

Lecture 3

System Programming

Fundamentals of Language Processing

Language Processing
Analysis of SP + Synthesis of TP

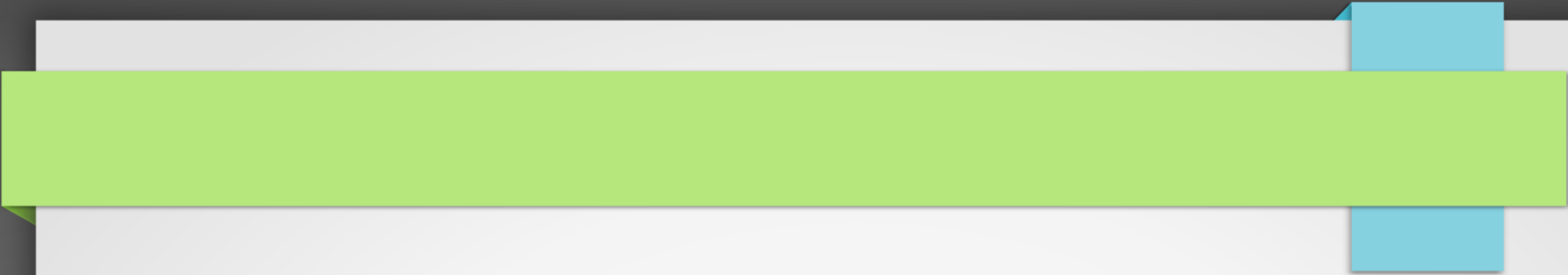
Specification of source language forms the basis of source program analysis

Specification consists of three components

- **Lexical rules** :- which govern the formation of valid lexical units in source language
- **Syntax rules** :- which govern the formation of valid statements in source language
- **Semantic rules** :- which associate meaning with valid statements of the language

Analysis Phase

- Lexical analysis
- Syntactic analysis
- Semantic analysis



```
int a;  
float c;  
c=a;
```

Synthesis phase

Construction of target language statement (s) which have the same meaning as a source statement

Two main activities

- Creation of data structures in the target program (Memory allocation)

- Generation of target program (Code generation)

A language processor generates the following assembly statements for the source statement

```
MOVER    AREG,A
ADD      AREG,B
MOVEM    AREG,C
```

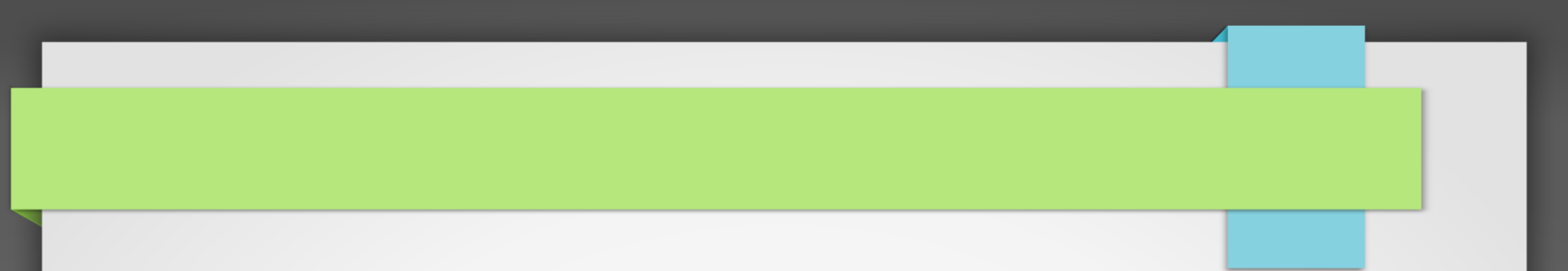
.....

```
A        DW      |
B        DW      |
C        DW      |
```

```
int a;
float b,c;
c=a+b;
```

MOVER and MOVEM move a value from a memory location to CPU register and vice versa

