INDEX

UNIT-I

INTRODUCTIONTOLANGUAGEPROCESSING:

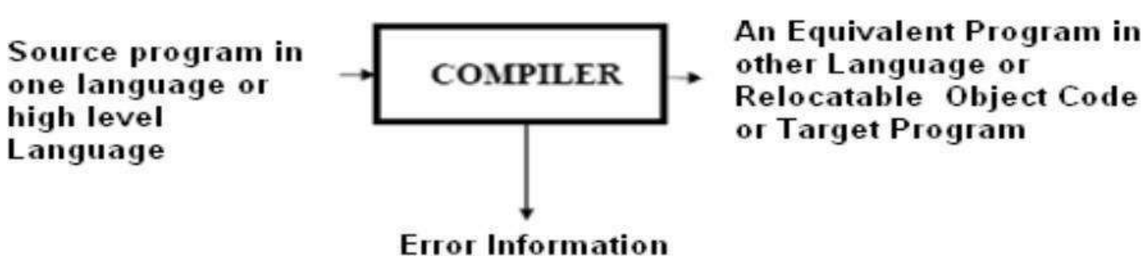
As Computers became inevitable and indigenous part of human life, and several languageswithdifferentandmoreadvancedfeaturesareevolvedintothisstreamtosatisfyorcomforttheuser in communicating with the machine, the development of the translators or mediator Software's have become essential to fill the huge gap between the human and machine understanding. This process is called Language Processing to reflect the goal and intent of the process. On the way to this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

LANGUAGETRANSLATORS:

Is a computer program which translates a program written in one (Source) language to its equivalentprograminother[Target]language.TheSourceprogramisahighlevellanguagewhereas the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level language program.

∑ Two commonly Used Translators are Compiler and Interpreter

1. **Compiler:**Compilerisaprogram,readsprograminonelanguagecalledSourceLanguage andtranslatesintoitsequivalent programinanotherLanguagecalledTarget Language, in addition to this its presents the error information to the User.



∑ Ifthetarget programisanexecutable machine-languageprogram, it canthenbecalled by the users to process inputs and produce outputs.

Figure1.1:RunningthetargetProgram

2. **Interpreter:** Aninterpreterisanothercommonlyusedlanguageprocessor.Insteadofproducing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by theuser.

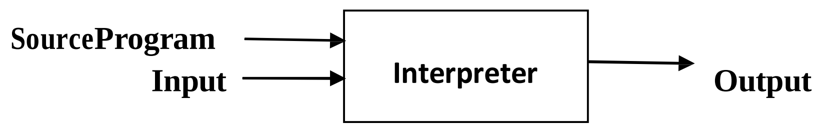


Figure1.2:Running thetargetProgram

LANGUAGE PROCESSING SYSTEM:

Basedonthe inputthetranslatortakesandtheoutputit produces,alanguagetranslatorcanbe called as any one of the following.

Preprocessor: Apreprocessortakestheskeletalsourceprogramasinput andproducesanextended version of it, which is the resultant of expanding the Macros, manifest constants if any, and includingheader filesetcinthesourcefile.Forexample,theCpreprocessorisa macro processor thatisusedautomaticallybytheCcompilertotransformoursourcebeforeactualcompilation.Over and above a preprocessor performs the following activities:

∑Collectsallthemodules,filesincaseifthesourceprogramisdivided intodifferent modules stored at different files.

∑Expandsshorthands/macrosintosourcelanguagestatements.

Compiler: Is atranslator that takes as input a source program written in high level language and convertsitinto itsequivalent target programinmachine language. Inadditiontoabovethecompiler also

∑Reportstoitsuserthepresenceoferrorsinthesourceprogram.

∑Facilitatestheuserinrectifyingtheerrors,andexecutethecode.

Assembler: Isaprogramthattakesas input anassemblylanguageprogramandconverts it intoits equivalent machine language code.

Loader/Linker: This isaprogramthattakesasinput arelocatable codeand collectsthe library functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

∑**Loading**consistsoftakingtherelocatable machinecode,alteringtherelocatableaddresses, and placing the altered instructions and data in memoryat the proper locations.

∑**Linking**allowsustomakeasingleprogramfromseveralfilesofrelocatable machine code. These files may have been result of several differentcompilations, one or more may be libraryroutines provided by the system available to anyprogramthat needs them.

In addition to these translators, programs like interpreters, text formatters etc., may be used in language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

Normally the steps in a language processing system includes Preprocessing the skeletal Source program which produces an extended or expanded source program or a ready to compile unit of the source program, followed by compiling the resultant, then linking / loading , and finally its equivalentexecutablecodeisproduced.AsIsaidearliernotallthesestepsaremandatory.Insome cases, the Compiler only performs this linking and loading functions implicitly.

The steps involved in a typical language processing system can be understood with following diagram.

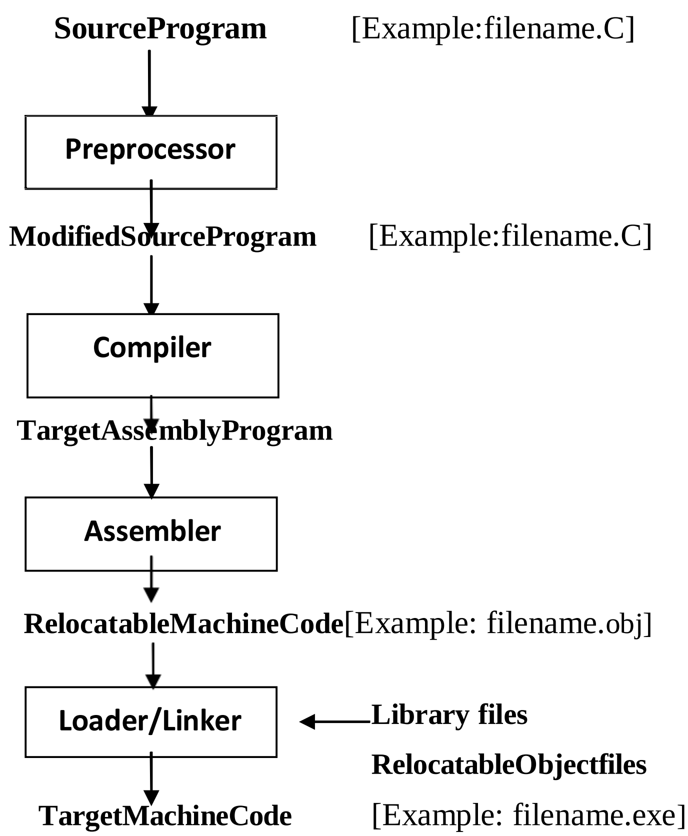


Figure1.3:ContextofaCompilerinLanguageProcessingSystem

TYPESOF COMPILERS:

Basedonthespecific input ittakesandtheoutputitproduces,theCompilerscanbeclassified into the following types;

TraditionalCompilers(C,C++,Pascal): TheseCompilersconvert asourceprograminaHLL into its equivalent in native machine code or object code.

Interpreters(LISP, SNOBOL, Java1.0): These Compilers first convert Source code into intermediate code, and then interprets (emulates) it to its equivalent machine code.

Cross-Compilers: Thesearethecompilersthatrunononemachineandproducecodeforanother machine.

Incremental Compilers: These compilers separate the source into user defined–steps; Compiling/recompiling step- by- step; interpreting steps in a given order

Converters (e.g. COBOL to C++): These Programs will be compiling from one high level language to another.

Just-In-Time (JIT) Compilers (Java, Micosoft.NET): These are the runtime compilers from intermediate language (byte code, MSIL) to executable code or native machine code. These perform type –based verification which makes the executable code more trustworthy

Ahead-of-Time (AOT) Compilers (e.g., .NET ngen): These are the pre-compilers to the native code for Java and .NET

BinaryCompilation: Thesecompilers willbecompilingobject codeofoneplatformintoobject code of another platform**.**

PHASESOFACOMPILER:

Due to the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces. GenerallyaninterfacecontainsaDatastructure(e.g.,tree),Setofexportedfunctions.Eachphase worksonanabstract **intermediate representation**ofthesourceprogram, notthesourceprogram text itself (except the first phase)

Compiler Phases arethe individual modules which are chronologicallyexecutedto performtheir respective Sub-activities, and finally integrate the solutions to give target code.

It is desirable to have relativelyfew phases, since it takes time to read and write immediate files. Following diagram(Figure1.4) depictsthe phasesofa compiler through which it goesduring the compilation. There fore a typical Compiler is having the following Phases:

1. LexicalAnalyzer(Scanner),2.SyntaxAnalyzer(Parser),3.SemanticAnalyzer, 4.IntermediateCodeGenerator(ICG),5.CodeOptimizer(CO),and6.CodeGenerator(CG)

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not all the phases are mandatory in everyCompiler. e.g, Code Optimizer phase is optional in some cases.Thedescriptionisgiveninnextsection.

ThePhasesofcompilerdivided intotwo parts,firstthreephaseswearecalledasAnalysis part remaining three called as Synthesis part.

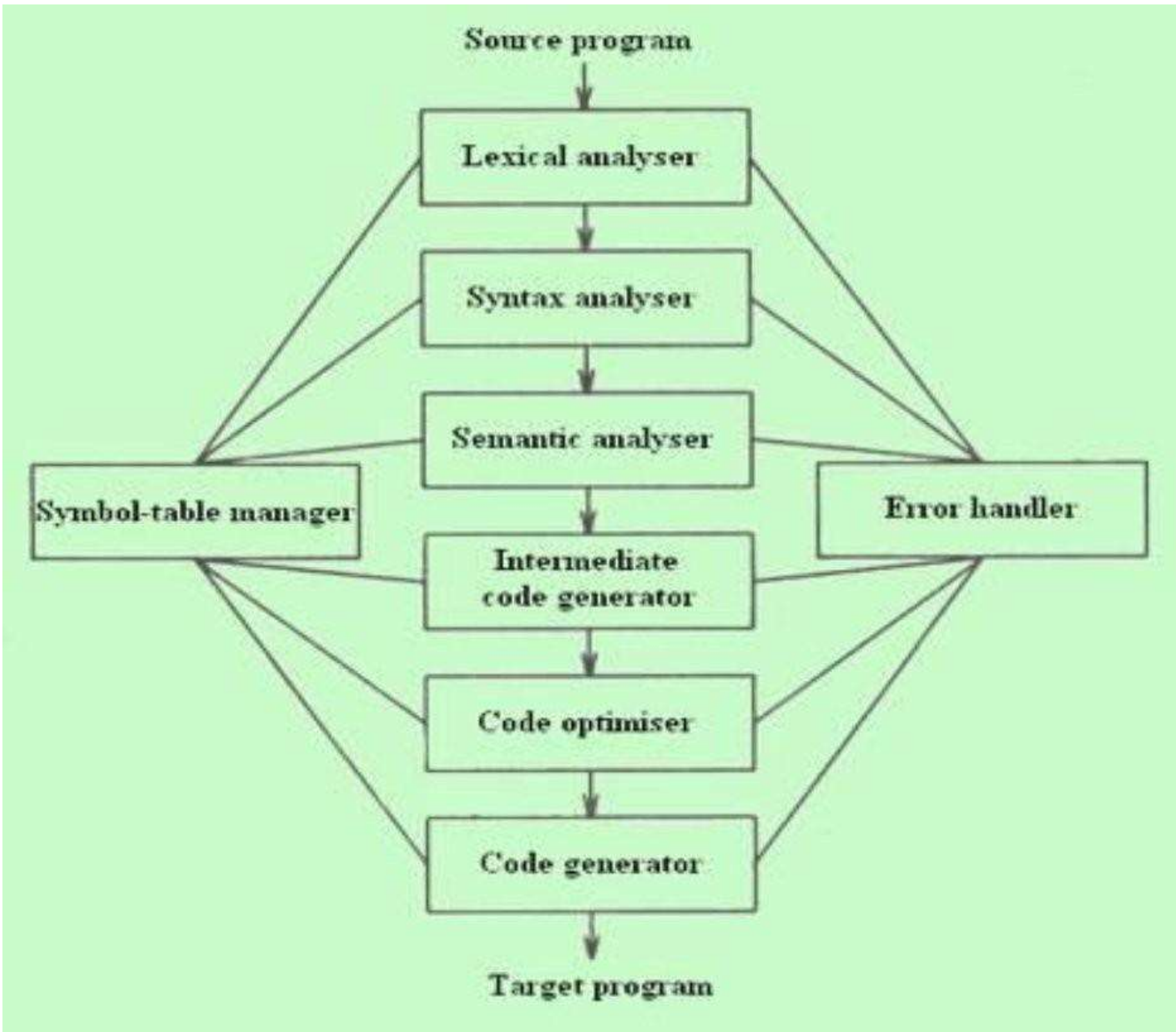


Figure1.4:PhasesofaCompiler

PHASE,PASSESOFACOMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completelydeferentrepresentation. Eachpassmakesacompletescanoftheinput andproducesits output to be processed bythe subsequent pass. For example a two pass Assembler.

THEFRONT-END&BACK-ENDOFACOMPILER

All of these phases of a general Compiler are conceptually divided into **The Front-end**, and**TheBack-end**.Thisdivisionisduetotheir dependenceoneithertheSourceLanguageorthe Target machine. This model is called an Analysis & Synthesis model ofa compiler.

The **Front-end** of the compiler consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.

The **Back-end** of the compiler consists of phases that depend on the target machine, and thoseportionsdon‘t dependent ontheSourcelanguage, just theIntermediate language. Inthiswe havedifferentaspectsofCodeOptimizationphase,codegenerationalongwiththenecessaryError handling, and Symbol table operations.

LEXICALANALYZER(SCANNER): TheScanneristhefirstphasethatworksasinterface betweenthecompilerandtheSourcelanguageprogramandperformsthefollowingfunctions:

- ∑ReadsthecharactersintheSourceprogramandgroupsthemintoastreamoftokensinwhich each token specifies a logically cohesive sequence of characters, such as an identifier , a Keyword , a punctuation mark, a multi character operator like := .
- ∑Thecharactersequenceforming a tokeniscalled a**lexeme** ofthetoken.
- ∑TheScannergeneratesatoken-id,andalso entersthatidentifiersname intheSymbol table if it doesn‘t exist.
- ∑AlsoremovestheComments,andunnecessaryspaces.

Theformatofthetokenis**<Token name,Attributevalue>**

SYNTAXANALYZER(PARSER): TheParserinteractswiththeScanner,anditssubsequent phase Semantic Analyzer and performs the following functions:

- ∑Groupstheabovereceived, andrecordedtokenstreamintosyntacticstructures,usually into a structure called **Parse Tree** whose leaves are tokens.
- ∑The interiornodeofthistreerepresentsthestreamoftokensthat logicallybelongs together.
- ∑Itmeansitchecksthesyntaxofprogramelements.

SEMANTICANALYZER: This phase receives the syntax tree as input, and checks the semanticallycorrectnessoftheprogram.Thoughthetokensarevalidandsyntacticallycorrect,it mayhappenthattheyarenotcorrectsemantically. Thereforethesemanticanalyzerchecksthe semantics (meaning) of the statements formed.

∑TheSyntacticallyandSemanticallycorrect structuresareproducedhereinthe formofa Syntax tree or DAG or some other sequential representation like matrix.

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

∑Itshould beeasytoproduce,andEasytotranslateintothetargetprogram.Example intermediate code forms are:

∑Three addresscodes,

∑Polishnotations,etc.

CODEOPTIMIZER: Thisphase isoptional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

∑Attemptsto improvetheICso asto havea faster machinecode.Typicalfunctions include – LoopOptimization, Removalofredundant computations, Strengthreduction, Frequency reductions etc.

∑Sometimesthedatastructuresusedinrepresentingthe intermediateforms mayalsobe changed.

CODE GENERATOR: This is the final phase of the compiler and generates the target code, normallyconsistingoftherelocatable machinecodeorAssemblycodeorabsolutemachinecode.

∑Memorylocationsareselectedforeachvariable used,andassignmentofvariablesto registers is done.

∑Intermediateinstructionsaretranslated intoasequenceofmachineinstructions.

TheCompileralso performsthe**Symboltablemanagement**and**Errorhandling**throughoutthe compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

Forexamplethesourceprogramisanassignment statement;thefollowing figureshowshowthe phases of compiler will process the program.

Theinputsourceprogramis**Position=initial+rate*60**

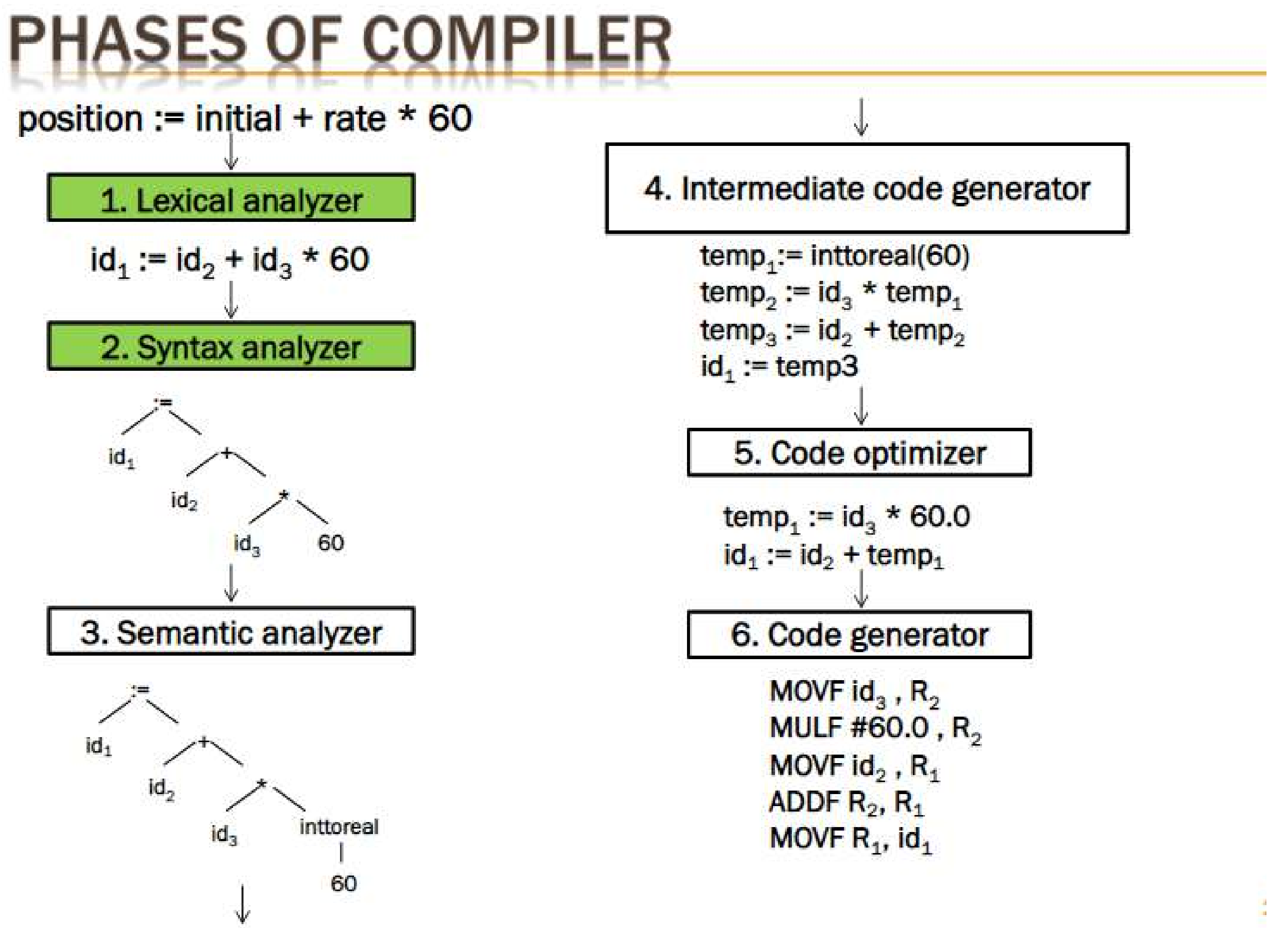


Figure1.5:TranslationofanassignmentStatement

LEXICALANALYSIS:

Asthe first phaseofacompiler, the maintaskofthelexicalanalyzeristoreadthe input charactersofthesourceprogram, grouptheminto lexemes, andproduceasoutputtokens for each lexeme inthe source program. This streamoftokens is sent to the parser for syntaxanalysis. It is common for the lexical analyzer to interact with the symbol table as well.

Whenthe lexicalanalyzer discoversa lexemeconstitutinganidentifier,it needsto enter that lexeme into the symboltable. This process is shown in the following figure.

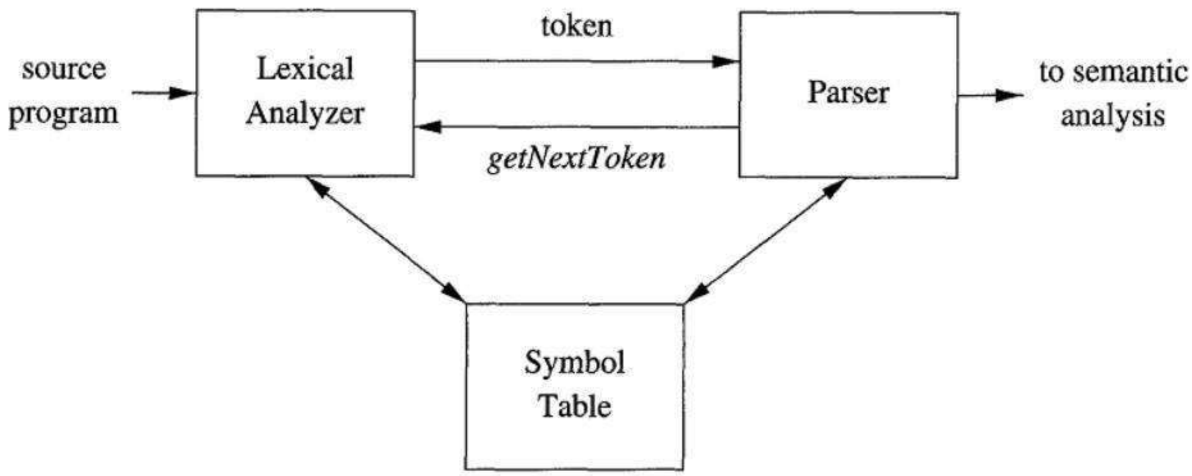


Figure1.6:LexicalAnalyzer

. When lexical analyzer identifies the first token it will send it to the parser, the parser receivesthetokenandcallsthe lexicalanalyzertosendnexttokenbyissuingthe**getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS,PATTERNS ANDLEXEMES:

A token is a pair consistingofatokennameandanoptionalattribute value.The tokenname is an abstract symbolrepresenting a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.Inwhatfollows, weshallgenerallywritethenameofatokeninboldface. Wewilloften refer to a token by its token name.

Apattern isadescriptionoftheformthatthelexemesofatokenmaytake[ormatch]. Inthe case ofa keyword as atoken, the pattern is just the sequence ofcharactersthatformthe keyword. For identifiersandsomeothertokens,thepatternisa morecomplexstructurethatis matched bymany strings.

Alexeme isasequenceofcharactersinthesourceprogramthat matchesthepatternfora token and is identified by the lexical analyzer as an instance of that token.

Example:InthefollowingClanguagestatement, printf

("Total = %d\n‖, score) ;

both**printf**and**score**arelexemesmatchingthe**pattern** fortoken**id**,and**"Total=%d\n‖** is a lexeme matching **literal [or string]**.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure1.7:ExamplesofTokens

LEXICALANALYSISVsPARSING:

Thereareanumberofreasonswhytheanalysisportionofacompiler isnormallyseparated into lexical analysis and parsing (syntax analysis) phases.

∑**1.Simplicityofdesignisthemostimportantconsideration.** TheseparationofLexicaland Syntactic analysis often allows us to simplify at least one ofthesetasks.For example,a parser thathad to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume commentsand whitespace have already been removed by the lexicalanalyzer.

∑**2. Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

∑**3.Compilerportabilityisenhanced**:Input-device-specificpeculiaritiescanbe restricted to the lexical analyzer.

