Basic Language Translator

2020-2021

الدراسات الاولية/ البكلوريوس / الفصل الثاني

أستاذ المادة

ا.م. وجدان عبد الامير حسن

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator
2nd Class/2nd Sem
Lecture 1

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

Basic Language Translator

FIRST LECTURE

Programming Languages

In computer programming ,programming language serves as means of communication between the person with a problem and the computer used to solve it.

Programming language is a set of symbols ,words, and rules used to instruct the computer.

Programming Languages

A hierarchy of programming languages based on increasing machine independence include the following:

1- Machine Language

2- Assembly Language

3- High-Level Language

4- Problem-Oriented Language

Programming Languages

1- Machine Language: is the actual language in which the computer carries out the instructions of program. otherwise ,it is the lowest form of computer language, each instruction in program is represented by numeric code, and numeric of addresses are used throughout the program to refer to memory location in the computer memory.

Programming Languages

2- Assembly Languages: is a symbolic version of a machine language, each operation code is given a symbolic code such **ADD,SUB,…**

3- High-Level Language(HLL): is a programming language where programming not require knowledge of the actual computing machine to write a program in the language. HLL offer a more enriched set of language features such as control structures, nested statements, block,…ect.

Programming Languages

4-**Problem-Oriented Language: it** provides for the expression of problems in a specific application. Examples of such language are SQL for database application.

Programming Language

Advantage of HLL over LLL

1- HLL are easier to learn then LLL.

2- A programmer is not required to know how to convert data from external form to internal within memory.

3- Most HLL offer a programmer a variety of control structures which are not available in LLL.

4- Programs written in HLL are usually more easily debugged than LLL equivalents.

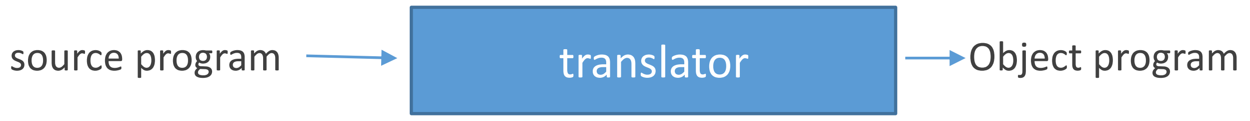
5- Most HLL offer more powerful data structure than LLL.

6- HLL are relatively machine-independent .

Translator

High-level language programs must be translated automatically to equivalent machine language program. A translator converts a source program into an object or target program.

The source program is written in a source language and the object program belong to an object language.



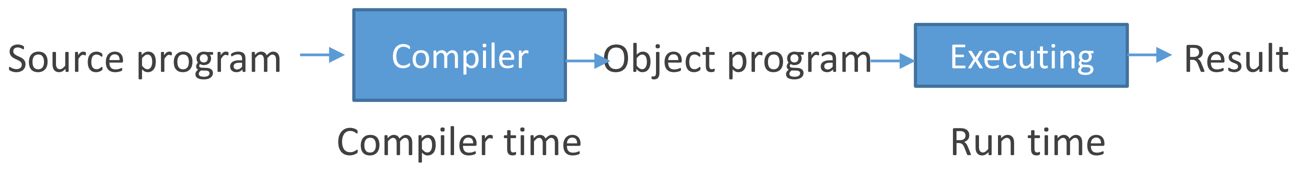
Translator

1- If the source program is written in assembly language and the target program in machine language the translator is called **Assembler.**

2- If the source program is written in HLL language and the object language is LLL then the translator is called **Compiler**.

3-If the source program is written in LLL language and the object language is HLL then the translator is called **Decompiler.**

Compilation Process

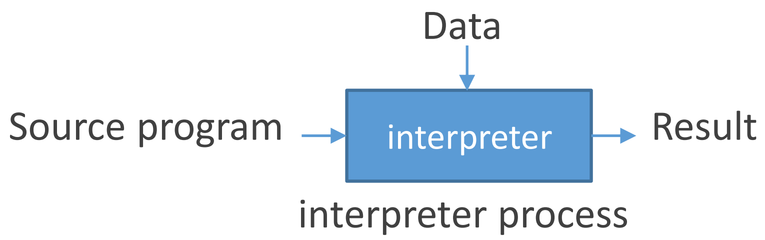


The time at which conversion of a source program to an object program occurs is called Compile time ,the object program is executed at Run time .

<u>Note</u> that the source program and data are process at different time.

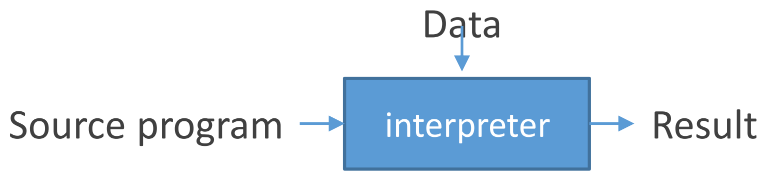
Interpreter

Another kind of translator called an **interpreter** in which processes an internal form of source program and data at the same time. That is interpretation of the internal source form occurs at run time and no object program is generated.



Interpreter

Another kind of translator called an interpreter in which processes an internal form of source program and data at the same time. That is interpretation of the internal source form occurs at run time and no object program is generated.



interpreter process

Comparison between compiler and interpreter

Compiler program usually run faster than interpreter ones because the overhead of understanding and translating has already been done.

However ,Interpreters are frequent easier to write than Compilers, and can more easily support interactive debugging of program.

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator

2nd Class/2nd Sem

Lecture 2

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

Basic Language Translator

Second Lecture

Introduction

- **A compiler** is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language). It also reports to its user the presence of errors in the source program.

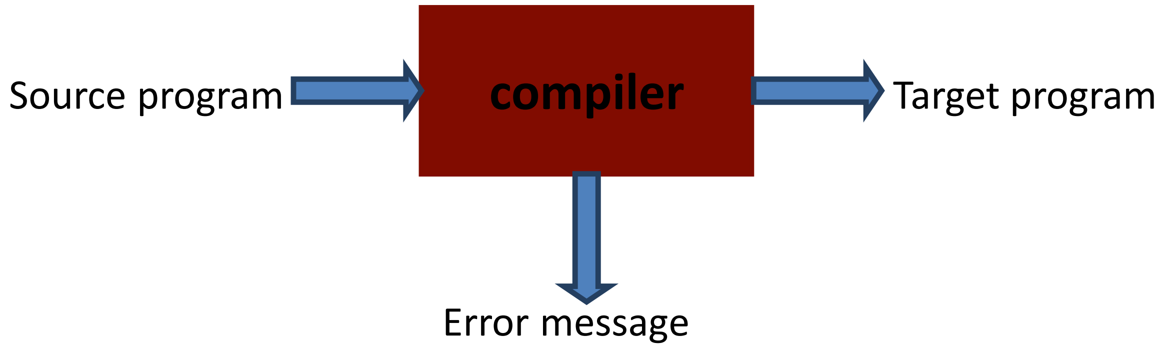


Fig 1: A compiler

Introduction

- Compilers are sometimes classified as single-pass, multi-pass, load and go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform.
- There are two parts to compilation: analysis (including lexical, syntax, and semantic phases) part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation.