DW reserves one or more words in memory



Pass I : Perform analysis of the source program and note relevant information

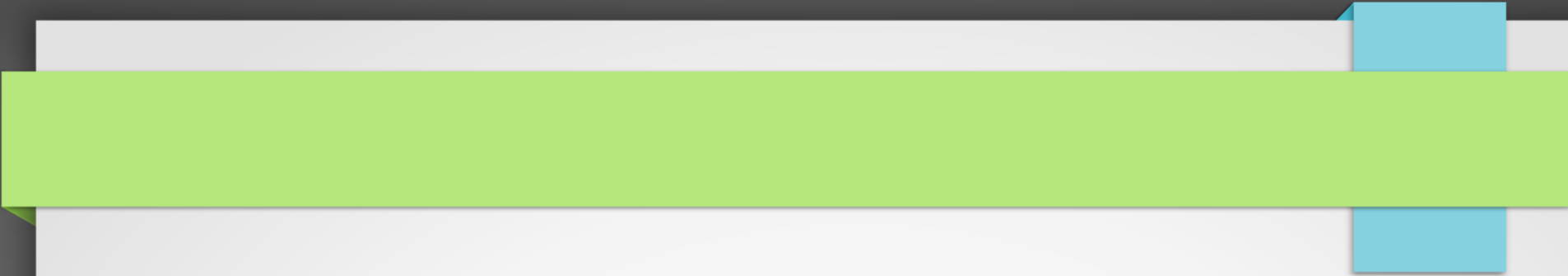
Pass II : Perform synthesis of target program

Intermediate Representation

In pass I, it analyses the source program to note the type of information

In pass II, it once again analyses the source program to generate target code using the type information noted in pass I

Avoided using an intermediate representation of the source program



Intermediate Representation (IR) is a representation of a source program which reflects the effect of analysis and synthesis tasks performed during language processing

The IR is the 'equivalent representation'

The first pass performs analysis of source program and reflects its results in the intermediate representation

The second pass reads and analyses the IR, instead of the source program to perform synthesis of target program which avoids repeated processing of source program



First pass is concerned exclusively with source language issues
- front end of language processor

Second pass is concerned with program synthesis for a specific target language - back end of language processor

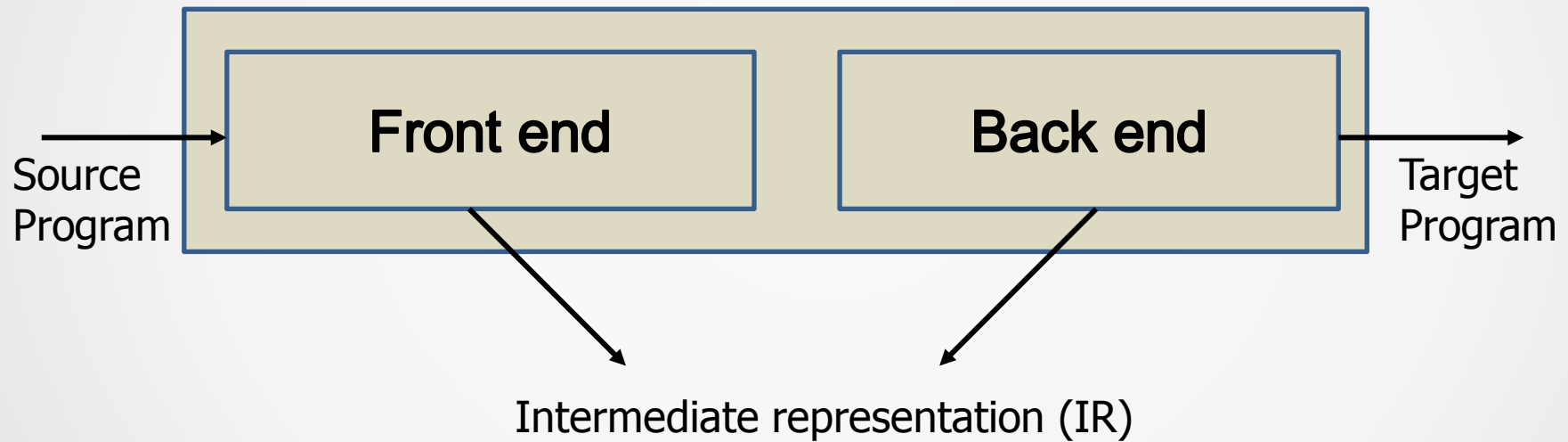


Fig : Two pass schematic for language processing



Desirable properties of an IR are

Ease of use :- IR should be easy to construct and analyse

Processing efficiency :- Efficient algorithms must exist for constructing and analysing the IR

Memory efficiency :- IR must be compact

Semantic Actions

All actions performed by the front end , except lexical and syntax analysis, are called semantic actions

Following Actions

- Checking semantic validity of constructs in SP
- Determining the meaning of SP
- Constructing an IR

Toy compiler – An example

Front end

It performs lexical, syntax and semantic analysis of source program.

