

Language Processors

Chapter 1.

By: Bhargavi H Goswami
Assistant Professor
Sunshine Group of Institutes
Rajkot, Gujarat, India

Is it ok if Analysis of Source Statement is followed by synthesis of equivalent target statement?

- Answer: No.
- Consequences:
 - 1. Forward Reference: The program entity having reference to the entity which precedes its definition in the program.
i.e Declaration Later, Reference First.
Eg: `Percent_Profit = (Profit * 100)/cost_price;`

`long Profit;`
 - 2. Issue concerning memory requirement and organization of a language processor.
- So, now I don't want statement to statement application of Analysis and then synthesis of instruction of SP to TP, what?
- Solution? Multi-pass model of language processor.

Language Processor Pass:

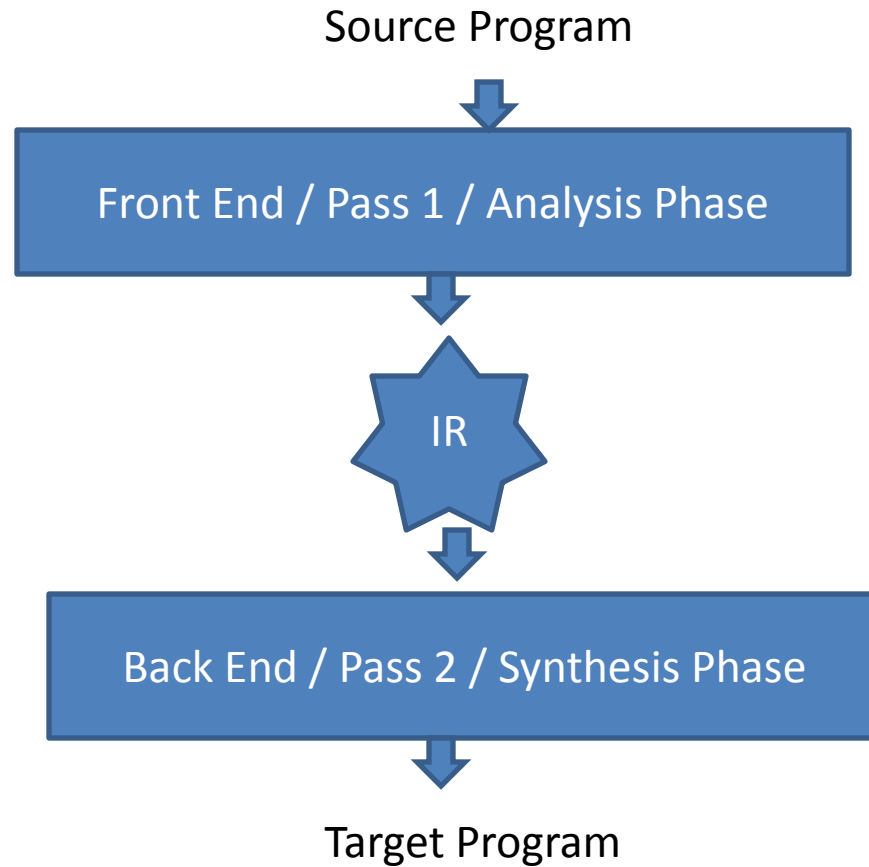
- A language processor pass is a processing of every statement in a source program, or its equivalent representation, to perform a language processing function/s.
- Pass1:
 - Performs analysis of source program and note relevant information.
- Pass 2:
 - Performs synthesis of target program.
- Eg: $\text{Percent_Profit} = (\text{Profit} * 100) / \text{cost_price};$
Pass 1: Performs notification of type of “profit”.
Pass 2: Using information of Pass-1 code generation is performed.

- What are the problems with Multi-pass model?
- Such processors perform certain operations more than once.
- How?
- Answer:
 - Pass 1: Analysis of SP
 - Pass 2: Analysis of SP + Synthesis of TP.
- Solution?
- IR – Intermediate Representation.

IR- Intermediate Representation

- An IR is a representation of a source program which reflects the effect of some, but not all, analysis & synthesis task performed during language processing. (see next fig).
- Benefit of IR:
 - 1. Memory Requirement Reduced.
 - 2. Now; No need of co-existence of front end & back end in memory.
- Desired Property of IR:
 - 1. Ease of Use: IR should be easy to construct & analysis.
 - 2. Processing Efficiency: Efficient algorithm must exist for constructing & analyzing IR.
 - 3. Memory Efficiency: IR must be compact.

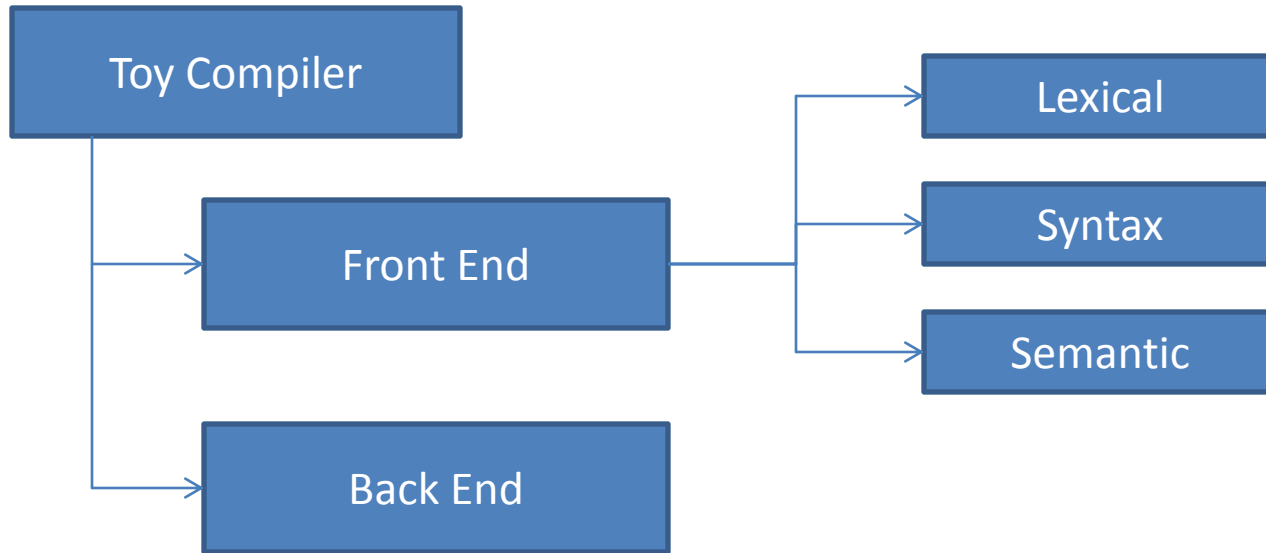
IR :



