Lectures in :

Compilers
Principles & Techniques

By

Dr. Esam T. Yassen

Uni. Al-Anbar, Computers College

Adapted By

Dr. Sameeh Abdulghafour Jasim

Uni. of Al-MAARIF, Col. of Sciences

Bottom-Up Parsing

The term "Bottom-Up Parsing" refer to the order in which nodes in the parse tree are constructed, construction starts at the leaves and proceeds towards the root. Bottom-Up Parsing can handle a large class of grammars.

1. **Shift-Reduce Parsing:** Is a general style of Bottom-up syntax analysis , it attempts to construct a parse tree for an input string beginning at leaves and working up towards the root,(reducing a string *w* to the start symbol of grammar).At each reduction step a particular substring matching the right side of production is replaced by the symbol on the left of that production.

Example : consider the grammar

S ⟶ aABe

A ⟶ Abc | b

B ⟶ d

And the input is **abbcde**

The implementation Bottom-Up Parsing is

a <u>b</u> b c d e
a <u>A b c</u> d e
a A <u>d</u> e
<u>a A B e</u>
S
Accept

Handle : Is a substring that matches the right side of a production.

<u>Stack Implementation of Shift-Reduce Parsing:</u>

A convenient way to implement a shift-reduce parser is to use a *Stack* to hold a grammar symbols and an input buffer to hold the sting *w* to be parsed. We use $ to mark the bottom of *stack* and also the right end of the input string. There are actually four possible actions:

1. **Shift :** The next input symbol is Shifted onto the top of *stack*.
2. **Reduce :** Replace the handle with nonterminal.
3. **Accept :** The parser announces successful completion of parsing .
4. **Error :** The parser discovers that syntax error has occurred and calls an error recovery routine.

Example: Consider the following grammar

$$\mathbf{E \longrightarrow E+E \mid E*E \mid (E) \mid id}$$

And the input string is **id + id * id,** then the implementation is :

Stack	Input Buffer	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce: E→id
$E	+id*id$	Shift
$E+	id*id$	Shift
$E+id	*id$	Reduce: E→id
$E+E	*id$	Shift(*)
$E+E*	id$	Shift
$E+E*id	$	Reduce: E→id
$E+E*E	$	Reduce: E→E*E
$E+E	$	Reduce: E→E+E
$E	$	Accept

Conflicts During Shift-Reduce Parsing:

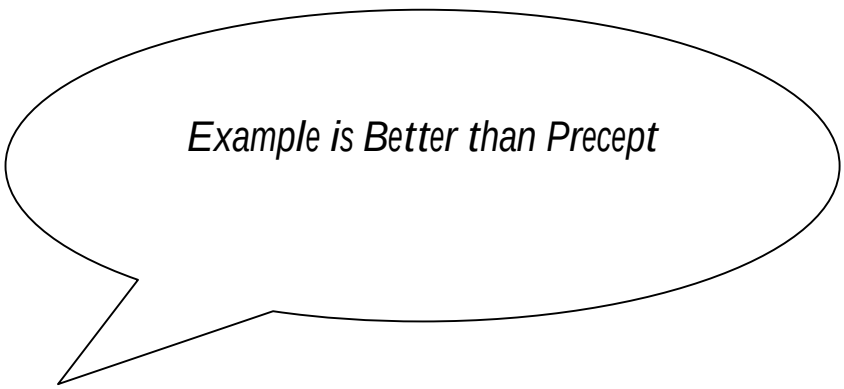
There are context free grammars for which shift-reduce parsing cannot be used. Ambiguous grammars lead to parsing conflicts. Can fix by rewriting grammar or by making appropriate choice of action during parsing. There are two type of conflicts :

1. **Shift/Reduce** conflicts: should we shift or reduce? (See previous example (*))
2. **Reduce/Reduce** conflicts: which production should we reduce with? for example:

stmt → id(param)
param → id
expr → id(expr) | id

Stack	**Input Buffer**	**Action**
$...id(id	,id)...$	Reduce by ??

Should we reduce to **param** or to **expr** ?