Each kind of analysis involves the following functions

- Determine validity of a source statement from the view point of analysis
- Determine the 'content' of source statement
- Construct a suitable representation of source statement for use by subsequent analysis functions, or by the synthesis phase of language processor



‘content’ has different connotations in lexical, syntax and semantic analysis

- Lexical analysis - lexical class to which each lexical unit belongs
- Syntax analysis - syntactic structure of a source statement
- Semantic analysis - meaning of a statement

Declaration statement, it is the set of attributes of a declared variable (eg. Type, length and dimension)

Imperative statement, it is the sequence of actions implied by the statement



Each analysis represents the 'content' of a source statement in the form of

- (1) tables of information
- (2) description of source statement

Subsequent analysis uses this information for its own purposes and either adds information to these tables and descriptions, or constructs its own tables and descriptions

Semantic analysis uses information concerning the syntactic structure and constructs a representation for the meaning of the statement

The tables and descriptions at the end of semantic analysis form the IR of front end

Output of front end

The IR produced by front end consists of two components

- Tables of information
- An intermediate code (IC) which is a description of source program

Tables

Contain information obtained during different analyses of SP

Most important table is the **symbol table** which contains information concerning all identifiers used in SP



Symbol table is built during lexical analysis

Semantic analysis adds information concerning symbol attributes while processing declaration statements

It may also add new names designating temporary results

Intermediate code (IC)

It is a sequence of IC units, each IC unit representing the meaning of one action in SP

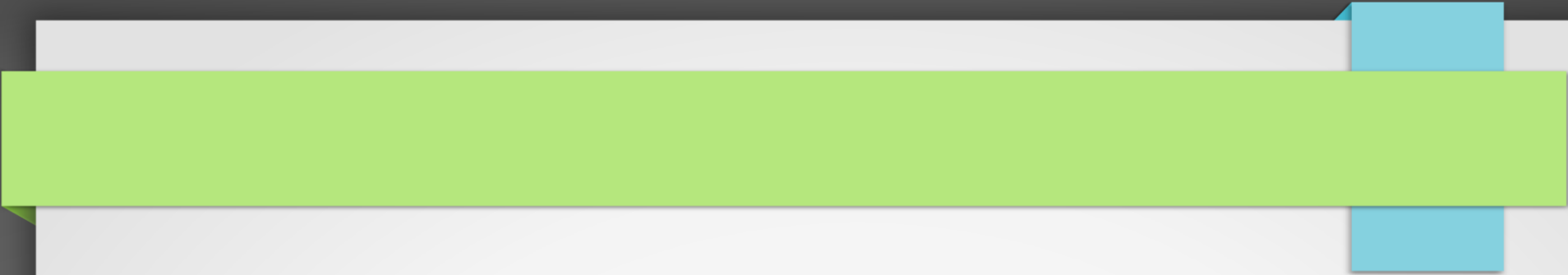
IC units may contain references to the information in various tables

int i;
float a,b;
a:=b+i;

Intermediate Code (IC)
Convert (ld,#1) to real, giving (ld,#4)
Add (ld,#4) to (ld,#3), giving (ld,#5)
Store (ld,#5) in (ld,#2)

Symbol table

| | symbol | type | length | address |
|----------|---------------|-------------|---------------|----------------|
| 1 | i | int | | |
| 2 | a | float | | |
| 3 | b | float | | |
| 4 | i* | float | | |
| 5 | temp | float | | |



The symbol table contains information concerning the identifiers and their types

This information is determined during lexical and semantic analysis

In IC, the specification (Id,#1) refers to the id occupying the first entry in table

i* and temp are temporary names added during semantic analysis of the assignment statement

Lexical analysis (scanning)

It identifies the lexical units in a source statement

It then classifies the units into different **lexical classes**, e.g identifiers, constants, reserved identifiers, etc and enters them into different tables

