics of intermediate code is that generation of assembly/machine code on the target computer is a relatively much simpler problem than generating target code directly from the source program.

BASIC CONCEPTS AND ISSUES

WHAT IS STATIC SEMANTIC ANALYSIS?

Static semantic analysis ensures that a program is correct, in that it conforms to certain extra-syntactic rules.

The analysis is called

Static: because it can be done by examining the program text, and

Semantic: because the properties analysed are beyond syntax, i.e. they cannot be captured by context free grammars.

Examples of such analyses are:

a. Type analysis,

b. Name and scope analysis,

c. Processing of declarations

d. Processing of expressions and statements

e. Intermediate code generation.

TYPE ANALYSIS

The type of an object should match that expected by its context.

Examples of issues involved in type analysis are:

1. Is the variable on the left hand side of an assignment statement ‘compatible’ with the expression on the right hand side?
2. Is the number of actual parameters and their types in a procedure call compatible with the formal parameters of the procedure declaration?
3. Are the operands of a built-in operator of the right type.

   If not, can they be forced (coerced) to be so?

What is the resultant operator, then, to be interpreted as?

For example, the expression **3 + 4.5** can be interpreted as:

a. Integer addition of 3 and 4 (4.5 truncated).

b. Real addition of 3.0 (3 coerced to real) and 4.5.

Note: The notion of ‘type compatibility’ is non-trivial. For example, in the declaration

```
type ptr = ↑ object;
var a : ↑ object;      b : ptr;
```

are a and b type compatible?

To handle the issues related to type compatibility, two notions of type equivalence were introduced by PL designers.

- Structural Equivalence
- Name Equivalence

Before analysing a program in a given L for type equivalence, one must find out the type equivalence adopted by the designers of L.

NAME AND SCOPE ANALYSIS

The issues are:

1. What names are visible at a program point? Which declarations are associated with these names?
2. Conversely, from which regions of a program is a name in a declaration visible? Example :

DECLARATION PROCESSING

Declaration processing involves:

1) *Uniqueness checks*: An identifier must be declared uniquely in an declaration.
2) *Symbol Table updation*: As declarations provide most of the information regarding the attributes of a name, this information must be recorded in a data structure called the symbol table. The main issues here are:
    a) What information must be entered into the symbol table, and how should such information be represented?
    b) What should be the organization of the symbol table itself.
3) *Address Resolution*: As declarations are processed, the addresses of variables may also be computed. This address is in the form of an offset relative to the beginning of the storage allocated for a block (refer to the module on runtime environments for more details).

INTERMEDIATE CODE GENERATION

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

There is a choice of intermediate representations such as abstract syntax trees, postfix code and three address code, Gimple among many others.

EXAMPLE

Three address code generated for the assignment statement **x := A[y,z] < b**, where A is a 10 x 20 array of integers, and b and x are reals is:

```
1. t1 := y * 20
2. t1 := t1 + z
3. t2 := addr(A) - 21
4. t3 := t2[t1]
5. t4 := intoreal(t3)
6. if t4 < b goto 9
7. t5 := 0
8. goto 10
9. t5 := 1
10. x:= t5
```

SYNTAX DIRECTED ANALYSIS

Can we always represent the properties stated above through a context free grammar? There are theoretical results to show that the answer is in the negative.

1. Declaration of a variable before its use is a requirement in most programming languages. The formal language {w c w | w is a string from some alphabet Σ } can be thought of as an abstraction of languages with a single identifier declaration (the first occurrence of w), which must be declared before use (the second occurrence of w).

   This language is not context free.

2. The language $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is an abstraction of languages with two procedures and their corresponding calls, which require that the number of formal parameters (represented by the a's and b's) be the same as the number of actual parameters (c's and d's).

   This language is not context free.

However, semantic analysis has traditionally been done in a *syntax directed* fashion, i. e., along with parsing, Why?

SYNTAX DIRECTED DEFINITION

Associate with each terminal and non-terminal a set of values called *attributes*. The evaluation of attributes takes place during parsing, i. e., when we use a production of the form, A ⇡ X Y Z, for *derivation* or *reduction*, the attributes of X, Y and Z could be used to calculate the attributes of A.

EXAMPLE: For the grammar shown below, suppose we wanted to generate intermediate code:

$S \rightarrow id := E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow - E_1$

$E \rightarrow id$

It is convenient to have the following attributes:

S.code - The intermediate code associated with S.

E.code - The intermediate code associated with E.

E.place - The temporary variable which holds the value of E.

id.place - The lexeme corresponding to the token id.

The grammar is augmented with the following semantic rules:

$S \rightarrow$ id := E

```
        S.code := E.code || gen(id.place ':=' E.place);
```

$E \rightarrow E_1 + E_2$

```
        E.place := newtemp;
          E.code := E1.code || E2.code ||
       gen(E.place ':=' E1.place '+' E2.place);
```

$E \rightarrow E_1 * E_2$

```
             E.place := newtemp;
             E.code := E1.code || E2.code ||
       gen(E.place ':=' E₁.place '*' E2.place);
```

$E \rightarrow - E_1$

```
             E.place := newtemp;
             E.code := E1.code ||
             gen(E.place ':='  'uminus' E1.place);
```

$E \rightarrow$ id

```
             E.place := id.place;  E.code := ' ';
```

where

1) newtemp is a function which generates a new temporary variable name, each time it is invoked.
2) '||' is the concatenation operator.
3) gen( ) evaluates its non-quoted arguments, concatenates the arguments in order, and returns the resulting string.

4) Symbols in quotes are inserted in place after stripping off the quotes.

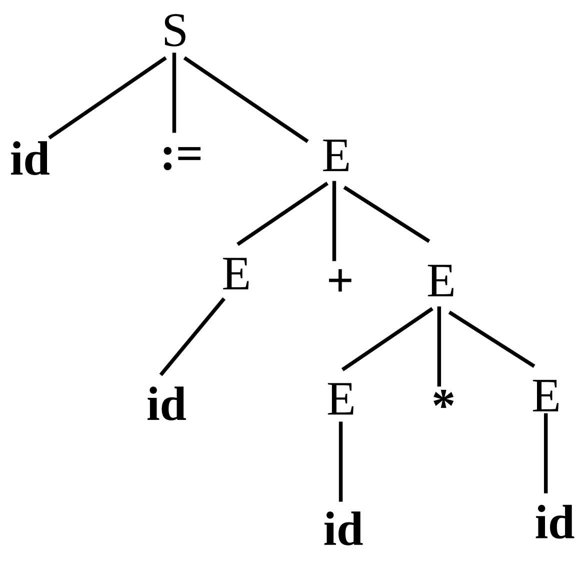
The semantic rule

S.code := E.code || gen(id.place ':=' E.place);

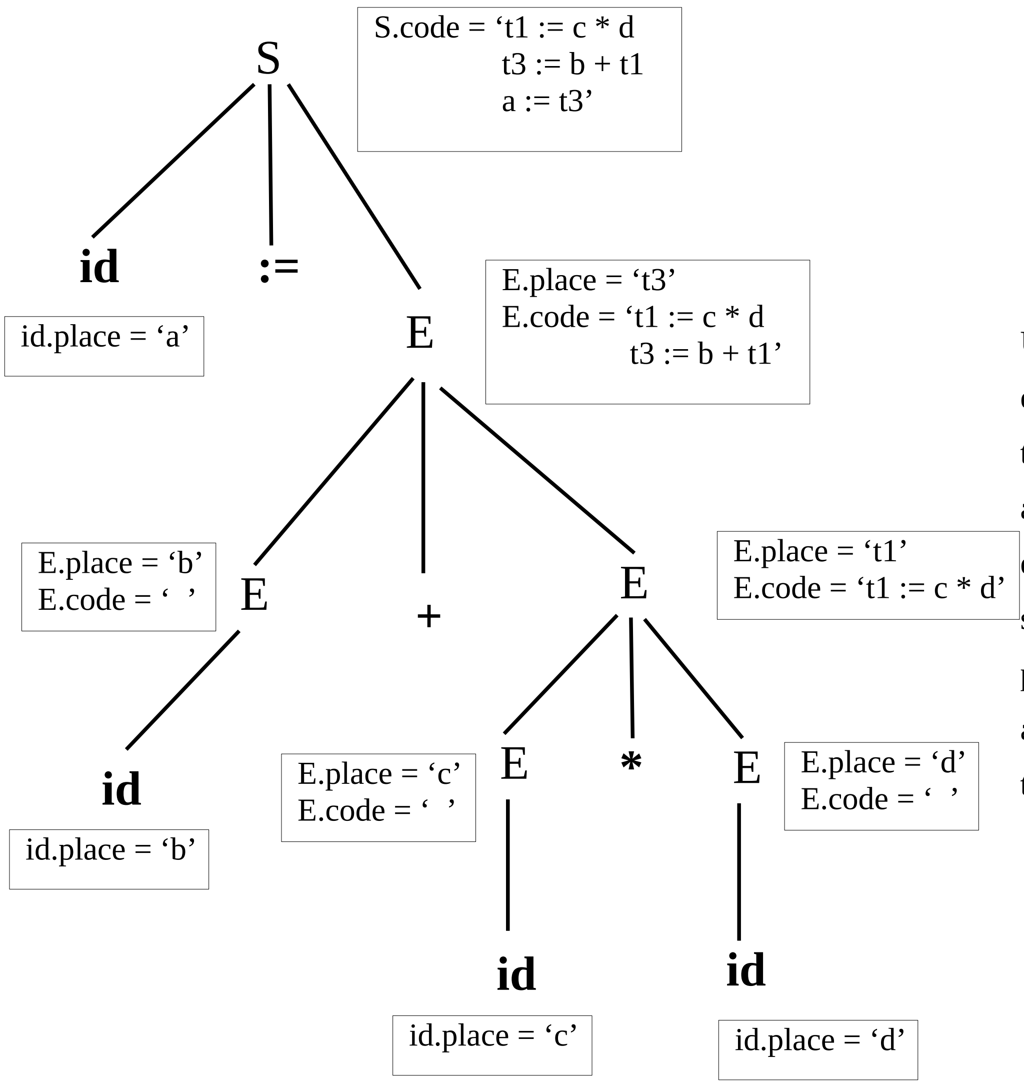
associated with the rule S → id := E is interpreted as :

- When the the right hand of the rule, id := E, is seen by a bottom up parser, the semantic attributes of the symbols in the rhs are assumed to be known.
- The semantic atributes, id.place and E.code, are available with the parser.
- After reducing the rhs, id := E, by its lhs nonterminal S, the parser appends to the attribute E.code another code fragment expressed as id.place := E.place
- The physical interpretation is that after the code for E has been generated in E.code, the result of := is to assign to id.place the value that is currently held in E.place.

Consider the sentence, **a := b + c * d** generated by the assignment grammar and its parse tree shown below.



SYNTAX DIRECTED ANALYSIS



NOTE:

1. Unlike the example, the attribute evaluation should proceed along with the construction of the parse tree. This would introduce restrictions on the possible relations between attribute values.
2. Nothing has been said about the implementational details of attribute evaluation.
3. The attributes of terminals are usually supplied by the lexical analyser.

SYNTAX DIRECTED DEFINITION: FORMALIZATION

A syntax directed definition is an **augmented context free grammar**, where each production **A** → **α**, is associated with a set of semantic rules of the form

b := f (c$_1$, c$_2$, ..., c$_k$), where f is a function, and

1. b is a *synthesized attribute* of A and **c$_1$, c$_2$, ..., c$_k$** are attributes belonging to the grammar symbols of **α** , or

2. b is an *inherited attribute* of one of the grammar symbols **α**, and $\mathbf{c_1, c_2, ..., c_k}$ are attributes belonging to A or **α**.

In either case, we say that the attribute b depends on the attributes $\mathbf{c_1, c_2, ..., c_k}$.

In the previous example, S.code, E.code, and E.place were all synthesized attributes.

TRANSLATION SCHEMES

A translation scheme is a refinement of a syntax directed definition in which:

i. Semantic rules are replaced by actions.

ii. The exact point in time when the actions are to be executed iis specified. This is done by embedding the actions within the right hand side of productions.

In the translation scheme

$$A \rightarrow X_1 \; X_2 \, . \, . \, X_i \; \{\mathit{action}\} \; X_{i+1} \, . \, . \, . \, X_n$$

action is executed after completion of the parse for X_i.

EXAMPLE : A translation scheme to map infix expressions into postfix expressions. Expressions involve binary *addop* operators {+, –} and token **num**.

For the infix expression : 10 + 7 – 3

Desired translation : 10 7 + 3 –

Using expression grammar writing rules for attributes of *addop*

$$E \rightarrow E \text{ addop } T \mid T$$

$$T \rightarrow \text{num}$$

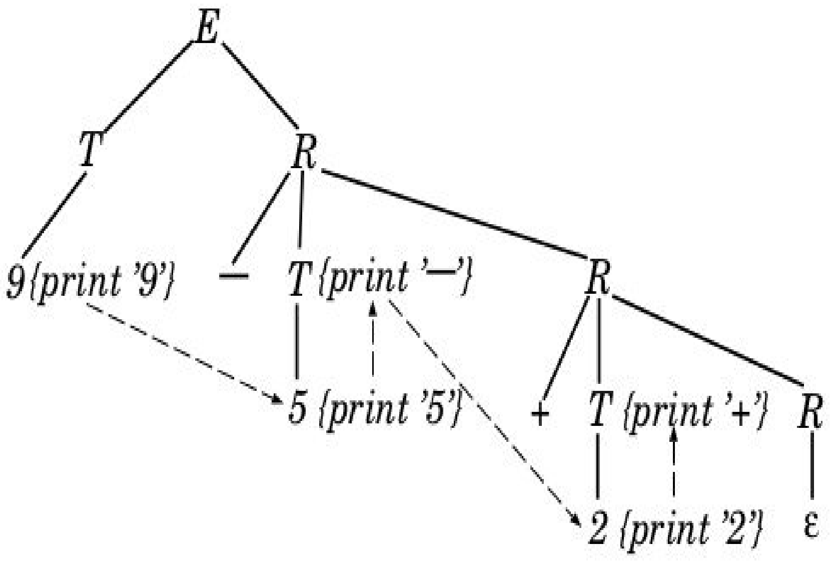
We however use an equivalent grammar that works for both parsing methods.

Consider the translation scheme given below.