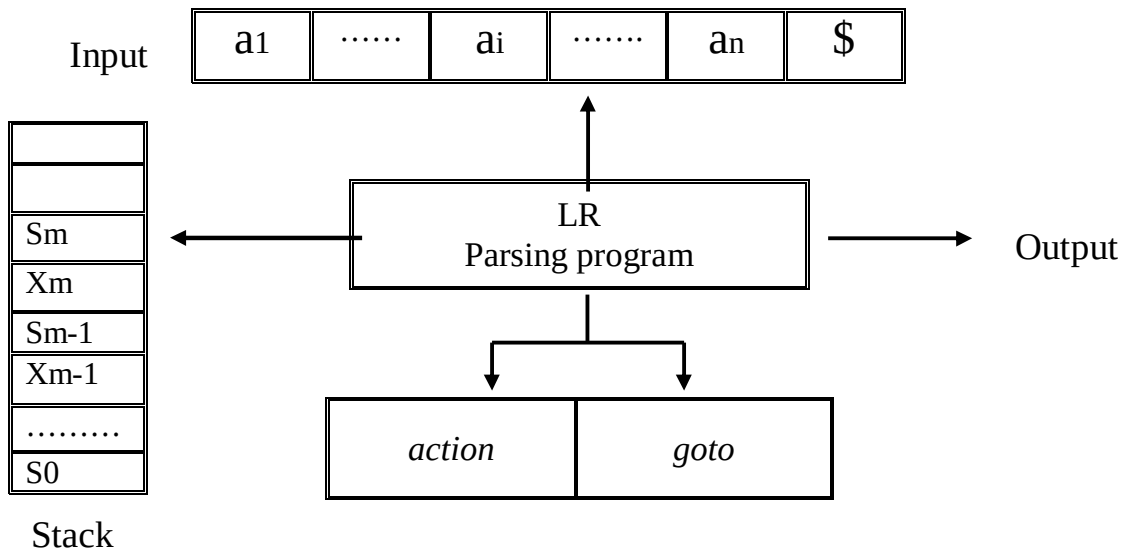


LR Parsers

This section presents an efficient Bottom-Up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing, the "L" is for left to right scanning of input, the "R" for constructing a rightmost derivation in reverse, and "k" for the number of input symbols of lookahead that are used in making parsing decisions-when "k" is omitted , k is assumed to be 1).
LR parsing is attractive for a variety of reasons:-

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general *nonbacktracking* shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

The schematic form of an LR parser is shown in following figure .It consists of **an Input**,**an Output**,**a Stack**,**a Driver program**,and **a Parsing table** that has two parts (*action* and *goto*) .



Model of an LR parser

There are three techniques for LR parser depending on the construct of LR parsing table for a grammar :

1. **Simple LR parser (SLR for short):**Is the easiest to implement but the least powerful of the three.It may be fail to produce a parsing table for certain grammars on which the other methods succeed.
2. **Canonical LR parser:** It is most powerful, and most expensive.
3. **Lookahead LR parser (LALR for short):**It is intermediate in power and cost between other two. The LALR method will work on most programming-language grammars and ,with some effort ,can be implemented efficiently.

The LR parsing Algorithm :- The LR program is the same for all LR parsers, only the parsing table changes from one parser to another.

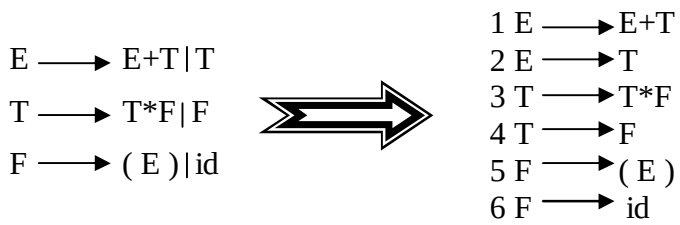
```
push the start state s0 onto the stack.
while (true) begin
    s = state on top of the stack and
    a = input symbol pointed to by input pointer ip
    if action[s,a] = shift s' then begin
       push a then s' onto the stack
       advance ip to the next input symbol
    end
    else if action[s,a] = reduce A → β then begin
       pop 2*|β| symbols off the stack, exposing state s'
       push A then goto[s',A] onto the stack
       output production A → β
    end
    else if action = accept then return
    else error()
end
```

Implementation of SLR parser:-

The SLR-parser Extremely tedious to build by hand, so need a generator. The following steps represents the main stages, which are used to build system that is used for implementing the SLR-parser:

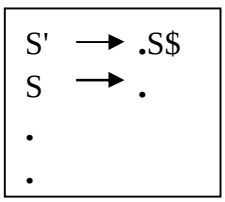
1. Input stage : In this state the grammar has been reading and the symbols of grammar (*terminals* and *nonterminals*) could be specified and each production of grammar must be on one straight line. Finally, the productions has been numbered.

For example, consider the grammar



2. Compute First & Follow stage : Through this state First & Follow could be detected for each *nonterminal*.

3. Construct DFA stage: By using a deterministic finite automaton (DFA)the SLR-parser know when to *shift* and when to *reduce*. the edges of DFA are labeled by symbols of grammar(terminals & nonterminals).In this state, where the input begins with S'(root),that means that it begins with any possible right-hand side of an S-production we indicate that by



Call this **state1** or **state0**,a productions combined with the **dot**(**.**) that indicates a position of parser.Firstly ,for each production in state1 we exam the symbol that occur after dot, there are three cases :

1. If the symbol is ***null*** (the dot has been occurred in the end of right side of production),then there are no new state .
2. If the symbol is **"$"** sign, then there are no new state.