Lexical analysis builds a descriptor, called a token, for each lexical unit

Token contains **two fields** – **class code**, and **number** in class code

Class code :- identifies the class to which a lexical unit belongs

Number in class :- is the entry number of lexical unit in relevant table

We depict a token as *Code #no*, eg. *Id#10*

The IC for a statement is thus a string of tokens

The statement $a=b+i$; is represented as string of tokens

Id#2 Op#5 Id#3 Op#3 Id#1 Op#10

Here Id#2 stands for 'identifier occupying entry #2 in symbol table' i.e. the id a

Op#5 similarly stands for the operator '='

Syntax analysis (Parsing)

It **processes the string of tokens built by lexical analysis to determine the statement class**, e.g. assignment statement, if statement, etc

It then builds an IC which represents the structure of the statement

The IC is passed to semantic analysis to determine the meaning of statement

Fig shows IC for statements float a, b; and a:=b+i;

A tree form is chosen for IC because a tree can represent the hierarchical structure of a PL statement appropriately.

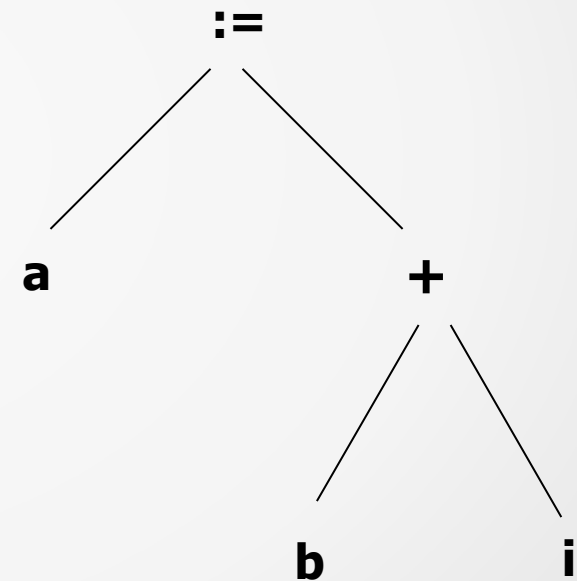
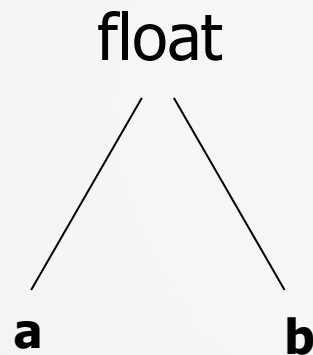


Fig : IC for the statements :
`float a,b;`
`a:=b+i;`

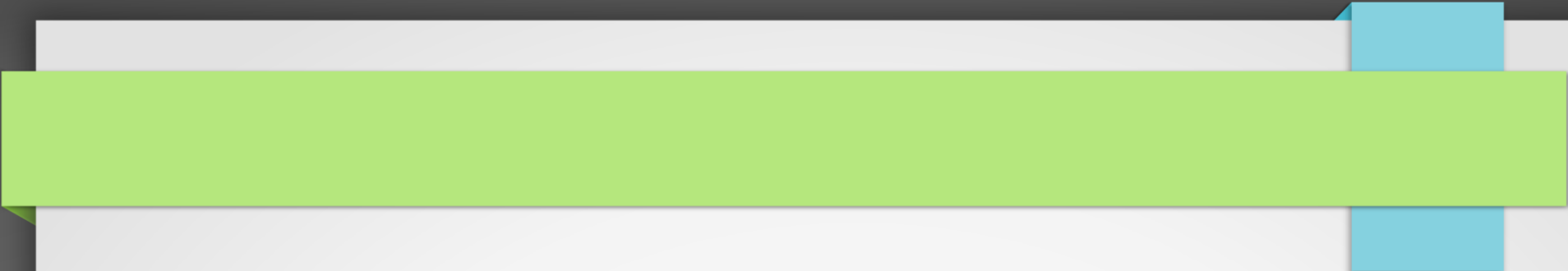
Semantic analysis

Semantic analysis of declaration statements differs from semantic analysis of imperative statements

Former results in addition of information to the symbol table, e.g. type, length and dimensionality of variables