INPUTBUFFERING:

Before discussing the problemofrecognizinglexemesinthe input,let us examine some waysthatthesimplebutimportanttaskofreadingthesourceprogramcanbespeeded.This taskismadedifficult bythe factthat weoftenhavetolookoneormorecharactersbeyond thenext lexemebeforewecanbesurewehavetheright lexeme. Therearemanysituationswhereweneed tolookat leastoneadditionalcharacterahead. Forinstance, wecannot besure we'veseentheend ofan identifier until we see a character that is not a letter or digit, and therefore is not part ofthe lexeme for id.InC, single-characteroperators like-,=,or<could also be the beginning ofa two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

BufferPairs

Because of the amountof time taken toprocess characters and thelarge number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.

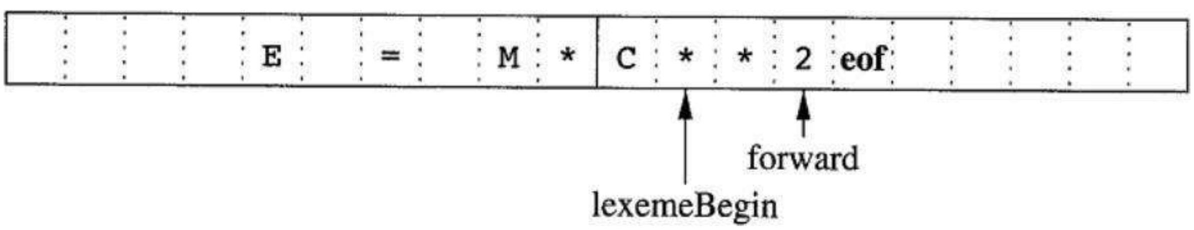


Figure1.8:UsingaPairofInputBuffers

EachbufferisofthesamesizeN,andNisusuallythesizeofadisk block,e.g.,4096bytes. Using one systemread command we can read N characters in toa buffer,rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.

∑Twopointerstotheinputaremaintained:

1. ThePointer**lexemeBegin**,marksthebeginningofthecurrent lexeme,whoseextent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy wherebythisdeterminationis madewillbecoveredinthebalanceofthischapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set tothe character immediatelyafter the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forwardrequiresthat wefirst testwhether we havereachedtheendof oneof the buffers, and if so, we mustreload the other bufferfrom the input, and move forward to the beginning ofthe newly loaded buffer. As long aswenever need to lookso far ahead ofthe actual lexemethat thesumofthe lexeme's lengthplusthedistancewelookahead isgreaterthanN, we shall never overwrite the lexeme in its buffer before determining it.

SentinelsTo ImproveScannersPerformance:

If we use the above scheme as described, we must check, each time we advance forward, thatwehavenot movedoffoneofthebuffers;ifwedo,thenwe must alsoreloadtheotherbuffer. Thus, for each character read, we make two tests: one for the end of the buffer, and oneto determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special characterthat cannot be partofthe source program, andanaturalchoice isthecharacter**eof**.Figure1.8showsthesamearrangement asFigure1.7, but with the sentinels added. Notethat eof retains its use as a marker for the end of the entire input.

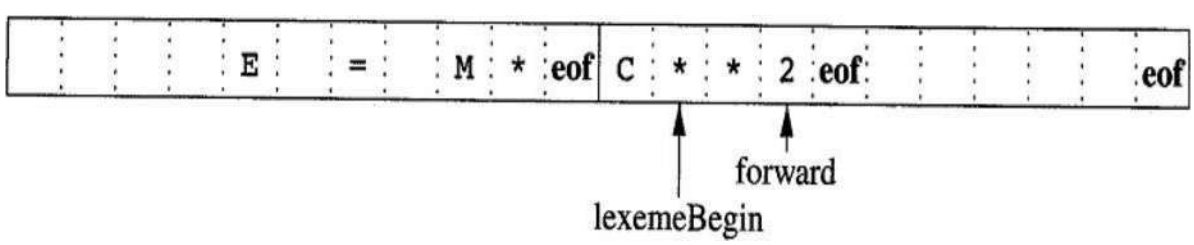


Figure1.8:Sententialattheendofeachbuffer

Anyeofthatappearsotherthanattheendofabuffermeansthatthe input isat anend. Figure1.9 summarizesthe algorithm for advancing forward.Notice howthe first test,whichcanbepart of amultiwaybranchbasedonthecharacterpointedtobyforward,istheonlytest wemake,except in the case where we actually are at the end ofa buffer or the end ofthe input.

```
switch(*forward++)
{
        caseeof:if(forward isatendoffirstbuffer)
                  {
                    reloadsecondbuffer;
                    forward=beginningofsecond buffer;
                   }
                  elseif(forwardisatendofsecondbuffer)
                  {
                    reloadfirstbuffer;
                    forward=beginningoffirstbuffer;
                  }
                    else    /*eofwithinabuffer markstheendofinput */
                     terminate lexical analysis;
           break;
}
```

Figure1.9:useofswitch-caseforthesentential

SPECIFICATIONOFTOKENS:

Regular expressions areanimportant notationfor specifyinglexemepatterns. Whiletheycannot express allpossiblepatterns, theyareveryeffectiveinspecifyingthosetypes of patterns that weactuallyneedfor tokens.

LEXtheLexicalAnalyzergenerator

Lex is a toolused to generate lexicalanalyzer, the input notation for the Lex tool is referredtoastheLexlanguageandthetoolitselfis theLexcompiler.Behindthescenes,the Lexcompilertransformstheinputpatterns intoatransitiondiagramandgeneratescode,ina filecalledlex.yy.c, it isacprogramgivenforCCompiler, givestheObject code.Hereweneed to know how to write the Lex language. The structure of the Lex program is given below.

StructureofLEX Program: ALexprogramhasthefollowingform:

Declarations
%%
Translationrules
%%
Auxiliaryfunctionsdefinitions

Thedeclarationssection : includesdeclarationsofvariables, manifest constants(identifiers declaredtostandforaconstant, e.g.,thenameofatoken), andregular definitions. It appears between %{. . .%}

Inthe **Translation rules** section, We place PatternActionpairswhere eachpair have the form

Pattern {Action}

Theauxiliary function definitionssectionincludesthedefinitionsoffunctionsusedto install identifiers and numbers in the Symbol tale.

LEXProgramExample:

```
%{
/*definitionsofmanifestconstantsLT,LE,EQ,NE,GT,GE,IF,THEN,ELSE,ID,NUMBER,
RELOP */
%}
/*regulardefinitions*/
delim       [\t\n]
ws    {     delim}+
letter       [A-Za-z]
digit        [o-91
id          {letter}({letter}| {digit})*
number       {digit}+(\.{digit}+)?(E[+-I]?{digit}+)?
%%
{ws}          {/*noactionandnoreturn*/}
if            {return(1F);}
```

```
then            {return(THEN);}
else            {return(ELSE);}
(id)            {yylval=(int)installID();return(1D);}
(number)        {yylval=(int)installNum();return(NUMBER); }
‖<‖             {yylval=LT;return(REL0P);)}
—<=‖            {yylval= LE;return(REL0P);}
—=‖             {yylval= EQ;return(REL0P);}
—<>‖            {yylval= NE;return(REL0P);}
—<‖             {yylval=GT;return(REL0P);)}
—<=‖            {yylval=GE;return(REL0P);}
%%
intinstallID0(){/*functiontoinstallthe lexeme,whose first characterispointedto byyytext, and
                    whose length is yyleng, into the symbol table and return a pointer thereto
                    */
intinstallNum(){/*similarto installID,butputsnumericalconstantsintoaseparatetable*/}
```

Figure1.10:LexProgramfortokens commontokens

SYNTAXANALYSIS(PARSER)

THEROLEOFTHEPARSER:

In our compiler model, the parser obtains a string of tokens from thelexical analyzer,as shown in the below Figure, and verifiesthatthestringoftoken names canbe generated by the grammarfor the source language.We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder ofthe program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest ofthe compiler for further processing.

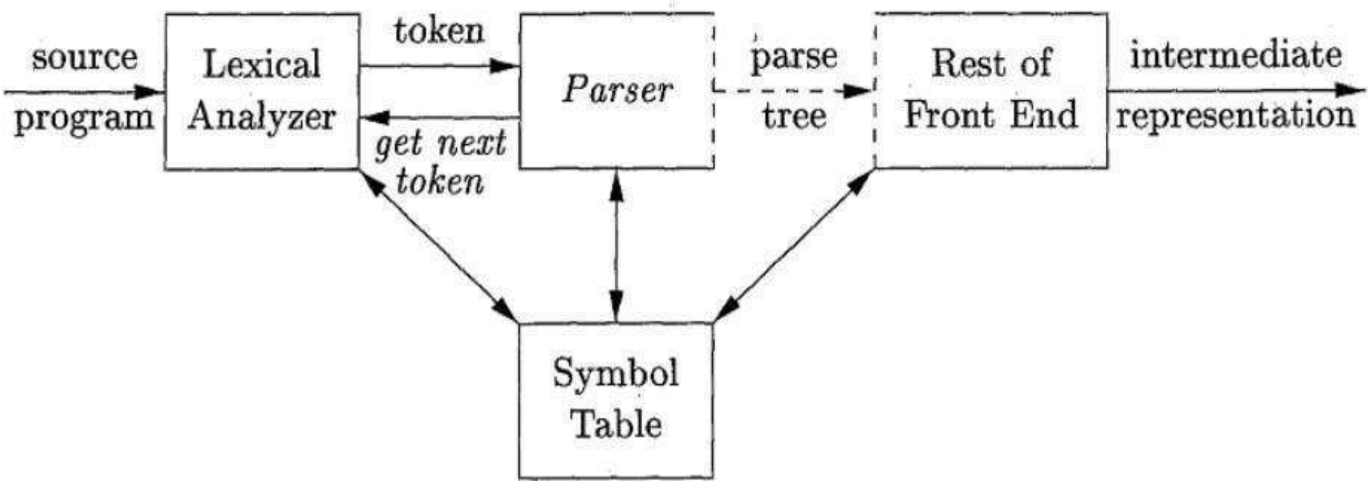


Figure2.1: ParserintheCompiler

Duringtheprocessofparsing itmayencountersomeerrorandpresenttheerrorinformationback to the user

Syntacticerrorsincludemisplacedsemicolonsorextraormissingbraces;thatis, ―{" or"}."Asanotherexample,inCorJava,the appearance ofacasestatementwithout anenclosing switch is a syntactic error (however, this situationisusuallyallowedbythe parser and caught later in the processing, as the compiler attempts to generate code).

Basedontheway/ordertheParseTreeisconstructed, **Parsing** isbasically**classified** into following two types:

1. **TopDownParsing:**Parsetreeconstructionstartattherootnodeandmovestothe children nodes (i.e., top down order).
2. **BottomupParsing:**Parsetreeconstructionbegins fromthe leafnodesandproceeds towards the root node (called the bottom up order).

IMPORTANT(OR)EXPECTEDQUESTIONS

1. WhatisaCompiler?ExplaintheworkingofaCompilerwithyourownexample?
2. WhatistheLexicalanalyzer?DiscusstheFunctionsofLexicalAnalyzer.
3. Writeshortnotesontokens,patternandlexemes?
4. WriteshortnotesonInput bufferingscheme?Howdoyouchangethebasic input buffering algorithm to achieve better performance?
5. Whatdoyou meanbyaLexicalanalyzergenerator?Explain LEXtool.

ASSIGNMENTQUESTIONS:

1. Writethedifferencesbetweencompilersandinterpreters?
2. Writeshortnotesontoken reorganization?
3. WritetheApplicationsoftheFiniteAutomata?
4. ExplainHowFiniteautomataareusefulinthelexicalanalysis?
5. ExplainDFAandNFAwithanExample?

UNIT-II

TOPDOWNPARSING:

∑ Top-down parsing can be viewed as the problem of constructing a parse tree for the given input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first left to right).

∑Equivalently, top-downparsingcanbeviewedasfindingaleftmostderivationforaninput string.

Itisclassified intotwodifferent variantsnamely;onewhichusesBackTrackingandtheotheris Non Back Tracking in nature.

NonBackTrackingParsing: Therearetwovariantsofthisparser asgivenbelow.

1. **TableDrivenPredictiveParsing:**
   i. LL(1) Parsing
2. **RecursiveDescentparsing**

BackTracking

1.BruteForcemethod

NONBACKTRACKING:

LL(1)ParsingorPredictiveParsing

LL(1)standsfor,left toright scanofinput,usesaLeft mostderivation, andtheparser takes 1 symbol as the look ahead symbol fromthe input in taking parsing action decision.

Anonrecursivepredictiveparsercanbebuilt bymaintainingastackexplicitly,ratherthan implicitly via recursive calls. The parser mimics a leftmost derivation. Ifw istheinput that has been matchedso far, thenthestackholdsa sequence ofgrammar symbols a such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

Thetable-drivenparserinthefigurehas

∑Aninput bufferthatcontainsthestringto beparsed followedbya$Symbol,usedto indicate end of input.

∑Astack, containinga sequenceofgrammar symbolswitha$atthebottomofthestack, which initially contains the start symbol of the grammar on top of$.

∑Aparsing table containingtheproductionrulestobeapplied.Thisisatwo dimensional array M [Non terminal, Terminal].

∑AparsingAlgorithmthattakesinput Stringanddeterminesifit isconformantto Grammar and it uses the parsing table and stack to take such decision.

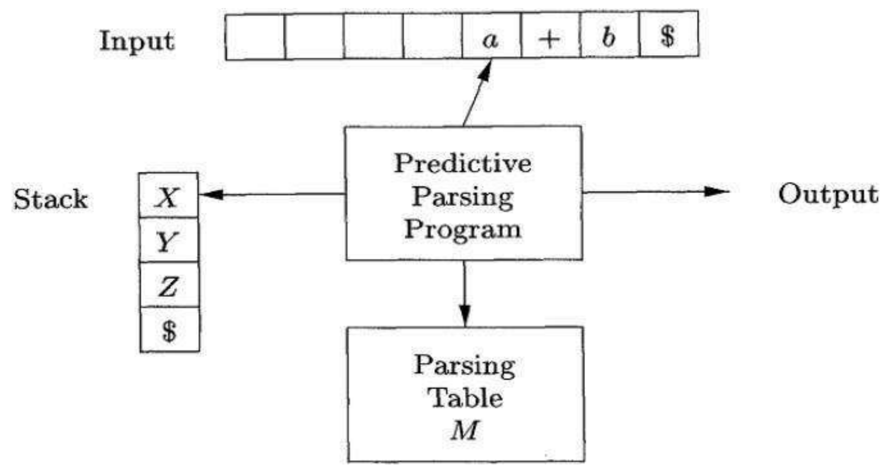


Figure2.2:Modelfortabledrivenparsing

TheStepsInvolvedInconstructinganLL(1) Parserare:

1. WritetheContextFreegrammarforgiveninputString
2. Checkfor Ambiguity.Ifambiguousremoveambiguityfromthegrammar
3. CheckforLeft Recursion.Removeleftrecursionifitexists.
4. CheckForLeftFactoring.Performleftfactoringifitcontainscommonprefixesin more than one alternates.
5. ComputeFIRSTandFOLLOWsets
6. ConstructLL(1) Table
7. UsingLL(1)AlgorithmgenerateParsetreeastheOutput

Context Free Grammar (CFG): CFG used to describe or denote the syntax of the programming language constructs.The CFG is denoted asG,and defined using a fourtuple notation.

Let GbeCFG,thenG iswrittenas, G=(V,T,P,S)

Where

∑V isa finite set ofNonterminal;Nonterminals are syntactic variablesthat denote setsof strings. The setsofstringsdenoted bynonterminalshelp definethe languagegenerated bythe grammar. Nonterminals impose a hierarchicalstructureonthe language that iskeytosyntaxanalysisandtranslation.

∑TisaFinitesetofTerminal;Terminalsarethebasicsymbolsfromwhichstringsareformed. The term "token name" is a synonym for '"terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer.

∑ S is the Starting Symbol of the grammar, one non terminal is distinguished as the start symbol, and the set ofstrings itdenotes isthelanguage generatedbythe grammar. P is finite set ofProductions;the productions ofa grammar specifythe manner inwhichthe terminalsandnonterminalscanbecombinedtoformstrings,eachproductionisinα->β form, where α is a single non terminal, β is (VUT)*.Each production consists of:

(a) A non terminal called the head or left side of the production;this production defines some of the strings denoted by the head.

(b) Thesymbol->.Some times:=hasbeenusedinplace ofthe arrow.

(c) Abodyorrightsideconsistingofzeroormoreterminalsandnon- terminals. The components ofthe bodydescribe one way in which strings of the nonterminalat the head can be constructed.

∑Conventionally,theproductionsforthestartsymbolarelistedfirst.

Example:ContextFreeGrammartoacceptArithmeticexpressions.

Theterminals are**+**,*****,**-**,**(**,**)**,**id**.

The**Nonterminalsymbols**are**expression**,**term**,**factor**andexpressionisthestartingsymbol.

expression → ***expression +term***
expression → ***expression –term***
expression → ***term***
term → ***term*factor***
term → ***term / factor***
term → ***factor***
factor → ***(expression)***
factor → ***id***

Figure2.3:GrammarforSimpleArithmeticExpressions

NotationalConventionsUsedInWritingCFGs:

To avoid always having to state that ―these are the terminals,""these are the non terminals,"andsoon,thefollowing notationalconventions forgrammarswillbeusedthroughout our discussions.

1. Thesesymbolsareterminals:

(a) Lowercaselettersearlyinthealphabet,suchasa,b,e.

(b) Operatorsymbolssuchas**+**,***,**andso on.

(c) Punctuationsymbolssuchasparentheses,comma,andsoon.

(d) Thedigits0,1...9.

(e) Boldfacestringssuchasidorif,eachofwhichrepresentsasingle terminal symbol.

2. Thesesymbolsarenonterminals:

(a) Uppercase lettersearlyinthealphabet,suchasA,B,*C*.

(b) TheletterS,which, whenitappears, isusuallythestartsymbol.

(c) Lowercase,italicnamessuchas*expr*or*stmt*.

(d) Whendiscussingprogrammingconstructs,uppercase lettersmaybeusedtorepresent Nonterminals for the constructs. For example, non terminal for expressions, terms, and factors are often represented by E, T, and F, respectively.

Usingtheseconventionsthegrammarforthearithmeticexpressionscanbewrittenas

E → E+T |E–T |T
T → T*F|T/F|F F

→ (E)| id

DERIVATIONS:

Theconstructionofaparsetreecanbemadeprecisebytakingaderivationalview,inwhich productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replacesa Nonterminal bythe bodyofone ofitsproductions. Thisderivationalview corresponds to the top-down construction of a parse tree as well as the bottom construction of theparse tree.

∑Derivationsareclassifiedinto**LetmostDerivation**and**RightMostDerivations.**

LeftMostDerivation(LMD):

Itistheprocessofconstructing theparsetreeoracceptingthegiveninput string,inwhich at everytime we need to rewrite the production rule it is done with left most nonterminalonly.

Ex:-IftheGrammarisE->**E+E| E*E|-E|(E)|id** andtheinputstringis**id +id* id**

The production**E->- E**signifies that ifE denotesanexpression, then - E must also denote an expression. The replacement of a single E by - E will be described bywriting

E=>-Ewhichisread as**"Ederives_E"**

Forageneraldefinitionofderivation,consideranonterminalAinthemiddleofasequence ofgrammar symbols, as inαAβ, where α and βarearbitrarystringsofgrammar symbol. Suppose A ->γ is a production. Then, we write αAβ => αγβ. The symbol => means "derives in one step". Often, we wish to say, "Derives in zero or more steps." For this purpose,we can use the symbol $\overset{*}{\Rightarrow}$,Ifwe wishto say, "Derives in$\overset{+}{\Rightarrow}$oneormore steps." We cnuse the symbol$\overset{+}{\Rightarrow}$.IfS $\overset{*}{\Rightarrow}$ a,whereSisthe start symbolofa grammar G, wesaythat αisa sententialformofG. The Leftmost Derivation for the given input string **id + id* id** is

E=><u>E</u>+E

=>id+E

=>id+ E*E

=>id+ id*E

=>id+ id*id

NOTE: Everytimewe needto startfromtherootproductiononly,theunder lineusingat Non terminal indicating that, it is the non terminal (left most one) we are choosing to rewrite the productions to accept the string.

RightMostDerivation(RMD):

Itistheprocessofconstructingtheparsetreeoracceptingthegiveninput string,every time we need to rewrite the production rule with Right most Nonterminal only.

TheRightmostderivationforthegiveninputstring**id+id*id**is

E=>E+ E
=>E+E *E
=>E+E*id
=>E+ id*id
=>id+ id*id

NOTE: Everytimeweneedtostart fromtherootproductiononly, theunder lineusingat Non terminalindicating that,it isthe non terminal(Right most one) weare choosing to rewrite the productions to accept the string.

WhatisaParseTree?

Aparsetreeisagraphicalrepresentationofaderivationthat filtersouttheorderinwhich productions are applied to replace non terminals.

- ∑Eachinteriornodeofa parsetreerepresentstheapplicationofaproduction.
- ∑Alltheinteriornodesare Nonterminalsand alltheleafnodesterminals.
- ∑Alltheleafnodesreadingfromtheleftto rightwillbetheoutputoftheparsetree.
- ∑If anodenislabeledXand haschildrenn1,n2,n3,…nkwithlabelsX1,X2,…Xk respectively, then there must be a production A->X1X2…Xk in the grammar.

Example1:-Parsetreefortheinputstring- **(id+id) using**theaboveContextfreeGrammaris

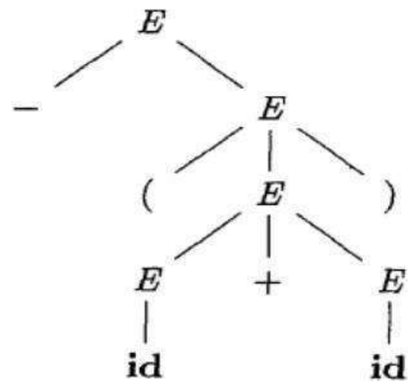


Figure2.4: ParseTreefortheinputstring**-(id+id)**

TheFollowingfigureshowsstepbystepconstructionofparsetreeusingCFG fortheparsetree for the input string - **(id + id).**

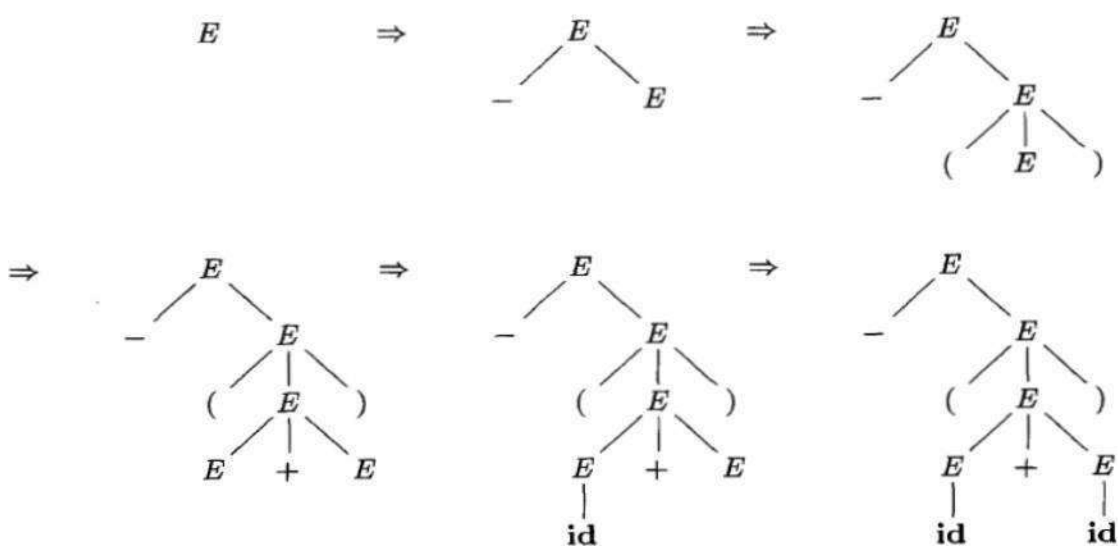


Figure2.5: SequenceoutputsoftheParseTreeconstructionprocessfortheinputstring**–(id+id)**

Example2:-Parsetreefortheinputstring**id+id*id**usingtheaboveContextfreeGrammaris

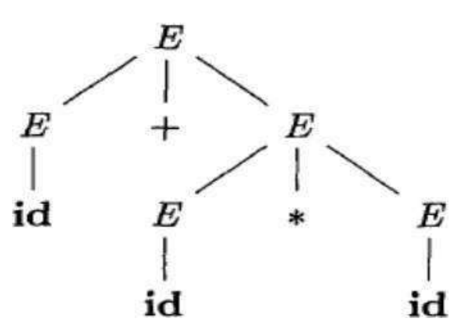


Figure2.6:Parsetreeforthe inputstringid+id*id

AMBIGUITYinCFGs:

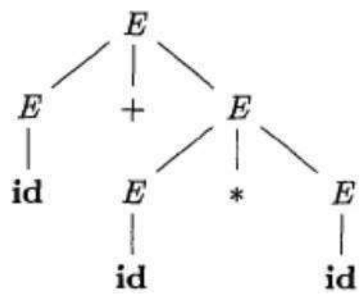
Definition: Agrammarthat producesmorethanoneparsetreeforsomesentence(input string) is said to be ambiguous.