$E \rightarrow T\ R$

$R \rightarrow$ *addop* T { print(addop.lexeme) } R_1 | ε

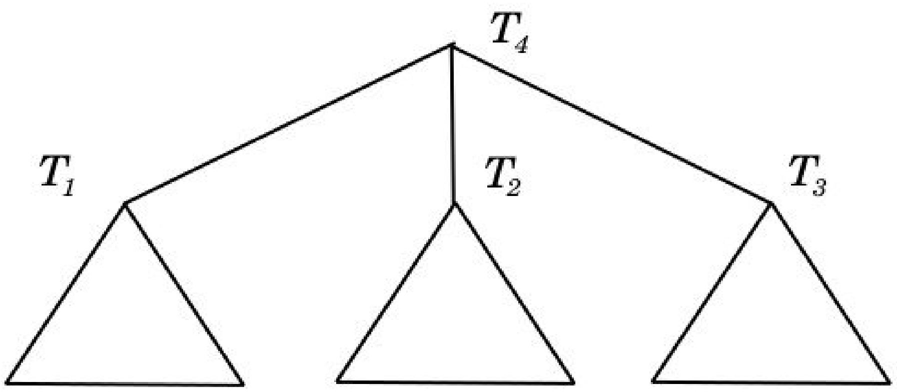
$T \rightarrow$ num { print(num.val) }



The order of execution of actions for the string 9 – 5 + 2 is shown by dotted arrows. Examine how this translation scheme works for top-down and bottom-up parsers and also whether it changes the properties of the operators.

RESTRICTIONS ON ATTRIBUTE DEPENDENCES

Certain restrictions must be placed on the dependences between attributes, so that they can be evaluated along with parsing. It is interesting to note that top-down parsers or bottom-up parsers would construct the parse tree shown in the figure in the order T1, T2, T3, T4.



Clearly an attribute of T2 cannot depend on any attribute of T3.

The extent of restriction gives rise to two classes of syntax directed definitions,

a. **S-attributed definition**

b. **L-attributed definition**

We will not discuss L-attributed definitions in this course.

S-ATTRIBUTED DEFINITIONS

A translation scheme is *S-attributed* if:

(i) Every non-terminal has synthesized attributes only.

(ii) All actions occur on the right hand side of productions.

The syntax directed definition shown below is an example of an S-attributed definition:

```
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);
```

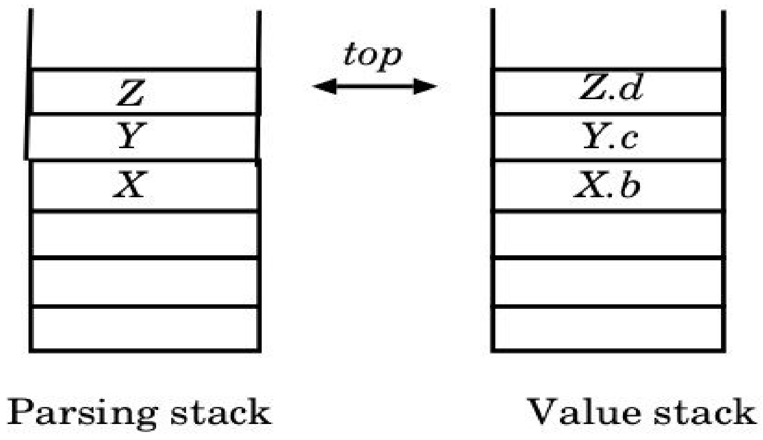
E → id

E.place := id.place; E.code := ' ';

IMPLEMENTATION OF S-ATTRIBUTED DEFINITION

Assume a bottom up parsing strategy. We augment the parsing stack with a value stack (val). If location i in the parsing stack, parse[i], contains a grammar symbol A, then val[i] will contain all the attributes of A.

Prior to a reduction using a production A → X Y Z :

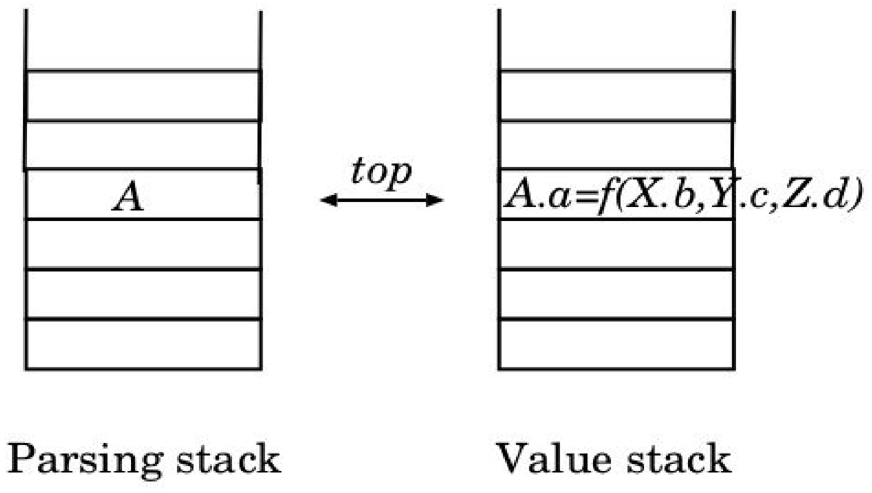


Note that rhs X Y Z is on parse stack in the reverse order.

If an attribute a of lhs nonterminal A is defined as f(X.b, Y.c, Z.d),

A.a = f(X.b, Y.c, Z.d)

then, after the reduction, the contents of parse and val stacks change as shown.



We can now replace the semantic rules by actions which refer to the actual data structures used to hold and manipulate the attributes. This is illustrated for the assignment grammar discussed earlier.

Let each element of val stack be a record with two fields val[i].code and val[i].place. Then the syntax directed definition is :

- First grammar rule with semantic rules given earlier

  S → id := E

  S.code := E.code || gen(id:place ':=' E:place);

  The same rule with actions is written as :

  S → id := E

  val[top−2].code := val[top].code ||

  gen(val[top−2].place ':=' val[top].place);

- The second rule

  E → E_1 + E_2

  E.place := newtemp;
  E.code := E1.code || E2.code ||
  gen(E.place ':=' E1.place '+' E2.place);

  is rewritten as
  E → E1 + E2

  T := newtemp;

  val[top−2].code := val[top−2].code || val[top].code ||
  gen(T ':=' val[top – 2].place) '+' val[top].place;

  val[top – 2].place := T;

- The action for the rule for E → E1 * E2 is rewritten as

```
        T := newtemp;
        val[top−2].code  := val[top−2].code ||  val[top].code
                 ||  gen( T  ':='  val[top – 2 ].place) '*'
                     val[top].place;
        val[top – 2 ].place := T;
```

- E → − E_1

```
        T := newtemp;

        val[top−1].code  := val[top].code ||

  gen( T  ':='  'uminus' val[top].place) '*'
                      val[top].place;
        val[top – 1].place := T;
```

- The last rule is rewritten as

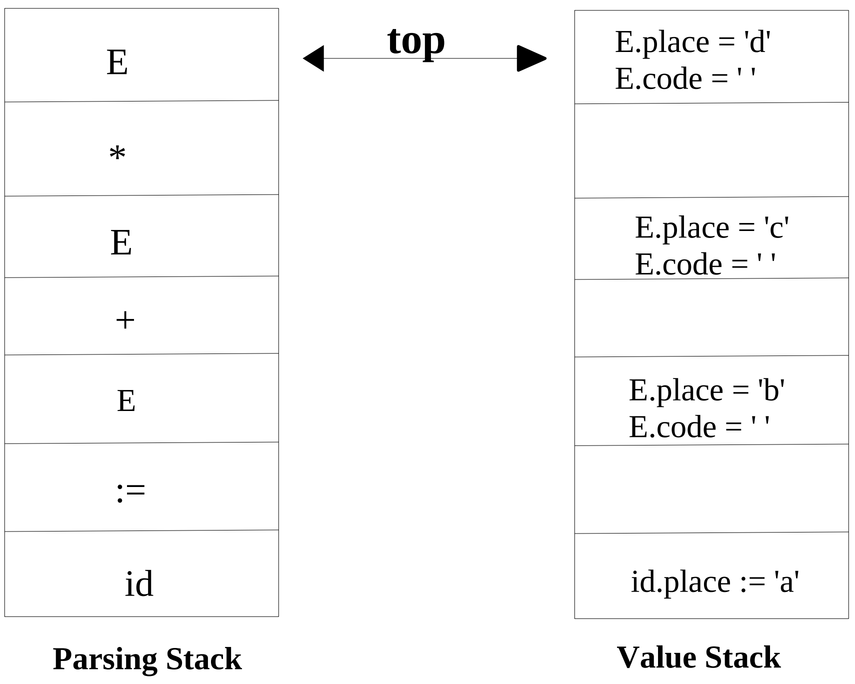
  E → id

```
        val[top].place := val[top].place;  val[top].code := ' ';
```

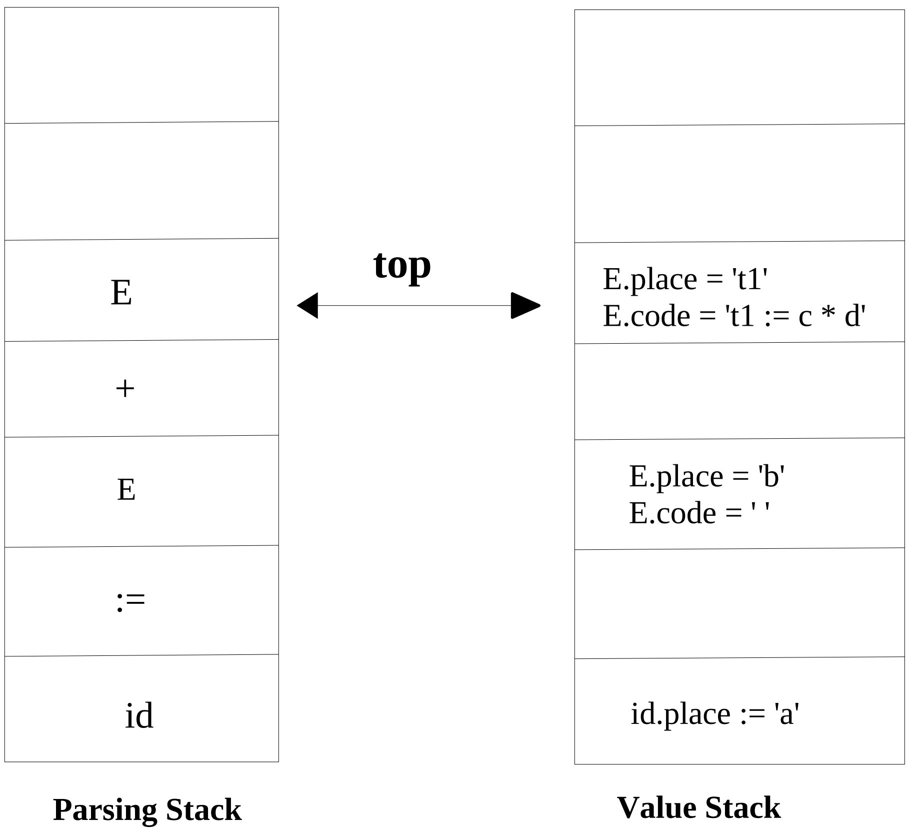
- The illustration shows how semantic rules can be transformed to semantic actions that are implementable for a specific grammar with an additional stack for semantic attributes.
- The approach is generic in nature and hence can be extended to other context free grammars.

Stack contents before reduce by production E → E1 * E2

Input string   a := b + c * d

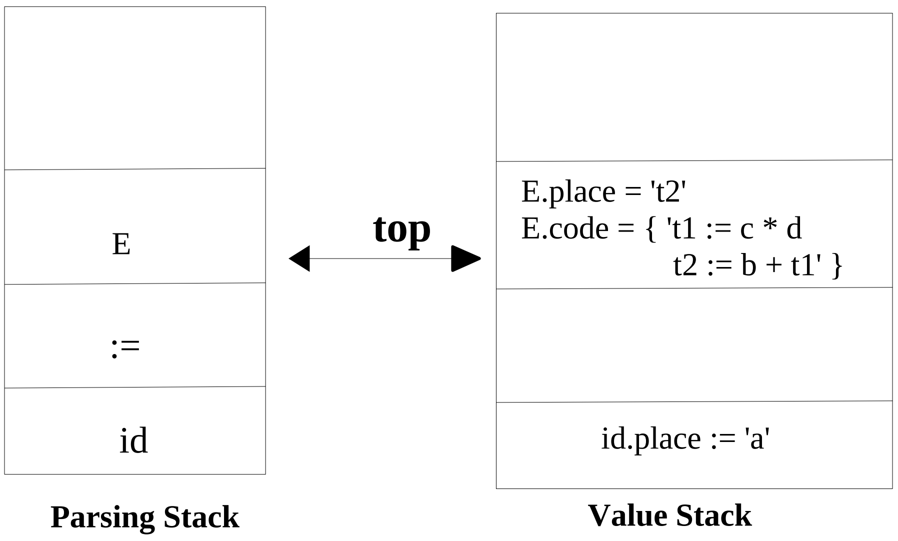


Stack contents immediately after reduction by E → E1 * E2

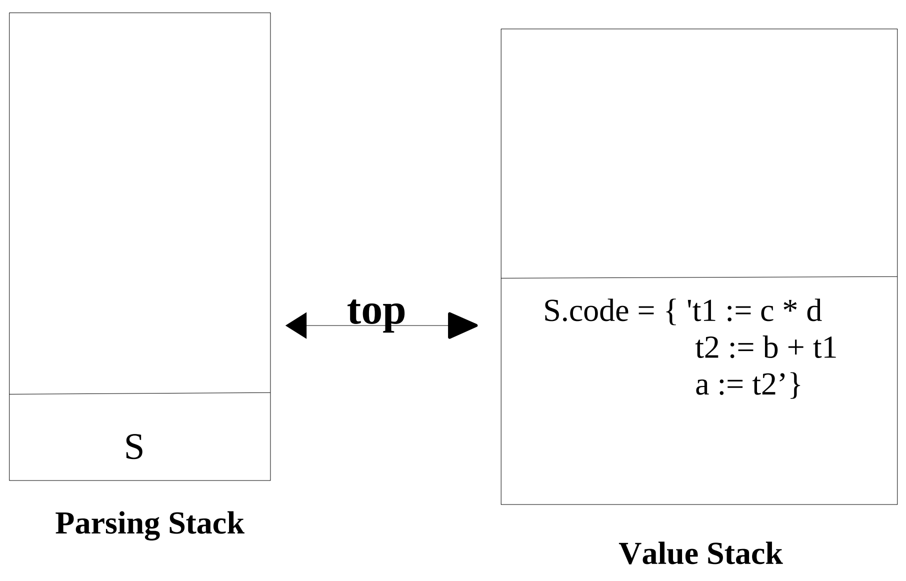


The actions specified in the S-attributed scheme are used to obtain the new stack configuration shown above.

Stack contents immediately after reduction by the production rule E → E1 + E2



Configuration of the two stacks just after the reduce by the rule S→id := E, and just before the accept action by the parser is shown below.



METHODS FOR SEMANTIC ANALYSIS

We summarize the diiferent methods introduced in this module for performing semantic analysis in a syntax directed manner.