1.2 phases of Compiler

A compiler operates in phases, each of which transform the source program from one representation to another. These phases are:

1- **Lexical Analyzer (Scanner)**

It reads the stream of characters (making up the source program)from left to right and group them into tokens (are sequence of characters having a collective meaning).

2- **Syntax Analyzer (parser)**

It grouped characters or tokens into nested collections with collective meaning.

3- **Semantic Analyzer**

It performs certain checks to ensure that the components of a program fit together meaningfully.

4- **Intermediate Code Generation**

It generates an explicit intermediate representation of the source code. This representation should have two important properties: it should be easy to product, and easy to translate into the target program.

5- **Code Optimizer**

It attempts to improve the intermediate code , so that faster running machine code will result .

6- **Code Generator**

It translates the intermediate code and generate the target code (normally relocatable machine code or assembly code).

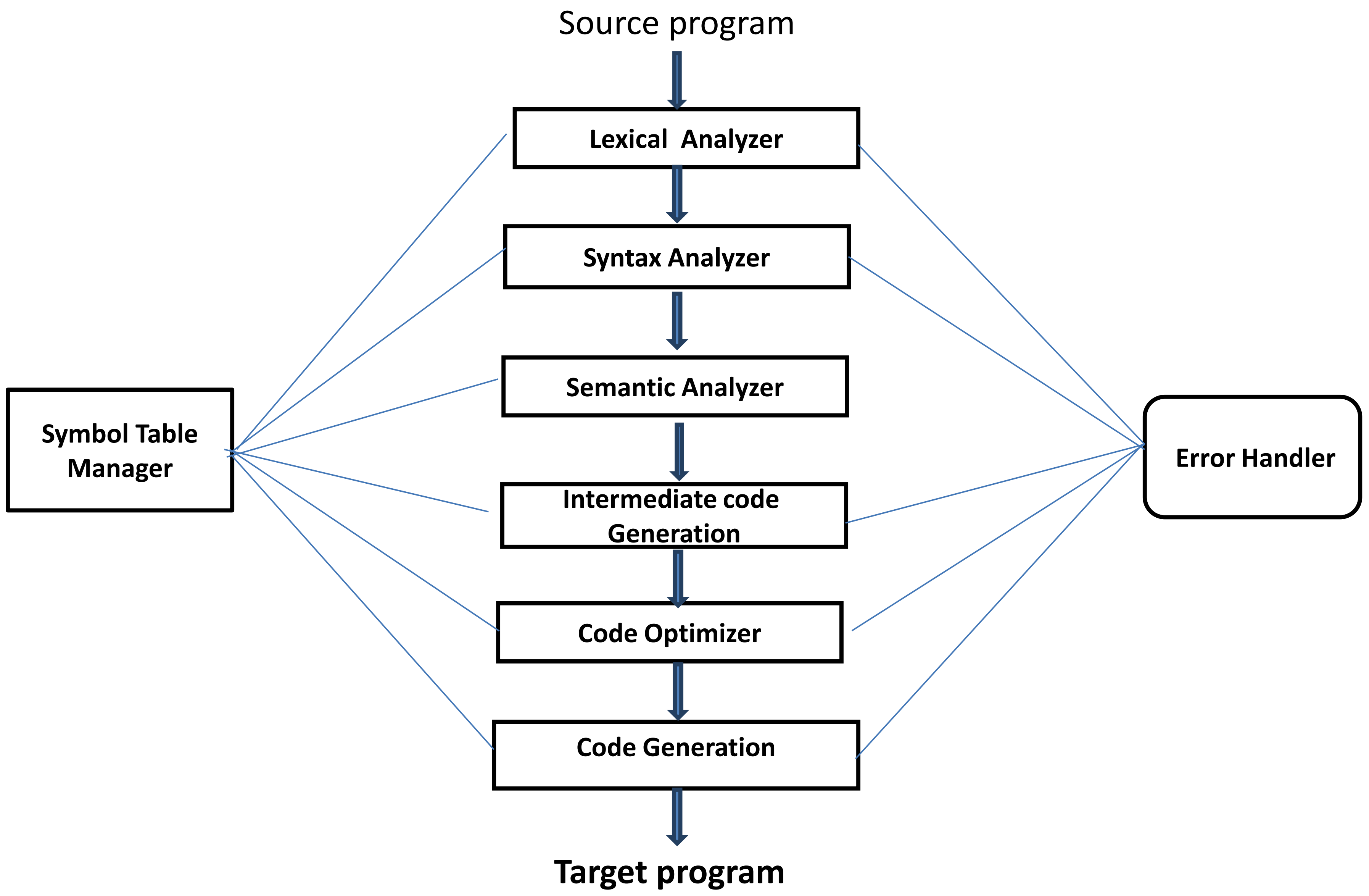


Fig 1.2 phases of compiler

Symbol Tables Management:-

- A symbol table is a data structure containing record for each identifier with field for the attributes of the identifier. These attributes may provide information about the storage allocated for an identifier, its types, and its scope (where in the program it is valid), number and type of the procedure arguments, the method of passing each argument, and the type returned by the procedure (if any).
- When an identifier in the source program is detected by the lexical analyzer, the identifier is enter into the symbol table and then use this information in various ways.

1.3 Symbol Tables Management

- A symbol table is a data structure containing record for each identifier with field for the attributes of the identifier. These attributes may provide information about the storage allocated for an identifier, its types, and its scope (where in the program it is valid), number and type of the procedure arguments, the method of passing each argument, and the type returned by the procedure (if any).
- When an identifier in the source program is detected by the lexical analyzer, the identifier is enter into the symbol table and then use this information in various ways.