Semantic Action

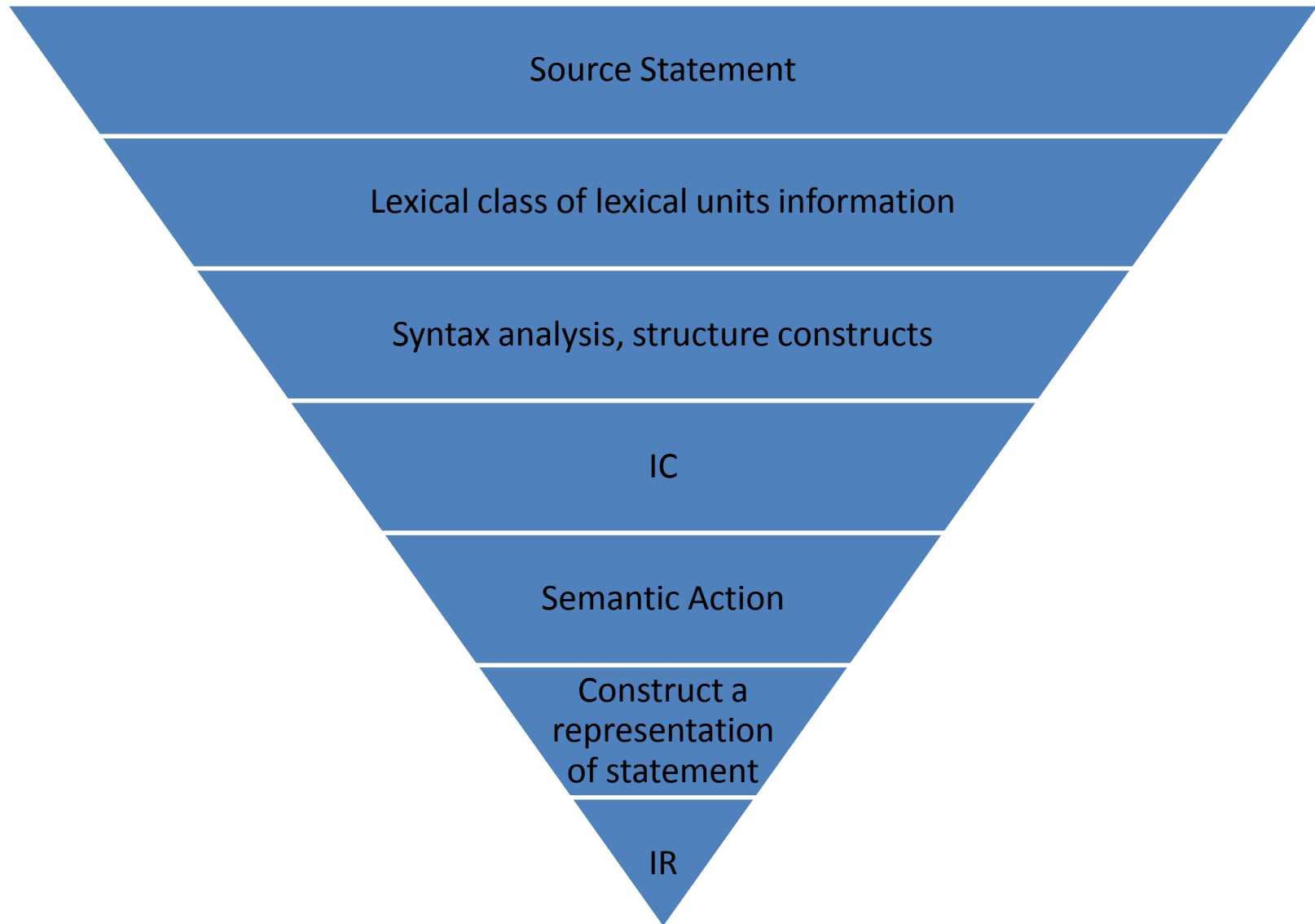
- All actions performed by front end except by lexical and syntax analysis are called semantic action.
- This include actions like:
 - 1. Checking Semantic validity of constructs in source program.
 - 2. Determining meaning of source program.
 - 3. Constructing of IR.

Toy Compiler



1. Front End:

1. It includes 3 analysis:
 1. Lexical
 2. Syntax
 3. Semantic
2. Functions of Front End:
 1. Determine validity of source statement from view point of analysis.
 2. Determine 'content' of source statement.
 3. Construct a suitable representation of source statement for the use of subsequent analysis function or synthesis phase of a language processor.
3. Result is stored in the following forms:
 1. Table of Information
 2. IC: Description of source statement.
4. Output of Front End:
 1. IR : Intermediate Representation. It has two things with it. 1. Table of Information. 2. Intermediate Code (IC).



a. Table of Content

- Contains information obtained during different analysis of source program.
- Eg. Symbol Table: Has information containing all the identifiers used in source program.
- Semantic analysis adds more information to symbol table.
- So, Symbol table have information about lexical analysis + new names designated by temporary results.
- Eg: IR produced by analysis phase of program:
i : integer;
a, b : real;
a := b + i;

Sr. No.	Symbol	Type	Length	Address
1	i	Integer	-	-
2	a	Real	-	-
3	b	Real	-	-
4	i*	Real	-	-
5	temp	Real	-	-

b. IC: Intermediate Code

- Has sequence of IC units.
- What does IC do?
- Each IC unit represent the meaning of each individual actions performed based on source program.
- IC unit may contain reference to information in various tables.
- Actions:
 - Convert (id,#1) to real, giving (id,#4)
 - Add (id,#4) to (id,#3), giving (id,#5)
 - Store (id,#5) in (id,#2)

Lexical Analysis of Front End

- Also called Scanning.
- Identifies lexical units and then classifies these units further into lexical classes like id's, constants, identifiers, reserved keywords and then enter them into different tables.
- Output: Token, Is the description of analysis. It has two parts. 1. Class Code, 2. Number.
- Eg: OP Table
- Class code identifies class to which a lexical unit belongs.
- Number in class is the entry number in lexical unit in relevant table.

String of token:

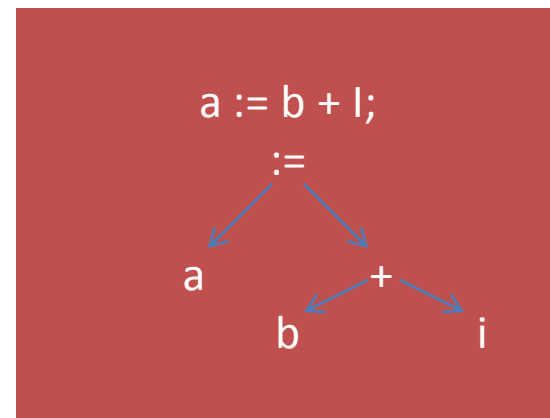
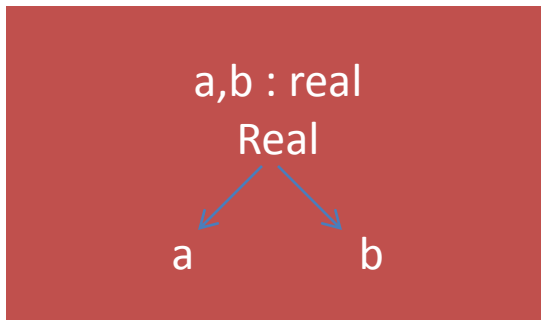
a := b + i;