Carefully distinguish between

1) *Syntax directed definition* : A context free grammar augmented with semantic rules. No assumptions are made regarding when the semantic rules are to be evaluated.
2) *Translation scheme* : A syntax directed definition with actions instead of semantic rules. Actions are embedded within the right hand side of productions, their positions indicating the time of their evaluation during a parse.
3) *Implementation of a syntax directed definition* : A translation scheme in which the actions are described in terms of actual data structures (e.g. value stack) instead of attribute symbols.

Semantic analysis of most programming features covered in this module are demonstrated using translation actions.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

This part of the module deals with writing S-attributed grammars for performing static semantic analysis of various language constructs.

1) **Processing of declarations** : defining required attributes, symbol table organization for handling nested procedures and scope information, translation scheme for sample declaration grammars.
1) **Type analysis and type checking** : concept of type expression and type equivalence, an algorithm for structural equivalence, performing type checking in expressions and statements and generating intermediate code for expressions.
1) **Intermediate code forms** : three address codes and their syntax.
2) **Translation of assignment statement.**
3) **Translation of Boolean expressions and control flow statements**.

Most of the semantic analysis and translation will be illustrated using synthesized attributes only.

PROCESSING OF DECLARATIONS

Declarations are typically part of the syntactic unit of a procedure or a block. They provide information such as names that are local to the unit, their type, etc.

Semantic analysis and translation involves,

- collecting the names in a symbol table
- entering their attributes such as type, storagerequirement and related information
- checking for uniqueness, etc., as specified by the underlying language
- computing relative addresses for local names which is required for laying out data in the activation record

We begin with a simple grammar for declarations

1. **Type associated with a Single Identifier**

   In the grammar given below, type is specified after an identifier.

- We use a synthesized attribute, T.type for storing the type of T.
- **T.width** is another synthesized attribute that denotes the number of memory units taken by an object of this type.
- **id.name** is a synthesized attribute for representing the lexeme associated with id.
- **offset** is used to compute the relative address of an object in the data area. It is a global variable.

PROCESSING OF DECLARATIONS

- Procedure **enter(name, type, width, offset)** creates a symbol table entry for id.name and sets the type, width and offset fields of this entry.

$P \rightarrow MD$

$M \rightarrow \varepsilon$ {offset := 0}

$D \rightarrow D; D$

$D \rightarrow$ id : T {enter (id.name, T.type, T.width, offset);offset := offset + T.width}

$T \rightarrow$ integer { T.type := integer; T.width := 4 }

$T \rightarrow$ real { T.type := real; T.width := 8 }

$T \rightarrow$ id [num] of T_1

{ T.width := num.val * T_1.width }

$T \rightarrow \uparrow T_1$

{ T.type := pointer(T_1.type); T.width := 4 }

In the code for semantic analysis given above,

- Arrays are assumed to start at 1 and num.val is a synthesized attribute that gives the upper bound (integer represented by token num).
- If a nonterminal occurs more than once in a rule, its references in the rhs are subscripted and then used in the semantic rules , e.g., use of T in the array declaration.

2. **Type associated with a List of Identifiers**

If a list of ids is permitted in place of a single id in the above grammar, then

- all the ids must be available when the type information is seen subsequently.
- A possible approach is to maintain a list of such ids, and carry forward a pointer to the list as a synthesized attribute, say, L.list
- The procedure **enter**( ) given below has a different semantics now which creates a symbol table entry for each member of a list and also sets the type information.
- The two functions, **makelist**() and **append**() perform the tasks of creating a list with one element and for inserting an element in the list respectively. These functions return a pointer to the created / updated list.

$P \rightarrow D$
$D \rightarrow D; D$

$D \rightarrow L : T$ {**enter** (L.list, T.type, T.width, offset)

$L \rightarrow$ id, L_1 {L.list := **append**(L1.list, id.name)}

$L \rightarrow$ id {L.list = **makelist** (id.name)}

$T \rightarrow$ integer { T.type := integer; T.width := 4 }

T → real { T.type := real; T.width := 8 }

T → id [num] of T_1 { T.width := num.val * T_1.width;
T.type := *array* (num.val, T_1.type) }

T → ↑T_1 { T.type := *pointer* (T_1.type); T.width := 4 }

3. **Type Specified at the beginning of a list**

   The following grammar provides an example.

   D → D; D | T L

   L → id , L | id

   T → integer | real | ...

- The grammar is rewritten to make writing of semantics easier.
- Note the role played by D.syn in propagating the type information

D → D; D

D → D_1, id {enter (id.name, D_1.syn);
D.syn := D_1.syn }

D → T id {enter (id.name, T.type);
D.syn := T.type}

T → integer | real | ... {same as earlier }

SCOPE INFORMATION

The method described above can be used to process declarations of names local to a procedure.

For a language that permits nested procedures with lexical scoping, the static scoping rules are as follows. We shall not discuss static sccoping rules as they are not applicanle in many PLs such as C/C++

- We use the earlier grammar for illustration but add another rule to it.

  P → D

  D → D **;** D

  D → **id :** T

  D → **proc id ;** D **;** S

  T → **integer**

  T → **real**

  T → **array [num]** of T_1

  T → ↑T1

- The rule, D $\rightarrow$ **proc id ;** D **;** S permits nested procedures, but without parameters.
- The design solution is to create a separate table for handling names local to each procedure. The tables are linked in such a manner that lexical scoping rules are honoured
- The following attributes are defined

  id.num T.type T.width

  The global variable nest is used to compute the nesting level for a procedure.
- Two new nonterminals, M and N, are introduced so that semantic actions can be written at these points

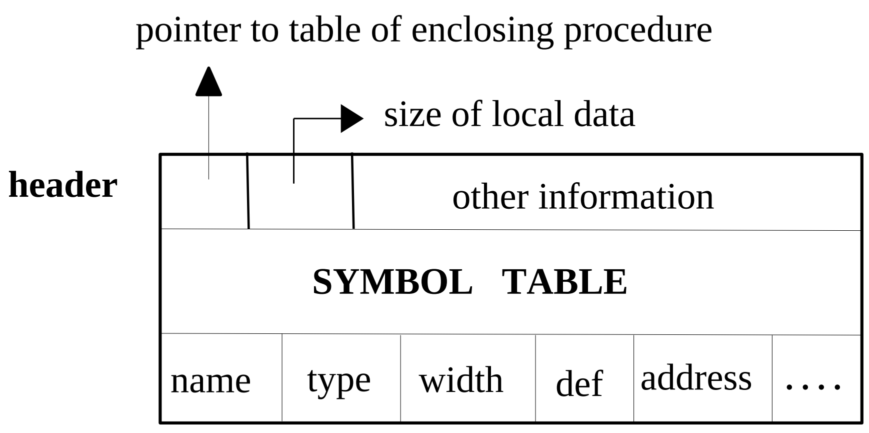
  P $\rightarrow$ M D

  D $\rightarrow$ **proc id ;** N D **;** S

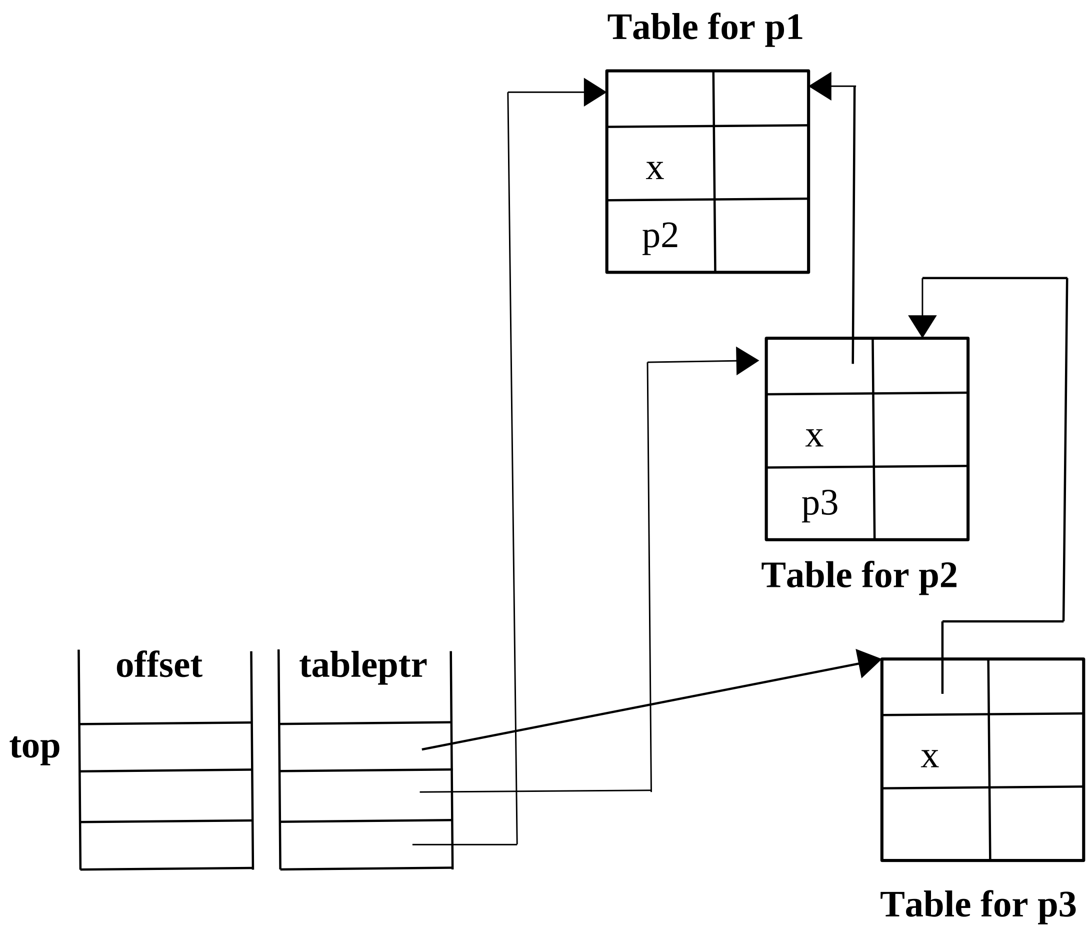
  M $\rightarrow \varepsilon$

  N $\rightarrow \varepsilon$
- The place marked by M is used to initialize all information related to the symbol table associated with this level. Similarly N would signal the start of activity for another table.
- The various symbol tables have to be linked properly so as to correctly resolve nonlocal references.
- A stack of symbol tables is used for this purpose.
- The symbol table structures and their interconnection are shown in Figure.
- For symbol table management, the following organization and routines are assumed.
  1. Each table is an array of a suitable size which is linked using the fields as shown.
  2. There are 2 stacks, called **tableptr** and **offset**. The stack tableptr points to the currently active procedure's table and the pointers to the tables of the enclosing procedures are kept below in the stack.
  3. The other stack is used to compute the relative addresses for locals within a procedure.
  4. **mktable(previous)** is a procedure that creates a new table and returns a pointer to it. It also gives the pointer to the previous table through the argument, **previous.** This is required in order to link the new table with that of the closest enclosing procedure's.

Symbol Table Organization for Nested Structures



(b) Organization of a single table



(c) Symbol Tables and their interconnection when control reaches the marked point

4. **enter**(table, name, type, width, offset) is a procedure for creating a new entry for the local variable, name, along with its attributes.

5. **enter**(table, name, type, width, offset) is a procedure for creating a new entry for the local variable, name, along with its attributes.

6. **addwidth**(table,width) is used to compute the total space requirement for all the names of a procedure.

7. **enterproc** (table, name, nest level, newtable) creates a new entry for a procedure name. The last argument is a pointer to the corresponding table.

The semantic rules are given now.

```
P → M D          {addwidth(top(tableptr),top(offset)) ;
                  pop(tableptr) ; pop(offset)}
```

```
M → ε  {p := mktable(nil);push(p,tableptr);
                push(0, offset); nest := 1}
D →  D ; D
D→  id : T
        {enter(top(tableptr), id.name, T.type, T.width);
         top(offset) := top(offset) + T.width }
D → proc  id ; N D ; S
                {p := top(tableptr); addwidth(p,top(offset));
                  pop(tableptr) ; pop(offset);
                  enterproc (top(tableptr), id.name, nest, p);
           nest --;
          }
N → ε  { p := mktable(top(tableptr)) ;
                  push(p,tableptr); push(0,offset) ; nest++;
          }
T → integer    { T.type := integer; T.width := 4 }
T → real                { T.type := real; T.width := 8 }

T → id [ num ] of T1
                { T.width := num.val * T1.width;
                  T.type :=  array (num.val, T1.type)
      }
T → ↑T1

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

- The translation scheme described is inadequate to handle recursive procedures. The reason being that the name of such a procedure would be entered in its closest enclosing symbol table only after the entire procedure is parsed.

- However a recursive procedure would have a call to itself within its body and this call cannot be translated since the procedure name would not be present in the table.
- A possible remedy is to enter a procedure's name immediately after it has been found. Semantic actions for only the changed rules are shown.