1.4 Error Detection And Reporting.

- Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error so that compilation can precede allowing further errors in the source program to be detected.
- The syntax and semantic analysis phases usually handle a large function of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input don’t form any token of the language. Errors where the token stream violates the structure rules of the language are determined by the syntax phase. Semantic analysis phase tried to detect constructs that have the right syntactic structure but no meaning to the operation involved (for example, trying to add two identifiers, one of which is the name of an array, and the other the name of a procedure)

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

VIEDEO LECTURES

Basic Language Translator

2nd Class/2nd Sem

Lecture 3

ا.م وجدان عبد الامير حسن

COMPILER

Lexical analyzer

Lexical Analysis(scanner)

Introduction

The main task of lexical analysis is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

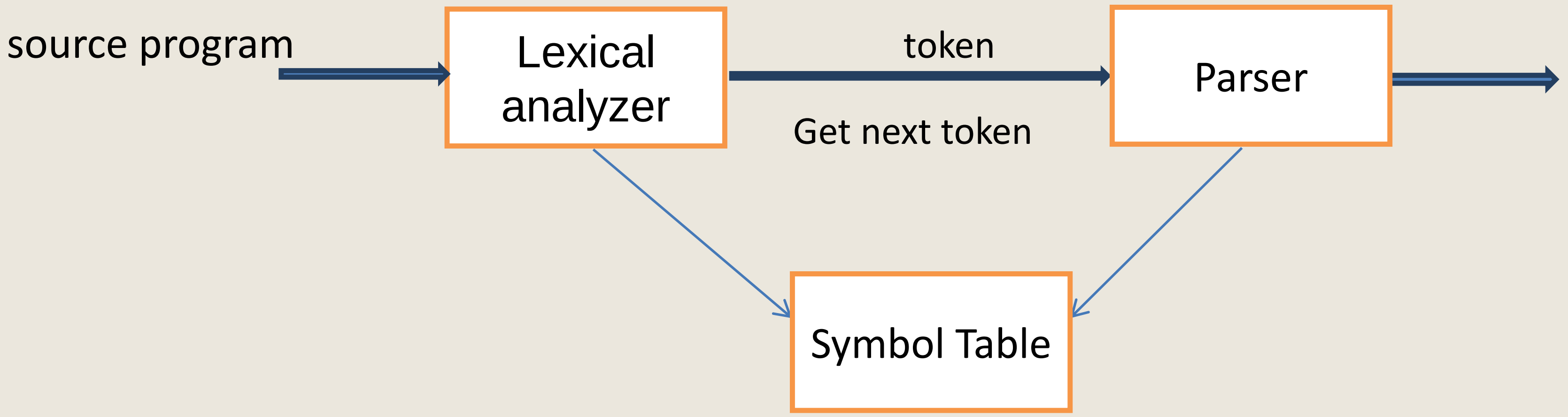


Fig 2.1 Interaction of lexical analyzer with parser.

Lexical Analysis(scanner)

In order to understand and implement an efficient lexical analyzer, three terms must be dealt with. These are

Token: a set of constructs forming the source programming language like keywords, operators , identifiers , …etc.

Lexeme: A sequence of characters in the source program that is matched by the rule of token.

Rule: describing the set of lexemes that can represent particular token.

Token	Sample lexemes	Informal Description of Pattern
Const	const	Const
If	if	if
Relation	<,<=,=,<>,>,>=	<or<=or<>or>=or>
Identifier	Pi ,count ,D2	Letter followed by letters and digits
number	3.12 ,8 ,6.02E31	Any number constant

Lexical Analysis(scanner)

2.2 Recognition of token

To design an elementary lexical analyzer(scanner) for a particular language, we must establish the following:

1- Knowing what language constructs (tokens) are, classified them in to groups such as identifier, keywords, operators…etc.

2- Describe tokens in terms of rules, either in terms of patterns(informal) or terms of regular expressions (formal).

3- Build devices that are well suited to this recognition task , which Deterministic Finite Acceptors (DFA).

Lexical Analysis(scanner)

Example:

Build a scanner that can recognize the following group of tokens:

1- identifier

2- constant number

Solution

Informal rules

Identifier: letter followed by letters or digits.

Constant numbers: any numeric constant.

Lexical Analysis(scanner)

Regular expressions (formal)

id $\longrightarrow$ letter (letter|digit)*

Letter $\longrightarrow$ A|B|C|........|a|b|c|........z

digit $\longrightarrow$ 0|1|2|3........|9

num $\longrightarrow$ digits Optional-fraction Optional –exponent

digits $\longrightarrow$ digit$^+$

Optional-fraction $\longrightarrow$ .digits |∈

Optional –exponent $\longrightarrow$ E(+|-| ∈)digits| ∈

Lexical Error

2.3 Lexical Errors

The lexical analyzer may correlate error message from the compiler with the source program.

Some possible error recovery actions are:

1- Detecting an extraneous character.

2- Inserting a missing character.

3- Replacing an incorrect character by a correct one.

4- Transposing two adjacent characters.

Symbol table

2.4 Symbol table

A simple technique for separating Keywords from identifiers is initialize the symbol table in which information about identifiers is saved.

A simple table is the data structure use to store information about various identifiers. The information is collected by the analysis phases of compiler and then used for checking the semantics correctness and aiding in the proper generation of code. The stored information (know as attributes) is as follows:

1) Variable name

It is the means by which a particular variable is identified for semantic analysis and code generation. This attributes should be inserted into the symbol table during lexical analysis.

2) Object code address

It denote the relative location for value (s) of a variable. This address is entered into the symbol table when a variable is declared and recalled from the table when the variable is referenced in the source program.

Symbol table

3) Type

It represents the type of variable and used for:

-Determining how many spaces in the memory is needed to store the variable.

-checking the semantic error in arithmetic equation.

4) Number of Dimensions and Number of parameters

- scalar dim 0
- vector dim 1
- metrics dim 2
- function according to number of parameters.

5) Line Declare

The source line number at which a variable is declared.

6) Lines Referenced

The source lines numbers of all other references to the variable.

Example

Draw a diagram of a cross reference table that would result when compiling the following program segment (report error and warning messages if any).

```
Main()

{

Int  i, j[5];

Char c, index[5][6], block[5];

float  f;

i=0;

i= i+ k;

f=f+i;

C=’x’;

block[4]=c;

}
```

Name	Object Address	Type	Dimension	Line Declared	Line Referenced
i	0	int	0	3	6,7,8
j	2	int	1	3	
c	12	char	0	4	9,10
Index	13	char	2	4	
block	43	char	1	4	10
f	48	float	0	5	8
k					7

error k is un defined
warning j is not used
warning index is not used

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator
2nd Class/2nd Sem
Lecture 4

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

SYNTAX ANALYZER

Part 1

Syntax Analyzer

3.1 introduction

In compiler model, the parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

Methods used in compilers are classified as being either **top-down** or **bottom-up.** Top-down parser builds parse tree from the top to the down (leaves) while bottom-up parser starts from the leaves and work up to the root.

Syntax Analyzer

There are number of tasks that might be conducted during parsing:

1- Collecting information about various tokens from the symbol table

2- Performing type checking and other kinds of semantics analysis.

3- Generating intermediate code.

Syntax Analyzer

3.2 Syntax Error and Handling

Program can contain errors at many different levels. For example, errors can be

Lexical such as misspelling an identifier, keyword, or operator.

Syntactic such as an arithmetic expression with unbalanced parentheses.

Semantic such as an operator applied to an incompatible operand.

Logical such as infinitely recursive call.

The error handler in a parser has the following goals:

1- It should report the presence of errors clearly and accurately.

2- It should recover from each error quickly enough to be able to detect subsequent error.