Id#2 op#5 id#3 op#3 id#1 op#1

Number	Class Code: OP
1	;
2	-
3	+
4	*
5	:=

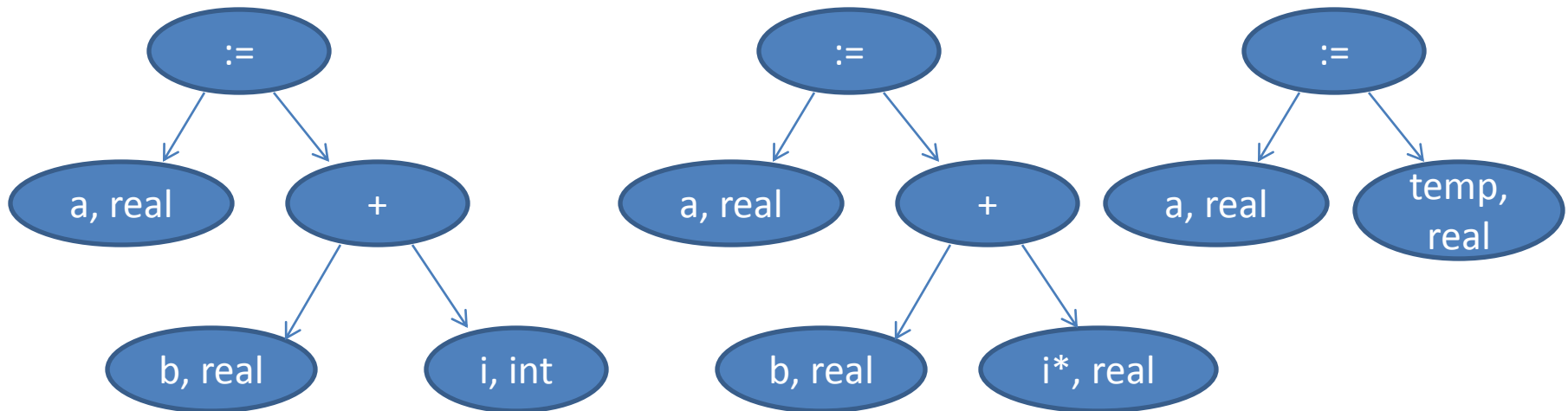
Syntax Analysis of Front End

- Also called Parsing.
- Syntax analysis process is the string of tokens built by lexical analyzer to determine the statement class.
- i.e Assignment Statement Class,
For Loop Class,
Switch Case, etc
- It builds IC representing the structure of statements in tree form.
- Why tree is chosen to represent IC?
- Answer: Tree can represent hierarchical structure of PL statement appropriately.
- IC is then passed to semantic analysis to determine meaning of the statement.
- Eg: IC



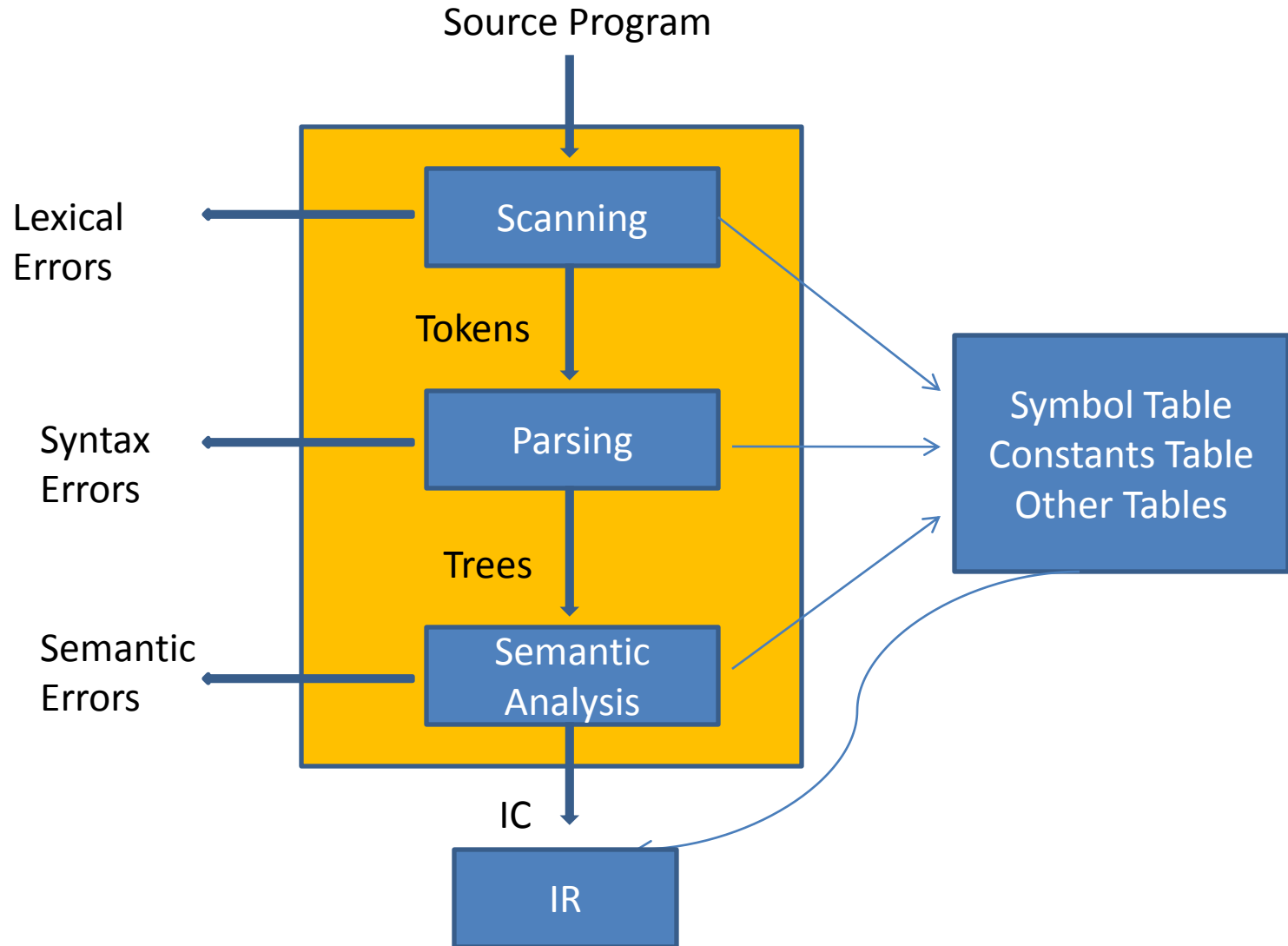
Semantic Analysis of Front End

- When semantic analysis determine the meaning of sub-tree in IC, it adds
 - Declaration (added to symbol table)
 - Imperative Statement Analysis (determines the action sequence).



- Steps:

1. Information concerning the type of operands is added to IC tree. See first tree.
2. Rules of meaning governing assignment statement on right hand side should be evaluated first. Hence focus on right sub-tree.
3. Rules of addition indicate type conversion of i. i.e convert i to real, giving i^* . See 2nd sub tree.
4. Rules of addition indicate feasibility of addition which leads to “add i^* to b, giving temp”.
5. The assignment is performed leads to action “Store temp in a”.



Back End

- Back End
 1. Memory Allocation
 2. Code Generation
- 1. Memory Allocation:
 - A memory allocation requirement of an identifier is computed from its
 - Size,
 - Length,
 - Type,
 - Dimension
 - And then memory is allocated to it.
 - The address of memory area is then entered in symbol table. See next figure.
 - i* & temp are not shown because memory allocation is not required now for this id.
 - i* and temp should be allocated memory but this decision are taken in preparatory steps of code generation.

Sr. No.	Symbol	Type	Length	Address
1	i	Integer	-	2000
2	a	Real	-	2016
3	b	Real	-	2020

- 2. Code Generation:
- Uses knowledge of target architecture.
 - 1. Knowledge of instruction
 - 2. Addressing mode in target computer.
- Important issues effecting code generation:
 - Determine the place where the intermediate results should be kept. i.e in memory or register?
 - Determine which instruction should be used for type conversion operation.
 - Determine which addressing mode should be used for accessing variable.
- Eg: for sequence of actions for assignment statement $a:=b+c$;
 1. Convert I to real, giving i^* ;
 2. Add i^* to b, giving temp;
 3. Store temp in a.

The synthesis phase:

a. decide to hold the values of i^* & temp in machine register.