```
D → proc  id ; N D ; S
                {p := top(tableptr); addwidth(p,top(offset));
                  pop(tableptr) ; pop(offset); nest --;
```

```
            }
    N → ε  { p := mktable(top(tableptr)) ;
                 enterproc(top(tableptr),id.name, nest, p);
               push(p,tableptr); push(0,offset) ; nest++;
           }
```

SYMBOL TABLE ORGANIZATION

We now consider the issues related to the design and imple-mentation of symbol tables

- Two major factors are the features supported by the programming language and the need to make efficient access to the table.
- The scoping rules of the language indicate whether single or multiple tables are convenient.
- The various types and attributes of names decide the internal organization for an entry in the table.
- Need for efficient access of the table influences the implementation onsiderations, such as whether arrays, linked lists, binary trees, hash tables or an appropriate mix should be used.
- We examine in brief two possible hash table based implementations in this course.

MULTIPLE HASH TABLES

1. Each symbol table is maintained as a separate hash table.
2. For nonlocal names, several tables may need to be searched which, in turn, may result in searching several chains in case of colliding entries in a hash table.
3. An important issue in this design is the access time for globals
4. Another concern relates to space. The size of each table, if fixed at compile time, could turn out to be a disadvantage, specially if the estimates are wrong.

SINGLE HASH TABLE

A single hash table for block structured languages is possible, provided one stores the scope number associated with each name. A feasible organization is shown in Figure.

- New names are entered at the front of the chains, hence search for a name terminates with the first occurrence on the chain.
- When a scope is closed, the table has to be updated by deleting all entries with the current scope number. The operation is not too expensive, since the search stops at the first entry where the scope values mismatch.
- Basic table operations, enter and search, are efficient because there is a single table. However, storage gains may be compensated due to the inclusion of scope value with each entry.

- All the locals of a scope are not grouped together, hence additional effort in time and space is needed to maintain such information, if so required.

Single hash table organization is depicted in the figure. The entry for a name in a node also contains its scope value.

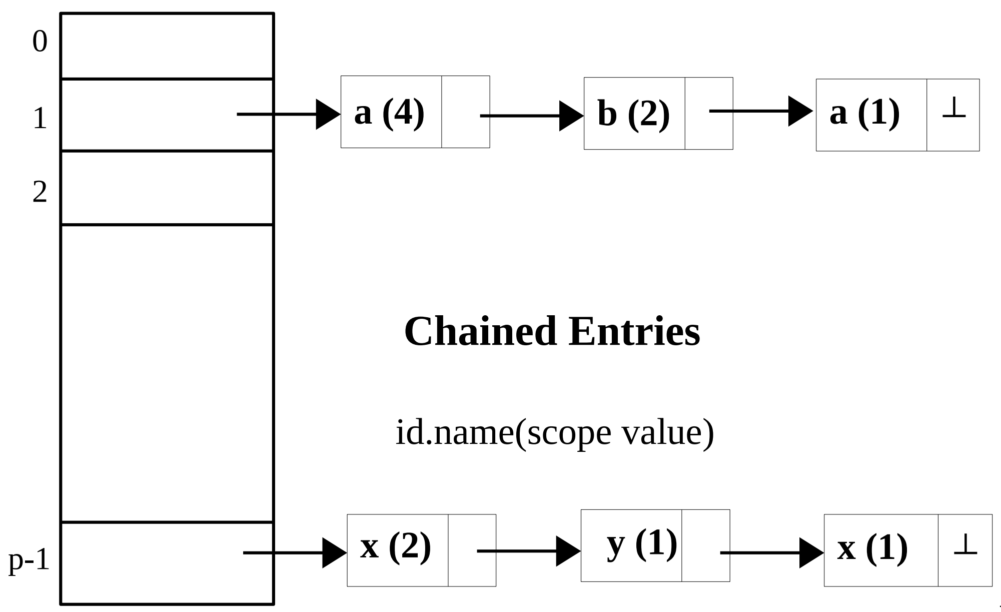


FIGURE : Single Table for Handling Nested Scopes

The attributes for each name have to be predecided, depending on the processing they undergo in various phases of the compiler.

From the viewpoint of semantic analysis, relevant attributes have been identified in the preceding examples.

- The other phases, like error analysis and handling, code generation and optimization, may introduce more attributes for an entry in the table.

The design discussed for handling of nested procedure declarations can also be used for doing semantic analysis of record declarations. In language C, struct is similar to a record of Pascal.

The earlier grammar is first augmented as given below with a marker nonterminal L. We introduce marker nonterminals to facilitate writing of semantic actions at points of interest.

D → D ; D | id : T

T → **integer** | **real**

T → **id [num] of** T1

T → **record** L D **end**

L → ε

Two stacks tableptr and offset are used in the same sense as introduced earlier.

Attribute T.width gives the width of all the data objects declared within a record.

Attribute T.type is used to hold the type of a record. The expression used here for T.type is explained later during Type Analysis.

The semantic actions for the grammar rules for record are as follows. The actions for other rules remain unchanged.

T → **record** L D **end**

```
        {T.type := record(top(tableptr);
           T.width := top(offset);
            pop(tableptr); pop(offest)
        }
```

L → ε

```
        { p := mktable(nil);
    push(p, tableptr); push (0, offset)
        }
```

Semantic errors that are detected during processing of declarations of a program include the following among many others

- uniqueness checks
- names referenced but not declared
- names declared but not referenced
- names declared but not initialized properly
- ambiguity in declarations
- and many more

Code for detection and handling of such errors are incorporated in the semantics actions at the relevant places.

TYPE ANALYSIS AND TYPE CHECKING

Static type checking is an important part of semantic analysis.

- detect errors arising out of application of an operator to an incompatible operand, and
- to generate intermediate code for expressions and statements.

The notion of types and the rules for assigning types to language constructs are defined by the source language.

Typically, an expression has a type associated with it and types usually have structure.

Usually languages support types of two kinds, basic and constructed.

- Examples of basic types are integer, real, character, boolean and enumerated
- array, record, set and pointer are examples of constructed types. The constructed types have structure.

How to express type of a language construct ?

For basic types, it is straightforward but for the other types it is nontrivial. A convenient form is known as a **type expression**.

1. A **type expression** is defined as follows.
2. A **basic type** is a type expression. Thus boolean, char, integer, real, etc., are type expressions.

   For the purposes of type checking, two more basic types are introduced, type **error** and **void**.
3. A **type name** is a type expression. In the example given below, the type name 'table' is a type expression.

   type table = array[1 .. 100] of array[1 .. 100] of integer ;
4. A type constructor applied to type expressions is a type expression. The constructors array, product, record, pointer and function are used to create constructed types as described below.

Array : If T is a type expression, then array(I, T) is also a type expression; array elements are of type T and index set is of type I (usually a range of integer).

Example : For the array declaration

var ROW : array [1 .. 100] of integer;

the type expression for variable ROW is **array(1..100, integer)**

Product : If T_1 and T_2 are type expressions, then their cartesian product T_1 X T_2 is a type expression; X is left associative. It is used in other constructed types.

Records : The constructor record applied to a product of the type of the fields of a record is a type expression. For example,

```
        type entry = record
                                token : array [ 1.. 32 ] of char;
                                salary : real
                          end;
        var employee : array[1 .. 20] of entry;
```

The type name **entry** has the type expression as given below.

record((token X array(1..32, char)) X (salary X real))

Exercise : Write the type expression for employee.

Pointers : If T is a type expression, then pointer(T) is a type expression. For example for the declaration

var ptr : ↑ entry

is added to the type name entry, then

type expression for **ptr** is :

pointer (record ((token X array(1..32, har)) X (salary X real)))

Function : A function in a programming language can be expressed by a mapping D → R, where D and R are domain and range type.

The type of a function is given by the type expression D → R. For the declaration :

function f (a :real,b:integer) : ↑integer;

the type expression for f is

real X integer → pointer (integer)

Programming languages usually restrict the type a function may return, e.g., range R is not allowed to have arrays and functions in several languages.

How to use the type expression ?

The type expression is a linear form of representation that can conveniently capture structure of a type.

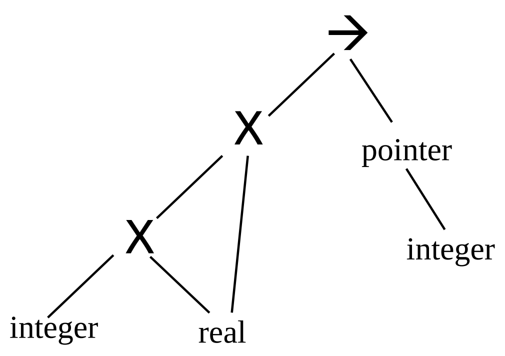
This expression may also be represented in the form of a DAG (Directed Acyclic Graph) or tree.

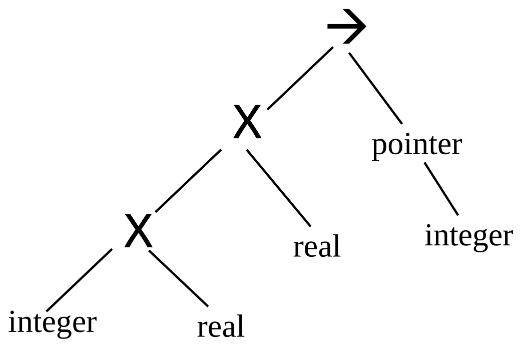
Consider the function **f (a :integer, b,c : real) : ↑integer;**

the type expression for f is

integer X real X real → pointer (integer)

Two equivalent representations in the form of a DAG and tree.





DAG form of type graph **Tree form of type graph**

Figure : Type Expressions And Type Graphs

The tree (or dag) form, as shown, is more convenient for automatic type checking as compared to the expression form.

A **type system** is a collection of rules for assigning type expressions to linguistic constructs of a program.

A **type checker** implements a type system.

Checking for type compatibility is essentially finding out when two type expressions are equivalent.

For type checking, the general approach is to compare two type expressions and **if two type expressions are equivalent** then return an **appropriate type** else return **type error**.

This, in turn, requires a definition of the notion of **equivalence of type expressions**.

The type expressions of variables in Example 1 are given in the second column.

Example 1 :

```
type link = ↑ node;
var first, last : link;
p : ↑ node;
q , r : ↑ node;
```

Variable	**TypeExpression**
first	link
last	link
p	pointer (node)
q	pointer (node)
r	pointer (nod

Example 2 :

```
type person  = record
                        id : integer;
                        weight : real
                        end;
type car   =     record
                        id : integer;
                        weight : real
                        end;
var x : person;
var y : car;
```

The issue is to decide whether all the type expressions given above are type equivalent.

Presence of names in type expressions give rise to two distinct notions of equivalence.

One is called **Name Equivalence** and the other **Structural Equivalence**.

Name Equivalence treats each type name as a distinct type, two expressions are name equivalent if and only if they are identical.

For Example 1, the type expressions for variables are reproduced below.

Variable **Type Expression**

first	link
last	link
p	pointer (node)
q	pointer (node)
r	pointer (node)

Variables **first** and **last** are name equivalent since both have identical type expression, **link**.

Similarly variables p, q and r are name equivalent, but first and p are not.

In Example 2, x and y are not name equivalent because their respective type expressions person are car which are different type names.

If type names are replaced by the type expressions defined by them, then we get the notion of structural equivalence.

Two type expressions are structurally equivalent if they represent structurally equivalent type expressions after the type names have been substituted in.

In Example 1, when the type expression for link is used

Type Name	**Type Expression**
link	pointer (node)

Variable	**Type Expression**	**Type Name Substitution**
first	link	pointer (node)
last	link	pointer (node)
p	pointer (node)	pointer (node)
q	pointer (node)	pointer (node)
r	pointer (node)	pointer (node)

As shown by the third column above, all the variables in Example 1 are structurally equivalent.

In Example 2, the type expressions for type names are

Type Name	**Type Expression**
person	record ((id X integer) X (weight X real))
car	record ((id X integer) X (weight X real))

After substitution of expression for type names gives

Variable	**Type Exp**	**Type Name Substitution**
x	person	record ((id X integer) X (weight X real))
y	car	record ((id X integer) X (weight X real))

Variables x and y are seen to be structurally equivalent.

- Example 2 indicates the problem with structural equivalence. If user declares two different types, they probably represent different objects in the users application. Name equivalent appears to be the right choice here.
- Declaring them type equivalent, merely because they have the same structure, may not be desirable.
- Name equivalence is safer but more restrictive. It is also easy to implement. Compiler only needs to compare the strings representing names of the types.

Two expressions are structurally equivalent if

i. the expressions are the same basic type, or

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

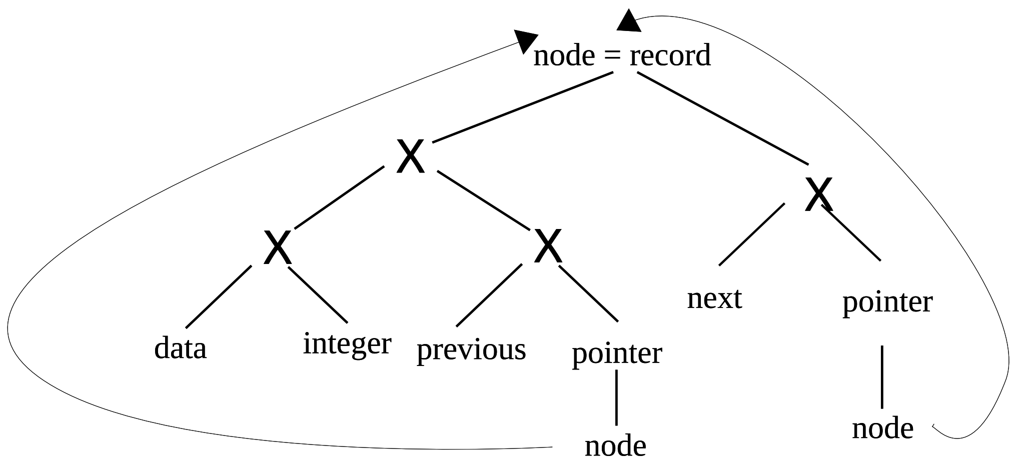
Pascal uses structural equivalence while C uses a mix of both forms of equivalence.

A compiler for a PL that uses structural equivalence would require an algorithm for detecting it at compile time.

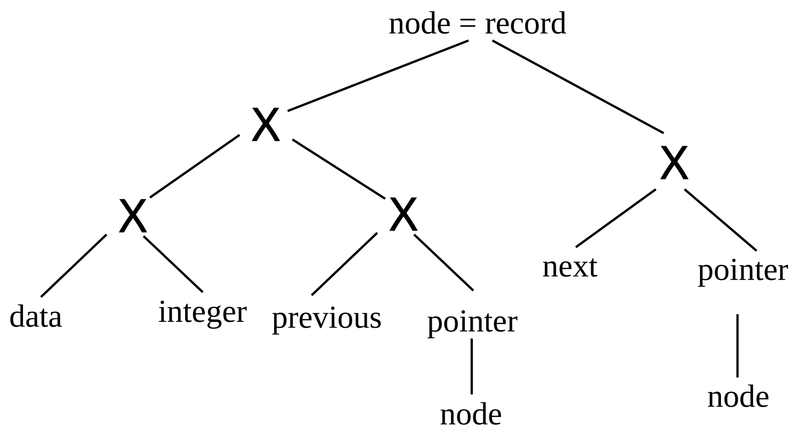
The complexity of such an algorithm is clearly dependent on the structure of the type graphs involved. When the type graph is a tree or a dag, a simpler algorithm is possible. For type graphs containing cycles, such an algorithm is nontrivial.

Cycle in a type graph typically arises due to recursively defined type names. A common situation is depicted by pointers to records.

```
link = ↑ node ;
node = record
                data : integer ;
                previous : link;
                next : link
          end;
```



Cyclic type graph



Equivalent acyclic graph

IMPLEMENTATION OF STRUCTURAL EQUIVALENCE

- Type graph for type name node is shown in the figure; a cyclic graph is a natural way of representing the tupe expression and it contains two cycles.
- The acylic equivalent does not substitute the recursive references to node.

Algorithm For Structural Equivalence

A function sequiv() is presented in the following for checking structural equivalence of two type expressions (or type graphs).

The function uses two formal parameters, s and t, which are the input type expressions or roots of the associated type graphs graphs (however the graphs must be acyclic).

The algorithm checks the equivalence of s and t using

- both are same basic type
- both are arrays of compatible types
- both are compatible cartesian products
- both are compatible pointers, and finally
- both are compatible functions

ALGORITHM FOR STRUCTURAL EQUIVALENCE

```
function sequiv(s,t) : boolean;
begin
    if s and t are the same basic type then return true
    else if s = array(s1, s2) and t = array(t1, t2 ) then
        return sequiv(s1, t1 ) and sequiv(s2, t2)
    else if s = s1 X s2 and t = t1 X t2 then
        return sequiv(s1, t1 ) and sequiv(s2, t2)
    else if s = pointer(s1) and t = pointer(t1) then
        return sequiv(s1, t1 )
    else if s = s1→s2 and t = t1→t2 then
        return sequiv(s1, t1 ) and sequiv(s2, t2)
    else return false
end
```

Rewriting the implementation of function sequiv, so that it can be used for detecting structural equivalence of general type graphs is an interesting exercise.

Writing A Type Checker

We use the material on types studied so far to implement a type checker for a sample language given below. The purpose is to demonstrate the working of a type checker only (and hence no code is generated).