Latter identifies the sequence of actions necessary to implement the meaning of a source statement

In both cases, structure of a source statement guides the application of semantic rules



When semantic analysis determines the meaning of a subtree in IC, it adds information to a table or adds an action to sequence of actions

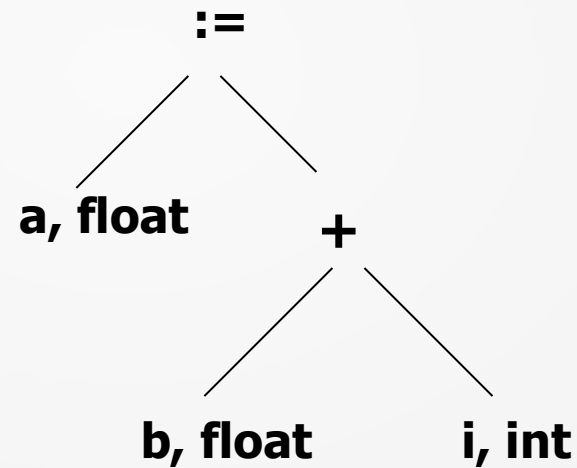
It then modifies the IC to enable further semantic analysis

The analysis ends when the tree has been completely processed

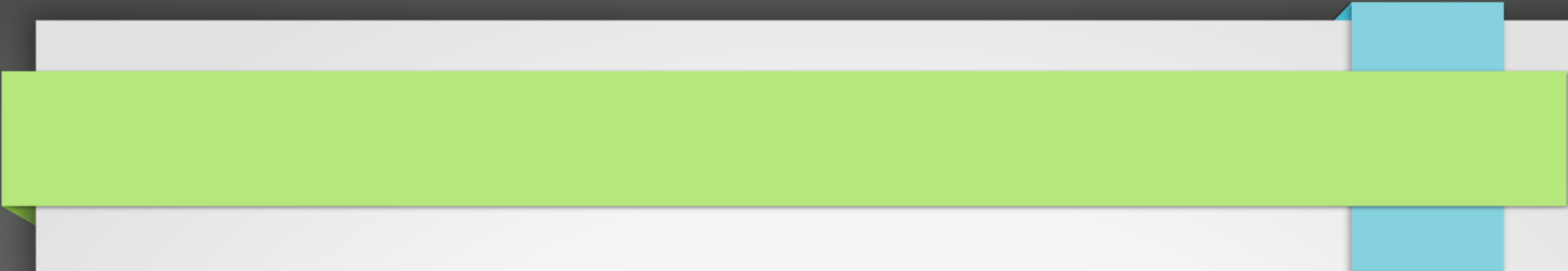
The updated tables and the sequence of actions constitute the IR produced by analysis phase

Semantic analysis of statement $a=b+i$;

Information concerning the type of the operands is added to IC tree. The IC tree now looks as in Fig (a)



(a)

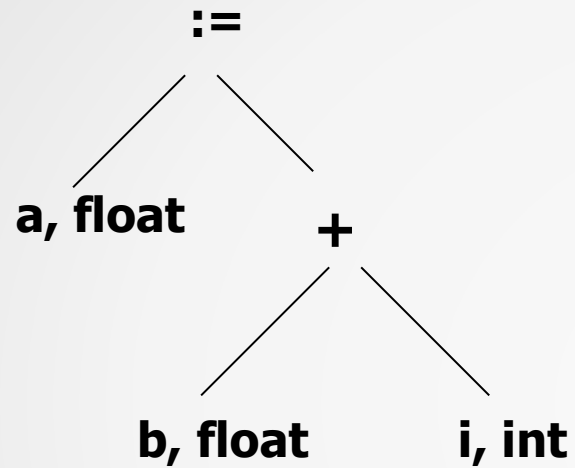


Rules of addition indicate that type conversion of i should be performed to ensure type compatibility of the operands of '+'.