Inotherwords,anambiguousgrammar isonethatproducesmorethanone leftmost derivation or more than one rightmost derivation for the same sentence.

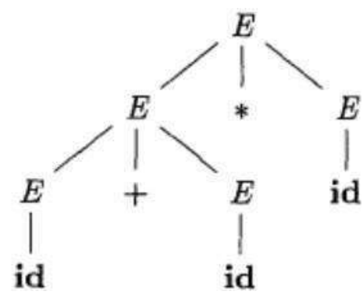
Or If the right hand production of the grammar is having two non terminals which are exactlysameasleft handsideproductionNonterminalthenit issaidtoanambiguousgrammar.

Example : Ifthe Grammaris **E-> E+E | E*E | -E|(E) | id** and the Input String is **id + id* id**

Twoparsetreesforgiveninputstring are



(a)



(b)

TwoLeftmostDerivationsforgiveninputStringare:

(a)	(b)
E=>$\underline{E}$+E	**E=>$\underline{E}$*E**
=>id+$\underline{E}$	**=>$\underline{E}$+E*E**
=>id+ $\underline{E}$*E	**=>id+ $\underline{E}$ *E**
=>id+id*$\underline{E}$	**=>id+ id*$\underline{E}$**
=>id+id*id	**=>id+ id*id**

TheaboveGrammar isgivingtwo parsetreesortwo derivations forthegiven input string so, it is an ambiguous Grammar

Note: LL (1) parser will not accept the ambiguous grammars or We cannot construct an LL(1) parser for the ambiguous grammars. Because such grammars may cause the Top Down parser to go into infinite loop or make it consume more time for parsing. If necessary we must remove all types of ambiguity from it and then construct.

ELIMINATING AMBIGUITY: SinceAmbiguous grammars may cause the top down Parser go into infinite loop, consume more time during parsing.

Therefore, **s**ometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. The general form of ambiguous productions that cause ambiguity in grammars is

$$A \rightarrow A\alpha|\beta$$

Thiscanbewrittenas(introduceonenewnonterminalinthe place ofsecondnonterminal)

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A'|\varepsilon$$

Example:Letthegrammar is **E⟶E+E|E*E|-E|(E) |id.**It isshownthatit isambiguousthat can be written as

E→ E+E
E→ E-E
E→ E*E
E→ -E
E→ (E)
E→ id

Intheabovegrammar the1st and 2nd productionsarehaving ambiguity. So,theycanbewritten as

E->E+E| E*Ethisproductionagaincanbe writtenas
E->E+E|β,whereβisE*E
Theaboveproductionissameasthegeneralform. so,thatcanbewrittenas E->E+T|T
T->β

ThevalueofβisE*Eso,abovegrammarcanbewrittenas

1) **E->E+T|T**
2) **T-> E*E** **Thefirstproductionisfreefromambiguity**andsubstituteE->Tin the 2nd production then it can be written as
T->T*T|-E|(E)|**id**thisproductionagaincanbewrittenas
T->T*T|βwhereβis-E|(E)|id, introducenewnonterminalintheRight handside production then it becomes
T->T*F|F
F->-E|(E)|id nowtheentiregrammarturnedintoitequivalentunambiguous,

TheUnambiguousgrammarequivalenttothe givenambiguousoneis

1) **E→E +T |T**
2) **T→T *F|F**
3) **F→-E |(E)|id**

LEFTRECURSION:

Another feature of the CFGs which is not desirable to be used in top down parsers is left recursion. A grammar is left recursive if it has a non terminal A such that there is a derivation A**=>**Aα for some string α in (TUV)*. LL(1) or Top Down Parsers can not handle the Left Recursive grammars, so we need to remove the left recursion from the grammars before being used in Top Down Parsing.

TheGeneralformofLeftRecursionis

$$A \rightarrow A\alpha|\beta$$

Theaboveleftrecursiveproductioncanbewrittenasthenonleftrecursiveequivalent:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A'|€$$

Example:-Isthe followinggrammar left recursive?Ifso,findanonleft recursivegrammar equivalent to it.

E→E +T |T

T →T * F | F

F →E | (E) | id

Yes,thegrammarisleftrecursiveduetothefirsttwoproductionswhicharesatisfyingthe

generalformofLeftrecursion,sotheycanberewrittenafterremovingleftrecursionfrom

E→E+T,and**T→T*F** is

E→TE′
E′→+TE′ |€
T→F T′
T′→*FT′|€
F →(E) | id

LEFTFACTORING:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictiveortop-downparsing.Agrammarinwhichmorethanoneproductionhascommonprefix is to be rewritten by factoring out the prefixes.

Forexample,inthefollowinggrammartherearenAproductionshavethecommonprefix**α,** whichshouldberemovedorfactoredoutwithoutchangingthelanguagedefinedfor A.

$$A \rightarrow \alpha A1|\alpha A2|\alpha A3|$$
$$\alpha A4 |\ldots | \alpha An$$

Wecanfactoroutthe**α**fromallnproductionsbyaddinga newAproduction**A→αA′** ,andrewritingtheA′productionsgrammar as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow A1|A2|A3|A4\ldots|An$$

FIRSTandFOLLOW:

Theconstructionofbothtop-downandbottom-upparsersisaidedbytwofunctions,FIRST and FOLLOW, associated with a grammar G. During top down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input (look a head) symbol.

ComputationofFIRST:

FIRSTfunctioncomputesthesetofterminalsymbolswithwhichtheright handsideofthe productions begin. To compute FIRST (A) **for all grammar symbols**, applythe following rules until no more terminals or € can be added to any FIRST set.

1. IfAisaterminal,thenFIRST{A}={A}.
2. IfAisaNonterminalandA->X1X2…Xi
   FIRST(A)=FIRST(X1) if X1is not null, if X1 is a non terminal and X1->€, add FIRST(X2)to FIRST(A), ifX2->€add FIRST(X3)to FIRST(A), …ifXi->€, i.e.,allXi'sfori=1..iarenull,add€FIRST(A).
3. IfA->€isaproduction,thenadd€toFIRST(A).

ComputationOfFOLLOW:

Follow(A) isnothing butthesetofterminalsymbolsofthegrammar thatareimmediately following the Nonterminal A. If**a** is to the immediate right ofnon terminal A, then Follow(A)= {**a**}.TocomputeFOLLOW(A) for **allnonterminals** A,applythe followingrulesuntilnomore symbols can be added to any FOLLOW set.

1. Place$inFOLLOW(S),where*S* isthestartsymbol,and$istheinput right end marker.
2. IfthereisaproductionA->αBβ,theneverything inFIRST(β)except €isin FOLLOW(B).
3. IfthereisaproductionA->αBoraproductionA->αBβwithFIRST(β) contains€, then FOLLOW (B) = FOLLOW (A).

Example:-ComputetheFIRSTandFOLLOWvaluesoftheexpressiongrammar

1. **E→TE′**
2. **E′→+TE′|€**
3. **T→FT′**
4. **T′→*FT′|€**
5. **F→(E)|id**

ComputingFIRSTValues:

FIRST(E)=FIRST(T)=FIRST(F)={(,id}

FIRST(E′)={+,€}

FIRST(T′)={*,€}

ComputingFOLLOWValues:

FOLLOW (E) = { \$,), } Becauseitisthestartsymbolofthegrammar.

FOLLOW (E′) = {FOLLOW (E)} satisfying the 3rd rule of FOLLOW()

= { \$,)}

FOLLOW(T)={FIRSTE′} ItisSatisfyingthe2ndrule.

U{FOLLOW(E′)}

= {+,FOLLOW(E′)}

= { +,\$,)}

FOLLOW(T′)={FOLLOW(T)} Satisfyingthe3rdRule

= {+, \$,)}

FOLLOW(F)={FIRST(T′)} ItisSatisfyingthe2ndrule.

U{FOLLOW(E′)}

={*,FOLLOW(T)}

={*,+,\$,)}

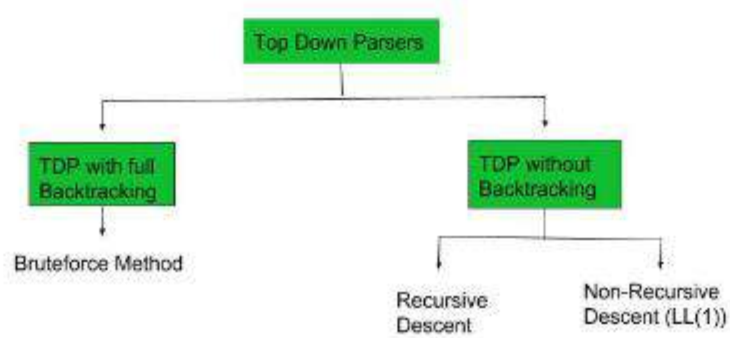
NONTERMINAL	FIRST	FOLLOW
E	{(,id}	{\$,)}
E′	{+,€}	{\$,)}
T	{(,id}	{ +,\$,)}
T′	{*,€}	{ +,\$,)}
F	{ (,id}	{*,+,\$,)}

Table2.1:FIRSTandFOLLOWvalues

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

1. Top-Down Parser with Backtracking
2. Top-Down Parsers without Backtracking

Top-Down Parsers without backtracking can further be divided into two parts:



ConstructingPredictiveOrLL(1)ParseTable:

Itistheprocessofplacing theallproductionsofthegrammar intheparsetablebased onthe FIRST and FOLLOW values of the Productions.

TherulestobefollowedtoConstructtheParsingTable(M)are:

1. ForEachproductionA->αofthegrammar,dothebellowsteps.
2. Foreachterminalsymbol_a‘inFIRST(α),addtheproductionA->αtoM[A,a].
3. i.If€ isinFIRST(α) addproductionA->αtoM[A,b],wherebisallterminalsin FOLLOW (A).
   ii.If€ is inFIRST(α) and$is inFOLLOW(A)thenaddproductionA->αto M [A, $].
4. Markotherentriesintheparsingtableaserror.

NON-TERMINALS	INPUTSYMBOLS					
	+	*****	**(**	**)**	**id**	**$**
E			E TE′		E id	
E′	E′ +TE′			E′ €		E′ €
T			T FT′		T FT′	
T′	T′ €	T′ *FT′		T′ €		T′ €
F			F (E)		F id	

Table2.2:LL(1)ParsingTablefortheExpressionsGrammar

Note:iftherearen o multipleentriesinthetable for singleaterminalthengrammar isaccepted by LL(1) Parser.

LL(1)ParsingAlgorithm:

The parseractsonbasis onthebasisoftwosymbols

i. A,thesymbolonthetopofthestack
ii. a,thecurrentinputsymbol

Therearethreeconditionsfor Aand_a‘,thatareusedfrotheparsing program.

1. IfA=a=$thenparsingisSuccessful.
2. IfA=a≠$thenparserpopsoffthestackandadvancesthecurrent input pointertothe next.
3. If A is a Nonterminalthe parser consults the entryM [A, a] inthe parsing table. If

M[A,a] isaProductionA->$X_1X_2..X_n$,thenthe programreplacestheAonthetopof the Stack by$X_1X_2..X_n$in such a way that X_1comes on thetop.

STRINGACCEPTANCEBYPARSER:

Iftheinput string fortheparser is**id+id*id,**thebelowtableshowshowtheparser accept the string with the help of Stack.

Stack	**Input**	**Action**	**Comments**
$E	id+id*id$	E TE`	EontopofthestackisreplacedbyTE`
$E`T	id+id*id$	T FT`	Tontopofthestackis replacedbyFT`
$E`T`F	id+id*id$	F id	Fontopofthestackis replacedbyid
$E`T`id	id+id*id$	popandremoveid	Condition2issatisfied
$E`T`	+id*id$	T` €	T`ontopofthestackis replacedby€
$E`	+id*id$	E` +TE`	E`ontopofthestackis replacedby+TE`
$E`T+	+id*id$	Popandremove+	Condition2issatisfied
$E`T	id*id$	T FT`	Tontopofthestackis replacedbyFT`
$E`T`F	id*id$	F id	Fontopofthestackis replacedbyid
$E`T`id	id*id$	popandremoveid	Condition2issatisfied

$E`T`	*id$	T` *FT`	T`ontopofthestackis replacedby*FT`
$E`T`F*	*id$	popandremove*	Condition2issatisfied
$E`T`F	id$	F id	Fontopofthestackis replacedbyid
$E`T`id	id$	Popandremoveid	Condition2issatisfied
$E`T`	$	T` €	T`ontopofthestackis replacedby€
$E`	$	E` €	E`ontopofthestackis replacedby€
$	$	Parsingissuccessful	Condition1satisfied

Table2.3: Sequenceofstepstakenbyparserinparsingtheinput**tokenstreamid+id*id**

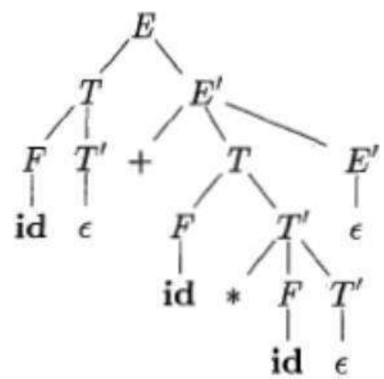


Figure2.7:Parsetreefortheinputid+id*id

ERRORHANDLING(RECOVERY)INPREDICTIVEPARSING:

Intabledrivenpredictiveparsing, it isclear astowhichterminaland Nonterminalsthe parser expects fromthe rest of input. An error can be detected in the following situations:

1. Whentheterminalontopofthe stackdoesnotmatchthe currentinputsymbol.
2. whenNonterminalA isontopofthe stack,aisthe current inputsymbol, and M[A, a] is empty or error

Theparser recoversfromtheerror andcontinues itsprocess. Thefollowingerrorrecovery schemes are use in predictive parsing:

PanicmodeErrorRecovery:

It is based on the idea that when an error is detected, the parser will skips the remaininginput untilasynchronizingtokenisencounteredinthe input.Someexamplesare listed below:

1. For a Non Terminal A, place all symbols in FOLLOW (A) are adde into the synchronizingsetofnonterminalA. ForExample, consider theassignmentstatement ―c=;‖ Here, the expression on the right hand side is missing. So the Follow of this is considered. It is ―;‖ and is taken as synchronizing token. On encountering it, parser emits an error message ―Missing Expression‖.
2. ForaNonTerminalA,placeallsymbolsinFIRST(A)areaddeintothesynchronizing set ofnon terminal A. For Example, consider the assignmentstatement ―22c=a+ b;‖Here,FIRST(expr) is22.It is ―;‖ and istakenas synchronizingtoken and then the reports the error as ―extraneous token‖.

PhraseLevelRecovery:

Itcanbeimplementedinthepredictiveparsingbyfillinguptheblankentries inthe predictiveparsingtablewithpointerstoerrorHandlingroutines.Theseroutinescan insert, modify or delete symbols in the input.

RECURSIVEDESCENTPARSING:

A recursive-descent parsing program consists of a set of recursive procedures, one for each non terminal. Each procedure is responsible for parsing the constructs defined by its non terminal, Executionbeginswiththeprocedureforthestartsymbol, whichhaltsandannouncessuccess if its procedure body scans the entire input string.

Ifthegivengrammaris

E→TE′
E′→+TE′|€
T→FT′
T′→*FT′|€
F→(E)|id

Reccursiveproceduresfortherecursivedescentparserforthegivengrammararegivenbelow.

```
procedureE()
{
        T();
         E′( );
}
procedureT()
{
        F();
        T′( );
}
ProcedureE′()
{
        ifinput=_+‘
        {
                advance();
                T ( );
                E′( );
                returntrue;
        }
        elseerror;
}
procedureT′()
{
        ifinput=_*‘
        {
                advance();
                F ( );
```

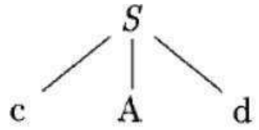
```
        T′( );
    returntrue;
    }
    elsereturnerror;
}
procedureF()
{
    ifinput=‗(‗
    {
        advance();
        E ( );
        ifinput=‗)‘
        advance( );
        return true;
    }
    elseifinput=―id‖
    {

        advance();
        returntrue;
    }
    elsereturnerror;
}
advance()
{
    input=next token;
}
```

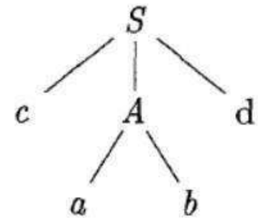
BACK TRACKING: This parsing method uses the technique called Brute Force method during the parsetree construction process. This allowsthe processto go back (back track)and redo the steps byundoing the work done so far in the point of processing.

Bruteforcemethod: It isaTopdownParsing technique,occurswhenthereismore than one alternative in the productions to be tried while parsing the input string. It selects alternativesintheordertheyappearandwhenit realizesthat somethinggonewrongittrieswith next alternative.
Forexample,considerthegrammarbellow.

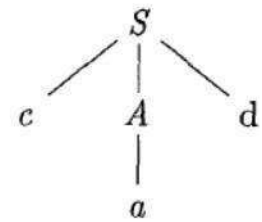
S→cAd
A→ab|a

To generatethe input string ―cad‖, initiallythe first parse tree given below is generated. Asthestringgeneratedisnot―cad‖,inputpointerisbacktrackedtoposition―A‖,toexaminethe nextalternate of ―A‖. Now a match to the input string occurs as shown in the 2nd parse trees given below.





(1)



(2)

IMPORTANTANDEXPECTEDQUESTIONS

1. ExplainthecomponentsofworkingofaPredictiveParserwithanexample?
2. WhatdotheFIRSTandFOLLOWvaluesrepresent?Givethealgorithmforcomputing FIRST n FOLLOW of grammar symbols with an example?
3. ConstructtheLL(1)Parsingtableforthefollowinggrammar? E→ E+T|T

   T→T*F

   F→(E)|id
4. FortheabovegrammarconstructandexplaintheRecursiveDescentParser?
5. WhathappensifmultipleentriesoccurringinyourLL(1)Parsingtable?Justifyyour answer? How does the Parser

ASSIGNMENTQUESTIONS

1. EliminatetheLeftrecursionfromthebelow grammar?

   A->Aab|AcB|b
   B->Ba|d
2. Explaintheprocedureto removetheambiguityfromthegivengrammar with yourown example?
3. Writethegrammarfortheif-elsestatement intheCprogrammingandcheckfortheleft factoring?
4. WillthePredictiveparseraccepttheambiguousGrammarjustifyyouranswer?
5. IsthegrammarG={S->L=R,S->R,R->L,L->*R|**id**}anLL(1)grammar?
6. **Construct an LR parsing table for the given context-free grammar –**

   S–>AA
   A–>aA|b

BOTTOM-UPPARSING

Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom nodes) and working up towards the root (the top node). It involves ―reducing an input string ‗w' to the Start Symbol of the grammar. in each reduction step, aperticular substring matching the right side ofthe production is replaced by symbolonthe left of that production and it is the Right most derivation. For example consider the following Grammar:

E→E+T|T

T→T*F

F→(E)|id

Bottomupparsing oftheinputstring**"id *id"**isas follows:

INPUTSTRING	**SUB STRING**	**REDUCINGPRODUCTION**
id*id	Id	F->id
F*id	T	F->T
T*id	Id	F->id
T*F	*	T->T*F
T	T*F	E->T
E		Startsymbol.Hence,theinput String is accepted

ParseTreerepresentationisasfollows:

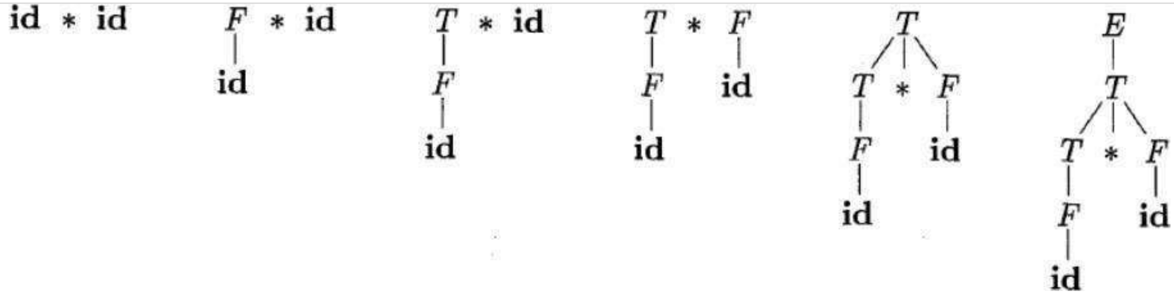


Figure3.1:ABottom-upParsetreeforthe inputString"id*id"

Bottomupparsing isclassified into 1.Shift-ReduceParsing, 2. OperatorPrecedenceparsing , and 3. [Table Driven] L R Parsing

i. SLR(1)
ii. CALR(1)
iii. LALR(1)

SHIFT-REDUCEPARSING:

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbolsandaninput bufferholdstherestofthestringto beparsed,Weuse$to markthebottom ofthestackandalsotheright endofthe input. And it makesuseoftheprocessofshift andreduce actionstoaccepttheinput string. Here,theparsetreeisConstructedbottomupfromthe leafnodes towards the root node.

Whenweareparsingthegiveninput string, ifthe matchoccurstheparsertakesthe reduce actionotherwise it willgo for shift action. And it can accept ambiguous grammarsalso.

Forexample,considerthebelowgrammartoacceptthe inputstring—id*id—,usingS-Rparser

E→E+T|T
T→T*F|F
F→(E)|id

ActionsoftheShift-reduceparserusing Stackimplementation

STACK	INPUT	ACTION
$	Id*id$	Shift
$id	*id$	ReducewithF→d
$F	*id$	ReducewithT→F
$T	*id$	Shift
$T*	id$	Shift
$T*id	$	ReducewithF→id
$T*F	$	ReducewithT→T*F
$T	$	ReducewithE→T
$E	$	Accept

Considerthefollowinggrammar:

S➝aAcBe
A➝Ab|b
B➝d

Lettheinputstringis―abbcde‖.Theseriesofshiftandreductionstothestartsymbolareas follows.

abbcde ⇨ aAbcde ⇨ aAcde ⇨ aAcBe ⇨ S

Note:intheaboveexamplethearetwoactionspossible inthesecondStep,theseareas follows :

1. Shiftactiongoingto3rdStep
2. Reduceaction,thatisA->b

Iftheparser istakingthe1stactionthenit cansuccessfullyacceptsthegiveninput string, ifitisgoing for second actionthen it can't accept given input string. This iscalled shift reduce conflict. Where, S-Rparser is notabletakeproperdecision, so it notrecommended for parsing.

OPERATOR PRECEDENCE PARSING:

Operatorprecedencegrammar iskindsofshift reduceparsing methodthatcanbeappliedtoa small class ofoperator grammars. And it can process ambiguous grammars also.

∑Anoperatorgrammarhastwo importantcharacteristics:

1. Therearenoۀproductions.
2. Noproductionwouldhavetwoadjacentnonterminals.

∑Theoperatorgrammartoacceptexpressionsisgivebelow:

E➡E+E/E➝E-E /E➝E*E/E➝E/E/E➝E^E/E➝-E/E➝(E)/E➝ id

TwomainChallengesintheoperatorprecedenceparsingare:

1. IdentificationofCorrecthandlesinthereductionstep,suchthatthegiveninput shouldbe reduced to starting symbol of the grammar.
2. Identificationofwhichproductionto useforreducing inthereductionsteps, suchthat we should correctlyreduce the given input to the starting symbol of the grammar.

Operatorprecedenceparserconsistsof:

1. Aninputbufferthatcontainsstringto beparsedfollowed bya$,asymbolusedto indicate the ending of input.
2. Astackcontaininga sequenceofgrammarsymbols witha $atthebottomofthestack.
3. Anoperator precedence relation table O, containing the precedence ralations between the pair ofterminal. There are three kinds of precedence relations will exist between the pair of terminal pair _a' and _b' as follows:
4. Therelationa<•bimpliesthatheterminal_a'haslowerprecedencethanterminal_b'.
5. Therelationa•>bimpliesthatheterminal_a'hashigherprecedencethanterminal_b'.
6. Therelationa=•bimpliesthatheterminal_a'haslowerprecedencethanterminal_b'.