3-It should not significantly slow down the processing of correct program.

Syntax Analyzer

How should an error handler report the presence of an error?

A common strategy is to print the offending line with a pointer to the position at which an error is detected. In some compilers, an informative, understandable message is also included.

Once an error is detected, how should the parser recover?

with a reasonable hope that correct input will be parsed. Usually there is some form of error recovery in which the parser attempts to restore itself to a state where processing of the input can continue.

Syntax Analyzer

Many strategies employ recovery:

1- Panic-Mod

On discovering an error, the parser discards input symbols one at a time until one of a designated set of tokens is found.

It has the advantage of simplicity and is guaranteed not to go into an infinite loop. It skips a considerable amount of input without checking it for additional errors.

2- Phrase-level

One discovering an error, the parser may perform local correction on the remaining input.

That is, it may replace a prefix of the remaining input by some string that allows the parser to continue.

A typical local correction would be to replace a comma by a semicolon, detecting an extraneous semicolon, or insert a missing semicolon.

Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Syntax Analyzer

3- Error productions

If we have a good idea of the common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. We then use the grammar augmented by these error productions to construct a parser. If an error production is used by the parser, we can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input.

4- Global Correction

Design algorithms to choose a minimal sequence of changes to obtain a globally least-cost correction. These methods are in general too costly to implement in terms of time and space.

Syntax Analyzer

3.3 Context-Free Grammar

Context Free Grammar consists of terminals, non-terminals, a start symbol, and productions. Terminals are the basic symbols from which strings are formed. Non-terminals are syntactic variables that denote sets of strings.

One non- terminal is distinguished as the start symbol. The productions of a grammar specify the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminals followed by an arrow followed by a string of non-terminals and terminals.

Syntax Analyzer

Example:-

Consider the following context – free grammar:

expr $\longrightarrow$ expr op expr |(expr)|-expr |id

op $\longrightarrow$ +|- |*|/| ↑

Non-terminal: expr,op

Terminal:+,-,/,(,),id, ↑

Star:expr

We can rewrite the above grammar as:

E $\longrightarrow$ EAE|(E)| -E| id

A $\longrightarrow$ +|-|*|/|↑

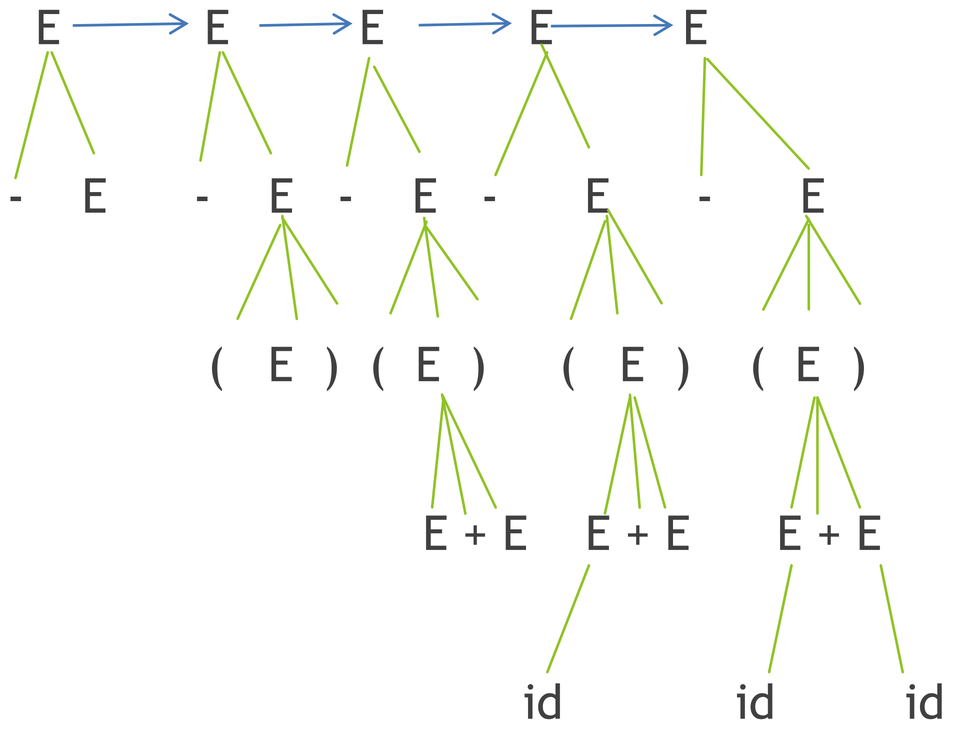
Syntax Analyzer

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order.

Example:-

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

The string –(id+id)



Example:-

$E \longrightarrow E+E \mid E-E \mid E/E \mid E*E \mid (E) \mid -E \mid id$

The string -(id+id)

Leftmost derivation:

$E \rightarrow -\underline{E} \rightarrow -(\underline{E}) \rightarrow -(\underline{E}+E) \rightarrow -(id+\underline{E}) \rightarrow -(id+id)$

Rightmost derivation

$E \rightarrow -\underline{E} \rightarrow -(\underline{E}) \rightarrow -(E+\underline{E}) \rightarrow -(\underline{E}+id) \rightarrow -(id+id)$

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator

2nd Class/2nd Sem

Lecture 5

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

SYNTAX ANALYZER

Part 2

3.4 Ambiguity , Left Recursion , and Left Factoring

An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

Example:

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

The String **id+id*id** has the following two **leftmost derivation**:

$E \rightarrow \underline{E}+E \rightarrow id+\underline{E} \rightarrow id+\underline{E}*E \rightarrow id+id*\underline{E} \rightarrow id+id*id$

$E \rightarrow \underline{E}*E \rightarrow \underline{E}+E*E \rightarrow id+\underline{E}*E \rightarrow id+id*\underline{E} \rightarrow id+id*id$

Ambiguous grammar may be detected if it has the following forms:

N $\rightarrow$ **NaN** where **a** is a string of terminals and non- terminals.

3.4 Ambiguity , Left Recursion , and Left Factoring

However, a simple set of sufficient conditions can be developed such that when they are applied to a grammar , then the grammar is guaranteed to be un ambiguous.

- Operator precedence.

- Associatively of operators.

Example:

$E \rightarrow E+E|E-E|E*E|E/E|E\uparrow E|(E)|-E\ |id$ ambiguous

By applying the above conditions, we obtain:

$E \rightarrow E+T|E-T|T$

$T \rightarrow T*P|T/P|P$

$P \rightarrow F \uparrow P|F$

$F \rightarrow -E|(E)|id$

+,-,*,/ are left associative

$\uparrow$ is aright associative

3.4 Ambiguity , Left Recursion , and Left Factoring

A grammar is said to be **left recursive** if it has a non-terminal A such that there is a derivation $A \longrightarrow Aa$ for some string a. Form this we can notice that there is two types of left recursion , either **immediate** or **non-immediate**.