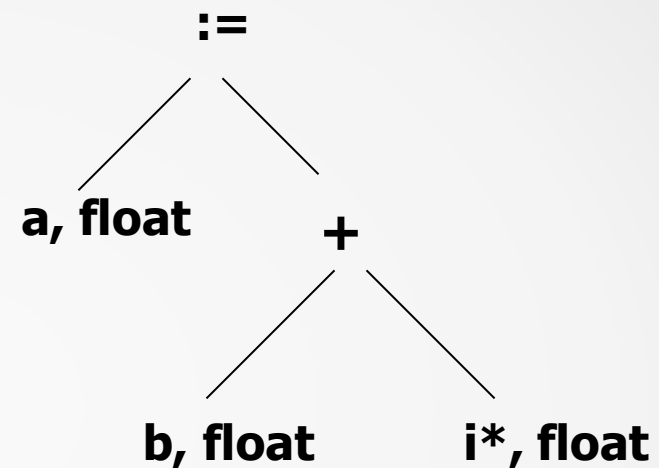
This leads to the action

(i) Convert i to real, giving i^*

which is added to sequence of actions. The IC tree under consideration is modified to represent the effect of this action (fig (b)). The symbol i^* is now added to symbol table

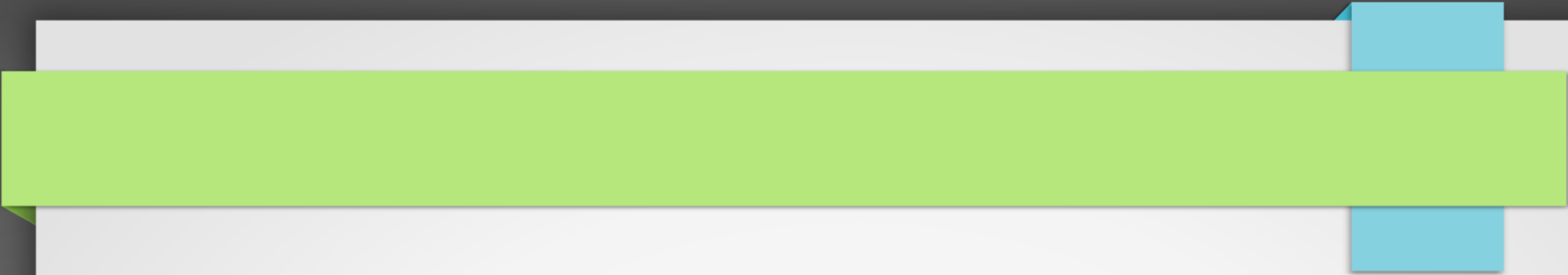


(a)



(b)

Fig : Steps in semantic analysis of an assignment statement



Rules of addition indicate that addition is now feasible. This leads to action

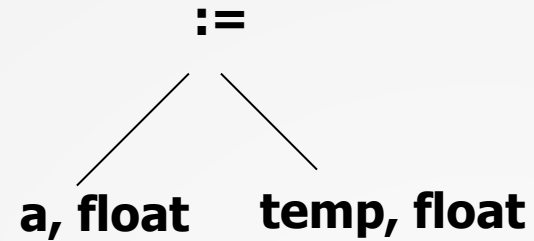
(ii) Add i^* to b , giving $temp$

The IC tree is transformed as shown in Fig (c), and $temp$ is added to symbol table

The assignment can be performed now. This leads to action

(iii) Store $temp$ in a

This completes semantic analysis of the statement



(c)