7. An operator precedence parsing program takes an input string and determines whether it conforms to the grammar specifications. It uses an operator precedence parse table and stack to arrive at the decision.

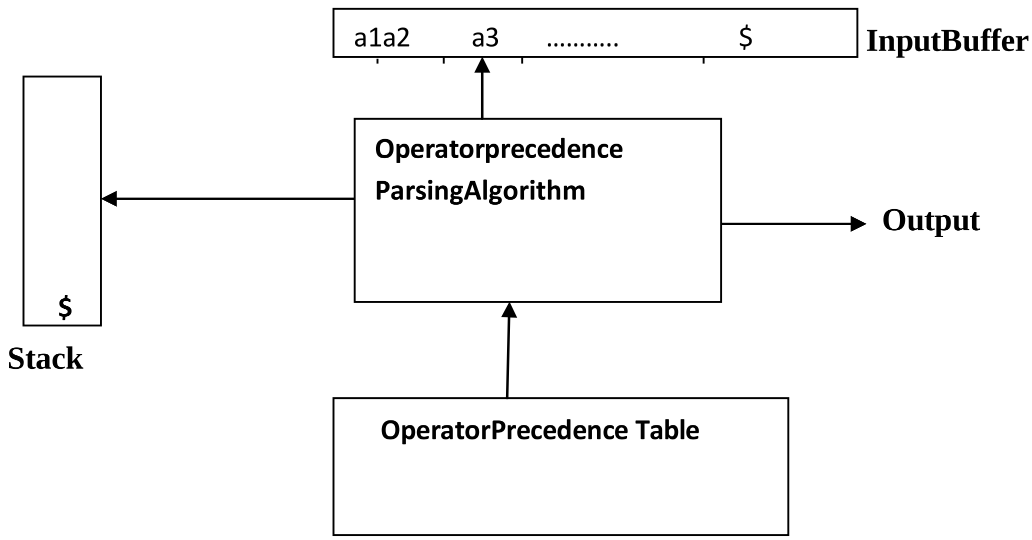


Figure3.2:Componentsofoperatorprecedenceparser

Example,Ifthegrammaris

E→E+E
E→E-E
E→E*E
E→E/E
E→E^E
E→-E
E→(E)
E→id,Constructoperatorprecedencetableandacceptinputstring**“id+id*id”**

Theprecedencerelationsbetweentheoperatorsare
(id)>(^)>(*/)>(+-)>$,,,^‟operatorisRight Associativeand reaming alloperators are Left Associative

	+	-	*	/	^	id	(	)	$
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
^	•>	•>	•>	•>	<•	<•	<•	•>	•>
Id	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
(	<•	<•	<•	<•	<•	<•	<•	=	**Err**
)	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	**Err**	**Err**

Theintentionoftheprecedencerelationsistodelimit thehandleofthegiveninput Stringwith<• marking the left end ofthe Handle and •> marking the right end ofthe handle.

ParsingAction:

Tolocatethehandlefollowingstepsarefollowed:

1. Add$ symbolat the bothendsofthegiveninputstring.
2. Scantheinputstringfromlefttorightuntiltherightmost•>isencountered.
3. Scantowardsleftoveralltheequalprecedence‘suntilthe first <•precedenceis encountered.
4. Everything between<•and•>isahandle.
5. $onSmeansparsingissuccess.

Example,Explaintheparsing ActionsoftheOPParserforthe input string is**“id*id”**andthe grammar is:

E→E+E
E→E*E
E→id

1. $<•id•>*<•id•>$

The first handle is ‗id‘ and match for the ‗id ‗in the grammar is E→ id. So, id is replaced with the Non terminalE. the given input string can be written as

2. $<•E•>*<•id•>$

Theparserwillnot considertheNonterminalasaninput. So,theyarenot considered in the input string. So , the string becomes

3. $<•*<•id•>$

Thenexthandleis‗id‘andmatchforthe‗id‗inthegrammarisE→id. So, id is replaced with the NonterminalE. the given input string can be written as

4. **$<•*<•E•>$**

Theparserwillnot considertheNonterminalasaninput. So,theyarenot considered in the input string. So, the string becomes

5. $<•*•>$

The next handle is ‗*‘ and match for the ‗ ‗in the grammar is E→ E*E. So, id is replaced with the Non terminal E. the given input string can be written as

6. **$E $**

Theparserwillnot considertheNonterminalasaninput. So,theyarenot considered in the input string. So, the string becomes

7. **$$**

Onmeansparsing successful.

OperatorParsingAlgorithm:

TheoperatorprecedenceParser parsingprogramdeterminestheactionoftheparser depending on

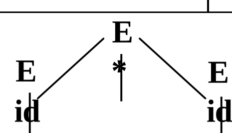
1. _a‘istopmostsymbolonthe Stack
2. _b‘isthecurrentinputsymbol

Thereare3conditionsfor _a‘and_b‘thatareimportant fortheparsingprogram

1. a=b=$,theparsingissuccessful
2. a<•bor a=b,theparser shiftsthe input symbolontothestackand advancesthe input pointer to the next input symbol.
3. a •>b, parser performs the reduce action. The parser popsout elementsone by one fromthe stackuntilwe find the current topofthe stack element has lower precedence than the most recently popped out terminal.

Example,thesequenceofactionstakenbytheparserusingthestackfortheinputstring—id*id —andcorrespondingParseTreeareasunder.

STACK	**INPUT**	**OPERATIONS**
$	id*id$	$<•id,shift_id‘ intostack
$id	*id$	id•>*,reduce_id‘using E->id
$E	*id$	$<•*,shift_*‘ intostack
$E*	id$	*<•id,shift_id‘intoStack
$E*id	$	id•>$,reduce_id‘using E->id
$E*E	$	*•>$,reduce_*‘usingE->E*E
$E	$	$=$=$,soparsingissuccessful



AdvantagesandDisadvantagesofOperatorPrecedenceParsing:

Thefollowing aretheadvantagesofoperatorprecedenceparsing

1. Itissimpleandeasytoimplementparsingtechnique.
2. Theoperatorprecedenceparsercanbeconstructedbyhandafterunderstandingthe grammar. It is simple to debug.

Thefollowingarethedisadvantagesofoperatorprecedenceparsing:

1. Itisdifficulttohandletheoperatorlike_-_whichcanbeeitherunaryorbinaryandhence different precedence‘s and associativities.
2. Itcanparseonlyasmallclass ofgrammar.

3. Newadditionordeletionoftherulesrequirestheparsertoberewritten.
4. Toomanyerrorentriesintheparsingtables.

LRParsing:

Most prevalent type of bottom up parsing is LR (k) parsing. Where, L is left to right scan of the giveninput string,RisRight Mostderivationinreverseand Kisno ofinputsymbolsastheLook ahead.

∑Itisthemostgeneralnonbacktrackingshiftreduceparsingmethod

∑Theclassofgrammarsthat canbeparsed usingtheLRmethodsisapropersupersetof the class of grammars that can be parsed with predictive parsers.

∑AnLRparser candetect asyntacticerrorassoonas it ispossibletodo so,onaleft to right scan of the input.

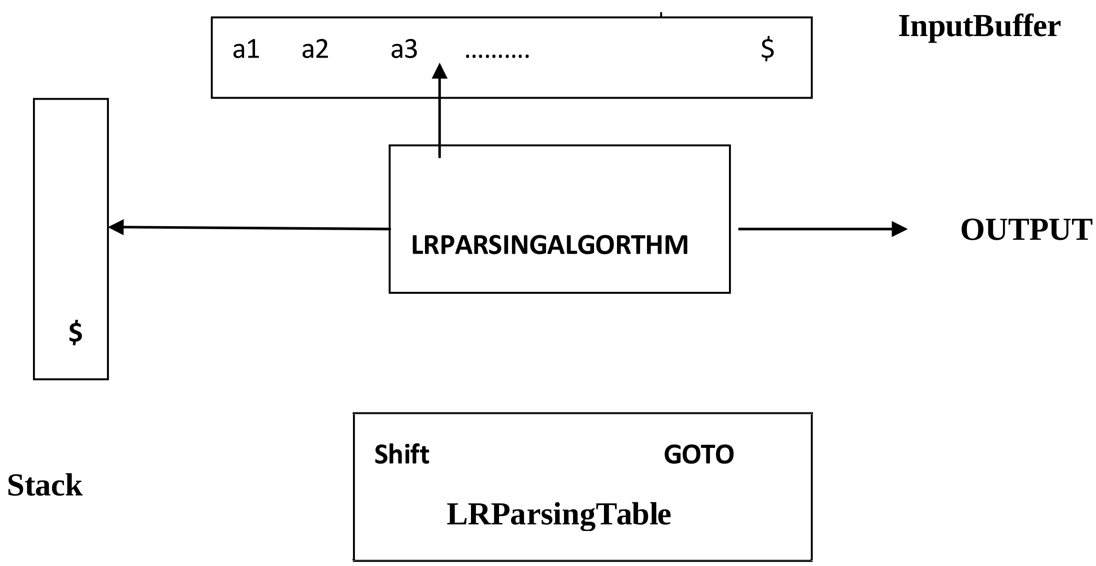


Figure3.3:ComponentsofLRParsing

LRParserConsistsof

∑Aninput bufferthat containsthestringtobeparsedfollowed bya$Symbol,usedto indicate end of input.

∑Astackcontaining asequenceofgrammar symbolswitha$atthebottomofthestack, which initially contains the Initial state of the parsing table on top of$.

∑Aparsingtable(M), it isatwodimensionalarrayM[state,terminalorNonterminal]and it contains two parts

1. **ACTIONPart**
   The ACTION part ofthe table is a two dimensionalarrayindexed bystateand the input symbol, i.e. **ACTION**[state][input], An action table entry can have one of following four kinds of values in it. They are:
   1. ShiftX,whereXisaStatenumber.
   2. ReduceX,whereXisaProductionnumber.
   3. Accept,signifyingthecompletionofasuccessfulparse.
   4. Errorentry.
2. **GOTOPart**
   TheGOTOpartofthetable isatwodimensionalarrayindexed bystateandaNon terminal, i.e. GOTO[state][NonTerminal]. A GO TO entry has astate number in the table.

∑ A parsing Algorithmuses the current State X, the next input symbol‗a' to consult the entryat action[X][a]. it makes one ofthe four following actions as given below:

1. If the action[X][a]=shift Y, the parser executes a shift of Y on to the top of the stack and advances the input pointer.
2. Ifthe action[X][a]= reduce Y (Y is the production number reduced in the State X), if the production is Y->β, then the parser pops 2*β symbols from the stack and push Y on to the Stack.
3. If the action[X][a]= accept, then the parsing is successful and the input string is accepted.
4. If the action[X][a]= error, then the parser has discovered an error and calls the error routine.

Theparsingisclassified into

1. LR(0)
2. SimpleLR(1)
3. CanonicalLR(1)
4. Lookahead LR(1)

LR(1)Parsing: VariousstepsinvolvedintheLR(1)Parsing:

1. WritetheContextfreeGrammarforthegiveninputstring
2. CheckfortheAmbiguity
3. AddAugmentproduction
4. Create CanonicalcollectionofLR(0)items
5. DrawDFA
6. ConstructtheLR(0)Parsingtable
7. BasedontheinformationfromtheTable,withhelpofStackandParsingalgorithm generate the output.

AugmentGrammar

The Augment Grammar G`, is G with a new starting symbol S` an additional production S`→S.thishelpstheparserto identifywhentostoptheparsing andannouncetheacceptanceofthe input.Theinput string isaccepted ifandonlyifthe parser isabouttoreducebyS`→S.Forexample let us consider the Grammar below:

E→ E+T|T
T→ T*F
F→ (E)|id

theAugmentgrammarG`isRepresented by

E`→ E
E→ E+T|T
T→ T*F
F→ (E)|id

NOTE: Augment Grammar issimplyaddingoneextraproductionbypreservingtheactual meaning of the given Grammar G.

CanonicalcollectionofLR(0)items

LR(0) items

AnLR (0) itemofa Grammar is a production G with dot at some position on the right sideoftheproduction. Anitemindicateshow muchofthe input has beenscanneduptoagiven point in the process ofparsing. For example, ifthe Production is X→YZ then, The LR (0) items are:

1. X→•AB,indicatesthattheparser expectsastring derivablefromAB.
2. X→A•B, indicatesthattheparserhasscannedthestringderivablefromtheAand expecting the string from Y.
3. X→AB•, indicatesthatheparserhasscannedthestringderivablefromAB. If the grammar is X→€ the, the LR (0) item is

X→•, indicating thattheproduction isreducedone.

CanonicalcollectionofLR(0)Items:
ThisistheprocessofgroupingtheLR(0)itemstogether basedontheclosureandGoto operations

Closureoperation

IfIisaninitialState,thentheClosure (I)isconstructedasfollows:

1. Initially,addAugment Productiontothestateandcheck forthe•symbolintheRight hand side production, if the • is followed by a Non terminal then Add Productions which are Stating with that Non Terminal in the State I.

2. If a production X→α•Aβ is in I, then add Production which are starting with X in the StateI.Rule2 isapplieduntilno moreproductionsaddedtotheStateI(meaningthat the•isfollowedbyaTerminalsymbol).

Example:

0.E` → E
1. E→ E+T
2. T→ F
3. T→ T*F
4. F→ (E)
5. F→ id

LR(0)itemsfortheGrammaris

E`→ • E
E →·E+T
T→ •F
T→ •T*F
F→ • (E)
F→ • id

Closure (I_0)State

Add**E`→•E**inI_0State

Since,the‗•‘symbolintheRight handsideproductionisfollowed byANon terminal E. So, add productions starting with E in to Io state. So, the state becomes

E ` •E →

0. E→•E+T

1. T→•F

The1st and2nd productionsaresatisfiesthe2nd rule.So,addproductions which are starting with E and T in I_0

Note:onceproductionsareadded inthestatethesameproductionshould not added for the 2nd time in the same state. So, the state becomes

0.E` → •E
1. E→ ·E+T
2.T → •F
3.T → ·T*F
4.F → ·(E)
5.F → • id

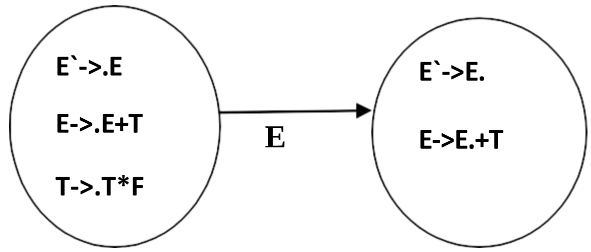
GO TOOperation

Go to (I_0, X), where I_0 is set of items and X is the grammar Symbolonwhichwe aremovingthe„•" symbol. It islike findingthe next stateoftheNFAfor agiveStateI$_0$andthe input symbol is X. For example, if the production is **E•E+T →**

Goto **(I_0,E)**is**E`→•E,E→E•+T**

Note: OncewecompletetheGotooperation,weneedtocomputeclosureoperationforthe output production

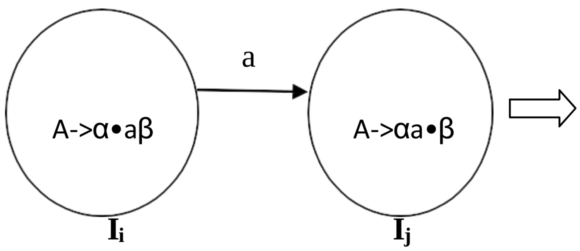
Goto**(I_0, E)**is**E→E•+T,E`→E.=Closure({E`→E•,E→E•+T})**



ConstructionofLR(0)parsingTable:

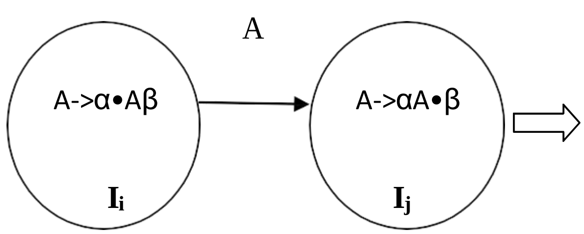
Oncewe haveCreatedthecanonicalcollectionofLR(0)items,needtofollowthesteps mentioned below:

Ifthereisatransactionfromonestate(**I_i**)to another state(**I_j**)onaterminalvaluethen, we should write the shift entry in the action part as shown below:



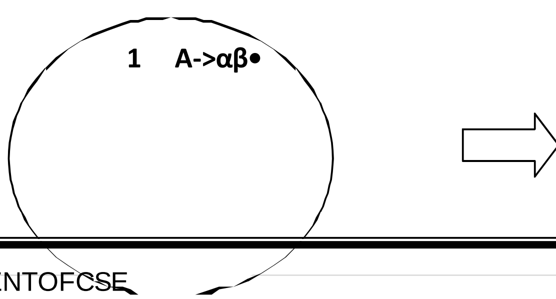
States	ACTION		GOTO
	a	$	A
I_i	**S_j**		
I_j			

Ifthereisa transactionfromone state(**I_i**)toanotherstate(**I_j**)onaNonterminalvalue then, weshouldwritethesubscript valueof**I_i**intheGOTOpart asshownbelow:part asshown below:



States	ACTION		GOTO
	a	$	A
I_i			**j**
I_j			

Ifthere is one state (**I_i**), where there is one production which has no transitions. Then, the productionissaidtobeareducedproduction. Theseproductionsshouldhavereducedentryinthe Actionpartalongwiththeirproductionnumbers.IftheAugmentproductionisreducingthen,write accept in the Action part.



States	ACTION		GOTO
	a	$	A
I_i	**r_1**	**r_1**	

I_i

I_i

ForExample,ConstructtheLR(0)parsing TableforthegivenGrammar(G)

S→aB

B→bB|b

Sol:1.AddAugmentProductionandinsert„•"symbolatthefirstpositionforevery production in G

0. **S′→•S**
1. **S→•aB**
2. **B→•bB**
3. **B →•b**

<u>I_0State:</u>

1. AddAugmentproductiontotheI_0StateandComputethe Closure

I_0=Closure(S′→•S)

Since_•'isfollowed bytheNonterminal,addallproductionsstartingwithSintoI_0State.So, the I_0State becomes

I_0= **S′→•S**

S→•aB Here,intheSproduction_.'Symbolisfollowedbyaterminalvalueso close the state.

I_1=Go to(I_0,S)

S`→S•

Closure(S`→S•)=S′→S• Here,TheProductionisreducedsoclosetheState.

I_1=S′→S•

I_2=Goto(I_0,a)=closure(S→a•B)

Here,the_•'symbolis followed byTheNonterminalB. So,addtheproductionswhichare Starting B.

I_2= **B→•bB**

B→•b Here,the_•'symbolintheBproductionis followedbytheterminalvalue. So, Close the State.

I_2= **S → a•B**

B→•bB

B→•b

I_3= Go to (I_2,B) = Closure(S →aB•)= S→aB•

I_4= Go to (I_2, b) =closure ({B →b•B, B→b•})

AddproductionsstartingwithBinI_4.

B→ •bB
B → •b TheDotSymbolis followedbytheterminalvalue.So,closetheState.

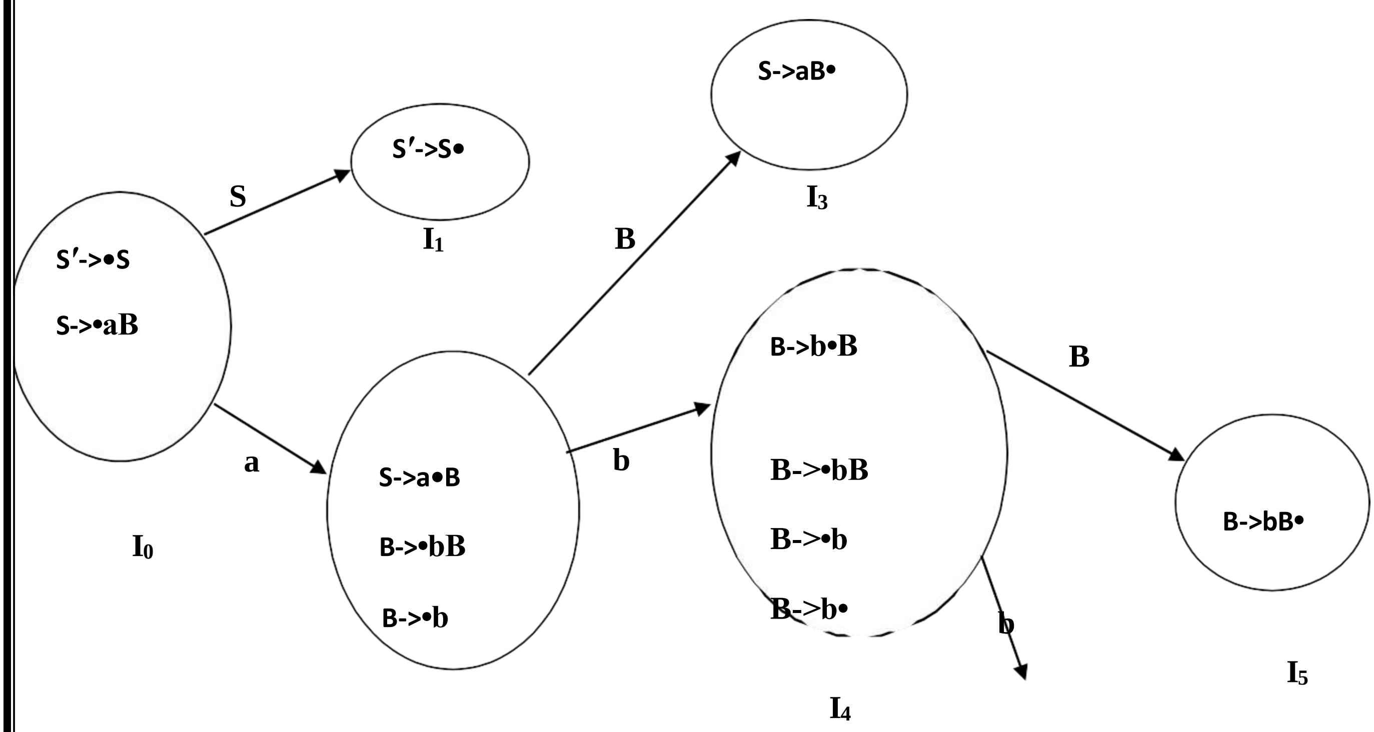
I_4= **B → b•B**
B→ •bB
B → •b
B → b•

I_5=Goto(I_2,b)=Closure(B→b•)=B→b•

I_6=Go to(I_4,B) =Closure(B →bB•)=B →bB• I_7 =

Go to (I_4 , b) = I_4

<u>DrawingFiniteStatediagramDFA:</u>Following DFAgivesthestatetransitionsoftheparser and is useful in constructing the LR parsing table.



$\mathbf{I_2}$ $\mathbf{I_4}$

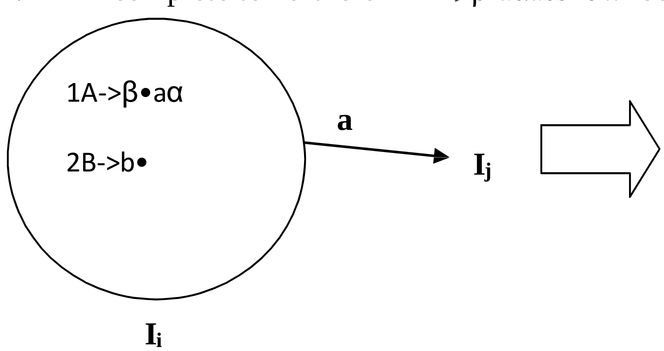
LRParsingTable:

States	ACTION			GOTO	
	a	B	$	S	B
I_0	S_2			1	
I_1			ACC		
I_2		S_4			3
I_3	R_1	R_1	R_1		
I_4	R_3	S_4/R_3	R_3		5
I_5	R_2	R_2	R_2		

Note:iftherearemultipleentriesintheLR(1)parsingtable,thenitwillnotacceptedbytheLR(1) parser. In the above table I_3 row is giving two entries for the single terminal value ‗b‘ and it is called as Shift-Reduce conflict.