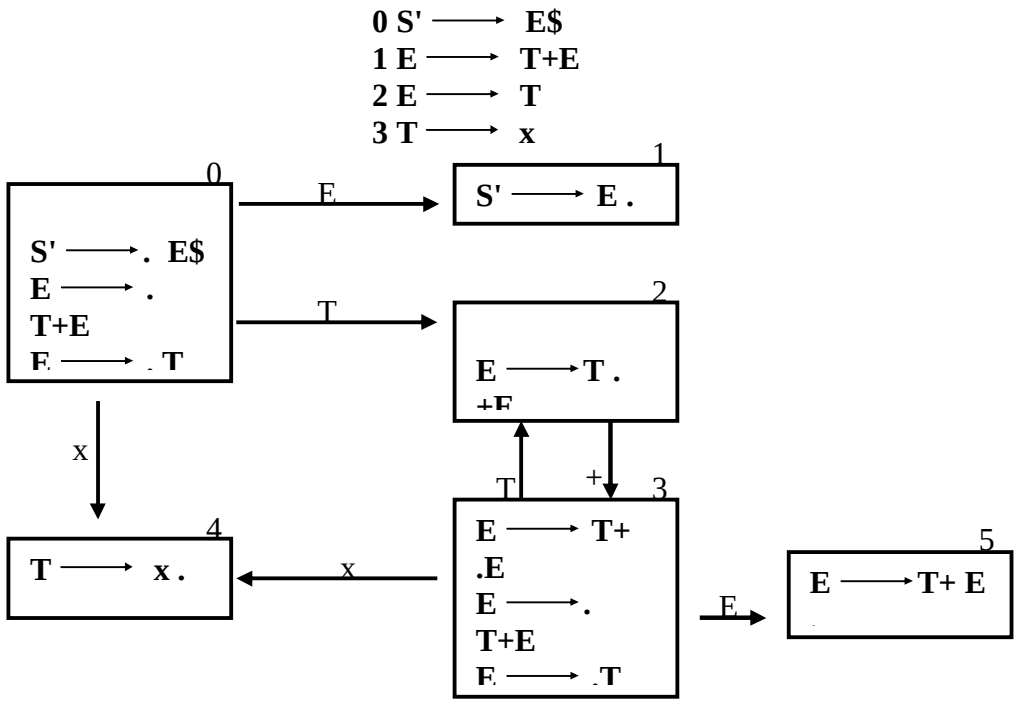
3. If the symbol is a ***terminal*** or ***nonterminal***, then there are new state, this state start with current production after the dot has been proceeded one step forward. If the symbol has been occurred after the dot(in new position)is *nonterminal* such as ***A***, then we add all possible right hand side of ***A*** to a new state, and so on.

You must know that any new state must built firstly in a buffer, and we compare it with a previous states in **DFA**, if there are no similarity situation then the new state is added to **DFA** and give it a new number equal to number of states in DFA plus one. Finally ,we repeat this steps on all new states until the **DFA** completed.

Example : consider grammar

E ⟶ T+E
E ⟶ T
T ⟶ x

Initially, it will have an empty stack, and the input will be a complete S-sentence followed by $;that is the right-hand side of the S' rule will be on the input. we indicate this as S' ⟶ . S$ where the dot indicates the current position of the parser. So:



Deterministic finite automaton (DFA)

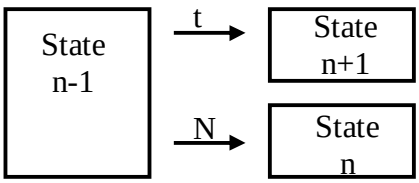
4. Construct SLR table stage: The SLR-table is a data structure consist of many rows equal to the number of the states in DFA,also many columns equal to the number of grammar symbols plus "$" sign .As know ,data structure presents fast in information treatment and information retrieve .In this stage SLR-table is constructed .this table had seen as two subtables:

1. **The Action table:** consist of many rows equal to number of states in DFA,and many columns equal to number of terminals plus "$" sign(the end of input).
2. **The Goto table:** consist of many rows equal to number of states in DFA,and many columns equal to number of nonterminals.

the elements(entries) in the SLR-table are labeled with four kinds of actions:

- Sn shift into state n
- gn goto state n
- rk reduce by production k
- a accept
- error (denoted by blank entry in the table)

For the construction of this table and the contribution the actions on the tables cells must pass to each state in DFA individually :



- Shift action & Goto action could be specified according to the edge which has been moved from the current state(n) to the new state.

  If the edge was terminal symbol (t) then

  Cell[n-1,t]= sn

  If the edge was nonterminal symbol (N) then

  Cell[n-1,N]= gn

- If there are production in current state has the form **A → ß.** (the dot in the end of right hand side, **ß** is any string),then the action is reduce

  Cell[n-1,f]=rk { **f** in Follow(A), **k** is the no. of production}
- If there are production in current state has the form **A → ß.**$ {the dot occurred before $ sign, **ß** is any string },then the action is accept

  Cell[n-1,$]=a
- Finally, any empty cell in row n-1 means error action.