Fig : Steps in semantic analysis of an assignment statement

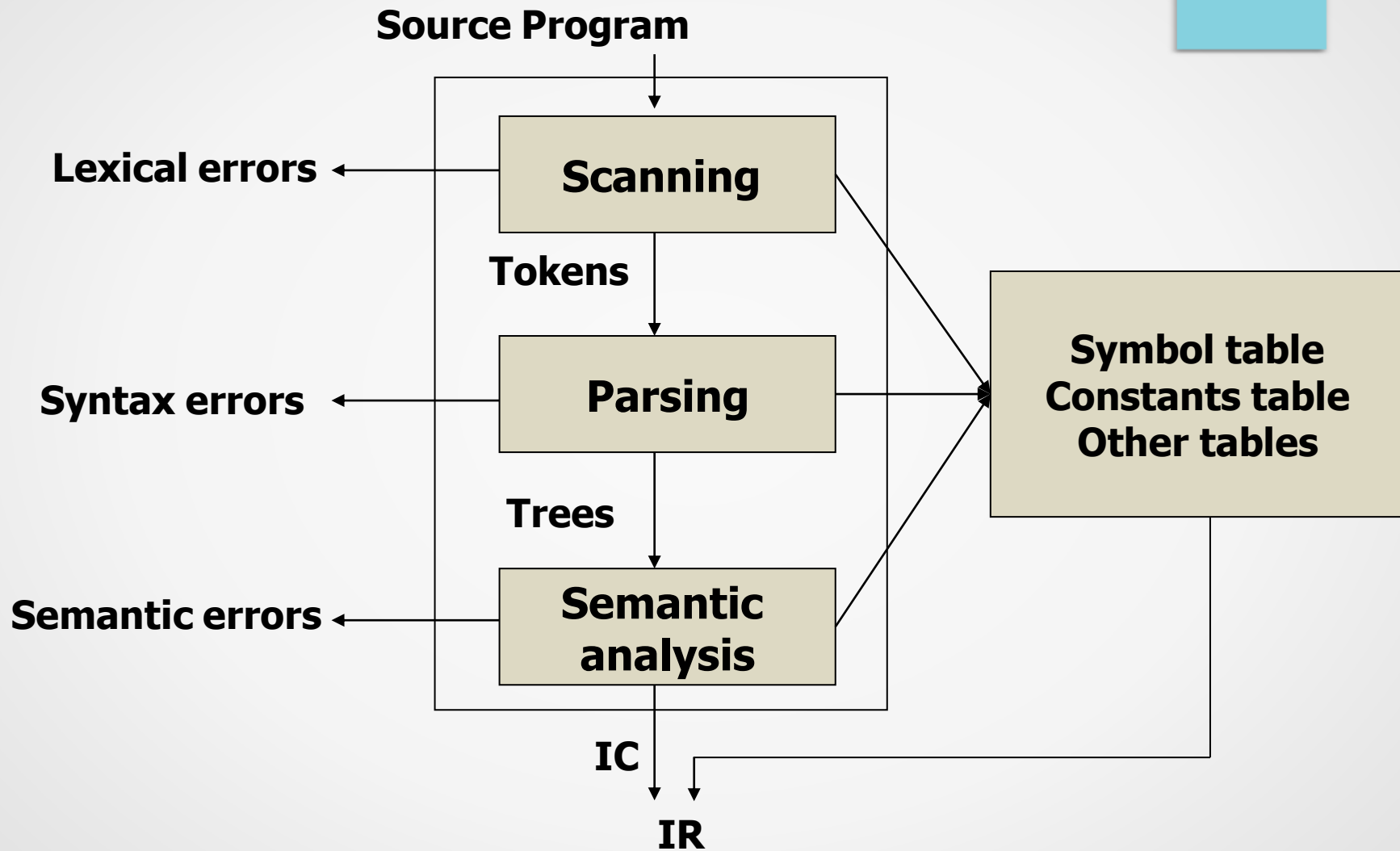


Fig : Front end of the toy compiler



Back end

It performs memory allocation and code generation

Memory allocation

It is a simple task given the presence of symbol table

The memory requirement of an identifier is computed from its type, length and dimensionality, and memory is allocated to it

Address of memory area is entered in symbol table

After memory allocation, the symbol table looks as shown. The entries for i^* and temp are not shown because memory allocation is not needed for these id's

| | symbol | type | length | address |
|---|--------|-------|--------|---------|
| 1 | i | int | | 2000 |
| 2 | a | float | | 2001 |
| 3 | b | float | | 2002 |

Note that certain decisions have to precede memory allocation, eg, whether i^* and temp should be allocated memory. These decisions are taken in the preparatory steps of code generation.