Shift-ReduceConflictinLR(0)Parsing: Shift ReduceConflict intheLR(0)parsing occurs when a state has

1. AReduceditemoftheformA→α•and
2. AnincompleteitemoftheformA→β•aαasshownbelow:

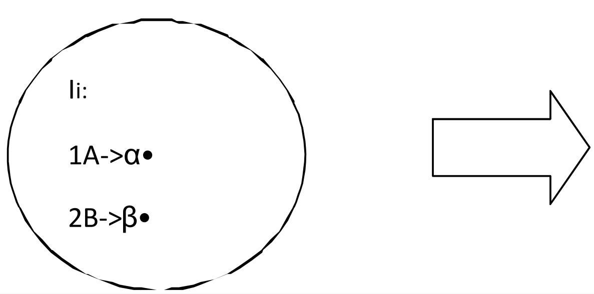


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2	r_2		
I_j				

Reduce-ReduceConflictinLR(0)Parsing:

Reduce-ReduceConflict intheLR(1)parsingoccurswhenastatehastwoormore reduced items of the form

1. **A→α•**
2. **B→β•**asshownbelow:



States	Action		GOTO	
	A	$	A	B
I_i	r_1/r2	r_1/r_2		

SLRPARSERCONSTRUCTION:WhatisSLR(1)Parsing

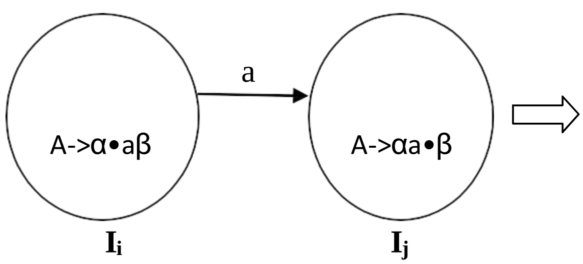
VariousstepsinvolvedintheSLR(1)Parsingare:

1. WritetheContextfreeGrammarforthegiveninputstring
2. CheckfortheAmbiguity
3. AddAugment production
4. Create CanonicalcollectionofLR(0)items
5. DrawDFA
6. Construct theSLR(1)Parsing table
7. BasedontheinformationfromtheTable,withhelpofStackandParsingalgorithm generate the output.

SLR(1)ParsingTableConstruction

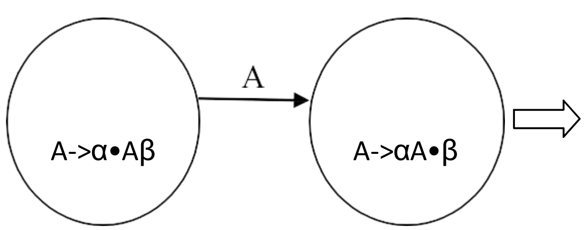
Oncewe haveCreatedthecanonicalcollectionofLR(0)items,needto followthesteps mentioned below:

Ifthereisatransactionfromonestate(**I_i**)to another state(**I_j**)onaterminalvaluethen, we should write the shift entry in the action part as shown below:



States	ACTION		GOTO
	a	$	A
I_i	**S_j**		
I_j			

Ifthere is a transaction fromone state (**I_i**) to another state (**I_j**) on a Non terminal value then, weshouldwritethesubscript valueof**I_i**intheGOTOpart asshownbelow:part asshown below:



States	ACTION		GOTO
	a	$	A
I_i			**j**
I_j			

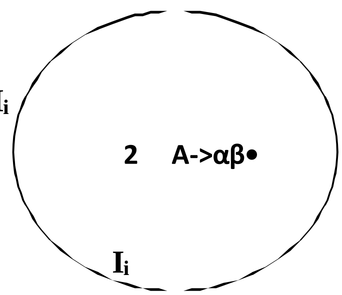
I_i I_j

1 If there is one state (I_i), where there is one production (**A->αβ•**) which has no transitions to the next State. Then, the production is said to be a reduced production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers. If the Augment production is reducing then write accept.

1 S->•aAb

2 A->αβ•
Follow(S)={$}
Follow(A)=(b}

I_i



States	ACTION			GOTO	
	a	b	$	S	A
I_i		**r_2**			

SLR(1) table for the Grammar

S→aB
B→bB|b

Follow(S)={$},Follow(B)={$}

States	ACTION			GOTO	
	A	**b**	**$**	**S**	**B**
I_0	S_2			1	
I_1			ACCEPT		
I_2		S_4			3
I_3			R_1		
I_4		S_4	R_3		5
I_5			R_2		

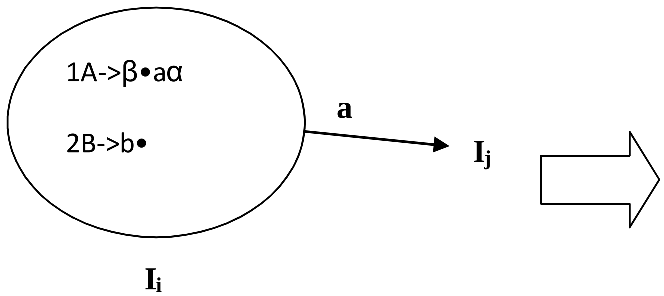
Note: When Multiple Entries occurs in the SLR table. Then, the grammar is not accepted by SLR(1) Parser.

Conflicts in the SLR(1) Parsing :

When multiple entries occur in the table. Then, the situation is said to be a Conflict.

Shift-ReduceConflictinSLR(1)Parsing: Shift ReduceConflict intheLR(1)parsingoccurs when a state has

1. AReduceditemoftheformA→α•andFollow(A)includestheterminalvalue ‗a‘.
2. AnincompleteitemoftheformA→β•aαasshownbelow:



States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce-ReduceConflictinSLR(1)Parsing

Reduce-ReduceConflict intheLR(1) parsingoccurswhenastatehastwoormore reduced items of the form

1. **A→α•**
2. **B→β•andFollow (A) ∩Follow(B)≠null**asshownbelow:

IfTheGrammaris

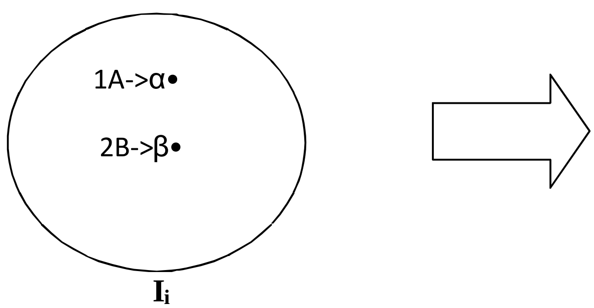
S->αAaBa

A->α

B->β

Follow(S)={$}

Follow(A)={a}andFollow(B)={a}



States	Action		GOTO	
	a	$	A	B
I_i	r_1/r2			

CanonicalLR(1)Parsing: Variousstepsinvolved intheCLR(1)Parsing:

1. WritetheContextfreeGrammarforthegiveninputstring
2. ChecrfortheAmbiguity
3. AddAugmentproduction

4. Create CanonicalcollectionofLR(1)items
5. DrawDFA
6. ConstructtheCLR(1)Parsing table
7. BasedontheinformationfromtheTable,withhelpofStackandParsing algorithm generate the output.

LR(1)items:

TheLR(1) itemisdefined by**production**,**positionofdata**anda**terminalsymbol**.The terminal is called as ***Look ahead symbol***.

GeneralformofLR(1)itemis

S->α•Aβ, $
A->•γ,FIRST(β,$)

Rulestocreatecanonicalcollection:

1. EveryelementofIisaddedtoclosureofI
2. If an LR (1) item [**X-> A•BC, a]** exists in I, and there exists a production B->b_1b_2….., then additem**[B->• b_1b_2, z]** where z is a terminal in **FIRST(Ca),**if itis not already in Closure(I).keep applying this rule until there are no more elements adde.

Forexample,ifthegrammaris

S->CC
C->cC
C->d

TheCanonicalcollectionofLR(1)itemscanbecreatedasfollows:

0. *S′*->•*S*(AugmentProduction)
1. *S*->•CC
2. C->•cC
3. C->•d

I_0State: AddAugmentproductionandcomputetheClosure, thelookaheadsymbolfor theAugment Production is $.

S′->•S,$=Closure(*S′*->•S,$)

ThedotsymbolisfollowedbyaNonterminalS.So,addproductionsstarting withSinI_0 State.

**S->•CC,FIRST($),**using2ndrule

S->•CC, $

ThedotsymbolisfollowedbyaNonterminalC.So,add productionsstartingwithCinI_0 State.

C->•cC,FIRST(C,$)
C->•d, FIRST(C, $)

FIRST(C) ={c,d}so,theitemsare

C->•cC,c/d
C->•d, c/d

Thedotsymbolisfollowedbyaterminal value.So,closetheI_0State.So,theproductionsinthe **I_0**are

S′->•S , $
S->•CC,$
C->•cC,c/d
C->•d,c/d

I_1=Goto(I_0,S)=S′->S•,$

I_2=Goto(I_0,C)=Closure(S->C•C,$)

S->**C->•cC ,$**
**C->•d,$**So,the$I_2$Stateis

S->C•C,$
C->•cC,$
C->•d,$

I_3=Goto(I_0,c)=Closure(C->c•C,c/d)
C->•cC,c/d
C->•d,c/dSo,theI_3Stateis

C->c•C,c/d
C->•cC,c/d
C->•d , c/d

I_4=Goto(I_0,d)=Colsure(C->d•,c/d)=C->d•,c/d

I_5=Goto(I_2,C)=closure(S->CC•,$)=S->CC•,$ I_6=

Goto (I_2, c)= closure(C->c•C , $)=
C->•cC,$
**C->•d,$**S0,the$I_6$Stateis

C->c•C,$
C->•cC,$
C->•d,$

I_7 =Goto(I_2, d)=Closure(C->d•,$)=C->d•, $

Goto(I_3, c)= closure(C->•cC, c/d)= I_3.

I_8=Goto(I_3, C)=Closure(C->cC•,c/d)=C->cC•,c/d Go

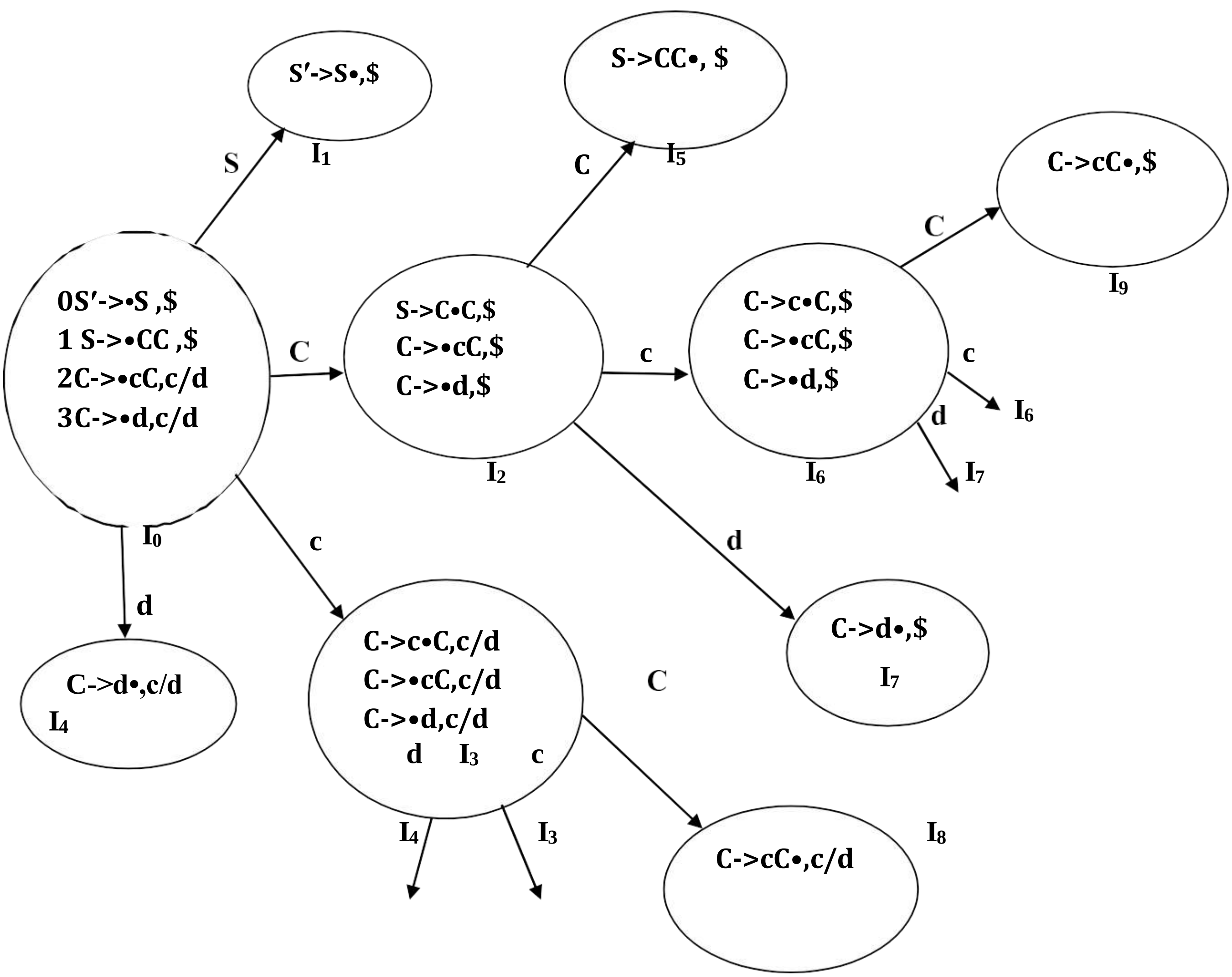
to (I_3 , c)= Closure(C->c•C, c/d) = I_3

Goto(I_3,d)=Closure(C->d•,c/d)= I_4

I_9=Goto(I_6, C)=Closure(C->cC•, $)= C->cC•,$
Goto(I_6, c)=Closure(C->c•C ,$)= I_6

Goto(I_6,d)= Closure(C->d•,$)=$I_7$

DrawingtheFiniteStateMachineDFAfortheaboveLR(1)items



Construction ofCLR(1)Table

Rule1:ifthere isanitem[A->α•Xβ,b] in$I_i$andgoto(I_i,X)isin$I_j$thenaction[I_i][X]=Shift j, Where X is Terminal.

Rule2:ifthere isanitem[A->α•,b] in$I_i$and(A≠S`) set action[I_i][b]=reducealongwith the production number.

Rule3:ifthereisanitem[S`->S•,$]in$I_i$thensetaction[$I_i$][$]=Accept.

Rule4:ifthere isanitem[A->α•Xβ,b] in$I_i$andgoto(I_i,X)isin$I_j$thengoto[I_i][X]=j, Where X is Non Terminal.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Table:LR(1)Table

LALR(1)Parsing

The CLR Parser avoids the conflicts in the parse table. But it produces more number of States when compared to SLR parser. Hence more space is occupied by the table in the memory. So LALR parsing can be used. Here, the tables obtained are smaller than CLR parse table. But it also as efficient as CLRparser. Here LR(1)items that have same productions but different look-aheads are combined to form a single set of items.

For example, consider thegrammar inthepreviousexample. Consider thestates$I_4$and I_7as given below:

$I_{4=}$Goto(I_0,d)=Colsure(C->d•, c/d)=C->d•,c/d I_7=

Go to (I_2, d)= Closure(C->d•,$) = C->d•, $

These statesaredifferring onlyinthe look-aheads. Theyhave thesameproductions. Hencethese states are combined to form a single state called as I_{47}.

SimilarlythestatesI$_3$andI$_6$differing onlyintheirlook-aheadsasgivenbelow:

$I_{3=}$Goto(I_0,c)=

C->c•C,c/d
C->•cC,c/d
C->•d , c/d

I_6=Goto(I_2,c)=
C->c•C,$
C->•cC,$
C->•d,$

Thesestatesaredifferingonlyinthe look-aheads.Theyhavethesameproductions. Hencethese states are combined to form a single state called as I_{36}.

SimilarlytheStatesI_8andI_9differingonlyinlook-aheads. Hencetheycombinedtoform the state I_{89}.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			ACCEPT		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		5
I_5			R_1		
I_{89}	R_2	R_2	R_2		

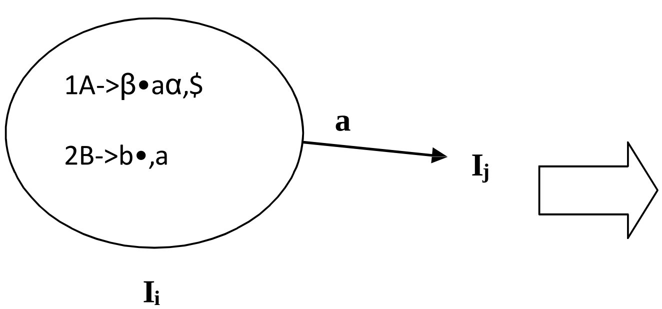
Table:LALRTable

ConflictsintheCLR(1)Parsing: Whenmultiple entriesoccurinthetable.Then,the situation is said to be a Conflict.

Shift-ReduceConflictinCLR(1)Parsing

ShiftReduceConflictintheCLR(1)parsing occurswhenastatehas

3. AReduceditemoftheformA→α•,aand
4. AnincompleteitemoftheformA→ β•aαasshownbelow:

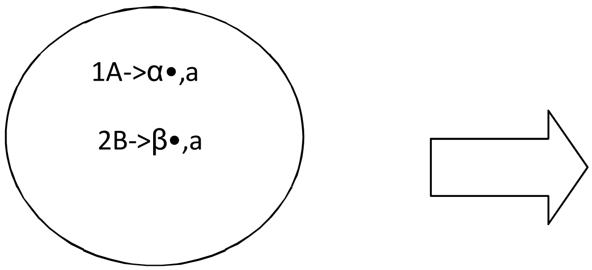


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce/ReduceConflictinCLR(1)Parsing

Reduce-ReduceConflict intheCLR(1)parsingoccurswhenastatehastwoormore reduced items of the form

3. **A→α•**
4. **B→β•Iftwoproductionsinastate(I)reducingonsamelookaheadsymbol as shown below:**



I_i

States	Action		GOTO	
	a	$	A	B
I_i	r_1/r2			

StringAcceptanceusingLRParsing:

Considertheaboveexample,iftheinputStringis**cdd**

States	ACTION			GOTO	
	c	**D**	**$**	**S**	**C**
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

0 S′->•S(AugmentProduction)

1 S->•CC

2 C->•cC

3 C->•d

STACK	**INPUT**	**ACTION**
$0	cdd$	ShiftS_3
$0c3	dd$	ShiftS_4
$0c3d4	d$	ReducewithR3,C->d,pop 2*βsymbolsfromthestack
$0c3C	d$	Goto(I_3,C)=8ShiftS_6

$0c3C8	d$	ReducewithR_2,C->cC,pop2*β symbolsfromthestack
$0C	d$	Goto(I_0,C)=2
$0C2	d$	ShiftS_7
$0C2d7	$	ReducewithR3,C->d,pop 2*βsymbolsfromthestack
$0C2C	$	Goto(I_2,C)=5
$0C2C5	$	ReducewithR_1,S->CC,pop2*βsymbolsfromthestack
$0S	$	Goto(I_0,S)=1
$0S1	$	Accept

HandingAmbiguousgrammar

Ambiguity:AGrammar canhave morethanoneparsetreeforastring.Forexample,consider grammar.

stringstring+string
|string- string
|0|1|.|9

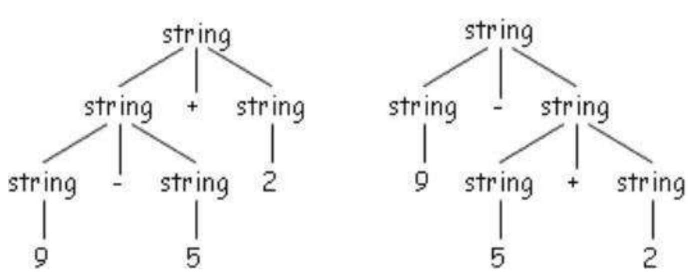
String9-5+2hastwoparsetrees

Agrammar issaidtobeanambiguousgrammar ifthereissomestringthat it cangeneratein more thanone way(i.e., the string has more thanone parse tree or morethanone leftmostderivation). A language is inherently ambiguous if it can only be generated by ambiguous grammars.

Forexample,considerthefollowinggrammar:

stringstring+string
|string- string
|0|1|.|9

Inthisgrammar,thestring9-5+2 hastwo possibleparsetreesasshowninthenextslide.



Consider the parse trees for string 9-5+2, expression like this has more than one parse tree. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9-(5+2). The second parenthesization gives the expression the value 2 instead of 6.

∑Ambiguityisproblematicbecausemeaningoftheprogramscanbeincorrect

∑ Ambiguitycanbehandledinseveralways

- Enforceassociativityandprecedence

- Rewritethegrammar(cleanestway)

Therearenogeneraltechniquesforhandlingambiguity,but

.Itisimpossibletoconvertautomaticallyanambiguousgrammartoanunambiguousone

Ambiguityisharmfultothe intent ofthe program. The input might be deciphered ina waywhich was not really the intention of the programmer, as shown above in the 9-5+2 example. Though there is no general technique to handle ambiguity i.e., it is not possible to develop some feature which automatically identifies and removes ambiguity from any grammar. However, it can be removed, broadly speaking, in the following possible ways:-

1) Rewritingthewholegrammarunambiguously.

2) Implementingprecedenceandassociativelyrulesinthegrammar. Weshalldiscussthis technique in the later slides.

Ifanoperand has operatoronboththe sides, the sideonwhichoperatortakesthis operand is the associativity of that operator

.Ina+b+c bistakenbyleft+
.+,-,*,/areleftassociative
.^,=arerightassociative

Grammartogeneratestringswithright associativeoperatorsright àletter=right |letterletter → a| b |.| z

A binary operation * on a set S that does not satisfy the associative law is called non-associative. A left-associative operation is a non-associative operation that is conventionally evaluated from left to right i.e., operand is taken bythe operator onthe left side.
Forexample,
6*5*4 =(6*5)*4andnot6*(5*4)
6/5/4 =(6/5)/4andnot6/(5/4)

Aright-associative operation isa non-associative operationthat isconventionallyevaluated from right to lefti.e., operand is taken by the operator on the right side.

Forexample,

6^5^4=>6^(5^4)andnot(6^5)^4)
x=y=z=5 => x=(y=(z=5))

Following isthegrammar to generatestringswithleft associativeoperators.(Notethatthis is left recursiveandmaygointoinfiniteloop.Butwewillhandlethisproblemlateronbymakingit right recursive)

left → left+letter|letter
letter → a | b |......| z

IMPORTANT QUESTIONS

1. DiscussthetheworkingofBottomupparsingandspecificallytheOperatorPrecedence Parsing with an exaple?
2. WhatdoyoumeanbyanLRparser?ExplaintheLR(1)Parsingtechnique?
3. WritethedifferencesbetweencanonicalcollectionofLR(0)itemsandLR(1)items?
4. WritetheDifferencebetweenCLR(1) andLALR(1)parsing?
5. WhatisYACC?Explainhowdoyouuseitinconstructingtheparserusingit.

ASSIGNMENTQUESTIONS

1. ExplaintheconflictsintheShiftreduceParsing withanexample?
2. E→E+T|T

   T→T*F

   F→(E)|id,constructtheLR(1)Parsing table?AndexplaintheConflicts?
3. E→E+T|T

   T→T*F

   F→(E)|id, constructtheSLR(1)Parsingtable?AndexplaintheConflicts?
4. E→E+T|T

   T→T*F

   F→(E)|id,constructtheCLR(1)Parsingtable?AndexplaintheConflicts?

5. E→E+T|T

   T→T*F

   F→(E)|id,constructtheLALR(1)Parsingtable?AndexplaintheConflicts?

UNIT-III

INTERMEDIATECODEGENERATION

In Intermediate code generation we use syntax directed methods to translate the source program into an intermediate form programming language constructs such as declarations, assignments and flow-of-control statements.

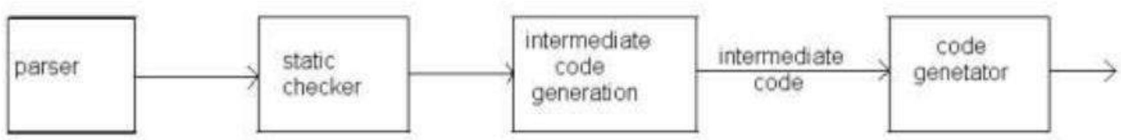


Figure4.1:IntermediateCodeGenerator

Intermediatecodeis:

- TheoutputoftheParserandtheinputtotheCodeGenerator.
- Relativelymachine-independentandallowsthecompilertoberetargeted.
- Relativelyeasytomanipulate(optimize).