Repeat the above steps for each states in DFA.

state	x	+	$	E	T
0	S4			g1	g2
1			Accept		
2		S3	r2		
3	S4			g5	g2
4		r3	r3		
5			r1		

5.Implement LR Algorithm : Suppose input string is **x+x**. After insert input string the **LR-program** is executed, as follows:

Stack	Input	Action
0	x+x $	shift
0S4	+x$	Reduce by **T ⟶ x**
0T2	+x$	shift
0T2S3	x$	Shift
0T2S3S4	$	Reduce by **T ⟶ x**
0T2S3T2	$	Reduce by **E ⟶ T**
0T2S3E5	$	Reduce by **E ⟶ T+E**
0E1	$	Accept

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.

Examples 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 to leave a 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 ;an error occurs if such an enclosing statement does not exist.
3. **Uniqueness checks:-** There are 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. For example, in **'Ada'** language a loop or block may have a name that appear at the beginning and end of the construct. The compiler must check that the same name is used at both places.

Type system:-

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

The following excerpts are examples 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*.

- 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, Subrange, Enumerated*, 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, set.*
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.*

Specification of a simple type checker:-

The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. In this section, we specify a type checker for simple language in which the type of each identifier must be declared before the identifier is used.

Suppose the following grammar to generates program, represented by *nonterminal* P, consisting of a sequence of declarations D followed by a single expression E.

P ⟶ D ; E

D ⟶ D ; D | id : T

T ⟶ char | int | array[num] of T | ↑T

E ⟶ literal | num | id | E mod E | E[E] | E↑

Type checker (translation scheme) produce the following part that saves the type of an identifier:

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

- **The type checking of expression:** the following some of semantic rules:

E ⟶	literal	{E.type=char}	//constants represented
E ⟶	num	{E.type=int}	// = =

We can use a function *lookup*(e) to fetch the type saved in *ST* ,if identifier " e " appears in an expression:

E ⟶ id {E.type=lookup(id.entry)}

The following expression formed by applying (mod) to two subexpression:

E ⟶ E1 mod E2 {E.type= if E1.type=int and E2.type=int then int
Else type-error }

An array reference:

E ⟶ E1[E2] { E.type= if E2.type=int 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}

- **The type checking of statements :**

S ⟶ id=E {S.type=if id.type = E.type then void
Else type-error)}

S ⟶ if E then S1 {S.type= if E.type=boolean then S1.type
Else type-error }

S ⟶ while E do S1 {S.type= if E.type=boolean then S1.type
Else type-error }

S ⟶ S1 ; S2 { S.type= if S1.type=void and S2.type=void then void
Else type-error}

Intermediate Code Generation (IR)

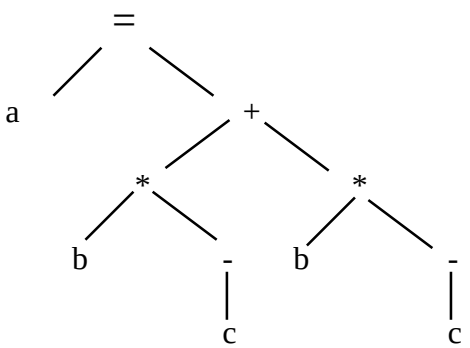
IR is an internal form of a program created by the compiler while translating the program from a *H.L.L* to *L.L.L.*(*assembly* or *machine code*),from IR the back end of compiler generates *target code*.

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

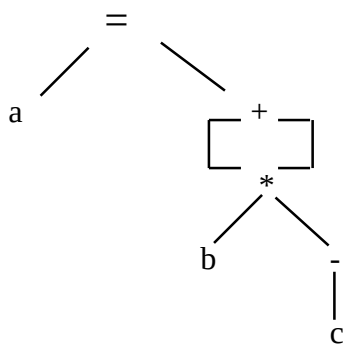
1. A compiler for different machine can be created by attaching a back end for a new machine into an existing front end.
2. Certain optimization strategies can be more easily performed on IR than on either original program or L.L.L.
3. An IR represents a more attractive form of target code.