Code generation

It uses knowledge of target architecture, viz. knowledge of instructions and addressing modes in the target computer, to select the appropriate instructions

Important issues in code generation are

- Determine the places where **intermediate results** should be kept, i.e. memory locations or machine registers
- Determine which instructions should be used for **type conversion** operations
- Determine which **addressing modes** should be used for accessing variables

For the sequence of actions for the assignment statement $a=b+i$;

- (i) Convert i to real, giving i^*
- (ii) Add i^* to b , giving temp
- (iii) Store temp in a

| | |
|--------|---------|
| CONV_R | AREG, I |
| ADD_R | AREG, B |
| MOVEM | AREG, A |

Some issues involved in code generation may require designer to look beyond machine architecture

Eg. Whether or not the value of temp should be stored in memory location would partly depend on whether the value of $b+i$ is used more than once in program

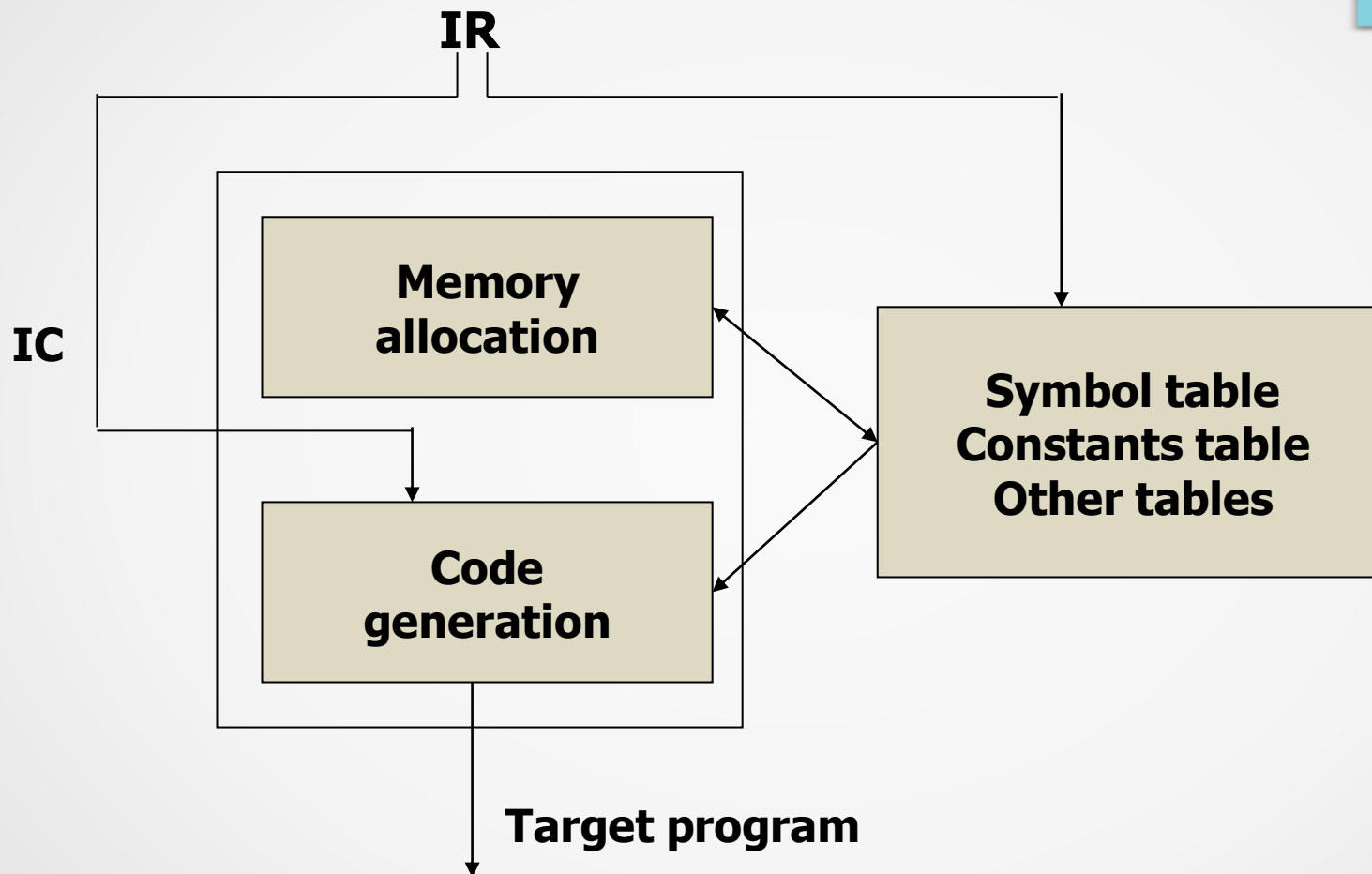


Fig : Back end of the toy compiler