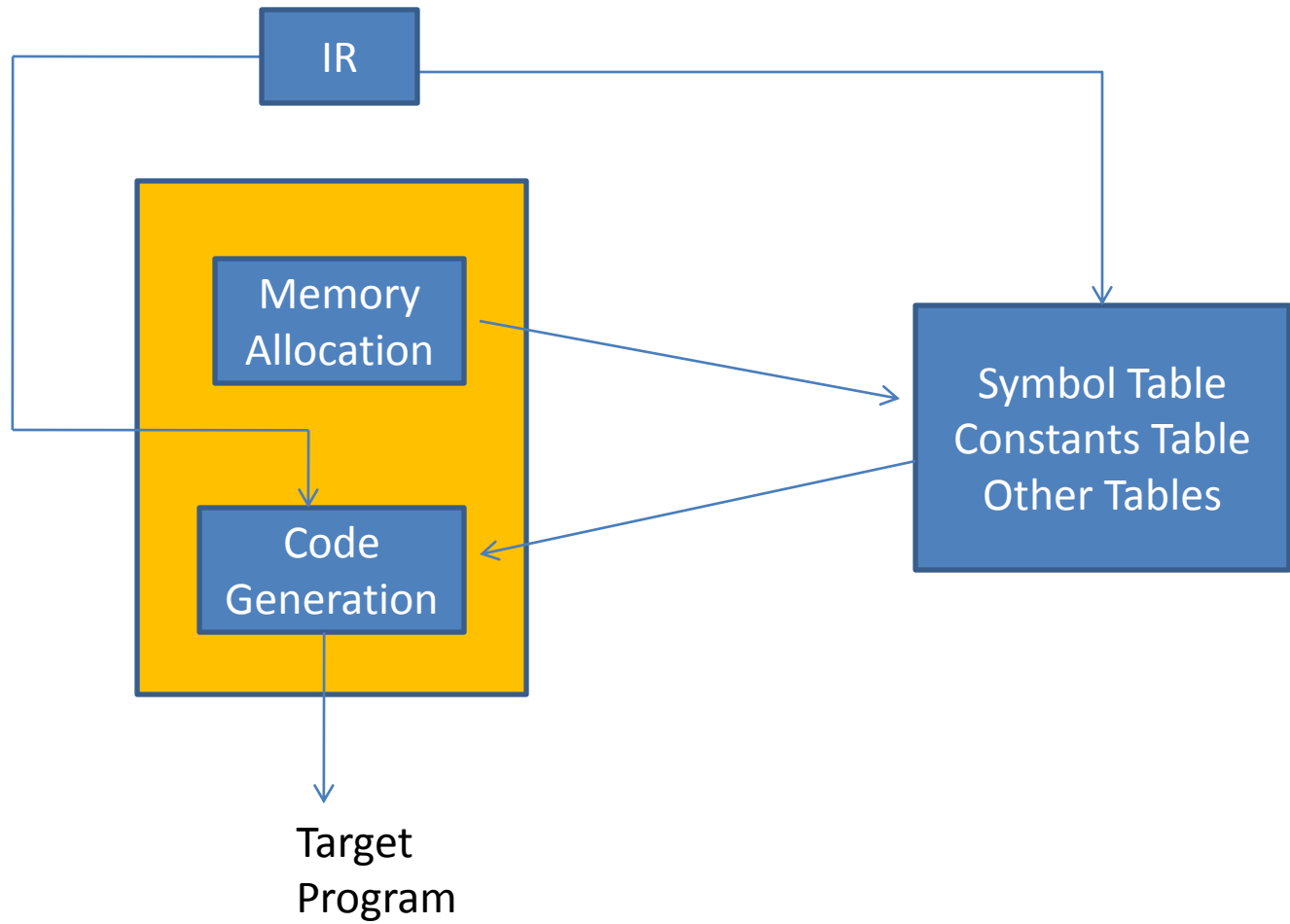
b. generate following assembly code.

```

CONV_R      AREG, I
ADD_R       AREG, B
MOVE_M      AREG, A

```

- See next figure of Back End.




Fundamentals of Language Specification:

- The lexical & synthetic feature of a programming language are specified by its grammar rules.
- Language L can be considered to be collection of valid statements.
- A former language grammar: It is set of rules which specify the sentences of L language.
- Natural languages are not formal languages due to there rich vocabulary.
- Terminal: However, PL are formal language. Now $E = \{0,1,....9,a,b,c.....z\}$
- The alphabet of L; denoted by 'E' is a collection of symbols in its character set.

- Alphabet: It is defined as finite set of symbols. Eg. Roman alphabet (a,b,...z)
- Terminal Symbol: Terminal symbols, or just terminals, are the elementary symbols of language defined by formal grammar.
- Non-Terminal Symbols: A non terminal symbol (NT) is the name of syntax category of a language. E.g noun, verb, etc. It is written in single capital letter. Eg. A or <Noun>. It represents instance of category. Non-terminal symbols, or just non-terminals are the symbols which can be replaced; thus they are composed of some combination of terminal and non-terminal symbols.
- Eg: <integer> ::= ['<->']<digit>{<digit>}
 <digit> ::= 0|1|2|3|.....|9.
- Here, 0 to 9 are terminal symbols and <digit> and <integer> are non-terminal symbols.
- String: A string over an alphabet is a finite sequence of symbols from the alphabet, which is usually written next to one another and not separated by commas.
- Meta symbols: Here, '{' , '}' are part of notations and we call them meta symbols.

- String Operations:

- Length of string: The length of a string is the number of symbols in it. The length of string is its length as a sequence. The length of a string w is written as $|w|$.
- Concatenation: Operation combining two strings into a single string represented by $\alpha\beta$ or $\alpha.\beta$
Eg: $\alpha = ab$ and $\beta = axy$ then, $\alpha.\beta = abaxy$.
If concatenation is done with null string, result is itself. Eg. $a.\epsilon = \epsilon.a = a$.

- Production: A production also called a re-writing rule, is a rule of the grammar having the following form:
A non-terminal symbol ::= string of Ts and NTs.
Eg: <article> ::= a | an | the
 <noun phrase> :: <article><noun> 
- Grammar (G): A grammar G of a language L_G is a quadruple (Σ, SNT, S, P) where,
 - Σ is an alphabet of L_G . i.e Set of Ts.
 - SNT is a set of Non-terminals.
 - S is a distinguished symbol
 - P is a set of productions
- Purpose of Grammar | G | :
 - To generate valid string of L_G .
 - To recognize a valid string of L_G .
- How to fulfill this purpose?
 - Derivation operation helps generate valid string.
 - Reduction operation helps recognize valid string.
 - Parse Tree is used to depict the syntactic structure of valid string as it emerges during the sequence of derivations or reductions.
- Lets see all the three stated above in detail.

- **Derivation:**
- $P: A ::= \alpha \quad \beta = \gamma A \Theta \quad \text{then, } \beta = \gamma \alpha \Theta.$
- Types:
 - Direct Derivation $N \Rightarrow n$
 - Indirect Derivation $N \Rightarrow^* n$
- Eg:
 - <Article> ::= a | an | the
 - <Noun Phrase> ::= boy | apple
 - <Verb> ::= ate
- Then, <Noun Phrase> \Rightarrow <Article><Noun>
 - \Rightarrow The <Noun>
 - \Rightarrow The boy

- Example 1.15:

<sentence> ::= <noun phrase><verb phrase>

<noun phrase> ::= <article><noun>

<verb phrase> ::= <verb><noun phrase>

Then,

<noun phrase><verb phrase>

<noun phrase><verb><noun phrase>

<noun phrase> ate <noun phrase>

The boy ate <noun phrase>

The boy ate an apple.

The last sentence “The boy ate an apple is a sentence.”