WhataretheAdvantagesofanintermediatelanguage?

AdvantagesofUsinganIntermediateLanguageincludes:

1. Retargetingisfacilitated-Buildacompiler foranew machine byattachinganewcode generator to an existing front-end.

2. Optimization-reuseintermediatecodeoptimizersincompilersfordifferentlanguages and different machines.

Note: the terms ―intermediate code‖, ―intermediate language‖, and ―intermediate representation‖ are all used interchangeably.

TypesofIntermediaterepresentations/forms: Therearethreetypesofintermediate representation:-

1. SyntaxTrees
2. Postfixnotation
3. ThreeAddressCode

Semanticrulesforgeneratingthree-addresscodefromcommonprogramminglanguage constructs are similar to those for constructing syntaxtrees of for generating postfix notation.

GraphicalRepresentations

A syntax tree depicts the natural hierarchical structure of a source program. A DAG (DirectedAcyclicGraph)givesthesameinformationbutinamorecompact waybecausecommon sub-expressions are identified. Asyntaxtree forthe assignment statement a:=b*-c+b*-cappear in the following figure.

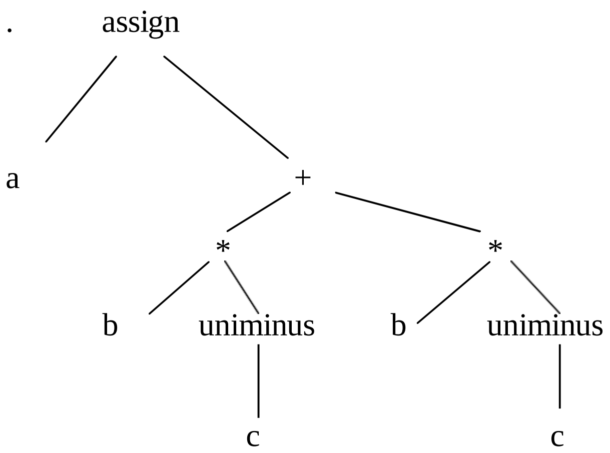


Figure4.2: AbstractSyntaxTreeforthestatementa:=b*-c+b*-c

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the in whichanodeappears immediatelyafter itschildren. Thepostfixnotationforthesyntaxtreeinthe fig is

a bc**uminus**+bc **uminus** *+**assign**

The edges in a syntax tree do not appear explicitly in postfix notation. They can be recoveredintheorderinwhichthenodesappearandtheno.ofoperandsthattheoperatoratanode expects.Therecoveryofedgesissimilartotheevaluation, usingastaff, ofanexpressioninpostfix notation.

WhatisThreeAddressCode?

Three-addresscodeisasequenceofstatementsofthe generalform:X:=YOpZ

where x, y, and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is only oneoperatorontheright sideofastatement. Thusasourcelanguageexpression likex+y*z might be translated into a sequence

t1 := y * z
t2:=x+t1

Wheret1andt2arecompiler-generatedtemporarynames. Thisunravelingofcomplicated arithmeticexpressionsandofnestedflow-of-controlstatementsmakesthree-addresscodedesirable fortargetcodegenerationandoptimization.Theuseofnamesfortheintermediatevaluescomputed bya programallow- three-address codeto be easily rearranged – unlike postfix notation. Three - address code is a linearzed representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

IntermediatecodeusingSyntaxfortheabovearithmeticexpression t1

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

The reason for the term‖three-address code‖ is that each statement usually contains three addresses, two for the operands and one for the result. In the implementations of three-address codegiven later inthis section, a programmer-defined name is replaced bya pointertcasymbol- table entry for that name.

Three Address Code is Used in Compiler Applications

Optimization: Three address code is often used as an intermediate representation of code during optimization phases of the compilation process. The three address code allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.

Code generation: Three address code can also be used as an intermediate representation of code during the code generation phase of the compilation process. The three address code allows the compiler to generate code that is specific to the target platform, while also ensuring that the generated code is correct and efficient.

Debugging: Three address code can be helpful in debugging the code generated by the compiler. Since three address code is a low-level language, it is often easier to read and understand than the final generated code. Developers can use the three address code to trace the execution of the program and identify errors or issues that may be present.

Language translation: Three address code can also be used to translate code from one programming language to another. By translating code to a common intermediate representation, it becomes easier to translate the code to multiple target languages.

General Representation

```
a = b op c
```

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Example-1: Convert the expression a * – (b + c) into three address code.

$$t_1 = b + c$$
$$t_2 = \text{uminus } t_1$$
$$t_3 = a * t_2$$

TypesofThree-AddressStatements

Three-address statements are akinto assemblycode. Statements canhave symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding inter- mediate code. Actual indices can be substituted for the labels either by making a separate pass, or byusing ‖back patching,‖ discussed in Section 8.6.Herearethecommonthree-addressstatementsusedintheremainderofthisbook:

1. **Assignment statements** of the form x: = y op z, where op is a binary arithmetic or logical operation.

2. **Assignment instructions** ofthe formx:= op y, where op is a unaryoperation. Essentialunary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3. **Copy statements**ofthe formx:=ywhere thevalueofyisassignedtox.

4. **Theunconditionaljump**gotoL.Thethree-addressstatement withlabelListhenexttobe executed.

5. **Conditionaljumps**suchasifxrelop ygoto L.Thisinstructionappliesarelationaloperator(<, =,>=,etc.)toxandy,andexecutesthestatementwithlabelLnextifxstandsinrelationrelopto y.Ifnot,thethree-addressstatement following ifxrelopygotoLisexecutednext,asintheusual sequence.

6. **paramxandcallp,n** forprocedurecallsandreturny,where yrepresentingareturnedvalue is optional. Their typical use is as the sequence of three-address statements

paramx1

paramx2

paramxn

call p, n

Generated as part of a call of the procedure p(x,, x~,..., x‖). The integern indicating the number ofactualparametersin‖callp,n‖isnotredundantbecausecallscanbenested.Theimplementation of procedure calls is outline d in Section 8.7.

7. **Indexedassignments**ofthe formx:= y[i]and x[i]:= y.The firstofthese setsxtothevalue in the location i memory units beyond location y. The statement x[i]:=y sets the contents ofthe locationiunitsbeyondxtothevalueofy.Inboththeseinstructions,x,y,andirefertodataobjects.

8. **Address and pointer assignments** of the form x:= &y, x:= *y and *x: = y. The first of these setsthevalueofxtobethelocationofy.Presumablyyisaname,perhapsatemporary,thatdenotes anexpressionwithanI-value suchas A[i, j], and x is a pointer name ortemporary. That is, the r- value of x is the l-value (location) of some object!. In the statement x: = ~y, presumablyy is a pointeror atemporarywhose r- value is a location. The r-value ofx is made equaltothe contents ofthat location. Finally, +x: = ysets the r-value ofthe object pointed to by x to the r- value of y.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must clearly be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restrictedinstructionsetmayforcethefront endtogeneratelongsequencesofstatementsforsome source, language operations. The optimizer and code generator may then have to work harder if good code is to be generated.

SYNTAXDIRECTEDTRANSLATIONOFTHREEADDRESSCODE

Whenthree-addresscodeisgenerated,temporarynamesaremadeup fortheinteriornodes of a syntax tree. The value of non- terminal E on the left side of E E1 + E will be computed into a new temporary t. In general, the three- address code for id: = E consists of code to evaluate E intosome temporaryt, followedbythe assignmentid.place: = t. Ifanexpression is asingle identifier, sayy,thenyitselfholdsthevalueoftheexpression. Forthemoment, wecreate a new name every time a temporary is needed; techniques forreusing temporaries are given in Section S.3. The S-attributed definition in Fig. 8.6 generates three-address code for assignment statements. Given input a: = b+ – c + b+ – c, it producesthe code inFig. 8.5(a). The synthesized attribute S.code represents the three- address code for the assignment S. The non- terminalE has two attributes:

1. E.place,thenamethatwillholdthevalueofE,and

2. E.code,thesequenceofthree-addressstatementsevaluatingE.

The function newtemp returns a sequence of distinct names t1, t2,... in response to successive calls. For convenience, we use the notation gen(x‘: =‘ y‘+‘ z) inFig. 8.6to represent thethree-address statement x: = y+ z. Expressions appearing instead ofvariables like x, y, and z are evaluated when passed to gen, and quoted operators or operands, like ‘+‘, are taken literally. In practice, three- address statements might be sent to an output file, rather than built up into the code attributes. Flow-of-controlstatements can be added to the language ofassignments in Fig. 8.6byproductionsandsemanticrules)liketheonesfor whilestatementsinFig. 8.7.Inthefigure, the code for S - while E do S, is generated using‘ new attributes S.begin and S.after to mark the first statement in the code for E and the statement following the code forS, respectively.

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} := E$	$S.code := E.code \Vert gen(\textbf{id}.place\ \text{':='}\ E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \Vert E_2.code \Vert$ $gen(E.place\ \text{':='}\ E_1.place\ \text{'+'}\ E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \Vert E_2.code \Vert$ $gen(E.place\ \text{':='}\ E_1.place\ \text{'*'}\ E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$ $E.code := E_1.code \Vert gen(E.place\ \text{':='}\ \text{'uminus'}\ E_1.place)$
$E \rightarrow (\ E_1\ )$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow \textbf{id}$	$E.place := \textbf{id}.place;$ $E.code := \text{''}$

These attributes represent labels created by a function new label that returns a new label every time itis called.

IMPLEMENTATIONSOF THREE-ADDRESSSTATEMENTS:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

QUADRUPLES:

Aquadrupleisarecordstructurewithfour fields,whichwecallop,argl, arg2,and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented byplacing y inarg 1. z in arg 2. and x in result. Statements with unaryoperatorslike x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 norresult. Conditional and unconditional jumps put the target label in result. The quadruples in Fig. H.S(a) are for the assignmenta: = b+ – c + b i– c. Theyare obtained fromthe three-address code .Thecontentsoffieldsarg1,arg2,andresult arenormallypointerstothesymbol-tableentries for the names represented by these fields. If so, temporary names mustbe entered into the symbol table as they are created.

TRIPLES:

To avoid entering temporary names into the symbol table. We might refer to a temporary value bi the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: op, arg 1 and arg2, as Shown below. The fields arg l and arg2, for the arguments of op, are either pointers to the symbol table (for programmer-definednamesorconstants)orpointersintothetriplestructure(fortemporaryvalues). Since three fields are used, this intermediate code format is known as triples.‘ Except for the treatment of programmer-defined names, triples correspond to the representation of a syntax tree or dag byan array of nodes, as in

	op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		A

Table8.8(a):Qudraples

	op	Arg1	Arg2
(0)	uminus	C	
(1)	*	B	(0)
(2)	uminus	C	
(3)	*	B	(2)
(4)	+	(1)	(3)
(5)	:=	A	(4)

Table8.8(b):Triples:Triples

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointersarerepresented bythe namesthemselves. Inpractice, the informationneeded to interpret the different kinds ofentries in the arg 1and arg2fields can be encoded into theopfield or some additional fields. The triples in Fig. 8.8(b) correspond to the quadruples in Fig. 8.8(a). Note that thecopystatementa:=t5isencoded inthetriplerepresentationbyplacinga inthearg1field and using the operator assign. A ternary operation like x[i]: = y requires two entries in the triple structure,asshowninFig.8.9(a),whilex:=y[i]isnaturallyrepresentedastwooperationsinFig. 8.9(b).

	op	arg 1	arg 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y

	op	arg 1	arg 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

(b) x := y[i]

Fig. 8.9. More triple representations.

IndirectTriples

Another implementation of three-address code that has been considered is that of listing pointerstotriples,ratherthanlistingthetriplesthemselves.Thisimplementationisnaturallycalled indirect triples. For example, let us use an arraystatement to list pointers to triples in the desired order. Then the triples in Fig. 8.8(b) might be represented as in Fig. 8.10.

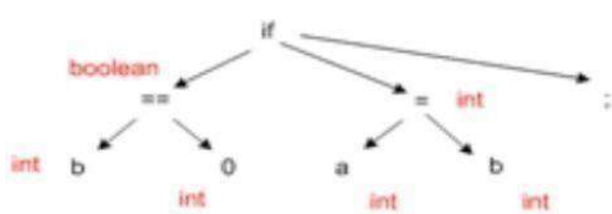
	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg 1	arg 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Figure 8.10 : Indirect Triples

SEMANTICANALYSIS: Thisphasefocusesmainlyonthe

.Checkingthesemantics,
.Errorreporting
.Disambiguateoverloadedoperators
.Typecoercion,
.Staticchecking



- Typechecking
-Controlflowchecking
- Uniquenesschecking
- Namecheckingaspectsoftranslation

Assume that the program has been verified to be syntactically correct and converted into somekindofintermediaterepresentation(aparsetree).Onenowhasparsetreeavailable.The next phase will be semantic analysis ofthe generated parse tree. Semantic analysis also includes error reporting in case any semantic error is found out.

Semantic analysis is a pass bya compiler that adds semantic information to the parse tree and performs certain checks based on this information. It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (Ina compiler implementation, it may be possible to fold different phases into one pass.) Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding). Sometimes also some early code optimization is done inthis phase. For this phase the compiler usually maintains symbol tables in which it stores what each symbol (variable names, function names, etc.) refers to.

FOLLOWINGTHINGSAREDONEINSEMANTICANALYSIS:

DisambiguateOverloadedoperators:Ifanoperatorisoverloaded,onewould liketospecifythe meaning ofthat particular operator because fromone willgo into code generation phase next.

TYPECHECKING: Theprocessofverifyingandenforcingtheconstraintsoftypesiscalledtype checking. This may occur either at compile-time (a static check) or run-time(a dynamic check). Static type checking is a primary task of the semantic analysis carried out by a compiler. If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

UNIQUENESSCHECKING: Whetheravariablenameisuniqueornot,intheitsscope.

Typecoersion: Ifsomekindofmixingoftypesisallowed.Done inlanguageswhicharenot strongly typed. This can be done dynamically as well as statically.

NAMECHECKS: Checkwhetheranyvariablehasanamewhichisnotallowed.Ex.Nameis same as an identifier (Ex. int in java).

∑ Parsercannotcatchalltheprogramerrors
∑ Thereisalevelofcorrectnessthatisdeeper thansyntaxanalysis
∑ Somelanguage featurescannotbemodeledusingcontextfreegrammarformalism

- Whetheranidentifierhasbeendeclaredbeforeuse,thisproblemisofidentifyingalanguage {w αw|wεΣ*}

- Thislanguage isnotcontextfree

A parser has its own limitationsin catching program errors related to semantics,something that is deeper than syntax analysis. Typical features of semantic analysis cannot be modeled using context free grammar formalism. If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore.

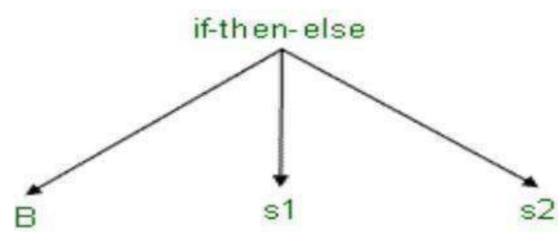
Example: in

stringx;inty;
y = x + 3 theuseofxisatypeerror int
a, b;
a = b+ccisnotdeclared

Anidentifiermayrefertodifferentvariables indifferentpartsoftheprogram.Anidentifier may be usable inone part ofthe programbut not another These are acouple ofexamples whichtellus thattypicallywhat acompiler has to do beyond syntaxanalysis. The third point can be explained like this: An identifier x can be declaredin twoseparate functions in the program, once of the type int and then of the type char. Hence the same identifier will have to be bound to these two differentpropertiesinthetwodifferent contexts.Thefourthpoint canbeexplainedinthismanner: A variable declared within one function cannot be used within the scope of the definition of the other function unless declared there separately. This is just anexample. Probably you can think ofmanymoreexamples inwhichavariabledeclaredinonescopecannotbeused inanother scope.

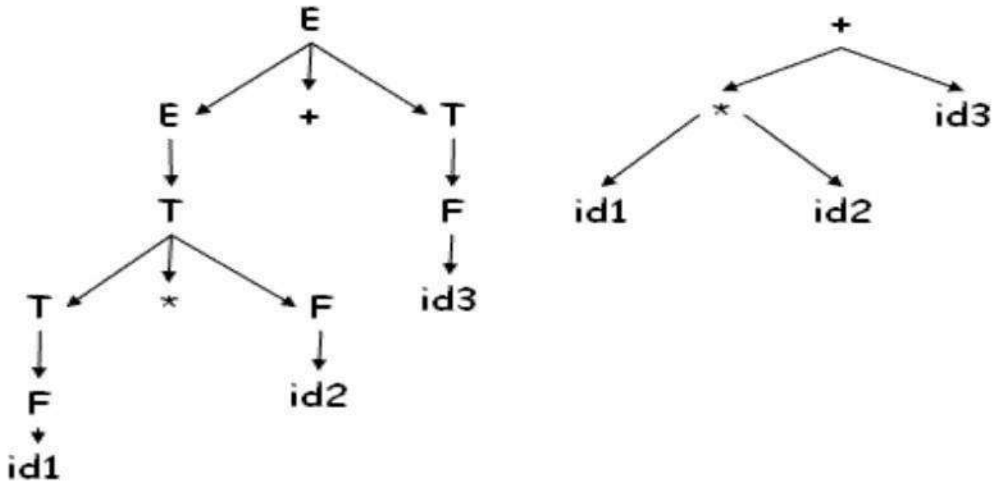
ABSTRACTSYNTAX TREE: Isnothingbutthecondensedformofaparsetree,Itis

∑Usefulforrepresentinglanguageconstructssonaturally.
∑TheproductionS⟶ifB thens1 else s2mayappearas



Inthenextfewslideswewillseehowabstractsyntaxtreescanbeconstructedfromsyntaxdirected definitions. Abstract syntax trees are condensed form of parse trees. Normally operators and keywordsappearasleavesbut inanabstractsyntaxtreetheyareassociatedwiththe interior nodes thatwouldbetheparentofthoseleaves intheparsetree.This isclearlyindicatedbythe examples in these slides.

Chainofsingleproductionsmaybecollapsed,andoperatorsmovetotheparentnodes



Chainofsingleproductionsare collapsed intoonenodewiththeoperatorsmoving upto become the node.

CONSTRUCTINGABSTRACTSYNTAXTREEFOREXPRESSIONS:

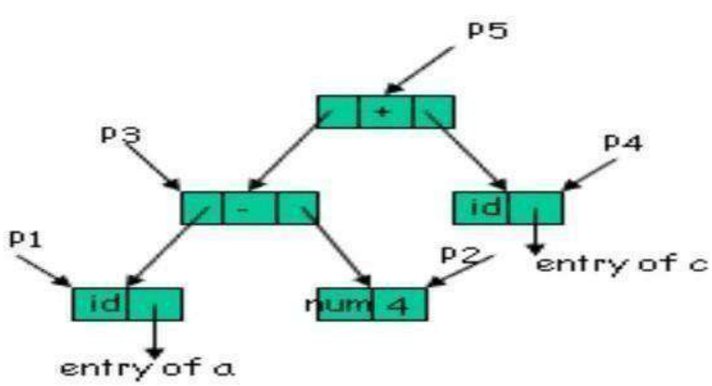
Inconstructingthe SyntaxTree,wefollowtheconventionthat:

.Eachnodeofthetreecanberepresented asarecordconsistingofat least twofieldstostore operators and operands.
.***operators*:**onefieldforoperator,remainingfieldsptrstooperands mknode(op,left,right)
.***identifier*:**onefieldwithlabelidandanotherptrtosymboltablemkleaf(id,id.entry)
.***number*:**onefieldwithlabelnumandanothertokeepthe valueofthenumbermkleaf(num,val)

Each node in an abstract syntax tree can be implemented as a record with several fields. In the node for an operator one field identifies the operator (called the label of the node) and the remaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may have additional fields to hold values (or pointers to values) of attributes attached to the node. The functions given in the slide are used to create the nodes of abstract syntax trees for expressions. Each function returns a pointer to a newly created note.

ForExample:thefollowing sequence of function callscreatesaparse treefor**w=a-4+c**

P1=mkleaf(id,entry.a) P 2 = mkleaf(num, 4) P3=mknode(-,P1,P2) P 4 = mkleaf(id, entry.c)



P5=mknode(+,P3,P4)

An example showing the formation of an abstract syntax tree by the given function calls for the expression a-4+c.The call sequence can be defined based on its postfix form, which is explained blow.

A-Writethepostfixequivalentoftheexpressionforwhichwewanttoconstruct asyntaxtree For

above string w=a-4+c, it is **a4-c+**

B-Callthe functionsinthesequence,asdefinedbythesequence inthepostfixexpressionwhich resultsinthedesiredtree.Inthecaseabove,callmkleaf()fora,mkleaf()for 4,mknode()for -,mkleaf()forc,andmknode()for+atlast.

1. P1=**mkleaf**(id, a.entry):Aleafnodemade fortheidentifier a,andanentryforais madein the symbol table.

2. P2=**mkleaf**(num,4):Aleafnodemadeforthenumber 4, andentryfor itsvalue.

3. P3=**mknode**(-,P1,P2):Aninternalnodeforthe-,takesthepointerto previouslymadenodes P1, P2 as arguments and represents the expression a-4.

4. P4=**mkleaf**(id, c.entry):Aleafnodemade fortheidentifierc,andanentryforc.entrymade in the symbol table.

5. P5=**mknode**(+,P3,P4):Aninternalnodeforthe+,takesthepointerto previouslymade nodes P3,P4 as arguments and represents the expression a- 4+c .

Followingisthesyntaxdirecteddefinitionfor constructing syntaxtreeabove

E ⟶E 1+ T	E.ptr= mknode(+,E1.ptr,T.ptr)
E⟶T	E.ptr=T.ptr
T ⟶T 1*F	T.ptr:=mknode(*,T1.ptr,F.ptr)
T ⟶F	T.ptr:=F.ptr
F⟶(E)	F.ptr :=E.ptr
F⟶id	F.ptr:=mkleaf(id,id.entry)
F⟶num	F.ptr:=mkleaf(num,val)

Nowwehave the syntaxdirected definitions to constructthe parsetreeforagivengrammar. All the rules mentioned in slide 29 are taken care ofand an abstract syntax tree is formed.

ATTRIBUTEGRAMMARS:ACFGG=(V,T,P,S),iscalledanAttributedGrammariff, where in G, each grammar symbol XƐ VUT, has an associated set of attributes, and each production,pƐP,isassociatedwithasetofattributeevaluationrulescalledSemantic Actions.

Inan**AG,**the**v**aluesofattributes at aparsetree node arecomputed bysemantic rules. There are two different specifications of**AGs** used bythe **Semantic** Analyzer inevaluating the semantics of the program constructs. They are,

- **Syntaxdirecteddefinition(SDD)s**
  - Highlevelspecifications
  - Hidesimplementationdetails
  - Explicit orderofevaluationisnotspecified
- **SyntaxdirectedTranslationschemes(SDT)s**
  - ∑Nothingbut anSDD, whichindicatesorderinwhichsemanticrulesaretobe evaluated and
  - ∑Allowsomeimplementationdetailstobeshown.

An **attribute grammar** is the formal expression of the syntax-derived semantic checks associated with a grammar. It represents the rules of a language not explicitly imparted by the syntax. In a practical way, it defines the information that is needed in the abstract syntax tree in order to successfully perform semantic analysis. This information is stored as attributes of the nodes ofthe abstract syntax tree. The values ofthose attributes are calculated bysemantic rule.

Therearetwowaysforwritingattributes:

1) **SyntaxDirectedDefinition(SDD):**Isacontextfreegrammar inwhichaset ofsemantic actions are embedded (associated) with each production of G.

It is a high level specification in which implementation details are hidden, e.g., S.sys = A.sys + B.sys;

/*doesnotgiveanyimplementationdetails. It justtellsus.Thiskindofattributeequation we will be using, Details like at what point oftime is it evaluated and in what manner are hidden from the programmer.*/

E → E1+ T	{E.val= E1.val+E2.val}
E → T	{E.val=T.val}
T → T 1*F	{T.val=T1.val+F.val)
T → F	{T.val=F.val}
F → (E)	{ F.val=E.val}
F → id	{F.val=id.lexval}
F → num	{F.val=num.lexval}

2) **Syntax directed Translation(SDT) scheme**: Sometimes we want to control the way the attributes are evaluated, the order and place where they are evaluated. This is ofa slightly lower level.