```
P →    D ;S
D →D ; D |  id : T
T → char | integer | boolean
  | id [num] of T |  ↑T
S → id := E | if E then S
        | while E do S | S ; S
E → literal | num | id | E mod E
```

| E [E] | E binop E | E↑

The first rule indicates that the type of each identifier must be declared before being referenced.

The first task is to construct type expressions for each name and code which does that is to be given against the relevant rules.

In case type graphs are used, these actions need to be recoded appropriately. A sythesized attribute, id.entry, is used to represent the symbol table entry for id.

The procedure enter() has the same purpose of installing attributes of a name in the symbol table.

T.type is a synthesized attribute that gives the type expression associated with type T. Attaching these semantic actions yields the following SDTS

$D \rightarrow$ id : T	{enter(id.entry, T.type) }
$T \rightarrow$ char	{T.type = char}
$T \rightarrow$ integer	{T.type = integer}
$T \rightarrow$ boolean	{T.type = boolean}
$T \rightarrow$ array [num] of T	
	{T.type = array (1..num.val, T_1.type)}
$T \rightarrow \uparrow T_1$	{T.type = pointer(T_1.type)}

TYPE CHECKING FOR EXPRESSIONS

Once the type expression for each declared data item has been constructed, semantic actions for type checking of expressions can be written, as explained below.

E.type is a synthesized attribute that holds the type expression assigned to the expression generated by E .

The function lookup(e) returns the type expression saved in the symbol table for e.

The semantic code given below assigns either the correct type to E or the special basic type, **type_error**.

The type checker can be enhanced to provide details of the semantic error and its location.

```
E → literal        {E.type := char}

E → num            {E.type := integer}
E → id             {E.type := lookup (id.entry)}
E → E₁ mod E₂      {E.type := if E₁. type = integer and
                        sequiv (E₁.type, E₂.type)
                       then integer else type_error
                     }

E → E₁ binop E₂    {E.type :=

                       if  sequiv (E₁.type, E₂.type)
                       then E₁.type else type_error
                     }


E → E₁ [ E₂ ]      {E.type :=
                       if  E₂.type = integer and
                          E₁.type = array (s, t)
                       then t else type_error
                     }


E → E₁↑            {E.type := if  E₁.type = pointer (t)
                       then t else type_error
                     }
```

TYPE CHECKING FOR STATEMENTS

Semantic code for type checking of the statement related rules of the sample grammar are given in the following.

S.type is a synthesized attribute that is assigned either type **void** or **type_error**, depending upon whether there was a type_error detected within it or not.

The semantic action in the last rule propagates type errors in a sequence of statements.

```
S → id := E          {S.type := if sequiv (id.type, E.type)
                        then **void** else **type_error**
                      }
S → if E then $S_1$
                     {S.type := if E.type = boolean
                        then $S_1$.type else **type_error**
                      }
S → while E do $S_1$
                     { S.type := if E.type = boolean
                        then $S_1$.type else **type_error**
                      }

S → $S_1$ ; $S_2$      {S.type := if $S_1$.type = void

                   and $S_2$.type = void
                        then **void** else **type_error**
                      }
```

Combining all the grammar rules and the semantic actions listed against them gives a type checker for the sample language.

In the type checker given above, it is assumed that for all operators, the types of the operands are compatible. This assumption is too restrictive and not followed in practice.

For example, **a op i** , where a is a real and i is an integer, is a legal expression in most programming languages.

The compiler is supposed to first convert one of the operands to that of the other and then translate the op.

Language definition specifies, for a given op, the kind of conversions that are permitted. Conversion of integer to real is usually common.

Semantic rules for performing type conversion for expressions and statements are included in the discussion on generation of intermediate code for these constructs.

INTERMEDIATE CODE FORM

The front end typically outputs an explicit form of the source program for subsequent analysis by the back end.

The semantic actions incorporated along with the grammar rules are responsible for emitting them.

Among the several existing intermediate code forms, we shall onsider one that is known as Three-Address Code.

In three address code, each statement contains 3 addresses, two for the operands and one for the result. The form is similar to assembly code and such statements may have a symbolic label.

The commonly used three address statements are given below.

1. Assignment statements of the form :
   x := y op z, where op is binary, or
   x := op y, when op is unary

2. Copy statements of the form x := y

3. Unconditional jumps, goto L. The three address code
   with label L is the next instruction where control flows
4. Conditional jumps, such as
   if x relop y goto L
   if x goto L

The first instruction applies a relational operator, relop, to x and y and executes statement with label L, if the relation is true. Else the statement following if x relop y goto L is executed. The semantics of the other one is similar.

5. For procedure calls, this language provides the following :

   param x
   call p,n
   where n indicates the number of parameters in the call of p.

6. For handling arrays and indexed statements, it supports statements of the form :

```
x := y[ i ]
x[ i ] := y
```

The first is used to assign to x the value in the location that is i units beyond location y, where x, y and i refer to data objects

7. For pointer and address assignments, it has the statements

```
x := & y
x := * y
* x := y .
```

In the first form, y should be an object (perhaps an expression) that admits a l-value. In the second one, y is a pointer whose r-value is a location. The last one sets r-value of the object pointed to by x to the r-value of y.

A three address code is an abstract form of intermediate code. It can be realised in several ways, one of them is called **Quadruples**.

A quadruple is a record structure with 4 fields, usually denoted by op, arg1, arg2 and result. For example, the quadruples for the expression **a + b * c** is

arg1	**arg2**	**result**	**op**	**Equivalent form**	
*		b	c	t_1	$t_1 = b * c$
+		a	t_1	t_2	$t_2 = a + t_1$

where t1 and t2 are compiler generated temporaries.

TRANSLATION OF ASSIGNMENT STATEMENTS

The semantic analysis and translation of assignment statements involve the following activities.

i. generating intermediate code for expressions (which are assumed to be free from type_error) and the statement it is a part of,

ii. generation of temporary names for holding the values of subexpressions during translation,

iii. handling of array references; performing address calculations for array elements and associating them with the grammar rules,

iv. performing type conversion (or coercion) operations while translating mixed mode expressions as specified by the underlying language.

We start with a simple grammar and incrementally add features to it. The first grammar, given below, excludes boolean expressions and array references.

$P \rightarrow M\ D$

$M \rightarrow \varepsilon$

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

$N \rightarrow \varepsilon$

$T \rightarrow integer \mid real$

$S \rightarrow id := E$

$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid id$

The rules for P , D and T give the declaration context in which an assignment statement is to be translated. Expressions of type real and integer only are considered here but other types can be handled similarly.

Ambiguous grammar for expressions have been chosen intentionally, in order to work with a small grammar. The semantic rules an be easily extended to an unambiguous version.

Three address code is generated, wherever required, during semantic analysis. The basic scheme for evaluation of expressions is described in the following.

For an expression of the form **E_1 op E_2**

- generate code to evaluate E_1 into a temporary t_1
- generate code to evaluate E_2 into a temporary t_2
- emit code, $t_3 := t_1$ op t_2

For compiler generated temporaries, we assume that a new temporary is created whenever required. This is achieved by means of a function newtemp() which returns a distinct name on successive invocations.

It is also assumed that temporaries are stored in the symbol table like any other user defined name.

The following synthesized attributes and routines are defined for writing Syntax Directed Translation Scheme (SDTS).

The attribute **id.name** holds the lexeme for the name id. The attribute E.place holds the value of E .

Function lookup(**id.name**) returns the entry for **id.name** if it exists in the symbol table, otherwise it returns nil.

Procedure emit(**statement**) appends the 3-address code **statement** to an output file.

TRANSLATION OF ASSIGNMENT STATEMENTS

```
S → id := E          {p := lookup (id.name);
                          if p ≠ nil then emit ( p ':=' E.place)
                        else error /* undeclared id */
                          }

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 /* undeclared id */ }
```

Example : The 3-address code generated for the expression

$d := -(a + b) * c$

is given below.

We assume that all variables are of same type and bottom up parsing is used.

Rule used	**Attribute Evaluation**	**Output File**
$E \rightarrow id$	E.place := a	------
$E \rightarrow id$	E.place := b	------
$E \rightarrow E_1 + E_2$	E.place := t_1	$t_1 := a + b$
$E \rightarrow - E1$	E.place := t_2	$t_2 :=$ uminus t_1
$E \rightarrow E_1 * E_2$	E.place := t_3	$t_3 := t_2 * c$
$S \rightarrow id := E$	-------	$d := t_3$

Type Conversion : In the translation given above, it is assumed that all the operands are of the same type. However that is not always the ase.

Languages usually support many types of variables and constants and permit certain operations on operands of mixed types.

Most languages allow an expression of the kind **a op b**, where a or b may be integer or real and op is an arithmetic operation, such as + and *.

For mixed mode expressions, a language specifies whether the operation is legal or not. If illegal, then the compiler has to indicate semantic error.

For legal mixed mode operations, the compiler has to perform type conversion (or type coercion) and then generate appropriate code.

Coercion is an implicit context dependent type conversion performed by a compiler.

To illustrate this concept, we use the earlier grammar where only real and integer types are permitted and convert integers to reals, wherever necessary.

In addition to the attributes mentioned earlier, E.place, id.name, we use E.type that holds the type expression of E.

We assume the existence of an unary operator, **inttoreal**, to perform the desired type conversion from integer value to a real value.

The reverse conversion in not permitted in our framework and hence the function **realtoint** is not needed here.

The function **newtemp()** is now replaced by two functions, **newtemp_int()** and **newtemp_real()** which generate distinct temporaries of type integer and real respectively.

The semantic actions for the rule E $\rightarrow$ E + E is given in the following. The semantics for the other rule E $\rightarrow$ E * E can be written analogously.

The operator + has been further qualified as **real+** or **integer+** depending on the type of operands it acts on.

The semantic actions for the rule S $\rightarrow$ id := E

uses a function called as type_of(p) whose purpose is to access the symbol table for the name p and return its type.

Structural equivalence is assumed for type checking, whereby sequiv() is used.

```
E → E_1 + E_2
        { if E_1.type = integer and sequiv( E_1.type, E_2.type)
          then begin
                        E.place := newtemp_int();
                        emit(E.place ':=' E_1.place 'int +' E_2.place);
                        E.type := integer
                  end
        else if E_1.type = real and sequiv( E_1.type, E_2.type)
                then begin
                                E.place := newtemp_real();
                                emit(E.place ':=' E_1.place 'real +' E_2.place);
                                E.type := real
                        end
        else if E_1.type = real and E_2.type = integer
                then begin
                                E.place := newtemp_real();
                                t := newtemp_real();
                                emit(t ':='  'inttoreal' E_2.place);
                                emit( E.place ‘:=’ E_1.place 'real +' t);
                                E.type := real
                        end
        else if E_1.type = integer and E_2.type = real
                then begin
                                E.place := newtemp_real();
                                t := newtemp_real();
                                emit(t ':='  'inttoreal' E_1.place);
                                emit( E.place ‘:=’ t ‘real +’ E_2.place);
                                E.type := real
                        end
        else  E.type := type_error
}
```

The semantic rules for the assignment statement is now given.

```
S → id : = E
    { p := lookup (id.name) ;
      if p = nil then error   /* undeclared id */
      else if sequiv (type_of(p), E.type)
          then if E.type = integer
              then begin
                  emit ( p 'int:=' E.place) ; S.type := void
              end
          else begin
              emit ( p 'real:=' E.place);
              S.type := void
          end
          else if type_of (p) = real and E.type = integer
              then begin
                      t := newtemp_real ();
                      emit ( t ':=' 'inttoreal' E.place);
                      emit ( p 'real:= ' t ) ;
                      S.type := void;
                  end
          else  S.type := type error; /* type mismatch */
    }
```

Note that the temporaries used above are of different types.

Only temporary t1 is of type integer and the rest are all real.

Example : A program fragment, alongwith the 3-address code generated for it, according to the SDTS, is given below.

Program Fragment	**Intermediate Code**
	t1 := b int* c
var a, b, : integer ;	t2 := inttoreal b
var d : real ;	t3 := t2 real* d
a := b * c + b * d ;	t4 := inttoreal t1
	t5 := t4 real+ t3
	type error

The type_error is indicated because **a** is of type **integer** while the type of rhs , i.e., **t5** is **real**. This is our interpretation that is used in the translation above (integer variable can not be assigned a real value).

In practice, a language defines the type conversion rules applicable. Therefore a language may cooerce the rhs to an integer value and then assign it to the lhs integer name.

Check out what your language and compiler do in this situation ?

Exercise : Change the sematic rules to what your language defines for such assignments.

TRANSLATION OF ARRAY REFERENCES

We now augment the grammar to include array references as well. The basic issues for arrays are explained through an example.

Consider the following declaration and assignment statement.