- **Reduction:**
- P: $A ::= \alpha \quad \delta = \gamma\alpha\ominus \quad \text{then, } \delta = \gamma A\ominus.$
- Types:
 - Direct Reduction $N \Rightarrow n$
 - Transitive reduction $N \Rightarrow^* n$
- Eg: To determine the validity of a string:

A boy ate an apple

<article> boy ate an apple

<article><noun> ate an apple

<article><noun><verb> an apple

<article><noun><verb><article> apple

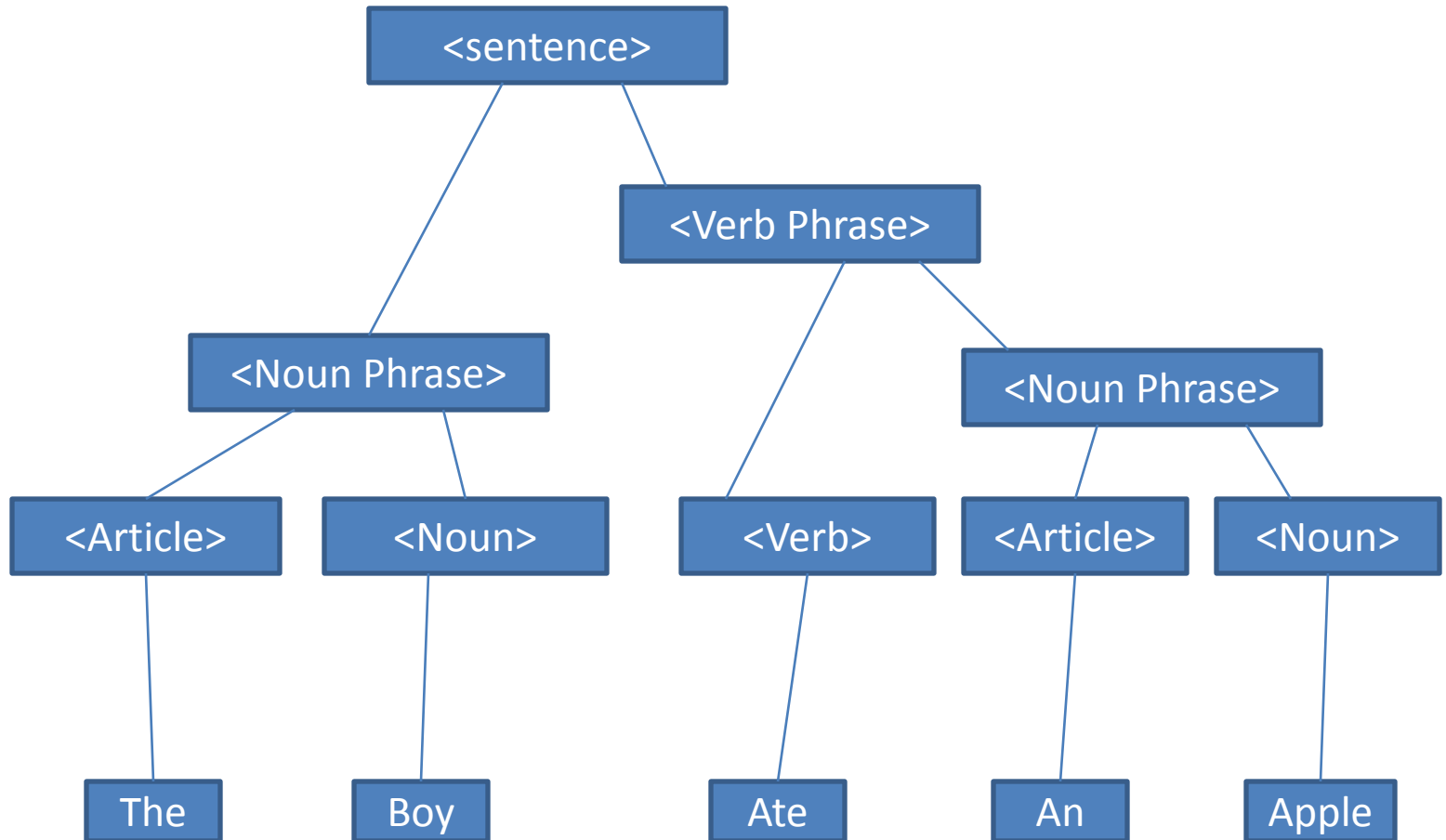
<article><noun><verb><article><noun>

<noun phrase><verb><noun phrase>

<noun phrase><verb phrase>

<sentence>

- **Parse Tree:**



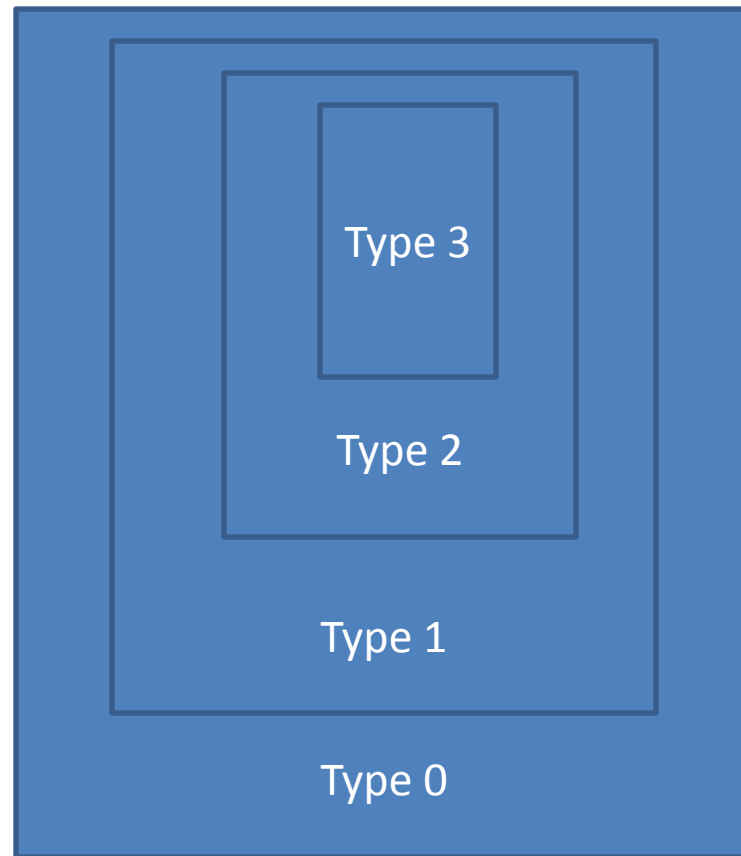
Recursive Specification:

- [BNF:- Backus Noun Form]
- $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$
- $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$
- $\langle \text{factor} \rangle ::= \langle \text{factor} \rangle ^ \langle \text{primary} \rangle \mid \langle \text{Primary} \rangle$
- $\langle \text{primary} \rangle ::= \langle \text{id} \rangle \mid \langle \text{const} \rangle \mid (\langle \text{exp} \rangle)$
- $\langle \text{id} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{id} \rangle \mid [\langle \text{letter} \rangle \mid \langle \text{digit} \rangle]$
- $\langle \text{const} \rangle ::= [+ \mid -] \langle \text{digit} \rangle \mid \langle \text{const} \rangle \langle \text{digit} \rangle$
- $\langle \text{letter} \rangle ::= a \mid b \mid c \dots \mid z$
- $\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \dots \mid 9$

- This noun form uses Rotation BNF (Backus Noun Form).
- It uses Recursive Specification.
- eg:
- ~~(a)~~ $\langle \text{exp} \rangle ::= \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{term} \rangle + \dots + \langle \text{term} \rangle \dots$
- ~~(b)~~ $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$
- 1st Alternative: a's right hand side is recursive because it allows unbounded number of + operation.
- 2nd Alternative: b is non recursive, as it provides escape from recursion by deriving expression.
- Recursive Rules:
 - Left Recursive Rule: where NT appears on the extreme left.
 - Right Recursive Rule: where NT appears on extreme right.
- Indirect Recursion: It occurs when two or more entries are defined in terms of one another. They are useful in specifying nested construct.
 - Eg: $\langle \text{primary} \rangle ::= \langle \text{exp} \rangle$ give rise to indirect recursion because $\langle \text{exp} \rangle \Rightarrow \langle \text{primary} \rangle$
- Direct Recursion: Permits unbounded number of character.
 - $\langle \text{id} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{id} \rangle [\langle \text{letter} \rangle \mid \langle \text{digit} \rangle]$
- But it is not correct. Solution? You need to bound the terms. Controlled occurrences may bound recursion.
 - $\langle \text{id} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \}_0^{15}$
 - indicates 0 to 15 occurrences of enclosed specification.