AnSDTisanSDD inwhichsemanticactionscanbeplacedat anypositioninthebodyofthe production.

Forexample,followingSDT printstheprefixequivalentofanarithmeticexpressionconsistinga +and *operators.

L⟶En{printf(„E.val")}
E⟶{printf(„+")}E1+TE
⟶T
T⟶{printf(„*")}T1*F T
⟶F
F⟶(E)
F⟶{printf(„id.lexval")}id
F⟶{printf(„num.lexval")}num

ThisactioninanSDT, isexecutedassoonasitsnodeintheparsetreeisvisited inapreorder traversal of the tree.

ConceptuallybboththeSDDand SDTschemeswill:

∑Parseinputtokenstream

∑Buildparsetree

∑Traversetheparsetreetoevaluatethesemanticrulesattheparsetreenodes Evaluation may:

∑Generatecode

∑Saveinformationinthesymboltable

∑Issue errormessages

∑Performanyotheractivity

Toavoidrepeatedtraversaloftheparsetree, actionsaretakensimultaneouslywhenatokenis found. So calculation of attributes goes along with the construction of the parse tree.

Along with the evaluation of the semantic rules the compiler may simultaneously generate code, save the information in the symbol table, and/or issue error messages etc. at the same time while building the parse tree.

Thissavesmultiplepassesoftheparsetree.

Example

Number⟶signlist
sign ⟶+ | -
list⟶listbit|bit
bit⟶0|1

Buildattributegrammar thatannotatesNumberwiththevalueitrepresents

.Associateattributeswithgrammarsymbols

symbol	**attributes**
Number	value
sign	negative
list	position,value
bit	position,value

productionAttributerulenumber ⟶ signlist
list.position ← 0

ifsign.negative

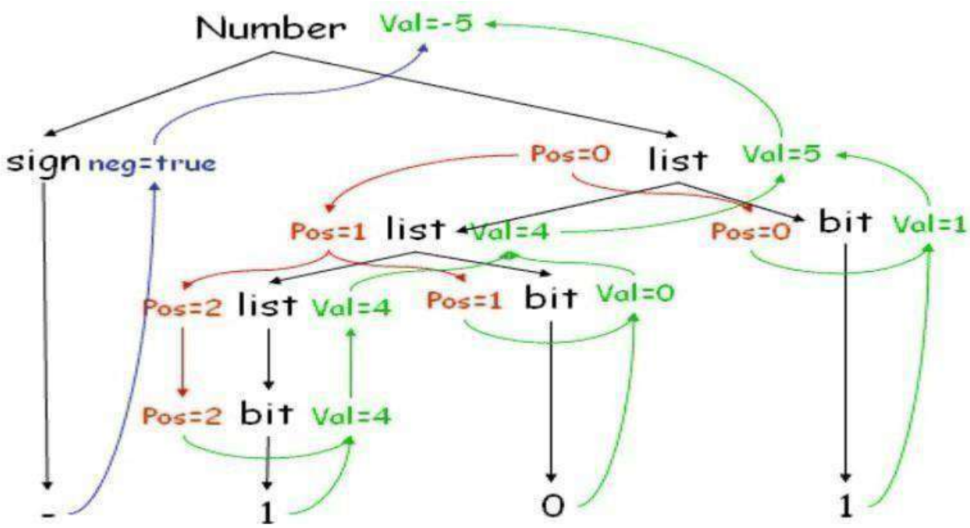
then number.value ← -list.value
else number.value ← list.value
sign ⟶ + sign.negative ← false sign ⟶ - sign.negative ← truelist ⟶ bit
bit.position ← list.position
list.value ← bit.value
$list_0 \longrightarrow list_1$ bit
$list_1$ .position ← $list_0$.position+1
bit.position ← list $_0$ .position
$list_0$ .value ← $list_1$.value+bit.value
bit ⟶ 0 bit.value ← 0 bit ⟶ 1 bit.value ← $2^{bit.position}$

Explanationofattribute rules

Num->signlist	/*sincelististherightmost soit isassignedposition0 *Signdetermineswhetherthevalueofthenumberwouldbe *sameorthe negative ofthe value of list*/
Sign-> +|-	/*SettheBooleanattribute(negative)for sign*/
List->bit	/*bitpositionisthesameaslist positionbecausethisbitistherightmost *value ofthe list is same as bit.*/
List0 -> List1 bit	/*positionand valuecalculations*/
Bit -> 0 | 1	/*set the corresponding value*/

AttributesofRHScanbecomputedfromattributesofLHSandviceversa.

TheParseTreeandtheDependencegraphareasunder



.

Dependence graph shows the dependence of attributes on other attributes, along with the syntaxtree.Top downtraversalis followed bya bottomuptraversalto resolve the dependencies. Number, val and neg are synthesized attributes. Pos is an inherited attribute.

Attributes : . Attributes fall into two classes namely *synthesized attributes* and *inherited attributes*.Valueofasynthesizedattributeiscomputedfromthevaluesofitschildrennodes.Value of an inherited attribute is computed fromthe sibling and parent nodes.

The attributes are divided into two groups, called synthesized attributes and inherited attributes. The synthesized attributes are the result of the attribute evaluation rules also using the values of the inherited attributes. The values of the inherited attributes are inherited from parent nodes and siblings.

Each grammar production A $\longrightarrow$ ahasassociatedwithit asetofsemanticrulesoftheform b=

f(c_1,c_2,...,c_k),Wherefisafunction,and either ,bisasynthesizedattributeofAOr

-bisan inheritedattributeofoneofthegrammarsymbolsontheright

.attributebdependsonattributesc_1,c_2,...,c_k

Dependence relation tells us what attributes we need to know before hand to calculate a particular attribute.

Here the value ofthe attribute b depends on the values ofthe attributes **c_1** to **c_k.** Ifc_1 to $c_k$belong to the children nodes and b to A then b will be called a synthesized attribute. And if b belongstooneamonga(childnodes)thenitisaninheritedattributeofoneofthegrammarsymbols on the right.

SynthesizedAttributes:Asyntaxdirecteddefinitionthat usesonlysynthesizedattributes is said to be an S- attributed definition

.Aparsetreefor anS-attributeddefinitioncanbeannotatedbyevaluatingsemantic rules for attributes

S-attributed grammars are a class of attribute grammars, comparable with L-attributed grammars butcharacterizedbyhavingnoinheritedattributesatall.Inheritedattributes,whichmustbepassed downfromparent nodesto childrennodesoftheabstract syntaxtreeduringthesemantic analysis, pose a problem for bottom-up parsing because in bottom-up parsing, the parent nodesof the abstract syntax tree are created*after* creation of all of their children.Attribute evaluation in S- attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing .

SyntaxDirectedDefinitionsforadeskcalculatorprogram

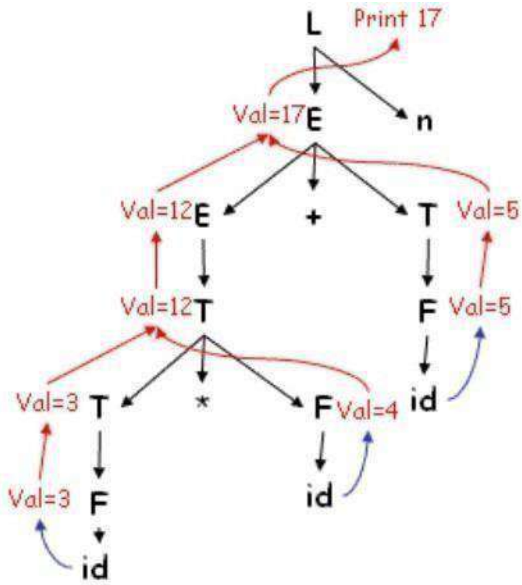
L ⟶ E n	Print(E.val)
E ⟶ E+ T	E.val=E.val+T.val
E ⟶ T	E.val=T.val
T ⟶ T*F	T.val=T.val*F.val
T ⟶ F	T.val=F.val
F ⟶ (E)	F.val=E.val
F ⟶ digit	F.val=digit.lexval

.terminals are assumed to have onlysynthesized attribute valuesofwhichare supplied bylexical analyzer

.startsymboldoesnothaveanyinheritedattribute

Thisisagrammarwhichusesonlysynthesizedattributes.Startsymbolhasno parents,henceno inherited attributes.

Parsetreefor3*4+5n



Usingthepreviousattributegrammar calculationshave beenworkedoutherefor3*4+5n. Bottom up parsing has been done.

InheritedAttributes:Aninheritedattributeisonewhosevalue isdefined intermsof attributes at the parent and/or siblings

.Usedforfindingoutthecontextinwhichitappears

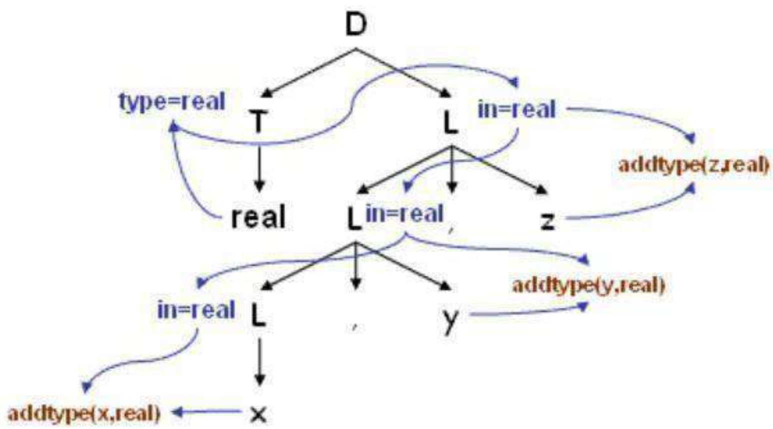
.possibletouseonlyS-attributesbut morenaturaltouseinheritedattributes D

$\longrightarrow$T L	L.in = T.type
T$\longrightarrow$real	T.type=real
T$\longrightarrow$int	T.type=int
L$\longrightarrow$$L_1$,id	L_1.in=L.in;addtype(id.entry,L.in)
L$\longrightarrow$id	addtype(id.entry,L.in)

Inherited attributes help tofind thecontext(type,scope etc.) ofa token e.g., the type of a token or scopewhenthe same variable name is used multiple times in a program indifferent functions. An inherited attribute system may be replaced by an S -attribute system but it is more natural to use inherited attributes in some cases like the example given above.

Hereaddtype(a,b)functionsaddsasymboltableentryfortheid aandattachestoitthetypeofb
.

Parsetreeforrealx,y,z



Dependence of attributes in an inherited attribute system. The value of in (an inherited attribute) at the three L nodes gives the type of the three identifiers x , y and z . These are determined by computing the value ofthe attribute T.type atthe left child ofthe root and thenvaluating L.intop down at the three L nodes in the rightsubtreeofthe root. Ateach L node the procedure addtype is called which inserts the type of the identifier to its entry in the symbol table. The figure also shows the dependence graph which is introduced later.

Dependence Graph: . Ifanattribute bdepends onanattribute cthenthe semantic rule for b must be evaluated after the semantic rule for c

.Thedependenciesamongthenodescanbedepictedbyadirectedgraphcalleddependency graph

DependencyGraph**:**Directedgraphindicatinginterdependenciesamongthesynthesizedand inherited attributes of various nodes in a parse tree.

Algorithmtoconstructdependencygraph for

each node **n** in the parse tree do

foreachattribute**a**ofthegrammarsymboldo construct a

node in the dependency graph

for**a**

foreachnodenintheparsetreedo

foreachsemanticrule b=f(c_1,c_2,...,c_k)do

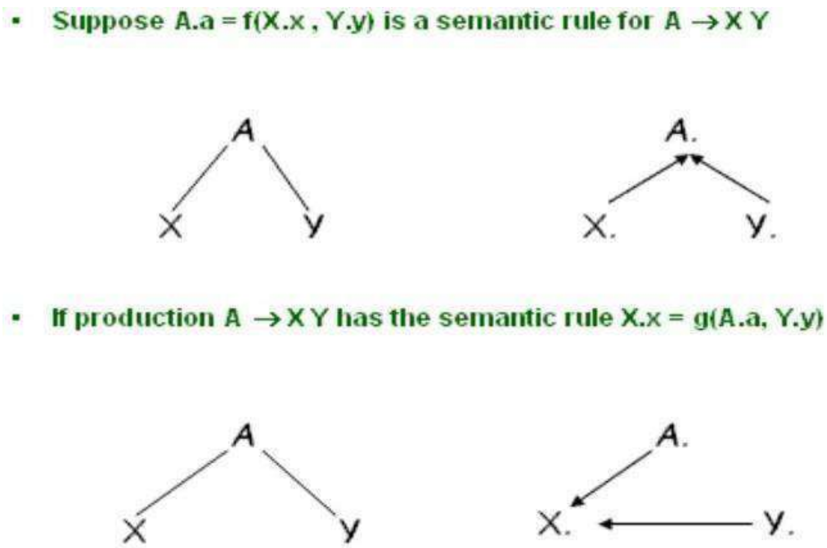
{associatedwithproductionatn}

fori=1tokdo

Constructanedgefrom$c_i$tob

Analgorithmtoconstructthedependencygraph.Aftermakingonenodeforeveryattribute of all the nodes of the parse tree, make one edge from each of the other attributes on which it depends.

Forexample,



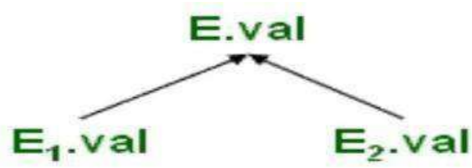
The semantic rule A.a = f(X.x , Y.y) for the production A -> XY defines the synthesized attribute a of A to be dependent on the attribute x of X and the attribute y of Y . Thus the dependency graph will contain an edge from X.x to A.a and Y.y to A.a accounting for the two dependencies.SimilarlyforthesemanticruleX.x= g(A.a,Y.y)forthesameproductiontherewill be an edge from A.a to X.x and an edg e from Y.y to X.x.

Example

.Wheneverfollowingproductionisusedinaparsetree E

$\longrightarrow$E 1 + E 2 E.val = E 1 .val + E 2 .val

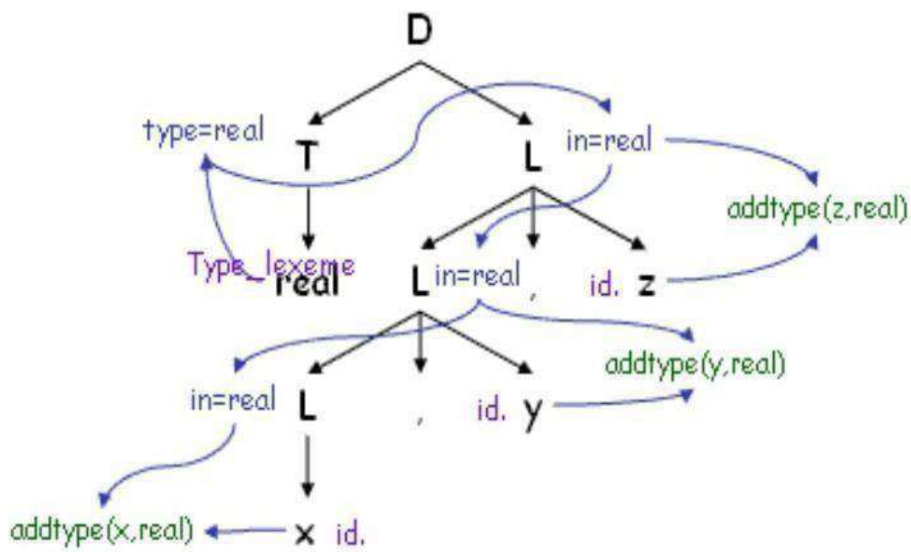
wecreate adependencygraph



ThesynthesizedattributeE.valdependsonE1.valandE2.valhencethetwoedgesoneeach from E 1 .val & E 2 .val

Forexample, thedependencygraphforthesting**real**id1,id2,id3

.Put adummysynthesized attributebfor asemanticrulethatconsistsofaprocedurecall



The figure shows the dependencygraph for the statement real id1, id2, id3 along with the parse tree. Procedure calls can be thought of as rules defining the values of dummy synthesized attributes of the nonterminal on the left side of the associated production. Blue arrows constitute thedependencygraphandblack lines,theparsetree.Eachofthesemanticrulesaddtype(id.entry, L.in) associated with the L productions leads to the creation of the dummy attribute**.**

<u>EvaluationOrder:</u>

Anytopologicalsortofdependencygraphgivesavalidorderinwhichsemanticrules must be evaluated

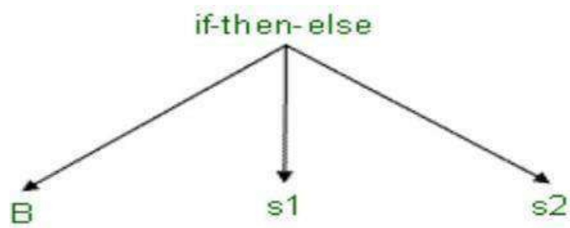
a4=real
a5 = a4
addtype(id3.entry,a5)
a7 = a5
addtype(id2.entry,a7)

a9:=a7addtype(id1.entry,a9)

Atopological sort ofa directed acyclic graph is anyordering m1, m2, m3mk ofthe nodesofthegraphsuchthatedgesgofromnodesearlierintheorderingtolaternodes.Thusifmi -> mj is an edge from mi to mj then mi appears before mj in the ordering. The order of the statementsshownintheslide isobtainedfromthetopologicalsortofthedependencygraphinthe previousslide. 'an'stands fortheattributeassociatedwiththenodenumbered ninthe dependency graph. The numbering is as shown in the previous slide.

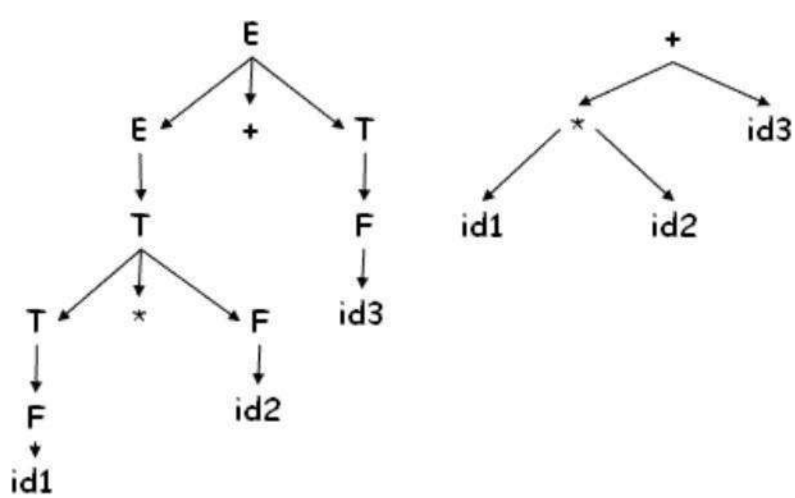
AbstractSyntaxTree isthecondensedformoftheparsetree,which is

.Usefulforrepresentinglanguageconstructs.
.Theproduction:**S**⟶**if**B**then**s1elses2mayappearas



Inthenext fewslideswewillsee howabstract syntaxtreescanbeconstructedfromsyntax directed definitions. Abstract syntax trees are condensed form of parse trees. Normallyoperators and keywords appear as leaves but in an abstract syntax tree theyare associated with the interior nodes that would be the parent of those leaves in the parse tree. This is clearly indicated by the examples in these slides.

.Chainofsingleproductionsmaybecollapsed,andoperatorsmovetotheparentnodes



Chainofsingleproductionarecollapsed intoonenodewiththeoperatorsmovingupto become the node.

ForConstructingtheAbstractSyntaxtreeforexpressions,

.Eachnodecanbe representedasarecord

.*operators*:onefieldforoperator,remainingfieldsptrstooperandsmknode(op,left,right)

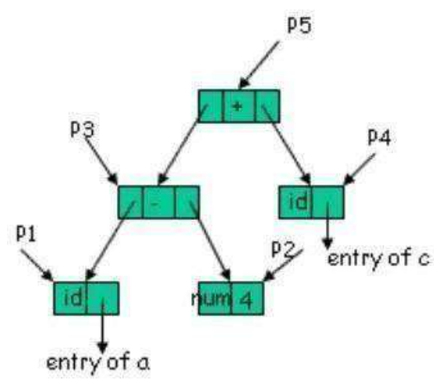
.*identifier*:onefieldwith labelidandanotherptrtosymboltablemkleaf(id,entry)

.*number*:onefieldwithlabelnumandanothertokeepthevalueofthenumber mkleaf(num,val)

Eachnode inanabstractsyntaxtreecanbe implemented asarecordwithseveralfields. In the node for an operator one field identifies the operator (called the label of the node) and the remaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may have additional fields to hold values (or pointers to values) of attributes attached to the node. The functions given in the slide are used to create the nodes of abstract syntax trees for expressions. Each function returns a pointer to a newly created note.

Example:**Thefollowing sequence of function calls creates a parse tree for a- 4 + c**

P1=mkleaf(id,entry.a) P 2 = mkleaf(num, 4) P3=mknode(-,P1,P2) P 4 = mkleaf(id, entry.c) P5=mknode(+,P3,P4)



Anexampleshowingthe formationofanabstract syntaxtreebythegivenfunctioncalls forthe expression a-4+c.The call sequence can be explained as:

1. P1=mkleaf(id,entry.a):Aleafnodemade fortheidentifierQaRandanentryforQaRis made in the symbol table.
2. P2=mkleaf(num,4):AleafnodemadeforthenumberQ4 R.
3. P3=mknode(-,P1,P2):Aninternalnode fortheQ-Q.Itakesthepreviouslymade nodesas arguments and represents the expression Qa-4 R.
4. P4=mkleaf(id,entry.c): Aleafnodemade fortheidentifierQcRandanentryforQcRis made in the symbol table.
5. P5=mknode(+,P3,P4):AninternalnodefortheQ+Q.Itakesthepreviouslymadenodesas arguments and represents the expression Qa- 4+c R.

Asyntaxdirecteddefinitionforconstructing syntaxtree

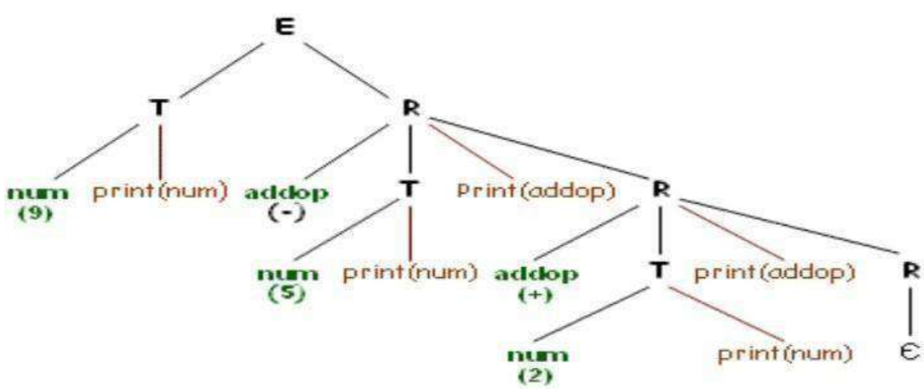
E ⟶E 1+ T	E.ptr=mknode(+,E1.ptr,T.ptr)
E⟶T	E.ptr=T.ptr
T ⟶T 1*F	T.ptr:=mknode(*,T 1.ptr,F.ptr)
T ⟶F	T.ptr:=F.ptr
F⟶(E)	F.ptr :=E.ptr
F⟶id	F.ptr:=mkleaf(id, entry.id)
F⟶num	F.ptr:=mkleaf(num,val)

Nowwehavethesyntaxdirecteddefinitionstoconstructtheparsetreeforagivengrammar.All the rules mentioned in slide 29 are taken care ofand an abstract syntax tree is formed.

Translationschemes : ACFGwheresemanticactionsoccurwithintheright handsideof production, A translation scheme to map infix to postfix.

E⟶TR
R⟶addopT{print(addop)}R|e T
⟶num {print(num)}

Parsetreefor9-5+2



Weassumethat theactionsareterminalsymbolsand Performdepthfirst ordertraversaltoobtain 9 5 - 2 +.

∑Whendesigningtranslationscheme, ensureattributevalueisavailablewhenreferredto

∑Incaseofsynthesized attributeitistrivial(why?)