Example:

Consider the following grammars:

$E \longrightarrow E+T|T$ (immediate left recursion)

$S \longrightarrow Aa|b$ (non-immediate left recursion)

$A \longrightarrow Ac|Sd|\epsilon$ (since $S \longrightarrow Aa \longrightarrow Sda$)

3.4 Ambiguity , Left Recursion , and Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. if A → a$\beta 1$|a$\beta 2$ are two A productions and the input begins with a string derived from a, we don't know whether to expand A or a$\beta 1$ or a$\beta 2$, we have to left factored the original productions to be: A→aA' and A'→$\beta 1$|$\beta 2$

Algorithm: Left factoring

For each non-terminal A, find the longest prefix a common to two or more of its alternatives.

If a ≠∈, replace all the productions:

A→a$\beta 1$ |a$\beta 2$|.........|aβn|y

By

A→ aA'|y

A'→$\beta 1$|$\beta 2$ |.....|βn

3.4 Ambiguity , Left Recursion , and Left Factoring

Example :-

Consider the following grammar:

$S \rightarrow iEtSeS \mid iEtS \mid a$

$E \rightarrow b$

Left factored this grammar becomes:

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

Note:

The three transformations must be done in order.

3.5 top-down parsing

The top-down construction of a parse tree is done by starting with the root and repeatedly performing the following two steps:

1- At node n, select one of the productions for A and construct children of n for the symbols on the right- hand side of the production.

Find the next node at which a sub tree is to be constructed.

We will consider two types of top-down parsing

1-Recursive-Descent Parsing

A special case of it called predictive parsing.

2-Non-recursive Predictive Parsing

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each non-terminal of a grammar. Recursive-descent parsing may involve backtracking (making repeated scans of the input).

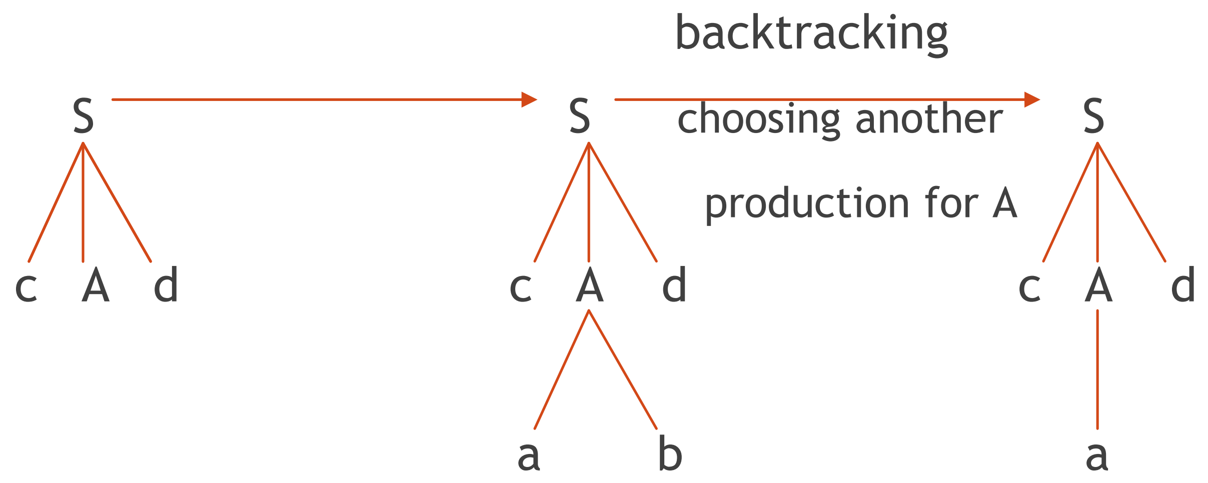
Example:

Consider the following grammar:

S→ cAd

A→ ab|a

The string w=cad



3.6 Bottom-Up Parsing

Shift-reduce parsing:

1- Operator precedence (easy form)

2- LR parsing (general form)

A general style of bottom up syntax analysis is known as shift reduce-parsing. Shift-reduce parsing Attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).At each reduction step a particular substring matching the right side of a production is placed by the symbol on the left of the production and if the substring is chosen correctly at each step, a right most derivation is traced out in reverse.

Example:

Consider the following grammar :

$S \rightarrow aABe$

$A \rightarrow Abc | b$

$B \rightarrow d$

The string **abbcde**

$a\underline{\mathbf{b}}bcde \rightarrow a\underline{\mathbf{Abc}}de \rightarrow aA\underline{\mathbf{d}}e \rightarrow \underline{\mathbf{aABe}} \rightarrow S$

$S \rightarrow aA\underline{B}e \rightarrow a\underline{A}de \rightarrow a\underline{A}bcde \rightarrow abbcde$

A handle of a string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of rightmost derivation.

Example:

Consider the following grammar:

E→E+E|E*E|(E)|id

The right most derivation for the string id+id*id is:

E→ E+**E**→ E+E***E**→ E+**E***id→ **E**+id*id→ id+id*id

The underlined symbols are the handles.

A convenient way to implement a **shift-reduce parsing** is to use a stack to hold grammar symbols an input buffer to hold the string w to be parsed. We use $ to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty and the string w is one the input as follow:

Stack	input
$	w$

The parser operators by shifting zero or more input symbols onto the stack until a handle β is on the top of the stack. The parser reduce β to the left side of the appropriate production. The parser repeat this cycle until it has detected an error or until the stack contains the start symbol and the input is empty as follow:

Stack	input
$ S	$

After entering this configuration, The parser halts and announces successful completion of parsing. There are four possible actions the parser can make:

Shift: the next input symbol is shifted onto the stack.

Reduce: parser knows right end of handle, it must locate left end of handle and decide what non-terminal to replace the handle.

Accept: successful completion.

Error: calls an error recovery routine.

Example:

Consider the following grammar:

E→E+E|E*E|(E)|id

The input string is **id+id*id**

Stack	input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by E→id
$E	+id*id$	shift
$E+	id*id $	shift
$E+id	*id$	reduce by E →id
$E+E	*id$	shift
$E+E*	id$	shift
$E+E*id	$	reduce by E →id
$E+E*E	$	reduce by E→ E*E
$E+E	$	reduce by E→ E+E
$E	$	accept

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator

2nd Class/2nd Sem

Lecture 6

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

SYNTAX ANALYZER

Part 3

3.4 Ambiguity , Left Recursion , and Left Factoring

A grammar is said to be **left recursive** if it has a non-terminal A such that there is a derivation A $\longrightarrow$ Aa for some string a. Form this we can notice that there is two types of left recursion , either **immediate** or **non-immediate**.