Classification of Grammer:

- Types:
 - Type 0
 - Type 1
 - Type 2
 - Type 3
 - OG



Type 0

- This grammar is also known as phrase structure grammar.
- Contains production of form $\alpha ::= \beta$, where, both can be strings of terminal & non-terminals.
- So, such productions permit arbitrary substitution of strings during derivation or reduction.
- Hence, they are not relevant to specification of programming language.

Type 1

- Also called Context Sensitive Grammar.
- Their production specify that derivation or reduction of string can take place only in specific content.
- The production form is $\alpha A \beta = \alpha \pi \beta$
- This grammar is also particularly not relevant for PL specification.

Type 2

- This grammar impose no context requirement on derivation or reduction.
- Also called context free grammar.
- The production form is $A ::= \pi$
- Can be applied independent of its context.
- This grammars are therefore known as context free grammar.
- Ideally suited for programming language specification.
- Eg: Pascal specification
ALGOL 60 Specification.

Type 3:

- Also called linear or regular grammars.
- Has the Production form $A ::= tB \mid t$ or $A ::= Bt \mid t$.
- Satisfies the requirement of type 2 grammar.
- Advantages in scanning.
- Restricts expression power of this grammar.
- Excluding:
 - 1. Nesting of constructs
 - 2. Matching of Parenthesis
- Hence, its use is restricted to specification of lexical units.
- Eg: $\langle id \rangle ::= l \mid \langle id \rangle l \mid \langle id \rangle d$
where, l for letter and d for digit.

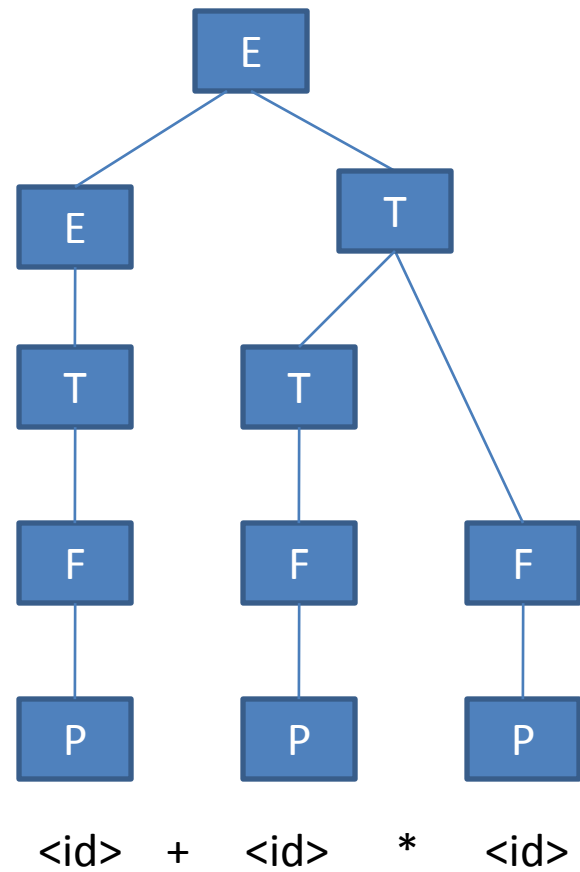
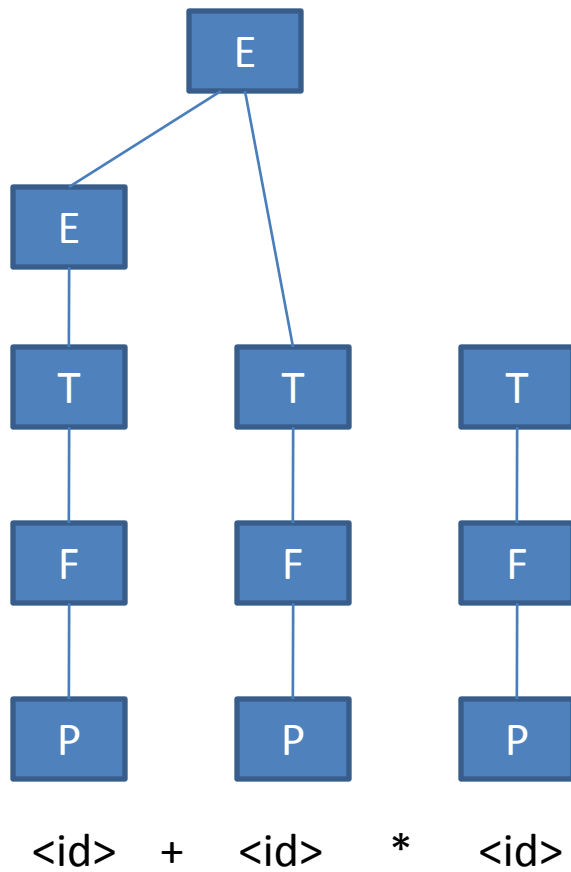
Operator Grammar:

- An operator grammar is a grammar none of whose productions contain two or more consecutive NTs in any RHS alternative.
- They are separated by terminals called operators in grammar.
- Eg: $a = b + i$

Ambiguity in Grammar Specification:

- Ambiguity implies the possibility of different interpretations of a source stream.
- A sentence may have multiple syntactic structure.
- Ambiguity is the problem for language specification.
- Solution?
 - Avoid Ambiguity
 - Eliminate Ambiguity
- How to Avoid Ambiguity?
 - We define a rule: Identical string cannot appear on RHS of more than one production in grammar.
 - At what level of analysis ambiguity is tested?
 - At lexical we can avoid ambiguity.
 - At syntax level we are suppose eliminate ambiguity.

- How to eliminate ambiguity in lang. grammar.
 - An ambiguous grammar should be re-written to eliminate ambiguity. $a+b*c$
 - Rewrite grammar such that $*$ is reduced before $+$. To achieve this, what can be done?
 - Solution? Use hierarchical structure of NTs in grammar.
 - Eg: at syntax level eliminate ambiguity
 - $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{tem} \rangle \mid \langle \text{term} \rangle$
 - $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$
 - $\langle \text{factor} \rangle ::= \langle \text{factor} \rangle ^ \langle \text{primary} \rangle \mid \langle \text{primary} \rangle$
 - $\langle \text{primary} \rangle ::= \langle \text{id} \rangle \mid \langle \text{constant} \rangle \mid (\langle \text{exp} \rangle)$
 - Parse tree drawn for above eg:



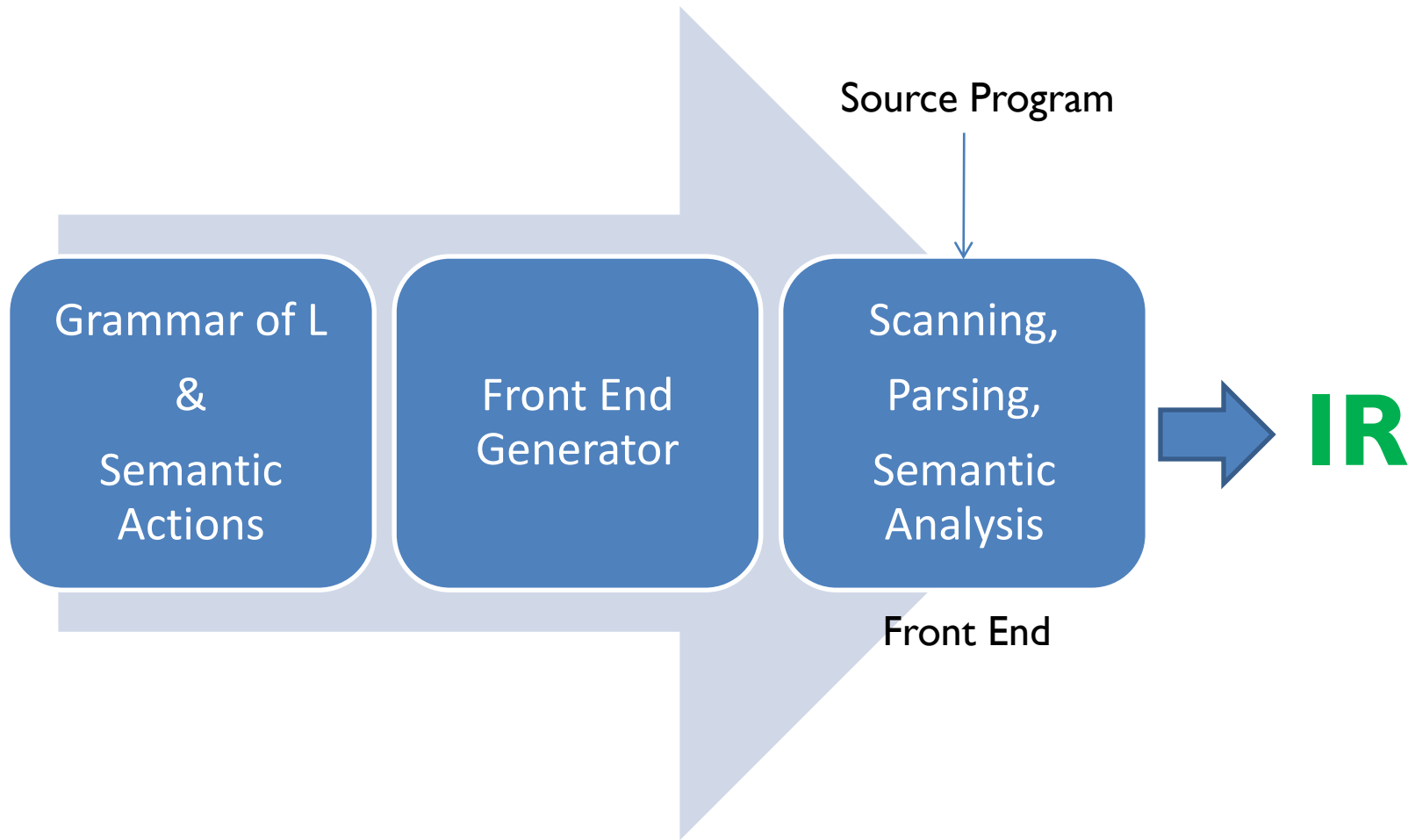
Binding:

- Each program entity = Pe_i in a program has attributes set $A_i = \{a_j\}$ associated with it.
- A set of attributes determine whether it is a variable, procedure or reserved identifier. i.e keyword.
- The value of attributes of entity should be determined and bind them before Lang. processor processes a declaration statement.
- Def: Binding: is the association of an attribute a program entity with a value.
- Def: Binding Time: Binding time is the time at which binding is performed.
- It determines efficiency of LP.
- Early Binding provides greater execution efficiency.
- Late Binding provides greater flexibility in writing the program.
- Def: Static Binding: A static binding is a binding performed before the execution of a program begins.
- Def: Dynamic Binding: A dynamic binding is a binding performed after the execution of program has begin.
- Static binding leads to more efficient execution of a program then dynamic binding.

Language Processor Development Tools

- Writing of language processors is a well understood and repetitive process which ideally suits the program generation approach to software development.
- Set of language processor development tools (LPDTs) focusing on generation of the analysis phase of language processors.
- LPDT requires the following two inputs:
 - Specification of grammar of language L.
 - Specification of semantic actions to be performed in the analysis phase.

LPDT



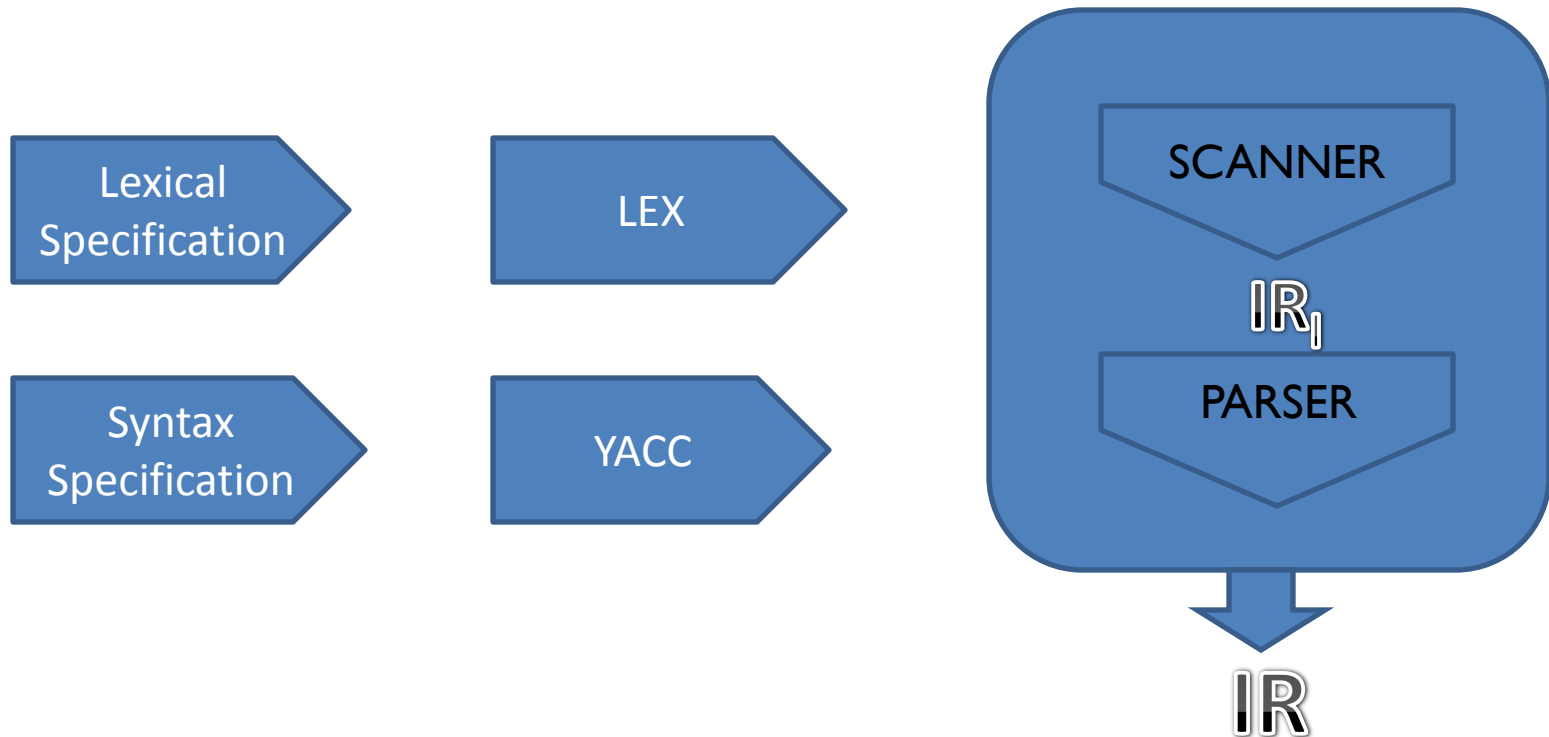
LPDT

- It generates programs that perform lexical, syntax and semantic analysis of the source program and construct the IR.
- These programs collectively form the analysis phase of the language processor.
- Lexical analyzer generator LEX, and the parser generator YACC.
- Input to those tools is a specification of the lexical and syntactic constructs of L, and the semantic actions to be performed on recognizing the constructs.