Intermediate Languages:-

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

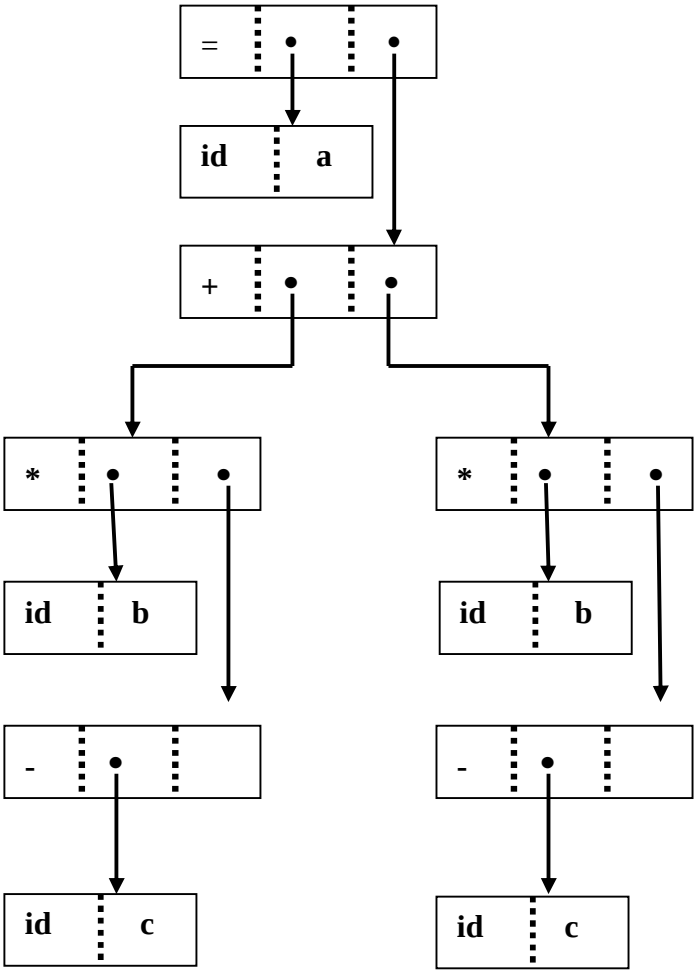


Syntax Tree



DAG

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



0	id	b	
1	id	c	
2	-	1	
3	*	0	2
4	id	b	
5	id	c	
6	-	5	
7	*	4	6
8	+	3	7
9	id	a	
10	=	9	8
	….	…..	…..
	…..	…..	…..

1 **2**

2. Three-Address Code is a sequence of statements 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. Assignment statements of the form *X=Y op Z* (where op is a binary arithmetic or logical operator).
2. Assignment instructions of the form *X= op Y* (op is a unary operator).
3. Copy statements of the form *X=Y* .
4. Unconditional jump (*Goto L*).
5. Conditional jump (*if X relop Y goto L*).
6. *Param X* & *Call P,N* for procedure call and and return *Y* , for example :

```
Param     x1
Param     x2
……..
Param     xn
Call      P,n
```

7. Index assignments of the form X=Y[i] & X[i]=Y.
8. Address & Pointer Assignments

X= &Y
*X= * Y*
**X= Y*

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 statements are a kin to assembly code statements can have symbolic labels and there are statements for flow of control.

Implementation of Three Address Code :-

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 fields:

   - **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

ii. By Triples

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

Code Optimization

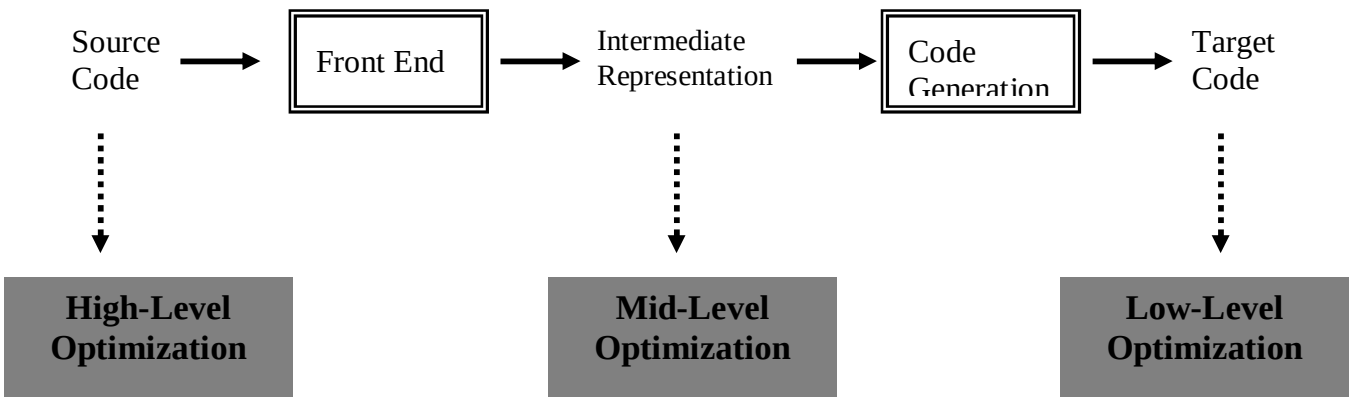
Compilers should produce target code that is as good as can be written by hand. This goal is achieved by program transformations that are called " Optimization " . Compilers that apply code improving transformations are called " Optimizing Compilers ".

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 & Less Space).