Example:

Consider the following grammars:

E $\rightarrow$ E+T|T (immediate left recursion)

S $\rightarrow$ Aa|b (non-immediate left recursion)

A $\rightarrow$ Ac|Sd|Є (since S $\rightarrow$ Aa $\rightarrow$ Sda)

3.4 Ambiguity , Left Recursion , and Left Factoring

Example1:

Eliminating left recursion from the following grammar

Consider the following grammars:

$E \longrightarrow E+T|E-T|T$

$T \longrightarrow T*P|T/P|P$

$P \longrightarrow F \uparrow P|F$

$F \longrightarrow -E|(E)|id$

Solution

$E \longrightarrow TE`$

$E` \longrightarrow +TE`|-TE`|\in$

$T \longrightarrow PT`$

$T` \longrightarrow *PT`|/PT`| \in$

$P \longrightarrow F \uparrow P|F$

$F \longrightarrow -E|(E)|id$

3.4 Ambiguity , Left Recursion , and Left Factoring

Example2:

Eliminating left recursion from the following grammar

$S \rightarrow Aa | b$

$A \rightarrow Ac | Sd$

Solution

$S \rightarrow Aa | b$

$A \rightarrow Ac | Aad | bd$

$S \rightarrow Aa | b$

$A \rightarrow bdA`$

$A` \rightarrow c A` | ad A` | \in$

3.5 top-down parsing

The top-down construction of a parse tree is done by starting with the root and repeatedly performing the following two steps:

1- At node n, select one of the productions for A and construct children of n for the symbols on the right- hand side of the production.

Find the next node at which a sub tree is to be constructed.

We will consider two types of top-down parsing

1-Recursive-Descent Parsing

A special case of it called predictive parsing.

2-Non-recursive Predictive Parsing

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each non-terminal of a grammar. Recursive-descent parsing may involve backtracking (making repeated scans of the input).

Type $\longrightarrow$ Simple|id|array[simple]of type

Simple $\longrightarrow$ integer|char|num..num

3.6 Bottom-Up Parsing

Shift-reduce parsing:

1- Operator precedence (easy form)

2- LR parsing (general form)

LR parsing:

It is an efficient bottom-up syntax analysis technique that can be used to parse a large class of context – free – grammar.

LR parser is non –backtracking parser.

Algorithm: LR parsing

Input: An input string **w** and **LR parsing table** with functions **action and goto** for the grammar G.

Otput: if w in L(G), a bottom-up parse for w; otherwise, an error indicated.

Metod :initially, the parser has S0 on its stack, where S0 is the initial state, and w$ in the input buffer. The parser then execute the following program until an accept or error action is encountered.

```
Set ip to point the first symbol of w$.
Repeat forever begin
Let S be the state on top of the stack and a the symbol pointed to by ip
If action[S,a]=shift S` then begin
Push a then S` on top of the stack
Advance ip to the next input symbol
End
Else if  action[S,a]=reduce $A \longrightarrow B$ then begin
Pop 2*|B| Symbols off the stack.
Lets S` be the state now on top of the stack.
Push A then goto[S`,A] on top of the stack.
```

Else if action [S,a]=accept then return accept

Else error()

End

Example :Consider the following grammar

1) E⟶E+T

2) E⟶T

3) T⟶T*F

4) T⟶F

5) F⟶(E)

6) F⟶id which has the following parsing table

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACC			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Si means shift and stack state i
Rj means reduce by production numbered j
Acc means accept
Blank means error

If the string is **id*id+id** then the parsing of this string will be

Stack	input	Action
0	id*id+id$	Shift S5
0id5	*id+id$	Reduce by R6(F$\rightarrow$id)
0F3	*id+id$	Reduce by R4(T$\rightarrow$F)
0T2	*id+id$	Shift S7
0T2*7	id+id$	Shift S5
0T2*7id5	+id$	Reduce by R6 (F$\rightarrow$id)
0T2*7F10	+id$	Reduce by R3 (T$\rightarrow$T*F)
0T2	+id$	Reduce by R2 (E $\rightarrow$T)
0E1	+id$	Shift S6
0E1+6	id$	Shift S5
0E1+6id5	$	Reduce by R6 (F$\rightarrow$id)
0E1+6F3	$	Reduce by R4(T$\rightarrow$F)
0E1+6T9	$	Reduce byR1(E$\rightarrow$E+T)
0E1	$	**accept**

Action[0,id]=S5	
Action[5,*]=R6	goto[0,F]=3
Action[3,*]=R4	goto[0,T]=2
Action[2,*]=S7	
Action[7,id]=S5	
Action[5,+]=R6	goto[7,F]=10
Action[10,+]=R3	goto[0,T]=2
Action[2,+]=R2	goto[0,E]=1
Action[1,+]=S6	
Action[6,id]=S5	
Action[5,$]=R6	goto[6,F]=3
Action[3,$]=R4	goto[6,T]=9
Action[9,$]=R1	goto[0,E]=1
Action[1,$]=**Acc**	

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator

2nd Class/2nd Sem

Lecture 7

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

Semantic Analysis

The Semantic analysis phase of compiler **connects variable definition to their uses**, and **checks that each expression has a correct type.**

This checking called "**static type checking**" to distinguish it from "**dynamic type checking**" during execution of target program. This phase is characterized be the maintenance of symbol tables mapping identifiers to their types and locations.

Semantic Analysis

Example of static type checking:

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

2- **Flow of control checks:**- Statements that cause flow of control leave construct must have some place to which to transfer the flow of control .

For example, a “break” statement in ‘C’ Language causes control to leave the smallest enclosing while ,for ,or switch statement.

Semantic Analysis

Example of static type checking:

3- **Uniqueness checks**:- There situations in which an object must be defined exactly once. For example in ‘Pascal’ Language an identifier must be declared uniquely.

4- **Name related checks**:- Sometimes, the same name must appear two or more times. The compiler must check that the same name Is used at both places.

Semantic Analysis

Type system:-

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

The following excerpts are example of information that a compiler writer might have to start with

- If both operands of the arithmetic operators "addition" , "subtraction", and "multiplication" are of type integer then the result is of type integer.

Semantic Analysis

- The result of Unary & operator is a pointer to the object referred to by the operand. If the type of operand is T, the type of result is pointer to T.

We can classify type into:

1- Basic type: this type are the atomic types with no internal structure , such as Boolean, Integer, Real, Char, and a special basic types ”type-error, void”.

2-Construct types: Many programming Languages allows a programmer to construct types from basic types and other constructed types. For example array, struct.

Semantic Analysis

3- complex type: Such as link list, tree, pointer.

Type system:- is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system.

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

VIEDEO LECTURES

Basic Language Translator

2nd Class/2nd Sem

Lecture 8

ا.م وجدان عبد الامير حسن

Intermediate Code Generation(IR)

IR is an internal form of a program created by the compiler while translating the program from a HLL to LLL(assembly or machine code),from IR the **back end** of compiler generates target code.