LPDT

- The Specification consists of a set of “Translation Rules” of the form
 < string specification > { < semantic action > }
 where,
 - < semantic action > consist of C code.
 - The code is executed when a string matching
 < string specification > is encountered in the input.
- LEX and YACC generate C programs which contain the code for scanning and parsing, respectively and the semantic actions contained in the specification.
- A YACC generated parser can use a LEX generated scanner as a routine if the scanner and parser use same conventions concerning the representation of tokens.
- A single pass compiler can be built using LEX and YACC if the semantic actions are aimed at generating target code instead of IR.
- Note that scanner also generates an intermediate representation of a source program for use by the parser, we call it IR_1 to differentiate it from the IR of the analysis phase.

Using LEX and YACC



LEX

- LEX accepts an input specification which consists of two components.
 - 1st is Specification of String i.e. in regular expression form. e.g id's and constants.
 - 2nd is Specification of Semantic Actions aimed at building an IR.
- Lets see the example to show sample input to LEX.
- It contains 3 components.
- 1st component is enclosed by %{ and %} which defines the symbols used in specifying the strings of L.
- 2nd component is enclosed between %% and %% which contains translation rules.
- 3rd component contains auxiliary routines which can be used in semantic actions.

LEX conti....

```
%{  
letter          [A-Za-z]  
Digit           [0-9]  
}%  
  
%%  
begin           {return(BEGIN);}  
end             {return(END);}  
“:=“           {return(ASGOP);}  
{letter} ({letter}|{digit})* {yyval=enter_id(); return(ID);}  
{digit}+       {yyval=enter_num(); return(NUM);}  
%%
```

enter_id()

{ /* enters the id in the symbol table and returns entry number */ }

enter_num()

{ /* enters the number in the constants table and returns entry number */ }

What does the code do?

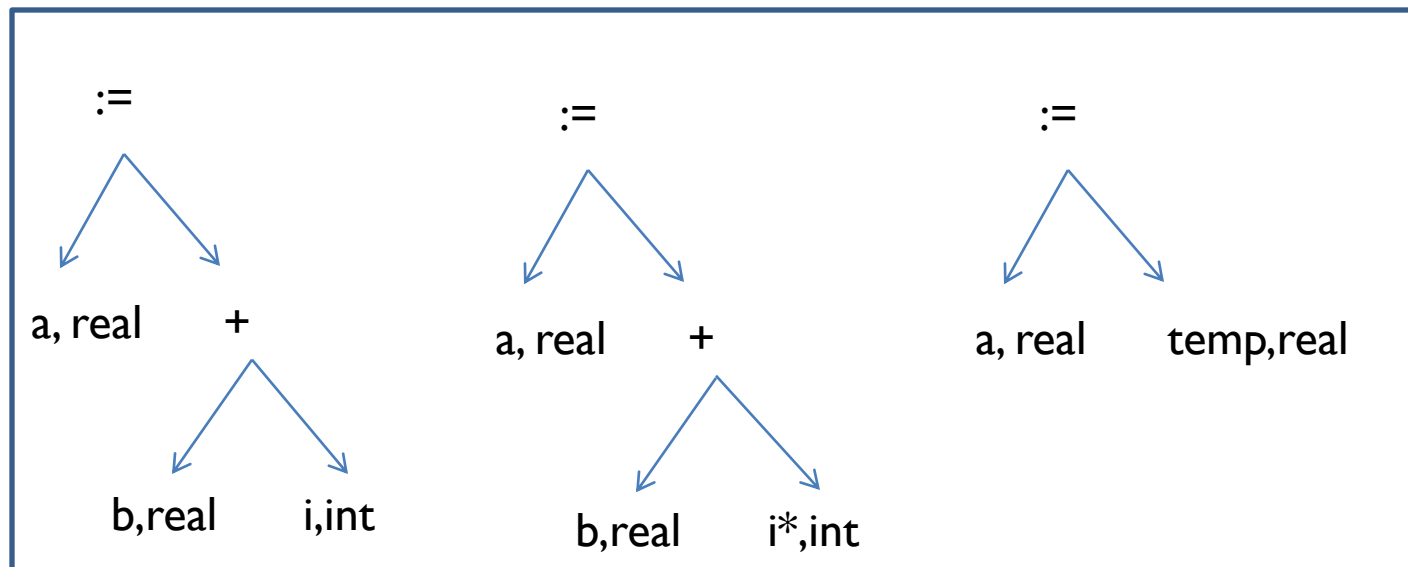
- The sample input defines the strings begin, end, := (the assignment operator), and identifier and constant strings of L.
- When an identifier is found, it is entered in the symbol table (if not already present) using routine enter_id.
- The pair (ID,entry#) forms the token for the identifier string.
- By convention entry# is put in the global variable yylval, and class code ID is returned as the value of the call on scanner.
- Similar actions are taken on finding a constant, the keywords 'begin' and 'end' and the assignment operator.

YACC

- Each string specification in the input to YACC resembles a grammar production.
- The actions associated with a string specification are executed when a reduction is made according to the specification.
- An attribute is associated with every non terminal symbol.
- The value of this attribute can be manipulated during parsing.
- The attribute can be given any user-designed structure.
- A symbol '\$n' in the action part of a translation rule refers to the attribute of the n'th symbol in the RHS of string specification.
- '\$\$' represents the attribute of the LHS symbol of the string specification.

YACC conti....

- Let us see the sample input to YACC.
- The input consists of 2 components.
- It is assumed that the attribute of a symbol resembles the attributes used in following fig.



YACC conti....

- The routine 'gendesc' builds a descriptor containing the name and type of an id or constant.
- The routine 'gencode' takes an operator and the attribute of two operands, generates code and returns with the attribute for the result of the operation.
- This attribute is assigned as the attribute of the LHS symbol of the string specification.
- In the subsequent reduction this attribute becomes the attribute of some RHS symbol.

YACC conti....

```
%%
```

```
E : E+T {$$ = gencode('+',$1,$3);}
```

```
  | T   {$$ = $1;}
```

```
;
```

```
T : T*V {$$ = gencode('*',$1,$3);}
```

```
  | V   {$$ = $1;}
```

```
;
```

```
V : id   {$$ = gendesc($1);}
```

```
;
```

```
%%
```

```
gencode(operator, operand_1,operand_2)
```

```
{ /* Generates code using operand descriptors.
```

```
   Returns descriptor for result. */ }
```

```
gendesc(symbol)
```

```
{ /* Refer to symbol/constant table entry.
```

```
   Build and return descriptor for the symbol. */ }
```

YACC conti....

- Parsing of the string $b+c*d$ where b , c and d are of type real, using the parser generated by YACC from the input of previous slide leads to following calls on C routine.

Gendesc(id#1);

Gendesc(id#2);

Gendesc(id#3);

Gencode(*, [c,real] , [d,real]);

Gencode(+, [b,real] , [t,real]);

- Where, an attribute has the form $\langle \text{name} \rangle, \langle \text{type} \rangle$ and t is the name of a location (a register or memory word) used to store the result of $c*d$ in the code generated by the first call on 'gencode'.

Our Chapter Ends Here
Soon You'll Get
Assignment Questions.
Soon your Quiz will be conducted.

THANK YOU.