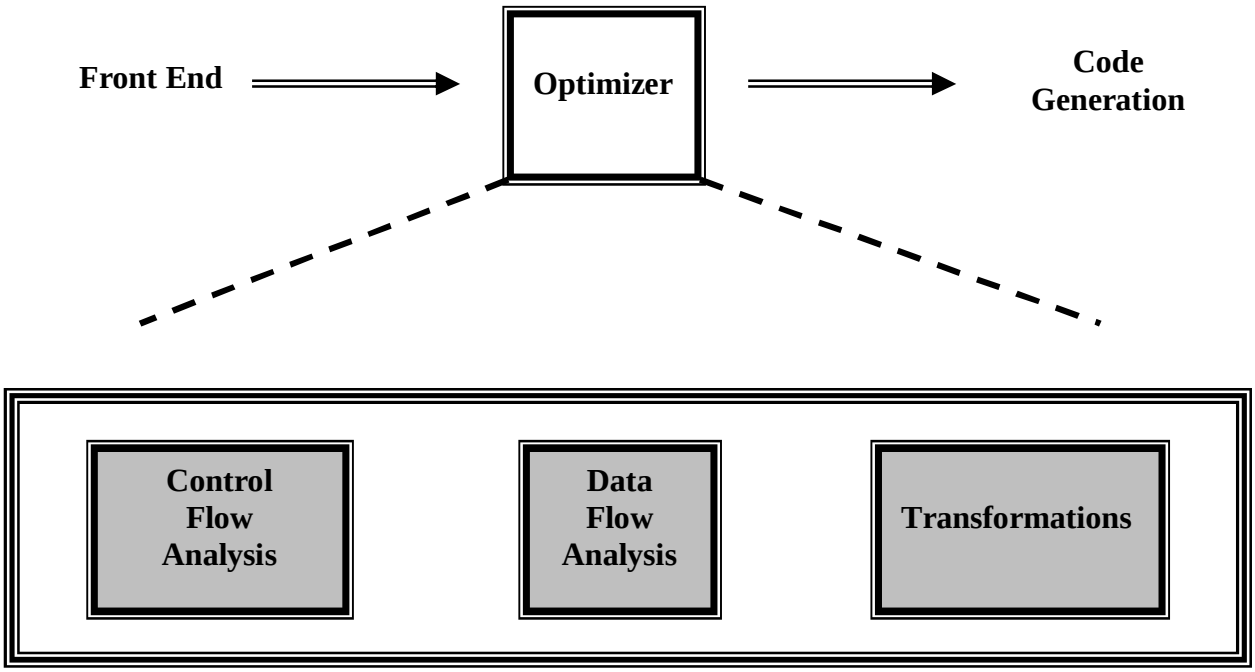
The transformations that are provided by an optimizing compiler should have several properties:-

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



Places for Optimization

This lecture concentrates on the transformation of intermediate code (Mid-Optimization or Independent Optimization),this optimization using the following organization:-



Organization of the Optimizer

This organization has the following advantages :-

1. The operations needed to implement high-level constructs are made explicit in the intermediate code.
2. The intermediate code can be independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for different machine.

Basic Blocks:-

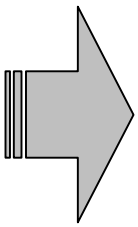
The code is typically devided into a sequence of "Basic Blocks". A Basic Block is a sequence of straight-line code,with no branches " In " or " Out " except a branch "In" at the top of block and a branch "Out" at the bottom of block.

- **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 & 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

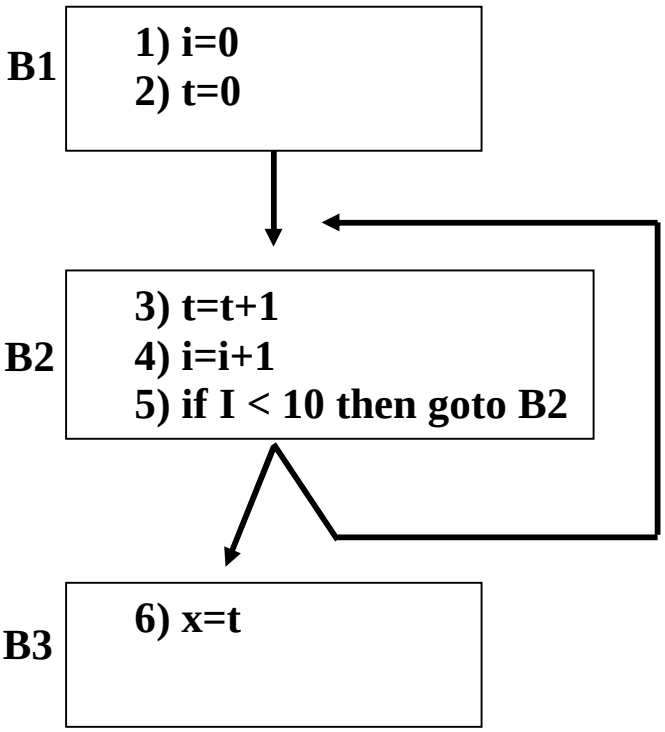


B1
1) i=0
2) t=0

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

B3
6) x=t

Basic Blocks



Control Flow

Data – Flow Analysis (DFA)

In order to do code optimization a compiler needs to collect information about program as a whole and to distribute this information to each block in the flow graph. DFA provides information about how the execution of a program may manipulate its data , and it provides information for *global optimization* .

There are many DFA that can provide useful information for optimizing transformations. One data-flow analysis determines how definitions and uses are related to each other, another estimates what value variables might have at a given point, and so on. Most of these DFAs can be described by data flow equations derived from nodes in the flow graph.