```
a : array [ 1..10, 1..20] of integer
i, j, k : integer
i := ___;  j := ___;
k := a[i+2, j–5] ;
```

In order to translate the array reference given above, we must know the address of this array element at compile time.

For statically declared arrays, it is possible to compute the relative address of each element (Data Structures course).

- Array is usually stored in contiguous locations.
- There are two possible layouts of memory, known as, row-major representation (used in Pascal /C/C++) and column-major representation (used in Fortran).
- In Fortran, array is indexed from 1; C/C++ start with index 0; there are languages that permit negative indices also.

TRANSLATION OF ARRAY REFERENCES

The row-major and column major forms for a 2 dimensional array of size 3x3 are shown below.

1	2	3	4	5	6	7	8	9
A[1, 1]	**A[1, 2]**	**A[1, 3]**	**A[2, 1]**	**A[2, 2]**	**A[2, 3]**	**A[3, 1]**	**A[3, 2]**	**A[3, 3]**

Row major representation of 2D array

1	2	3	4	5	6	7	8	9
A[1, 1]	**A[2, 1]**	**A[3, 1]**	**A[1, 2]**	**A[2, 2]**	**A[3, 2]**	**A[1, 3]**	**A[2, 3]**	**A[3, 3]**

Column major representation of 2D array

Address calculation at compile time

Let base be the relative address of a[1,1]. Then the relative address of a[i_1, i_2] is given by

$$\mathbf{base + ((i_1 - 1) * 20 + i_2 - 1) * w}$$

where w is the width of an array element (space for an element in bytes).

The above expression may be rewritten as

$$\mathbf{c + (20 * i_1 + i_2) * w}$$

where c = base – 21 * w

TRANSLATION OF ARRAY REFERENCES

The term, c = base – 21 * w, is a compilation time constant and can be pre computed and stored in the symbol table, while processing array declarations.

Assuming that c has been precomputed and saved in the symbol table for the array a[ ] of our example, the 3-address code generated for the array assignment

k := a [i + 2, j – 5];

using address of $a[i_1, i_2]$ as $c + (20 * i_1 + i_2) * w$, is

$t_1 := i + 2$

$t_2 := t_1 * 20$

$t_3 := j – 5$

$t_4 := t_2 + t_3$

$t_5 := 4 * t_4$ {assuming w is 4}

$t_6 := c$

$t_7 := t_6 [t_5]$

$k := t_7$

In the following we write a suitable grammar for array references and attach semantic actions to it.

TRANSLATION OF ARRAY REFERENCES

The address computation has two parts, one is a compile time constant and the other is variable part dependent on the index expressions.

Generalization of the address computation formula for n-dimensional arrays is required.

Let the declaration of a n-dimensional array be of the form : $a[u_1, u_2, \ldots, u_n]$

the lower bounds for each dimension is assumed to be 1.

In this setting, a[10, 20, 30, 40] declares a 4-dimensional array with 240000 elements which are indexed from a[1, 1, 1, 1] through a[10, 20, 30, 40].

An array reference, $a[e_1, e_2, \ldots, e_n]$, is given with e_i, $1 \leq i \leq n$, being the index expression at the ith subscript position. The address of a[1, 1, ..., 1] is the base address.

Relative address of $a[e_1, e_2, \ldots, e_n] = c + v$

where $c = base - ((\ldots((u_2+1)*u_3+1)*\ldots)*u_n+1)*w$, and

$v = ((\ldots((e_1 * u_2+e_2)*u_3 + e_3)*\ldots)*u_n + e_n)*w$

Q1. How does one derive such expressions ?

Q2. How does the compiler use these to calculate addresses?

TRANSLATION OF ARRAY REFERENCES

In order to calculate v incrementally, the following observation is useful.

Assume an array reference a[e_1,e_2 , ... ,e_n]

Expression seen	**Code Generated**
a[e_1	e_1 is evaluated in t_1
e_2	e_2 is evaluated in t_2
	$t_3 := t_1 * u_2$
...	$t_3 := t_3 + t_2$
...	...
	...
e_n	e_n is evaluated in t_{2n-2}
	$t_{2n-1} := t_{2n-3} * u_n$
	$t_{2n-1} := t_{2n-1} + t_{2n-2}, n \geq 2$
	$t_{2n} := c$

At this stage the reference may be replaced by $t_{2n}[t_{2n+1}]$.

Note that this scheme, as given, requires a large number of temporaries.

In view of the translation scheme outlined above, we need to write a grammar that exposes the index expressions one by one.

As an index expression is parsed the semantic rules should generate intermediate code for the address calculation for that expression.

TRANSLATION OF ARRAY REFERENCES

The skeleton grammar is given below.

S → L := E

E → L | other rules for expression

L → elist] | id

elist → elist , E | id [E

The rules for L take care of array references in both the lhs and rhs of an assignment statement.

The rules for the nonterminal elist are written in such a manner so as to match the intended translation scheme.

The semantic actions that need to be taken against the grammar rules are given.

1. The rule **elist → id [E** captures the array name and the first index expression. The location which holds the value of E must be noted in order to compute v.

2. The rule **elist → elist , E** exposes one by one the index expressions encountered from left to right. The part of v that has been computed so far is kept as an attribute of elist.

TRANSLATION OF ARRAY REFERENCES

3. Using the current value of E, in **elist → elist , E** the term corresponding to the dimension under consideration can now be added to v.

   Since each term of the address computation formula requires the corresponding upper bound, it is necessary to keep track of the dimension that corresponds to this occurrence of E.

4. The rule, **L → elist]** indicates that an array reference has been completely seen. The two parts c and v of the address computation are available at this point and can be propagated as attributes of L.

5. The rule, **E→L** indicates that instance of an array reference (r-value) has been found. The appropriate code can be generated and location where this value is held can be saved as an attribute of E.

6. The rule, **S → L := E** indicates an array reference whose lvalue is required. Appropriate code may be generated since all the required information is available.

TRANSLATION OF ARRAY REFERENCES

SDTS for array references are given below.

```
S →L := E

            { if L.offset = null
              then emit (L.place ':=' E.place)
              else emit (L.place '[' L.offset ']' ‘:=' E.place)
            }

E →   L

            { if L.offset = null
              then E.place := L.place)
              else begin
                          E.place := newtemp();
                          emit (E.place ‘:=’ L.place '[' L.offset ']')
              end
            }


L → id
            { L.place := id.place;
              L.offset := null
            }
```

TRANSLATION OF ARRAY REFERENCES

```
L → elist ]

        { L.place := newtemp();
      L.offset := newtemp() ;
         emit(L.place ':=' c(elist.array));
         emit(L.offset ':=' elist.place '*' width(elist.array))
        }

elist → elist_1, E
        { t := newtemp() ;
         m := elist_1.dim + 1 ;
        emit( t ':=' elist_1.place '*' limit(elist_1.array, m));
        emit( t ':=' t '+' E.place);
        elist.array := elist_1.array;
        elist.place := t ; elist.dim := m
    }


elist → id [ E
        { elist.array := id.place;
         elist.place := E.place; elist.dim := 1
     }
```

TRANSLATION OF ARRAY REFERENCES

Brief justifications about the attributes and the functions used in our SDTS.

For L the attributes, L.place and L.offset hold the values of c and v in case of arrays. In case of scalars, the second attribute has null value.

For E, the attribute E.place is used in the same sense.

For elist, the attributes used are elist.array, elist.place and
elist.dim.

- Attribute **elist.array** is a pointer to the symbol table entry for the array name.
- Attribute **elist.place** is a plaeholder for the v part of the address calculation performed so far.
- **elist.dim** is used to keep track of the dimension information.
- The function **c(array_name)** returns the c part of the address formula from the symbol table entry for array name.
- The function **width(array_name)** gives the width information from the symbol table.
- The function **limit(array_name, m)** returns the upper bound for the dimension m from the symbol table.

TRANSLATION OF ARRAY REFERENCES

Calculation of c and v parts of address formula

The complex part of c in

$c = base - ((\ldots ((u_2+1)*u_3+1)* \ldots)*u_n+1) * w$

is the part $((\ldots ((u_2+1)*u_3+1)* \ldots)*u_n+1)$

The calculations work out like

$x_1 = 1;\ x_2 = x_1*u_2 + 1;\ x_3 = x_2*u_3 + 1; \ldots.$

which leads to recurrence relation

$\mathbf{x_1 = 1;}$

$\mathbf{x_{i+1} = x_i*u_{i+1} + 1;\ i > 1}$

The above recurrence formulation can be used to incrementally compute c after parsing a index expression at a given dimension d.

$\mathbf{c = base - x_d * w}$

The v part of the address calculation is captured by another recurrence relation

$\mathbf{y_1 = e_1;}$ $\mathbf{y_{i+1} = y_i*u_{i+1} + e_i;\ i > 1}$

$\mathbf{v = y_i * w}$; after ith index expression is parsed.

Note : The SDTS given above is a generic array reference translation scheme where the arrays start with index 1 and the representation is row-major.

However C / C++ have the following differences.

- Array reference syntax is a[e1][e2] ..[en] instead of a[e1, e2, ..., en]
- Starting index is 0 and not 1

Upper bounds of any dimension is ≥ 0 (the lower bound), similar to that used for the general case.

In the following, we shall write SDTS for array reference in C/C++.

Let the declaration of a n-dimensional array be of the form :

$a[u_1]\ [u_2]\ [\ \ldots\ [u_n]$; the lower bounds for each dimension is ≥ 0.

Consider the array reference $a[e_1]\ [e_2]\ [\ \ldots\ [e_n]$, where e_i is the subscript expression at the i^{th} index position. The start address of the array is referred to as base(a), and the compiler is aware of where it will layout array at run time. The symbol w denotes the size in bytes of an element of the array and it depends on the type of the array.

Array reference	Number of dimension	Address of the reference a[ ] [ ] ....[ ]

a[e_1]	1	base(a) + e_1*w
a[e_1] [e_2]	2	base(a) + (e_1*U_2 + e_2)*w
a[e_1] [e_2] [e_3]	3	base(a) +(e_1*U_2*U_3+ e_2*U_3 + e_3)*w

The formulation in the address expression is captured by the following precise mathematical

$$addr(a[e1][e2]...[en]) = base(a) + \left(\sum_{i=1}^{i=n} e_i * \prod_{j=i+1}^{j=n} U_j\right) * w \qquad .... (1)$$

You may observe that putting n = 1 gives the address expression for dimension 1 and so on for higher dimensions. The key part of the address calculation is the expression :

$$\sum_{i=1}^{i=n} e_i * \prod_{j=i+1}^{j=n} U_j \qquad .... (2)$$

which is calculation intensive. Fortunately the same expression can be expressed using the recurrence relation given below.

$$y_1 = e_1;\ y_n = y_{n-1}*U_n + e_n;\ \ n \geq 2 \quad .... (3)$$

There are two distinct advantages of Equation (3) over that of (2).

- The calculation of y_i from y_{i-1}requires 1 addition and 1 multiplication, as a result computation of y_n requires n-1 additions and n-1 products, a linear function of the operations.
- The address calculation can be constructed in an incremental fashion as the index expressions are encountered one by one.

The incremental scheme is explicated in the table below.

Array reference seen so far	Key part of address calculation	Intermediate code
a[e_1	e_1	t_1 := e_1
a[e_1] [e_2	e_1*U_2 + e_2	t_2 := e_2 t_1 := t_1 *U_2 t_2 := t_1 + t_2
a[e_1] [e_2] [e_3	(e_1*U_2 + e_2)*U_3 + e_3	t_3 := e_3 t_2 := t_2 *U_3 t_3 := t_2 + t_3
a[e_1] [e_2] [e_3] [e_n	....	Let t_{n-1} be the place which holds the evaluation upto e_{n-1} t_n := e_n t_{n-1} := t_{n-1} *U_n t_n := t_{n-1} + t_n

$a[e_1]\ [e_2]\ [e_3]\ [e_n]$	....	$t_{n+1} := base(a)$ $t_n := t_n * w$ $t_{n+2} := t_{n+1}[t_n]$

SDTS requires a grammar, which enables the translation outlined above.

```
S → L := E
E → L              ## array reference in the rhs of assignment
     | E + E | E * E | ... other expressions for E
L → elist ]         ## last subscript expression
     | id            ## scalar not an array reference
elist → elist] [ E   ## one more subscript expression
        | id [ E     ## first subscript expression
```

Like the generic array reference case, we need the support of the symbol table and related information about the array declaration here. The same support functions are assumed to be available for getting information from the symbol table, such as name of the array, bounds at each dimension, width of an array element, etc. The SDTS is given below.

```
S →L = E  { if L.offset = null
              then emit (L.place ':=' E.place)
              else emit (L.place '[' L.offset ']' ‘:=' E.place)
            }

E →   L    { if L.offset = null
              then E.place := L.place)
              else
                   E.place := newtemp();
                   emit (E.place ‘:=’ L.place '[' L.offset ']')
              end
            }

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



L → elist ]

        { L.place := newtemp();
        L.offset := newtemp() ;
          emit(L.place ':=' base(elist.array));
          emit(L.offset ':=' elist.place '*' width(elist.array))
        }

elist → elist₁ ] [ E
        {  m := elist₁.dim + 1 ;
          emit(  elist₁.place ':=' elist₁.place '*' limit(elist₁.array, m));
          emit( E.place ':=' E.place '+' elist₁.place );
          elist.array := elist₁.array;   elist.place := E.place ; elist.dim := m
      }


elist → id [ E
        { elist.array := id.place;
          elist.place := E.place; elist.dim := 1
     }
```

The SDTS above uses less temporaries than the one given for the generic case.

Exercise : Show the intermediate code generated for the assignment :

```
b[i+2][2*j-1]  = x + a[i*i][j+10][k=2];
```

TRANSLATION OF BOOLEAN EXPRESSIONS

We wish to enrich our grammar by permitting boolean expressions in an assignment statement. The following rules

are representative for our purpose.

B →   B **or** B | B **and** B

| **not** B | (B)

| **true** | **false**

| E **relop** E | E

$E \rightarrow E + E$ | other rules for arithmetic expressions

The rules for B give the syntax of boolean expressions. For arithmetic expressions, the same rules for E are assumed.

The term **relop** stands for the class of relational operators such as { <, <=, ==, !=, >, >= }.

The common approach to translate boolean expressions (in the context of an assignment) is to encode **true** and **false** numerically and use the general translation scheme for arithmetic expressions.

We shall demonstrate the translation based on numerical encoding by using **1** for true and **0** for false.

TRANSLATION OF BOOLEAN EXPRESSIONS

Semantic actions for the rule B $\rightarrow E_1$ relop E_2 is the only one which needs explaining.

We generate a temporary, say t, for holding the value of the expression E_1 relop E_2.

The value assigned to t is either 1 or 0 depending on whether the expression is true or false.

In either case, t is carried forward in the translation of the rest of the expression.

We assume that all three address statements are labeled numerically and that nextstat gives the label of the next 3 address statement.

By using the unconditional and conditional branching 3 address statements and nextstat, the semantic actions for this rule are written.

The rules for B $\rightarrow$ **true** | **false** cause the generation of a new temporary initialized to 1 or 0 respectively.

The rules for B $\rightarrow$ B **or** B | B **and** B cause generation of code to evaluate the boolean expressions, **or** and **and** are operators available in our intermediate code language.

TRANSLATION OF BOOLEAN EXPRESSIONS

SDTS for Boolean expressions is now given.

$B \rightarrow B_1$ **or** B_2

```
{ B.place := newtemp();
  emit(B.place ':=' B₁.place 'or' B₂.place)
}
```

$B \rightarrow B_1$ **and** B_2

```
{ B.place := newtemp();
  emit(B.place ':=' B₁.place 'and' B₂.place)
}
```

$B \rightarrow$ **not** B_1

```
{ B.place := newtemp();
  emit(B.place ':=' ‘not’ B₁.place )
}
```

$B \rightarrow (B_1)$

```
{ B.place := B₁.place)
```

$B \rightarrow$ **true**

```
  { B.place := newtemp();
emit (B.place ‘:=’  ‘1’)
  }
```

$B \rightarrow$ **false**

```
  { B.place := newtemp();
emit (B.place ‘:=’  ‘0’)
  }
```

$B \rightarrow E$