Although a source program can be translated directly into the target language, some benefits of using a machine independent IR are:

1- A compiler for different machine can be created by attaching a back end for a new machine into an existing frond end.

2- Certain optimization strategies can be more easily performed on IR than on either original program or LLL.

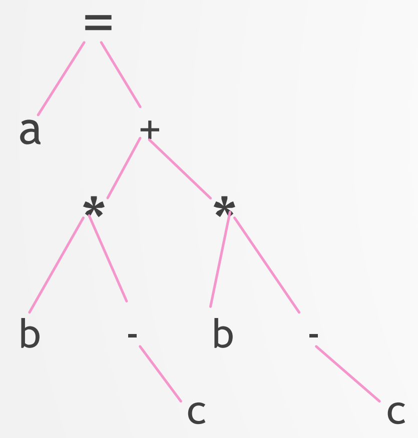
3- An IR represent a more attractive form of target code.

front end of compiler(lexical analysis , syntax analysis, semantic analysis)

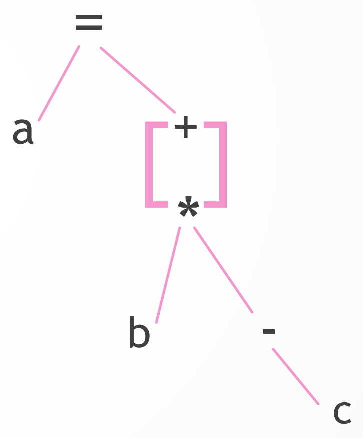
back end of compiler(code optimization, code generation)

Intermediate Languages:

1- Syntax Tree and Postfix Notation are two kinds of intermediate representation, for example a=b*-c + b*-c



Syntax tree



DAG

- A DAG give the same information in syntax tree but in compact way because common subexpressions are identified.
- Postfix notation is linearized representation of a syntax tree, for example abc-*bc-*+=

2-Three Address Code is a sequence of statement of the general form:

X=Y op Z //op is binary arithmetic operation

For example x+y*z

t1=y*z

t2=x+t1

Where t1,t2 are compiler generated temporary.

Types of three address code statement:

1- Assigning statement of the form X=Y op Z (where op is a binary arithmetic or logical operator)

2-Assigning instructions of the form X=op Y (op is a unary operator)

3- Copy statement of the form X=Y.

4- Unconditional jump(Goto l).

5- Conditional jump(if X relop Y goto L).

6- Param X and call P,N for function call and return Y.

7- Index assignments of the form X=Y[i] & X[i]=Y.

8- Address & pointer Assignments

x=&y

X=*Y

*X=Y

Implementation of three address code:

Example: a=b *-c + b*-c

```
t1=-c
t2=b*t1
t3=-c
t4=b*t3
t5=t2+t4
a=t5
```

Three address code for syntax tree

```
t1=-c
t2=b*t1
t5=t2+t2
a=t5
```

Three address code for DAG

Note: three address statement are kin to assembly code statement can have symbolic labels and there are statements for flow of control.

In compiler , three address code can be implement as records, with fields for operator and operands.

1- Quadruples:-It is a record structure with four fields:

- OP//operator
- Arg1,arg2//operands
- Result

2- Triples:-To avoid entering temporary into ST, we might refer to a temporary value by position of the statement that compute it. So three address can be represent by record with only three field:

- OP//operator
- Arg1,arg2//operands

Example: a=b*-c + b*-c

i.By Quadruples

Position	Op	arg1	arg2	result
0	-	c		t1
1	*	b	t1	t2
2	-	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

Example: a=b*-c + b*-c

ii.By Triples

Position	Op	arg1	arg2
0	-	c	
1	*	b	(0)
2	-	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator

2nd Class/2nd Sem

Lecture 9

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

Code Optimization

Compiler should produce target code that is as good as can be written by hand. This goal is achieved by program transformation called "**Optimization**". Compilers that apply code improving transformations are called "Optimizing compilers".

Code optimization attempts to increase program efficiency by restructuring code to simplify instruction sequences and take advantage of machine specific features:-

- Run faster, or

- Less Space, or

- Both(Run Faster and Less Space)

Code Optimization

The transformation that are provided by an optimizing compiler should have several properties:-

1- A transformation must preserve the meaning of program. That is , an optimizer must not change the output produce by program for a given input, such as **division by zero**.

2- A transformation must speed up programs by a measurable amount.

Code Optimization

Basic Blocks:-

The code is typically divided into a sequence of “Basic Bloks”.

A Basic Blok is a sequence of straight-line code, with no branches “In” or “out” except a branch “In” at the top of block and a branch “out” at the bottom of block.

- **Set of Basic Block :** The following steps are used to set the Basic Block :

1- Determine the Block beginning:

i- The First instruction.

ii- Target of conditional and unconditional Jumps.

iii-Instruction follow Jumps.

2- Determine the Basic Blocks:

i- There is Basic Block for each Block beginning.

ii- The Basic Block consist of the Block beginning and runs until the next Block beginning or program end.

Example:

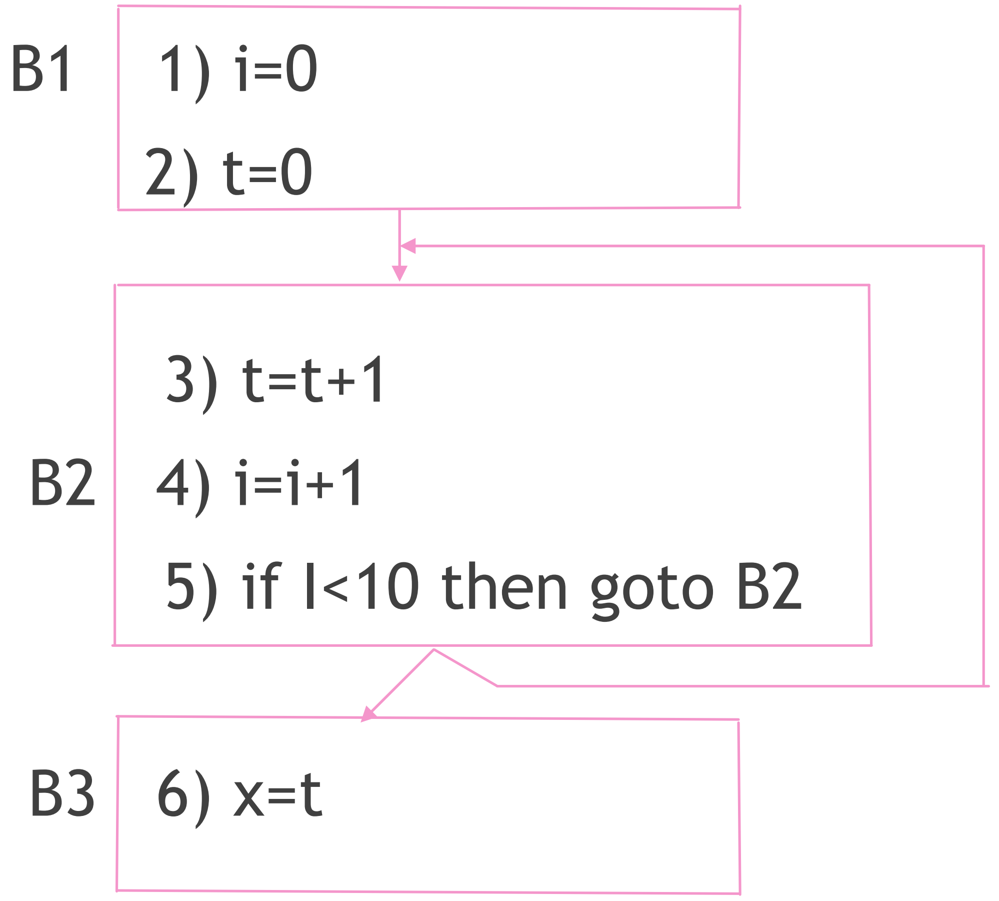
```
1) i=0
2) t=0
3) t=t+1
4) i=i+1
5) If I<10 then goto 3
6) x=t
```

```
B1   1)i=0
     2)t=0

     3)t=t+1
B2   4) i=i+1
     5)if I<10 then goto 3

B3   6)x=t
```