Reaching Definitions Analysis: All definitions of that variable, which reach the beginning of the block, as follow:

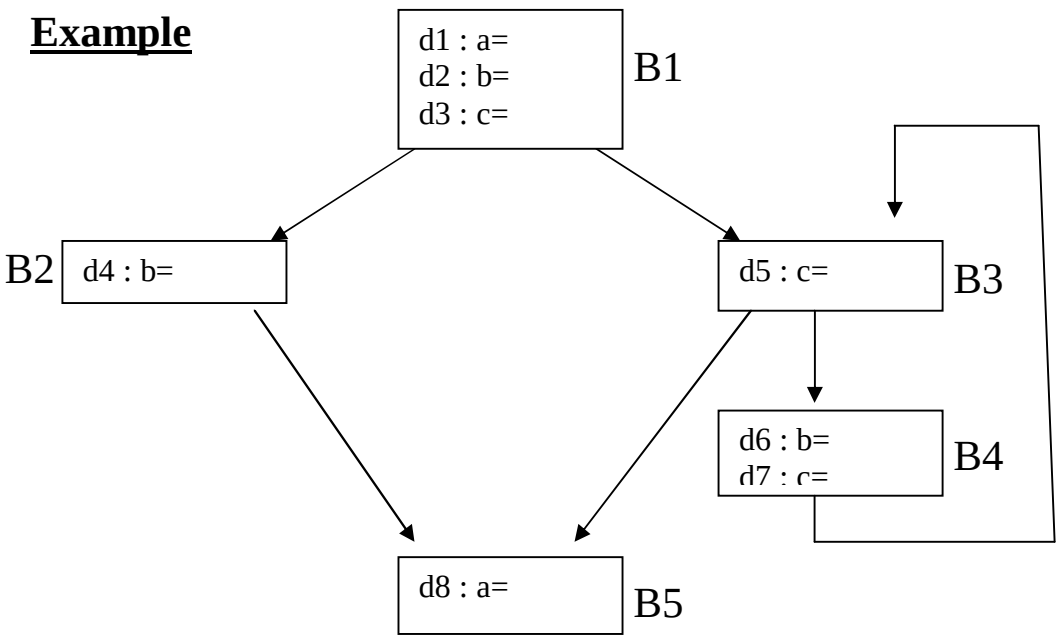
1. **Gen**[B] : contains all definitions *d:v=e* , in block B that *v* is not defined after *d* in B.
2. **Kill**[B] : if *v* is assigned in B , then Kill[B] contains all definitions *d:v=e,*in block different from B.
3. **In**[B] : the set of definitions reaching the beginning of B.

$$\mathbf{In[B]} = \cup\ \mathbf{Out[H]} \quad \text{where } H \in Pred[B]$$

4. **Out[B]** : the set of definitions reaching the end of B.

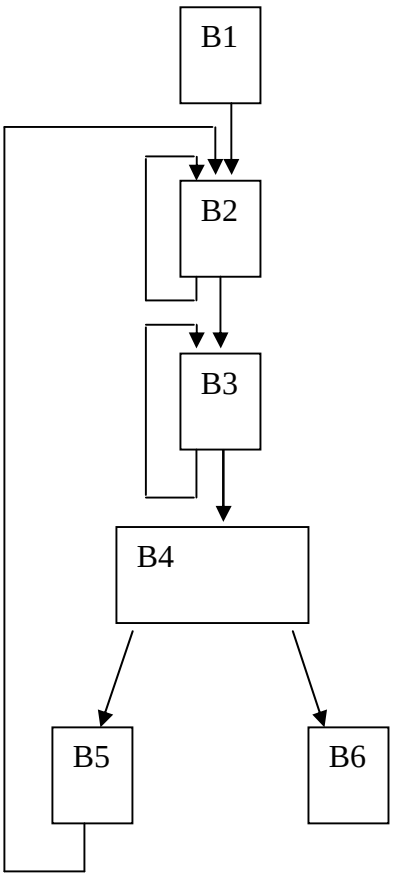
$$\mathbf{Out[B]} = \mathbf{Gen[B]} \cup (\mathbf{In[B]} - \mathbf{Kill[B]})$$

Example



Block	Gen	Kill	In	Out
B1	d1d2d3	d4d5d6d7d8	ø	d1d2d3
B2	d4	d2d6	d1d2d3	d1d3d4
B3	d5	d3d7	d1d2d3d6d7	d1d2d5d6
B4	d6d7	d2d3d4d5	d1d2d5d6	d1d6d7
B5	d8	d1	d1d2d3d4d5d6	d2d3d4d5d6d8

Loop Information: The simple iterative loop which causes the repetitive execution of one or more *basic blocks* becomes the prime area in which optimization will be considered.Here we determine all the loops in program and limit *headers* & *preheaders* for every loop, for example:



Flow Graph

Loop No.	Header	Preheader	Blocks
1	B2	B1	2-3-4-5-2
2	B2	B1	2-2
3	B3	B2	3-3

Loop Information

Code Optimization Methods

A transformation of program is called " *Local* " if it can performed by looking only at the statements in a *Basic Block*, otherwise, it is called " *Global* " .