```
{ B.place := E.place}
```

$B \rightarrow E_1$ **relop** E_2

```
  { B.place := newtemp();
emit (‘if’ E₁.place ‘relop.op’ E₂.place ‘goto’ nextstat+3);
  emit (B.place ‘:=’  ‘0’);
  emit (‘goto’ nextstat + 2);
  emit (B.place ‘:=’  ‘1’)
  }
```

The attribute **op** in **relop.op** is used to distinguish the relational operator in its class.

The sequence of emit statements in the semantic actions for the last rule gives the rationale behind the usage of nextstat+2 or nextstat+3.

The method used in the SDTS above evaluates a boolean expression completely is a safe implementation (generated code preserves semantics).

We shall refer to this method as **full** or **complete evaluation** of a boolean expression.

The translation ignores issues of type checking and type coercion. Such rules are required to correctly handle expressions that have mixed mode operands, one operand being boolean and other permitted numeric type in the language.

Semantic rules for type conversion is discussed in the context of arithmetic expressions. Similar rules will have to be written for boolean expressions that have permissible mixed mode operands.

SUMMARY OF SDTS FOR ASSIGNMENT STATEMENT

We have covered a wide range of issues dealing with translation of an assignment statement. However the treatment is not exhaustive and several details have been skipped for sake of clarity and brevity.

To be able to write a general grammar that includes expressions of all permissible types, such as boolean, integer, real and other numeric types.

Though the salient issues of translation have been exhibited with ambiguous grammars, it is desirable that unambiguous grammars that directly take care of properties of operators, such as precedence and others, be used.

The type checking and type coercion part of semantic analysis need to be integrated and extended over all permissible types.

The translation of array references made assumptions about

lower bounds, availability of c and type checking/ coercion.

The scheme needs to be elaborated by appropriate generalisations, and inclusion of missing analyses.

Once the design for the basic grammar is complete, other features may be added along the same lines. For instance, record accesses and function calls are common in expressions.

SUMMARY OF SDTS FOR ASSIGNMENT STATEMENT

The approach to incorporate more features comprises of the following activities.

- Add relevant rules to the existing grammar.
- Decide on the 3-address codes to be generated.
- Identify type checking and type coercion, if necessary, and insert semantic rules to that effect.
- Function calls may require elaborate type checking of all arguments and appropriate translation of parameter passing mechanisms employed.
- Integrate the grammar and semantic actions with the existing one.
- Detection of semantic errors at various places have been indicated. More elaborate detection of such errors, issuing of appropriate error messages and possible strategies for recovery need to be worked out and incorporated.

It should be obvious that semantic analysis for expressions is a nontrivial task and comprises the bulk of semantic analysis for the entire language.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

Translation Of Control Flow Constructs

Translation and generation of intermediate code for typical control flow constructs is considered here. We use the following grammar.

S	$\rightarrow$	if B then S	
		| if B then S else S	
		| while B do S	
		| begin Slist end	
		| L := E	
Slist	$\rightarrow$	Slist ; S	
		| S	
B	$\rightarrow$	B_1 **or** B_2	| B_1 **and** B_2
		| **not** B_1	| (B_1)
		| **true** | **false**	
		| E_1 **relop** E_2 | E	
E	$\rightarrow$	E1 + E2	| other rules for expression
L	$\rightarrow$	rules for lhs in assignment statement	

TRANSLATION OF CONTROL FLOW CONSTRUCTS

A comment about the grammar. It is capable of generating nested control flow statements such as nested if-then-else, nested while, etc.

The nonterminals used have the following purpose :

S Start nonterminal and also nonterminal for some type of statements – assignment and control flow

Slist List of one or more statements of type S

B Boolean expression

relop terminal symbol for relational operators, such as {<, <=, >, >=. =, !=}

The rules for assignment statement are assumed without elaboration.

Rule for a compound statement enclosed in begin and end is also included.

The translation issues are introduced through an example.

Example : Consider the boolean expression given below.

a+b > c **and** flag

where a,b and c are integer variables and flag is boolean.

Equivalent 3-address code sequence is given below.

```
10 :    t1 := a + b
11 :    if t1 > c goto 14
12 :    t2 := 0
13 : goto 15
14 :    t2 := 1
15 :    t3 := t2 and flag
```

Assume that the expression **a+b > c and flag** is part of an if-then-else statement, of the form below.

If a+b > c **and** flag **then**

begin

```
            a := b + c;
            flag := false
        end
        else   a := b – c ;
```

In order to generate code for above statement, it is required to append thefollowing two statements after the code for expression evaluation.

The two labels enable the control to flow to the then and else parts respectively.

```
16 :    if t3 goto label1
17 :    goto label2
```

The statements that need to be generated for the then and else parts are given below. Note the inclusion of a jump statement after the then part, in order to prevent control flow to the else part.

Code for Then part	**Code for Else part**
18 : t_4 := b + c	23 : t_6 := b – c
19 : a := t_4	24 : a := t_6
20 : t_5 := 0	25 :
21 : flag := t_5	
22 : goto label3	

It is clear that label1 should be 18 (or nextstat + 2) and may be filled in during generation of the statement 16.

The symbol label2 enables a jump to the begining of the else part, i.e., 23 in our example. Howvever when the statement 17 is being generated, it is not possible to know the value of this label.

The value clearly depends upon the number of statements in the then part.

Similarly, the value of label3 should be 25, where the code for the statement immediately following the if-then-else, would be placed.

Again its value depends on the number of statements in the else part and can not be resolved while generating statement 22.

The issues for translation of the if-then-else statement may be summarized to

- decide the code sequences for 3 components of this construct during parsing - the conditional, statements in the then-part, and statements in else part. Finally the inclusion of an unconditional jump after the then part to ensure control doe snot fall through into the else part.

- find a scheme to resolve the forward labels of the jump statements, such as label2 and label3, since the relevant information is not available till all the 3 components have been seen.

Instead of using the method of complete evaluation of a boolean expression, it is possible to use another method which is known as partial evaluation.

In partial evaluation of boolean expressions, the value of the
expression is not explicitly stored and it may not even be evaluated completely.

The essence of partial evaluation can be explained by considering the two 3-address code sequences given below.

Code fragment :

```
If  a+b > c  and flag  then
        begin
                a := b + c;
                flag := false
```

```
end

else   a := b – c ;
```

Complete Evaluation

```
10 :    t1 := a + b
11 :    if t1 > c goto 14
12 :    t2 := 0
13 :    goto 15
14 :    t2 := 1
15 :    t3 := t2 and flag
16 :    if t3 goto 18
17 :    goto 23
18 :    t4 := b + c
19 :    a := t4
20 :    t5 := 0
21 :    flag := t5
22 :    goto 25
23 :    t6 := b - c
24 : a := t6
25 :
```

Partial Evaluation

```
50 :    t1 := a + b
51 :    if t1 > c goto 53
52 :    goto 60
53 :    if flag goto 55
54 :    goto 60
55 :    t2 := b + c
56 :    a := t2
57 :    t3 := 0
58 :    flag := t3
59 :    goto 62
60 :    t4 := b - c
61 :    a := t4
62 :
```

The code generated for partial evaluation does not use the temporaries t2 and t3 to hold the values of subexpressions as they have been used in the first column.

Using the fact that for an expression of the form b1 **and** b2, evaluate b2 only when b1 is true. Similarly for an expression of the form b1 **or** b2, evaluate b2 only when b1 is false.

The code in the second column shows that it needs less number of temporaries and is also shorter in length as compared to that in the first column.

There are more jump statements in the code for partial evaluation. The number of forward resolutions of jump labels are also correspondingly higher in the second column.

It may be noted that in case of complete evaluation the generated code for expression has exactly two unresolved jump labels.

In order to translate using this method, the issues are

i. to select code sequences to be generated for the subexpressions of the form **e1 relop e2** ; **b1 and b2**, etc., and

ii. To resolve the target labels of jump statements. The context in which the expression occurs , e.g., **if-then-else**, **while**, also play a role in this part.

It may be noted that target labels of some jump statements may be resolved at generation time itself.

```
50 :    t1 := a + b
51 :    if t1 > c goto 53
52 :    goto 60
53 :    if flag goto 55
54 :    goto 60
```

Statement 51, shown above, is an example. The fact is that jump statements whose targets lie locally (within the code for the expression) can be resolved easily.

Jump statements whose targets are beyond the expression, such as the begining of then or else part, can not be resolved in this manner.

For example, after processing the expression a+b > c and flag, the statements numbered 50 to 54 are generated.

However the target labels of statements 52 and 54 are not known at this stage.

Also the target of 53 is known at generation time only because of the context of the if-then-else statement, and may not be known in a different situation.

The unresolved labels occuring in statements 52 and 54 have a common characteristic , the expression evaluates to false at these points in the code.

A general method to resolve such labels would be to remember all the statement numbers where the expression evaluates to false. When the target becomes known later it can be inserted in all these statements.

Why did we not need to remember all the statement numbers where the expression evaluates to true ?

This problem did not surface in our example. However, it can be easily seen that in general, we need to keep this information as well (try the same example, replacing **and** by **or**).

The observations above determine the actions that are required for partial evaluation of an boolean expression and to attach these actions at the appropriate place during the translation of control flow constructs.

- A CFG that generates boolean expressions in the context of their use in control flow structures is written first.

- Semantic actions are then added to generate code using partial evaluation of a boolean expression

SEMANTIC RULES FOR BOOLEAN EXPRESSIONS

A grammar alongwith semantic rules for partial evaluation of Boolean expressions is given below.

$B \rightarrow B_1$ **or** B_2

$B \rightarrow B_1$ **and** B_2

$B \rightarrow$ **not** B1

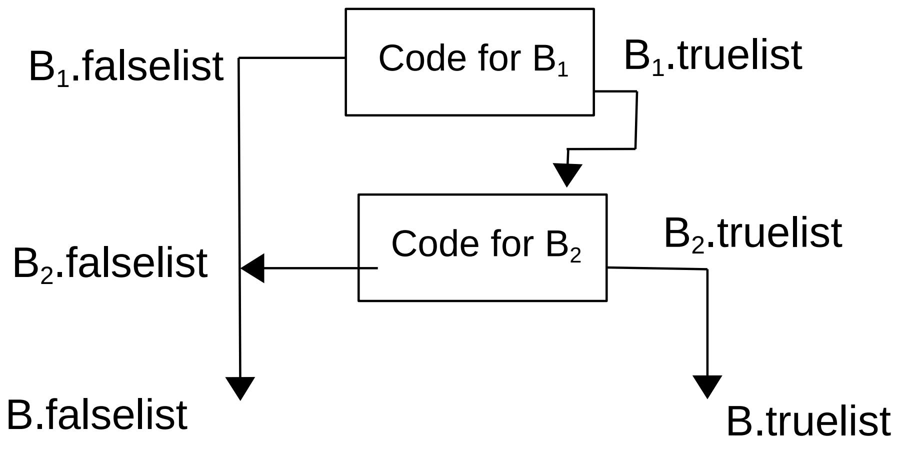
$B \rightarrow$ (B1)

$B \rightarrow$ **true** | **false**

$B \rightarrow E_1$ **relop** E_2 | E

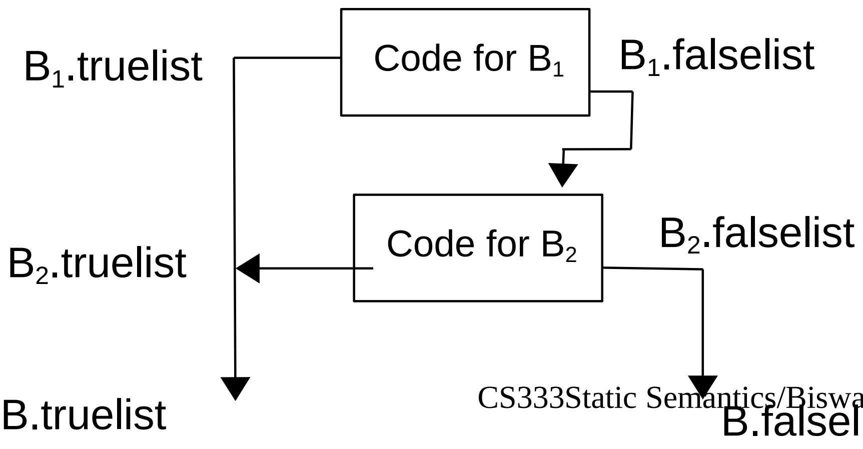
$E \rightarrow E_1 + E_2$ | other rules

Consider the second rule, $B \rightarrow B_1$ **and** B_2. Assume that parsing and semantic actions for the right hand side of this rule is complete and available. The scheme for linking the code sequences of the rhs in order to construct that of the lhs nonterminal B is illustrated in the figure below.



Actions for $B \rightarrow B_1$ **and** B_2

Similar linkage rule is as below.



for the first shown

Actions for $B \rightarrow B_1$ **or** B_2

The purpose of the semantic attributes used above are explained here.

- B.truelist and B.falselist are two attributes for nonterminal B, both of are lists of intermediate code statement numbers.
- B.truelist is a list of those intermediate code where B evaluates to true but the target address is not known. Each element on this list is an intermediate code whose label field for the destination is incomplete.
- B.falselist is a similar list where B evaluates to false.
- These lists contain incomplete code which contain jumps to the true and false exits of B.

The grammar complete with semantic actions is given below.

```
    B → B₁ or M B₂
        {   backpatch(B₁.falselist, M.stat);
            B.truelist := merge(B₁.truelist, B₂.truelist) ;
            B.falselist := B₂.falselist
        }
    B → B₁ and MB₂   {backpatch(B₁.truelist,M.stat);
                      B.falselist := merge(B₁.falselist, B₂.falselist) ;
                      B.truelist := B₂.truelist
                      }

B → not B₁
    { B.truelist := B₁.falselist ; B.falselist := B₁.truelist }

B →  ( B₁ )  { B.truelist := B₁.truelist ; B.falselist := B₁.falselist }

B →  true   { B.truelist := makelist (nextstat);  emit (‘goto __’) }

B →  false  { B.falselist := makelist (nextstat); emit (‘goto __’ }

B →  E₁ relop E₂
```

```
                  { B.truelist := makelist (nextstat);
                    B.falselist := makelist (nextstat + 1);
                    emit (‘if’ E1.place ‘relop’ E2.place ‘goto’ __);
                    emit (‘goto’ __);
                  }
B →     E
          { B.place := E.place;
           B.truelist := makelist (nextstat); B.falselist := makelist (nextstat + 1);
          emit (‘if’ B.place ‘goto’ __);
          emit (‘goto’ __);
          }

M→    ε                {M.stat := nextstat}
E →E1 + E2    |        other rules
                               { same actions as given earlier}
```

The attribute, M.stat, for nonterminal M is used to hold the index of the next 3-address statement at strategic points in the code.

We use the term marker nonterminal for nonterminals, such as M, which are used in this manner. The following functions and variables have also been used

1. makelist(i) is a function that creates a list containing a single element { i } and returns a pointer to the list created. The element i is an index of a 3-address statement.
2. merge(p1, p2) is a function that concatenates two lists pointed by p1 and p2 and returns a pointer to the merged list.
3. backpatch(p, i) is a function that inserts i in the place for the missing target label in each element of the list (of 3-address statements) pointed to by p.
4. nextstat is a global variable that is used to hold the index of the next 3-address statement.

The method for partial evaluation manages to generate code in a single pass over the source code.

It holds the list of incomplete statements and carries them around, along with parsing, till such time that till their target labels become available.

When the target labels are found, these are placed at the label fields of incomplete codes using the lists being carried along as semantic attributes.

The code generated is assuredly correct but may not be efficient.

Boolean expressions may be evaluated by either of the two methods. However, the choice of translation does not rest with the implementor, it is a part of the language specifations.

Partial evaluation is in some sense an efficient form of complete evaluation. However the two translations may not give identical results, particularly in the presence of side effects.

Partial evaluation is typically suited for translation of boolean expressions which are part of control flow constructs. Complete evaluation is used in the context of assignment statements.

TRANSLATION OF CONTROL FLOW CONSTRUCTS

We now take up translation of flow of control statements, such as if-then-else and while-do.

The boolean expression component of these statements are partially evaluated as explained already.

For each control flow construct, the following need to be done.

1. Determine from the context, how to resolve the true and false lists of the associated boolean expression.

2. **How to link the different components of the constructs so that the translation is semantically equivalent ?**

   For example, if-then-else statement has three components, boolean expression and two statement parts.

   These components have to be connected in a manner to ensure the correct flow of control and semantics as specified for this construct.

3. Three-address statements, which have jumps to outside the body of the construct, cannot be resolved at generation time.

These statements must be carried forward till the required target becomes known. Incomplete 3-address code may occur within the code generated for

i. boolean expression,

ii. other components of the construct, and

iii. 3-address jump statements inserted to link the various components.

We assume, for now, that the only kinds of jump allowed from within a control flow construct is to the statement following the construct (i.e., unconditional goto is absent in the source program).

In addition to the attributes, B.truelist, B.falselist, M.stat and the functions makelist(), merge() and backpatch() and a variable nextstat the following are required.

a) a nonterminal N , whose purpose is to prevent control flow from then part into the else part of an if-then-else statement. It causes the generation of an incomplete unconditional 3 address jump statement.

   The index of this statement is held in an attribute N.nextlist.

The target jump is resolved by backpatching it with the index of the address code following the if-then-else.

b) Indices of the incomplete 3-address statements generated for S are held in an attribute S.nextlist.

c) Similarly Slist.nextlist holds a list of indices of all the incomplete 3- address code corresponding to the list of statements denoted by Slist.