Basic Blocks



Control Flow

Code Optimization Methods

A transformation of program is called “local” if it can performed by looking only at the statement in a Basic Block,

Otherwise, it is called “Global”.

Local Transformations:

1- Structure-Preserving Transformation:-

- Common Subexpression Elimination.
- Dead Code Elimination.

Code Optimization Methods

2- Algebraic Transformations:- This transformations uses to change the set of expressions ,computed by a basic block ,with an algebraically equivalent set. The useful ones are those that simplify expressions or replace expensive operations by cheaper one, such as:

$$\left.\begin{array}{l} x:=x+0; \\ x:=x*1; \\ x:=x/1; \end{array}\right\} \text{Eliminated}$$

$$x:=y\hat{}2 \longrightarrow x:=y*y$$

Code Optimization Methods

Another class of algebraic transformations is **Constant Folding**,

That is, we can evaluate constant expressions at compiler time and replace the constant expressions by their values, for example, the expression 2 * 3.14 would be replaced by 6.28 .

Global Transformation:

1- Common Subexpression Elimination

```
a=b+c          a=b+c
c=b+c    ⇒     c=a
d=b+c          d=b+c
```

2-Dead Code Elimination: Variable is dead if never used

```
x=y+1
y=1         ⇒      y=1
x=2*λ              x=2* λ
```

3- Copy propagation

<u>Origin</u>	<u>Copy Propagation</u>	<u>Dead Code</u>
x=t3	x=t3	
a[t2]=t5	a[t2]=t5	a[t2]=t5
a[4]=x	a[4]=t3	a[4]=t3
Goto B2	Goto B2	Goto B2

4- Constant propagation

Origin	Copy Propagation	Dead Code
x=3	x=3	
a[t2]=t5	a[t2]=t5	a[t2]=t5
a[4]=x	a[4]=3	a[4]=3
Goto B2	Goto B2	Goto B2

5- **Loop Optimization**

- **Code Motion**: An important modification that decreases the amount of code in a loop is **Code Motion.** If result of expression does not change during loop(Invariant Computation),can hoist its computation out of the loop.

```
For(i=0; i<n; i++)
    a[i]=a[i]+(x*x)/(y*y);
c=(x*x)/(y*y);
 For (i=0; i<n; i++)
 a[i]=a[i]+c;
```

- **Strength Reduction**: Replaces expensive operations (Multiplies, Divides) by cheap ones (Adds, Subs). For example, suppose the following expression:

```
For(i=1; i<n; i++)   { v=4*i; s=s+v; }
```

Then

```
v=0;
For(i=1; i<n; i++)    { v=v+4; s=s+v; }
```

Thanks for listening

قسم علوم الحاسوب
Computer Dep.

التعليم الالكتروني
E-Learning

جامعة بغداد
University of Baghdad

كلية العلوم
College of Science

Basic Language Translator

2nd Class/2nd Sem

Lecture 10

ا.م وجدان عبد الامير حسن

VIEDEO LECTURES

Code Generation

In computer science, code generation is the process by which a compiler's code generator converts some internal representation of source code into a form(e.g., machine code)that can be readily executed by a machine.

Issues in the Design of a Code Generator:-

1- Input to the Code Generator: the input to the Code Generator consists of the intermediate representation of the source program(Optimized IR), together with information in ST(Symbol Table) that is used to determine the Run Time Addresses of the data objects denoted by names in .IR.

2-Target Programs: The output of the code generator is the target program. The output must be Correct and of high Quality, meaning that it should make effective use of the resources of the target machine.

Code Generation

Like the IR this output may take on a variety of forms:

a- Absolute Machine Language//Producing this form as output has the advantage that it can placed in a fixed location in memory and immediately executed.

A small program can be compiled and executed quickly.

b-Relocatable Machine Language// This form of the output

Allows subprograms to be compiled separately. A set of relocated object modules can be linked together and loaded for execution by linking-loader.

Code Generation

3-Memory Management: Mapping names in the source program to addresses of data objects in run time memory.

4-Major tasks in code generation: In addition to the basic conversion from IR into a linear sequence of machine instructions, a typical code generator tries to **optimize the generated code in some way**. The generator may try to use faster instructions, use fewer instructions, exploit available registers, and avoid redundant computations.

Tasks which are typically part of compiler’s code generation phase include:

i- instruction selection: Is a compiler optimization that transforms an internal representation of program into the final compiled code(either Binary or Assembly). The Quality of the generated code is determined by its Speed and Size.

For example, the three address code (x=y+z)can be translated into:

```
MOV al,y
ADD  al,z
MOV  x,al
```

If the three-address code is:

a=b+c

d=a+e

Then the target code is:

```
MOV al,b
ADD al,c
MOV a,al
MOV al,a
ADD al,e
MOV d,al
```

•

Finally ,A target machine with “Rich” instruction set may be provide several ways of implementing a given operation .For Example if the target machine has an “increment” instruction(INC), then the IR a=a+1 may be implemented by the single instruction(INC a) rather than by a more obvious sequence:

```
MOV al,a
ADD  al,1
MOV a,al
```

ii-Instruction Scheduling: In which order to put those instructions. Scheduling is a speed optimization. The order in which computation are performed can effect the efficiency of the target code ,because some computation orders require fewer registers to hold intermediate results than others.

ii-Register Allocation: Is the process of multiplexing a large number of target program variables onto a small number of CPU registers. The goal is to keep as many operands as possible in registers to maximize the execution speed of software programs(instructions involving register operands are usually shorter and faster than those involving operands in memory).

Thanks for listening