Local Transformations:

1. Structure-Preserving Transformations:-
   - Common Subexpression Elimination
   - Dead Code Elimination

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:

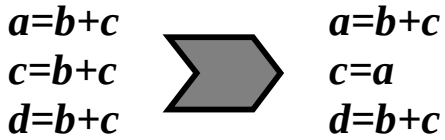
$$\left.\begin{array}{l} x:=x+0 \\ x:=x*1 \\ x:=x/1 \end{array}\right\} \quad \textit{Eliminated}$$

$$x:= y\text{^}2 \quad \Longrightarrow \quad x:=y*y$$

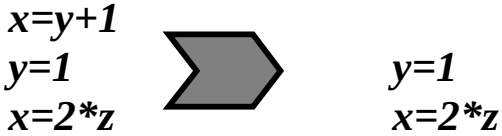
Another class of algebraic transformations is ***Constant Folding*** ,that is, we can evaluate constant expressions at compiler time and replace the constant expressions by their values, for example, the expression 2*3.14 would be replaced by 6.28.

Global Transformations:

1. Common Subexpression Elimination



2. Dead Code Elimination**:** Variable is *dead* if never used



3. Copy Propagation

Origin	Copy Propagation	Dead Code
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 operat--ions (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;} ***i*** *is induction variable*

Then:

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

Induction Variable: is a variable whose value changes by a constant amount on each loop iteration.

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 that is used to determine the Run Time Addresses of the data objects denoted by the names in IR. Finally, the code generation phase can therefore proceed on the assumption that its input is free of the errors.
2. **Target Programs :** The output of the code generator is the target program. The output code must be **Correct** and of **high Quality**, meaning that it should make effective use of the resources of the target machine. 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 relocatable object modules can be linked together and loaded for execution by linking-loader.
3. **Memory Management :** Mapping names in the source program to addresses of data objects in run time memory. This process is done cooperatively by the Front-end & code generator.
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 a 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 & Size. For example, the three address code (x=y+z) can be translated into:

```
MOV   y,R0
ADD   z,R0
MOV   R0,x
```

If three-address code is :

```
a=b+c
d=a+e
```

then the target code is :

```
MOV   b,R0
ADD   c,R0
MOV   R0,a
MOV   a,R0
ADD   e,R0
MOV   R0,d
```

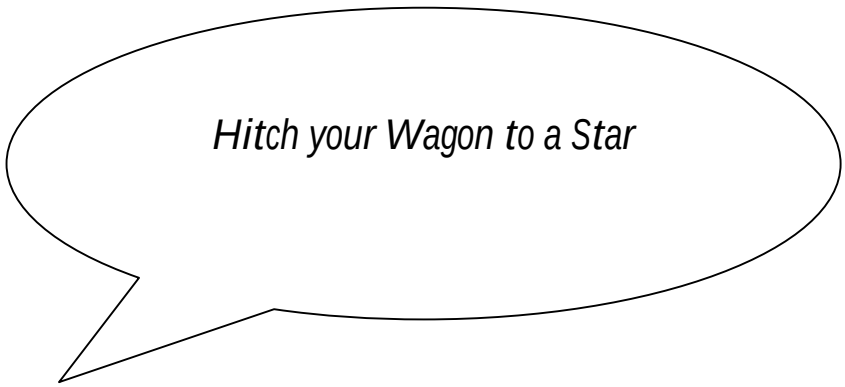
Finally, A target machine with "**Rich**" instruction set may be provide several ways of implementing a given operation. For example, if the target machine has an "increment" instruction (**INC**) ,then the IR a=a+1 may be implemented by the single instruction (INC a) rather than by a more obvious sequence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

```
MOV   a,R0
ADD   #1,R0
MOV   R0,a
```

ii. Instruction Scheduling : In which order to put those instructions. Scheduling is a *speed optimization*. The order in which computations are performed can

effect the efficiency of the target code, because some computation orders require fewer registers to hold intermediate results than others.

iii.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*).



References

1. MOGENSEN, Torben Ægidius. “**Introduction to compiler design**”. Springer Nature, 2024.

2. SINGH, Ajit. “***Compiler Design***”. Ajit Singh, 2024.

3. CAI, Shubin, et al. ComPAT: “**A Compiler Principles Course Assistant. In: *International Conference on Knowledge Science”,* Engineering and Management**. Singapore: Springer Nature Singapore, 2024. p. 74-83.

4. A.Aho,R.Sethi,J.D.Ullman," **Compilers- Principles, Techniques and Tools**" Addison-Weseley,2007

5. J.Tremblay,P.G.Sorenson,"**The Theory and Practice of Compiler Writing** ", McGRAW-HILL,1985

6. W.M.Waite,L.R.Carter,"**An Introduction to Compiler Construction**", Harper Collins,New york,1993

7. A.W.Appel, "**Modern Compiler Implementation in ML**", CambridgeUniversity Press,1998