```
S → if B then M S₁

        { backpatch(B.truelist, M.stat) ;

         S.nextlist := merge(B.falselist, S₁.nextlist)
        }


S → if B then M₁ S₁ N else M₂ S₂


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

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


S → begin Slist end
        { S.nextlist := Slist.nextlist }


S → L := E

        { semantic rules given earlier;
          S.nextlist := nil
        }


Slist → Slist₁ ; M S
```

```
            { backpatch (Slist1.nextlist, M.stat);
              Slist.nextlist := S.nextlist
            }

    Slist   → S     {Slist.nextlist := S.nextlist}

    M → ε {M.stat := nextstat}
    N → ε {  N.nextlist := makelist (nextstat);
                  emit ('goto' __)
                }
```

We illustrate the discussion with an example given below.

The symbols, ↕$_n$, are used to mark certain points of the program fragment, to illustrate generation of code, in an incremental fashion along with parsing. The marked points are clearly not part of the source program.

```
    begin
        if a > b ↕1 then
            if b > c ↕2 then
                begin
                d := b + c;  b := b / 2
                end ↕3
            else b := 2 * b  ↕4
        else
            while a > 0 ↕5 do
                begin
```

```
                                        d := b * b; a := a / 2
                                end ↕6 ;
                b : = b / 2 ↕7
        end ↕8
```

The source code has an outer if-then-else which has a nested if-then-else in its then part and a nested while loop in its else part.

It will be instructive to use the grammar over the input string and generate the intermediate code yourself along with parsing. You may assume a bottom up parser for this exercise.

The intermediate code that gets generated at the marked points, the values of semantic attributes, as parsing proceeds and semantic actions are executed are also given.

A number of stack configurations at intermediate stages of parsing are shown. The values of semantic attributes along with intermediate code fragments generated at that point of parsing show how syntax directed translation can be made to work correctly.

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Assume that nextstat is 10 at start. Relevant stack configurations around the marked points are shown.

Stack Contents **Parsing & semantic Actions**

Stack Configuration 1 :

begin if a > b

Reduction by B $\rightarrow$E1 relop E2 ;

B.truelist = {10}; B.falselist = {11} B ≡ a > b

begin if B$\updownarrow_1$

Code generated so far :

10 : if a > b goto __

11 : goto __

B.truelist = {10}; B.falselist = {11}

Stack Configuration 2:

begin if B then M_1 if b > c

M_1.stat = {12};

Reduction by B $\rightarrow$E1 relop E2 ;

B.truelist = {12}; B.falselist = {13} for B ≡ b > c

begin if B then M_1 if B$\updownarrow_2$

Code generated so far :

10 : if a < b goto __

11 : goto __

12 : if b > c goto __

13 : goto __

Stack Configuration 3:

begin if B then M_1 if B then M_1 begin d := b + c

M_1.stat = {14};

Reduction; S.nextlist = Φ

begin if B then M_1 if B then M_1 begin S

Code generated so far :

10 : if a < b goto __

11 : goto __

12 : if b > c goto __

13 : goto __

14 : t_1 := b + c

15 : d := t_1

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 4:

begin if B then M_1 if B then M_1 begin Slist; M b := b/2 reduction; Slist.nextlist = Φ; M.stat = {16}

begin if B then M_1 if B then M_1 begin Slist; M S

S.nextlist = Φ; reduction

Code generated so far :

10 : if a < b goto __	14 : t_1 := b + c
11 : goto __	15 : d := t_1
12 : if b > c goto __	16 : t_2 := b / 2
13 : goto __	17 : b := t_2

Stack Configuration 5:

begin if B then M_1 if B then M_1 begin S end

S.nextlist = Φ; reductions

Stack Configuration 6:

begin if B then M_1 if B then M_1 S_1 N $\updownarrow_3$

N.nextlist = {18}

Code generated so far :

10 : if a < b goto __	15 : d := t_1
11 : goto __	16 : t_2 := b / 2
12 : if b > c goto __	17 : b := t_2
13 : goto __	18 : goto __
14 : t_1 := b + c	

Stack Configuration 7:

begin if B then M_1 if B then M_1 S_1 N else M_2 b := 2/ b

M_2.stat = {19}; reduction

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 8:

begin if B then M_1 if B then M_1 S_1 N else M_2 S_2

S.nextlist = Φ; reduction;

Stack Configuration 9:

begin if B then M_1 S_1 $\updownarrow_4$

backpatch({12}, 14);

backpatch({13}, 19);

Code generated so far :

10 : if a < b goto __
11 : goto __
12 : if b > c goto **14**
13 : goto **19**
14 : t_1 := b + c
15 : d := t_1
16 : t_2 := b / 2
17 : b := t_2
18 : goto __
19: t_3 := 2 * b
20 : b := t_3
21 : goto __

Stack Configuration 10:

begin if B then M_1 S_1 N else M_2 while M_1 a > 0

M_2.stat = {22}; M_1.stat = {22}; reduction;

Stack Configuration 11:

begin if B then M_1 S_1 N else M_2 while M_1 B $\updownarrow_5$

B.truelist = {22}; B.falselist = {23};

Code generated so far :

10 : if a < b goto __
11 : goto __
12 : if b > c goto 14
13 : goto 19
14 : t_1 := b + c
15 : d := t_1
16 : t_2 := b / 2
17 : b := t_2
18 : goto __
19: t_3 := 2 * b
20 : b := t_3
21 : goto __
22 : if a > 0 goto __
23 : goto __

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 12:

begin if B then M_1 S_1 N else M_2 while M_1 B do M_2 begin d := b * b

M_2 .stat = {24}; reduction;

Code generated so far :

10 : if a < b goto __
11 : goto __
12 : if b > c goto 14
13 : goto 19
14 : t_1 := b + c
15 : d := t_1
16 : t_2 := b / 2
17 : b := t_2
18 : goto __
19: t_3 := 2 * b
20 : b := t_3
21 : goto __
22 : if a > 0 goto __
23 : goto __
24 : t_4 := b * b
25 : d := t_4

Stack Configuration 13:

begin if B then M_1 S_1 N else M_2 while M_1 B do M_2 begin Slist; M a := a/2

M_2 .stat ={26}; reductions; Slist.nextlist = Φ

Stack Configuration 14:

begin if B then M_1 S_1 N else M_2 while M_1 B do M_2 begin Slist; MS

Slist.nextlist = Φ; S.nextlist = Φ

Code generated so far :

```
10 : if a < b goto __
11 : goto __
12 : if b > c goto 14
13 : goto 19
14 : t1 := b + c
15 : d :=  t1
16 : t2 := b / 2
17 : b :=  t2
18 : goto __
19:  t3 := 2 * b
20 : b :=  t3
21 : goto __
22 : if a > 0 goto __
23 : goto __
24 : t4 := b * b
25 : d :=  t4
26 : t5 := a / 2
27 : a := t5
```

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 15:

begin if B then M_1 S_1 N else M_2 while M_1 B do M_2 begin Slist end

Slist.nextlist = Φ; reduction; M_1.stat ={22}

Stack Configuration 16:

begin if B then M_1 S_1 N else M_2 S_2

S_2.nextlist = {23}; S_1.nextlist = {18, 21};
backpatch({22}, 24); M_1.stat = {12}
M_2.stat = { }; N.nextlist ={21};
B.truelist = {10}; B.falselist = {11}; reduce;

Code generated so far :

```
10 : if a < b goto __
11 : goto __
12 : if b > c goto 14
13 : goto 19
14 : t1 := b + c
15 : d :=  t1
16 : t2 := b / 2
17 : b :=  t2
18 : goto __
19:  t3 := 2 * b
```

20 : b := t_3
21 : goto __
22 : if a > 0 goto **24**
23 : goto __
24 : t_4 := b * b

25 : d := t_4
26 : t_5 := a / 2
27 : a := t_5
28 : goto **22**

Stack Configuration 17:
begin S $\updownarrow_{6}$

S.nextlist = {18, 21, 23};
backpatch ({10}, 12); backpatch ({11}, 13);
reduce;

Code generated so far :

10 : if a < b goto **12**
11 : goto **13**
12 : if b > c goto 14
13 : goto 19
14 : t_1 := b + c
15 : d := t_1
16 : t_2 := b / 2
17 : b := t_2
18 : goto __
19: t_3 := 2 * b

20 : b := t_3
21 : goto __
22 : if a > 0 goto 24
23 : goto __
24 : t_4 := b * b
25 : d := t_4
26 : t_5 := a / 2
27 : a := t_5
28 : goto 22

Stack Configuration 18:
begin Slist ; M b := b / 2

Slist.nextlist = {18, 21, 23}; M.stat={29}
reduce;

Stack Configuration 19:
begin Slist ; M S

Slist.nextlist = {18, 21, 23}; M.stat={29}
S.nextlist = Φ; reduce;

Code generated so far :

10 : if a < b goto 12

11 : goto 13

12 : if b > c goto 14
13 : goto 19
14 : t_1 := b + c
15 : d := t_1
16 : t_2 := b / 2
17 : b := t_2
18 : goto __
19: t_3 := 2 * b
20 : b := t_3
21 : goto __
22 : if a > 0 goto 24
23 : goto __
24 : t_4 := b * b
25 : d := t_4
26 : t_5 := a / 2
27 : a := t_5
28 : goto 22
29 : t_6 := b / 2
30 : b := t_6

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 20:

begin Slist $\updownarrow_8$ backpatch ({18, 21, 23}, 29);
S.nextlist = Φ;

Code generated so far :

10 : if a < b goto 12
11 : goto 13
12 : if b > c goto 14
13 : goto 19
14 : t_1 := b + c
15 : d := t_1
16 : t_2 := b / 2
17 : b := t_2
18 : goto **29**
19: t_3 := 2 * b
20 : b := t_3
21 : goto **29**
22 : if a > 0 goto 24
23 : goto **29**
24 : t_4 := b * b
25 : d := t_4
26 : t_5 := a / 2
27 : a := t_5
28 : goto 22
29 : t_6 := b / 2
30 : b := t_6

Stack Configuration 21:

begin Slist end Slist.nextlist = Φ;

Stack Configuration 22:

S S.nextlist = Φ; Accept

Code generated :

10 : if a < b goto 12
11 : goto 13
12 : if b > c goto 14
13 : goto 19
14 : t_1 := b + c
15 : d := t_1
16 : t_2 := b / 2
17 : b := t_2
18 : goto 29
19: t_3 := 2 * b
20 : b := t_3
21 : goto 29
22 : if a > 0 goto 24
23 : goto 29

24 : t_4 := b * b
25 : d := t_4
26 : t_5 := a / 2
27 : a := t_5
28 : goto 22
29 : t_6 := b / 2
30 : b := t_6

TRANSLATION OF OTHER CONSTRUCTS

The basic issues in semantic analysis and translation have been covered. However, a compiler has to deal with more details, a few of which are listed below.

Translation of goto statement. It is handled by storing labels in the symbol table, associating index of 3-address code with the label and using backpatching to resolve forward references.

Procedure call statement

Let parameter passing be call by reference and storage be statically allocated.

a) Semantic analysis involves type checking of the formal and actual arguments.

b) Translation would require generating 3-address code for evaluating each argument (in case it is an expression). Then generate a list of 3-address **param** statements, one for each argument, followed by the 3-address call statement.

Translation of other constructs such as **case** statement, **for** statement and **repeat - until** statement can be worked out on similar lines.

We have ignored semantic analysis for checking structural integrity of single-entry control structures. Code for testing such violations also need to be included.

5. Translation of Other constructs : The basic issues in semantic analysis and translation have been covered. However, a compiler has to deal with more details, a few of which are listed below. Translation of goto statement or break statements. It is handled by storing labels in the symbol table, associating index of 3-address code with the label and using backpatching to resolve forward references.

Procedure call statement

Consider a function definition :

```
    void func (int a, int * p, float f)
     {  /* body skipped */   return; }
```

and a corresponding call to func() as given below.

func(i*j+5, &q+5, 25 *pi); // i, j are int; q is int*; and pi is float)

the desired intermediate code is. Note how each actual argument is evaluated into a temporary of the same type as that defined in the corresponding formal argument. The use of param statements, one for each actual parameter followed by a call to the function.

```
10: t1 := i * j          17: t8 := intoreal t6
11: t2 := t1 + 5         18: t9 := t8*pi
12: t3 := &q             19: param t2
13: t4 := 5*4            19 : param t6
14: t5 := t3+t4          20 : param t9
15: t6 := &t5            21 : call func
16 : t7 = 25
```

Parameter passing may be be call by value or call by reference and are processed accordingly..

a) Semantic analysis involves type checking of the formal and actual arguments.
b) Translation would require generating 3-address code for evaluating each argument (in case it is an expression). Then generate a list of 3-address **param** statements, one for each argument, followed by the 3-address call statement.

Translation of other constructs such as **case** statement, can be worked out on similar lines.