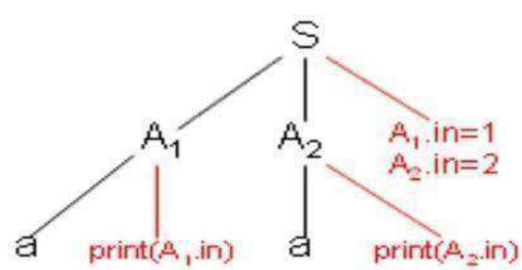
Inatranslationscheme,aswearedealingwithimplementation,wehavetoexplicitlyworry abouttheorderoftraversal. We cannowputinbetweentherulessomeactionsas partoftheRHS. We put this rules in order to control the order of traversals. In the given example, we have two terminals (num and addop). It can generally be seen as a number followed by R (which necessarily has to begin with an addop). The given grammar is in infix notation and we need to convert it into postfix notation. If we ignore all the actions, the parse tree is in black, without the rededges.Ifweincludetherededgeswegetaparsetreewithactions.Theactionsaresofartreated asaterminal.Now,ifwedoadepthfirsttraversal,andwheneverweencounteraactionweexecute it, we get a post-fix notation. Intranslation scheme, we have to take care ofthe evaluation order; otherwise some of the parts may be left undefined. For different actions, different result will be obtained. Actions aresomething we write and wehave to control it. Please note that translation scheme is different from a syntax driven definition.In the latter, we do not have any evaluation order;inthiscasewehaveanexplicit evaluationorder.Byexplicit evaluationorderwehavetoset correct action at correct places, in order to get the desired output. Place of each action is very important. We have to find appropriate places, and that is that translation scheme is all about. If we talk ofonly synthesized attribute, the translation scheme is verytrivial. This is because, when wereachweknowthatallthechildrenmust havebeenevaluatedandalltheirattributes must have also been dealt with. This is because finding the placefor evaluation is very simple, it is the rightmost place.

Incaseofbothinheritedand synthesizedattributes

. Aninherited attribute for asymbolonrhsofa production must be computed inanactionbefore that symbol

S→A1A2{A1.in=1,A2.in=2}

A→a {print(A.in)}



Depthfirstordertraversalgives error*undefined*

.Asynthesized attributefor nonterminalonthe lhscanbecomputedafter allattributes it references, have beencomputed. The action normallyshould be placed at the end ofrhs

We have a problem when we have both synthesized as well as inherited attributes. For the given example, if we place the actions as shown, we cannot evaluate it. This is because, when doing a depth first traversal, we cannot print anything for A1. This is because A1 has not yet been initialized. We, therefore have to find the correct places for the actions. This can be that the inheritedattributeofAmust becalculatedonitsleft.Thiscanbeseenlogicallyfromthedefinition of L-attribute definition, which says that when we reach a node, then everything on its left must have been computed. Ifwe do this, we will always have the attribute evaluated at the correctplace.Forsuchspecificcases(likethegivenexample)calculatinganywhereonthe left willwork, but generally it must be calculated immediately at the left.

Example:TranslationschemeforEQN

S →B	B.pts=10
	S.ht=B.ht
B →B_1 B_2	B_1.pts=B.pts
	B_2.pts=B.pts
	B.ht=max(B_1.ht,B_2.ht)
B→B_1subB_2	B_1.pts=B.pts;
	B $_2$ .pts = shrink(B.pts)
	B.ht=disp(B_1.ht,B_2.ht)
B→text	B.ht=text.h*B.pts

Wenowlookatanotherexample.ThisisthegrammarforfindingouthowdoIcomposetext.EQN was equation setting system which was used as an early type setting system for UNIX. It was earlier used as an latex equivalent for equations. We say that start symbol is a block: S - >B We can also have a subscript and superscript. Here, we look at subscript. A Block is composedof severalblocks:B->B1B2andB2isasubscriptofB1.Wehavetodeterminewhat isthepointsize (inherited) and height Size (synthesized). We have the relevant functionfor height and point size given along side. After putting actions in the right place

S → {B.pts = 10} B
{S.ht = B.ht}

B → {B_1.pts = B.pts} B_1
{B_2.pts = B.pts} B_2
{B.ht = max(B_1.ht,B_2.ht)}

B → {B_1.pts = B.pts} B_1 sub
{B_2.pts = shrink(B.pts)} B_2
{B.ht = disp(B_1.ht,B_2.ht)}

B → text {B.ht = text.h * B.pts}

We have put allthe actions at the correct places as per the rules stated. Read it from left to right, and topto bottom. We notethat all inherited attribute are calculated onthe left ofB symbols and synthesized attributes are on the right.

TopdownTranslation: UsepredictiveparsingtoimplementL-attributeddefinitions

E→E1+T E.val:= E1.val+T.val

E→E1-T E.val:= E1.val-T.val

E →T E.val:=T.val

T →(E) T.val:=E.val

T →num T.val:=num.lexval

We now come to implementation. We decide how we use parse tree and L-attribute definitions to construct the parse tree with a one-to-one correspondence. We first look at the top-down translation scheme. The firstmajor problem is leftrecursion. If we remove leftrecursion byour standard mechanism, we introduce new symbols, and new symbols willnot work withthe existing actions. Also, we have to do the parsing in a single pass.

TYPESYSTEMANDTYPECHECKING:

.Ifboththeoperandsofarithmeticoperators+,-,xareintegers thentheresultisoftypeinteger
.Theresultofunary&operatorisapointertotheobjectreferredtobytheoperand.
-Ifthe type ofoperandis***X***thentype ofresultis***pointertoX***

InPascal,typesareclassifiedunder:

1. *Basic*types: These areatomictypeswithno internalstructure.Theyinclude thetypesboolean, character, integer and real.

2. *Sub-range*types: Asub-range type defines a rangeofvalues withinthe range ofanothertype. For example, type A = 1..10; B = 100..1000; U = 'A'..'Z';

3. *Enumerated* types: An enumerated type is defined by listing all of the possible values for the type. For example: type Colour = (Red, Yellow, Green); Country = (NZ, Aus, SL, WI, Pak, Ind, SA, Ken, Zim, Eng); Both the sub-range and enumerated types can be treated as basic types.

4. *Constructed* types: A constructed type is constructed from basic types and other basic types. Examples of constructed types are arrays, records and sets. Additionally, pointers and functions can also be treated as constructed types.

TYPEEXPRESSION:

Itisanexpressionthat denotesthetypeofanexpression. Thetypeofa languageconstruct is denoted by a type expression

- ∑Itiseither abasictypeorit is formedbyapplyingoperatorscalled*typeconstructor*to other type expressions
- ∑Atype constructorapplied toatypeexpressionisatypeexpression
- ∑Abasic typeistype expression

- *typeerror*:errorduringtypechecking
- *void*:notypevalue

The type of a language construct is denoted by a type expression. A type expression is either a basictypeorisformedbyapplyinganoperatorcalledatypeconstructortoothertypeexpressions. Formally, a type expression is recursively defined as:

1. Abasictypeisatypeexpression.Amongthebasictypesare*boolean*,*char*,*integer*,and*real* .A special basic type, *type_error* , is used to signal an error during type checking. Another specialbasictypeis*void*whichdenotes"theabsenceofavalue"and isusedto checkstatements.
2. Sincetypeexpressionsmaybenamed,atypenameisatypeexpression.
3. Theresultofapplyingatypeconstructortoatypeexpressionisatypeexpression.
4. Typeexpressionsmaycontainvariableswhosevaluesaretypeexpressions themselves.

TYPECONSTRUCTORS:areusedtodefineorconstructthetypeofuserdefinedtypesbased on their dependent types.

Arrays: IfT isatypeexpressionandI isarangeofintegers,then*array*(*I*,*T*)isthetype expression denoting the type of arraywith elements oftype T and index set I.

Forexample,thePascaldeclaration, varA:array[1 .. 10]ofinteger;associatesthetype expression ***array* (1..10, *integer*)** with A.

Products*:* If*T1*and*T2*aretypeexpressions,thentheirCartesianproduct ***T1XT2***isalso atype expression.

Records*:*Arecordtypeconstructorisappliedtoatuple formed fromfield namesand field types. For example, the declaration

Considerthedeclaration

type row = record
addr:integer;
lexeme:array[1..15]ofchar
end;
vartable:array[1..10]ofrow;

Thetyperowhastypeexpression: **record((addrxinteger)x(lexemexarray(1..15,char)))** andtypeexpressionoftableis**array(1..10,row)**

Note:Includingthefieldnames inthetypeexpressionallowsustodefineanotherrecordtype with the same fields but with different names without being forced to equatethe two.

Pointers: IfT isatypeexpression,then***pointer***(***T***)isatypeexpressiondenotingthetype "pointer to an object of type T".
Forexample,inPascal,thedeclaration
var p: row declaresvariableptohavetype***pointer*(row)**.

Functions : Analogous to mathematical functions, functions in programming languages may be defined as mapping a domaintype Dto arangetype R. Thetype ofsucha function is denotedby the type expression D R. For example, the built-in function mod ofPascal has domain type int X int, and range type *int* . Thus we say mod has the type: **int xint -> int**

Asanotherexample,accordingtothePascaldeclaration
function f(a, b: char) : integer;
Herethetypeoffisdenotedbythetypeexpressionis**charxcharpointer(integer)**

SPECIFICATIONSOFATYPECHECKER: Consider alanguagewhichconsistsofa sequence of declarations followed by a single expression

P→D;E

D→D ;D |id:T

T→char| integer |array[num]ofT|^T E →

literal| num | E mod E | E [E] | E ^

A**typechecker**isatranslationschemethatsynthesizesthetypeofeachexpressionfromthetypes ofitssub-expressions. Considertheabovegivengrammarthat generatesprogramsconsistingofa sequence of declarations D followed by a single expression E.

Specificationsofatypecheckerforthelanguage oftheabovegrammar:Aprogramgenerated by this grammaris

key: integer;
keymod 1999

Assumptions:

1. Thelanguagehasthreebasictypes:*char*,*int*and*type-error*

2. Forsimplicity, allarraysstart at1.Forexample, thedeclarationarray[256]ofchar leadstothe type expression *array* (1.. 256, char).

RulesforSymbolTableentry

D→id:T	addtype(id.entry,T.type)
T →char	T.type=char
T→integer	T.type=int
T →^T_1	T.type=pointer(T_1.type)
T→array[num]ofT_1	T.type=array(1..num, T_1.type)

TYPECHECKINGOFFUNCTIONS:

ConsidertheSyntaxDirected Definition,

E $\rightarrow$ E$_1$(E$_2$)	E.type=ifE$_2$.type==sand
	E$_1$.type == s $\rightarrow$ t
	thent
	elsetype-error

Therules forthesymboltableentryarespecifiedabove. Thesearebasicallythewayinwhich the symbol table entries corresponding to the productions are done.

Typecheckingoffunctions

The production E -> E (E) where an expression is the application of one expression to another can be used to represent the application of a function to an argument. The rule for checking the type of a function application is

E ->E1(E2){*E.type*:=if*E2.type*== s and*E1.type*== *s* ->*t*then*t*else*type_error* **}**

Thisrulesaysthat inanexpressionformedbyapplyingE1toE2,thetypeofE1must bea function *s*->*t*fromthetype *s*ofE2to some range type *t* ;the type ofE1 (E2)is*t* .The above rule canbe generalizedtofunctionswithmorethanoneargument byconstructingaproducttype consistingof the arguments. Thus n arguments of type *T1* , *T2*

...*Tn*canbe viewedasasingleargumentofthetype*T1XT2...XTn*. Forexample, root : (real

real) X real real

declaresafunctionrootthattakesafunction fromrealstorealsandarealasargumentsand returns a real. The Pascal-like syntax for this declaration is

functionroot(functionf(real):real;x:real):real

TYPECHECKINGFOREXPRESSIONS:considerthefollowingSDDforexpressions

E $\rightarrow$ literal	E.type=char
E $\rightarrow$ num	E.type=integer
E $\rightarrow$ id	E.type=lookup(id.entry)
E $\rightarrow$ E$_1$modE$_2$	E.type=ifE $_1$.type==integerand
	E$_2$.type==integer
	then integer

	elsetype_error
E $\longrightarrow$ E_1[E_2]	E.type=ifE$_2$.type==integerand E$_1$.type==array(s,t) thent
	elsetype_error
E $\longrightarrow$ E_1^	E.type=ifE$_1$.type==pointer(t) then t
	elsetype_error

Toperformtypecheckingofexpressions,followingrulesareused.Wherethesynthesizedattribute typeforEgivesthetypeexpressionassigned bythetypesystemtotheexpressiongeneratedbyE.

Thefollowingsemanticrulessaythat constantsrepresentedbythetokensliteralandnumhave type *char* and *integer* , respectively:

E -> literal { *E.type* := *char* }

E->num{*E.type*:=*integer* }

.The function*lookup*(*e*)isusedtofetchthetypesavedinthesymbol-tableentrypointedtoby e.Whenanidentifierappearsinanexpression, itsdeclaredtype isfetchedandassignedtothe attribute type:

E ->id{ *E.type*:=*lookup*(id.*entry*)}

.Accordingtothefollowingrule, theexpressionformedbyapplyingthe modoperatortotwo sub-expressions oftype *integer* has type *integer* ; otherwise, its type is *type_error* .

E ->E1modE2{*E.type*:= if*E1.type*==*integer* and*E2.type*== *integer*then*integer* else *type_error*}

InanarrayreferenceE1[E2],theindexexpressionE2must havetype*integer*, inwhichcase the result is the element type *t* obtained fromthe type *array* (*s*, *t*) ofE1.

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

Withinexpressions,thepostfixoperator yieldstheobject pointedtobyitsoperand.ThetypeofE is the type *t* of the object pointed to bythe pointer E:

EE1{*E.type*:=if*E1.type* ==*pointer*(*t*)then*t*else *type_error*}

TYPECHECKINGOFSTATEMENTS: Statementstypicallydonothavevalues.Specialbasic type *void* can be assigned to them. Consider the SDD for the grammar below which generates Assignment statements conditional, and looping statements.

S → id := E	S.Type=ifid.type==E.type then void elsetype_error
S→ifE thenS1	S.Type=ifE.type== boolean then S1.type elsetype_error
S→whileEdoS1	S.Type=ifE.type== boolean thenS1.type elsetype_error
S →S1 ; S2	S.Type=ifS1.type==void and S2.type == void thenvoid elsetype_error

Sincestatementsdo nothavevalues,thespecialbasictype*void* isassignedtothem, but ifan error is detected within a statement, the type assigned to the statementis *type_error* .

The statements considered below are assignment, conditional, and whilestatements. Sequences of statements are separated by semi-colons. The productions given below can be combined with thosegivenbeforeifwechangetheproductionforacompleteprogramtoP->D;S.Theprogram now consists of declarations followed by statements.

Rulesfortypechecking thestatementsaregivenbelow.

1. Sid:=E{ *S.type*:=ifid.*type*==*E.type*then*void*else*type_error*}

Thisrulechecksthattheleftandrightsidesofanassignmentstatementhavethesametype.

2. SifEthenS1{*S.type* := if*E.type* == *boolean*then*S1.type* else *type_error*}

Thisrulespecifiesthattheexpressionsinanif-thenstatementmusthavethetype*boolean*.

3. Swhile Edo S1{*S.type*:=if*E.type*==*boolean*then*S1.type*else*type_error*}

Thisrulespecifiesthattheexpressioninawhilestatementmusthavethetype*boolean*.

4. SS1;S2 {*S.type*:=if*S1.type* ==*void*and *S2.type*==*void*then*void* else*type_error*}

Errorsarepropagatedbythis last rule becauseasequenceofstatementshastype*void*onlyif each sub-statement has type *void*.

IMPORTANT&EXPECTEDQUESTIONS

1. WhatdoyoumeanbyTHREEADDRESSCODE?Generatethethree-addresscodefor the following code.

```
begin
                    PROD:= 0;
     I:=1;
do
begin

     PROD:=PROD+A[I]B[I];
     I:=I+1
End
```

2. Writeashort noteonAttributed grammar&Annotated parsetree.
3. Defineanintermediatecodeform.Explainvariousintermediatecodeforms?
4. WhatisSyntaxDirectedTranslation?ConstructSyntaxDirectedTranslationschemeto convert a given arithmetic expression into three address code.
5. WhatareSynthesizedandInheritedattributes?Explainwithexamples?
6. ExplainSDTforSimpleTypechecker?
7. Defineandconstructtriples,quadruplesandindirecttriplenotationsofanexpression:a* -(b+c).

ASSIGNMENTQUESTIONS:

1. WriteThreeaddresscodeforthebelowexample

```
While( i<10)
{
a=b+c*-d;
i++;
}
```

2. What isaSyntaxDirectedDefinition?WriteSyntaxDirecteddefinitiontoconvert binary value in to decimal?

SYMBOLTABLE

SymbolTable(ST) : Isadatastructureused bythe **c**ompiler to keeptrackofscope and binding information about names

-Symboltableischangedeverytimeanameisencounteredinthesource;

Changestotableoccur whenever anew name isdiscovered;new informationaboutanexisting name is discovered

Asweknowthecompilerusesasymboltabletokeeptrackofscopeandbindinginformationabout names.ItisfilledaftertheAST is madebywalkingthroughthetree,discoveringand assimilating information about the names. There should be two basic operations - to insert a new name or information intothe symboltable asand whendiscovered and to efficiently lookup aname inthe symbol table to retrieve its information.

Twocommondata structuresused forthesymboltableorganizationare-

1. Linearlists:-Simpletoimplement,Poorperformance.
2. Hash tables:- Greater programming / space overhead, but, Good performance.

Ideallyacompilershouldbeableto growthesymboltabledynamically, i.e.,insert newentries or information as and when needed.

Butifthesizeofthetable isfixed inadvancethen(anarrayimplementationforexample),then the size must be big enough in advance to accommodate the largest possible program.

Foreachentryindeclarationofaname

- The formatneednot beuniformbecauseinformationdependsupontheusageofthename
- Eachentryisarecordconsistingofconsecutivewords
- Tokeeprecordsuniformsomeentriesmaybeoutsidethesymboltable

Information is entered into symbol table at various times. For example,

- keywordsareenteredinitially,
- identifierlexemesareenteredbythelexicalanalyzer.

.Symboltableentrymaybeset upwhenroleofname becomesclear,attributevaluesare filled in as information is available during the translation process.

Foreachdeclarationofaname,there isanentryinthesymboltable. Different entriesneed to store different information because of the different contexts in which a name can occur. An entrycorresponding to a particular name can be inserted into the symbol table at different stages dependingonwhentheroleofthe name becomesclear. The variousattributesthatanentryinthe symbol table can have are lexeme, type of name, size of storage and in case of functions - the parameter list etc.

Anamemaydenoteseveralobjectsinthesameblock

- intx;structx{floaty,z;}

The lexicalanalyzer returnsthe name itselfand not pointer to symboltable entry. Arecord inthe symboltableiscreatedwhenroleofthenamebecomesclear. Inthiscasetwo symboltableentries are created.

∑Aattributesofanameare entered inresponse todeclarations

∑Labelsareoften identifiedbycolon

Thesyntaxofprocedure/functionspecifiesthat certainidentifiersare formals, charactersina name.

There is a distinction between token id, lexeme and attributes of the names.

Itisdifficulttoworkwithlexemes

∑ifthereismodestupper boundonlengththenlexemescanbestoredinsymboltable

∑iflimitislargestorelexemesseparately

There might be multiple entries inthe symboltable forthe same name, allofthemhaving differentroles.Itisquiteintuitivethatthesymboltableentrieshavetobemadeonlywhenthe role of a particular name becomes clear. The lexical analyzer therefore just returns the name and not the symbol table entryas it cannot determine the context of that name. Attributes corresponding tothesymboltableareenteredforaname inresponsetothecorresponding declaration. Therehas to be an upper limit for the length of the lexemes for themto be stored in the symboltable.

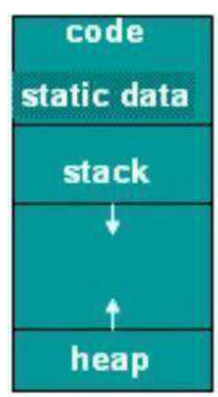
STORAGEALLOCATIONINFORMATION: Informationabout storagelocationsiskept in the symbol table.

Iftarget codeisassemblycode,thenassembler cantakecareofstorage forvariousnamesand the compiler needs to generate data definitions to be appended to assembly code

Iftarget codeis machinecode,thencompiler doestheallocation. Nostorageallocationisdone for names whose storage is allocated at runtime

Information about the storage locations that will be bound to names at run time is kept in thesymboltable. Ifthetarget isassemblycode,theassembler cantakecareofstoragefor various names. Allthecompiler hasto do istoscanthesymboltable, aftergeneratingassemblycode, and generateassemblylanguagedatadefinitionstobeappendedtotheassemblylanguageprogramfor eachname.Ifmachinecodeistobegeneratedbythecompiler,thenthepositionofeachdataobject relativetoafixedoriginmust beascertained. Thecompilerhastodothe allocationinthiscase. In the case of names whose storage is allocated on a stack or heap, the compiler does not allocate storage at all, it plans out the activation record for each procedure.

STORAGEORGANIZATION: Theruntimestoragemightbe subdivided into :



∑Targetcode,

∑Dataobjects,

∑Stacktokeeptrackofprocedureactivation,and

∑Heaptokeepallotherinformation

This kind of organization of run-time storage is used for languages such as Fortran, Pascal and C. The size of the generated target code, as well as that of some ofthe dataobjects, is known at compile time. Thus, these can be stored instaticallydeterminedareasinthememory.

STORAGEALLOCATIONPROCEDURECALLS: PascalandCusethe stack for procedure activations. Whenever a procedure is called, execution of activationgetsinterrupted,andinformationaboutthemachinestate(likeregister values) is stored on the stack.

When the called procedure returns, the interrupted activation can be restarted after restoring the saved machine state. The heap may be used to store dynamically allocated data objects, and also otherstuffsuchasactivationinformation(inthecaseoflanguageswhereanactivationtree cannot be used to represent lifetimes). Both the stack and the heap change in size during program execution,sotheycannotbeallocatedafixedamountofspace. Generallytheystartfromopposite ends of the memory and can grow as required, towards each other, until the space available has filled up.

ACTIVATION RECORD: An Activation Record is a data structure that is activated/ created when a procedure / function are invoked and it contains the following information about the function.

- ∑Temporaries:usedinexpressionevaluation
- ∑Localdata:fieldforlocaldata
- ∑Savedmachinestatus:holdsinfoaboutmachinestatusbefore procedure call
- ∑Accesslink:toaccessnonlocaldata
- ∑Controllink:pointstoactivationrecordofcaller
- ∑Actualparameters: fieldtohold actualparameters
- ∑Returnedvalue:fieldforholdingvaluetobereturned

The activation record is used to store the information required by a single procedure call. Not all the fields shown in the figure may be neededforalllanguages.Therecordstructurecanbemodifiedasperthe language/compiler requirements.

ForPascalandC,theactivationrecordisgenerallystoredontherun- time stack during the period when the procedure is executing.

Ofthefieldsshowninthefigure,accesslinkandcontrollinkareoptional(e.g.FORTRANdoesn't need access links). Also, actual parameters and return values are often stored in registers instead of the activation record, for greater efficiency.

- ∑Theactivationrecordforaprocedurecallisgeneratedbythecompiler. Generally, all field sizes can be determined at compile time.

However,thisisnotpossible inthecaseofaprocedurewhichhasalocalarraywhosesizedepends on a parameter. The strategies used for storage allocation in such cases will be discussedin forth coming lines.

STORAGEALLOCATIONSTRATEGIES: Thestorageisallocatedbasicallyinthefollowing THREE ways,

∑Staticallocation:laysoutstorageatcompiletimeforalldataobjects
∑Stackallocation:managestheruntimestorageasastack
∑Heapallocation:allocatesandde-allocatesstorageasneededatruntimefromheap

These represent the different storage-allocation strategies used in the distinct parts of the run-time memoryorganization(as shown inslide 8). We willnow look atthe possibilityofusing these strategies to allocate memory for activation records. Different languages use different strategies for this purpose. For example, old FORTRAN used static allocation, Algol type languages use stack allocation, and LISP type languages use heap allocation.

STATIC ALLOCATION: Inthisapproach memoryisallocated statically. So,Namesare bound to storage as the program is compiled

∑Noruntimesupportisrequired
∑Bindingsdonotchangeatruntime
∑Oneveryinvocationofprocedure namesareboundtothe samestorage
∑Valuesoflocalnamesare retainedacrossactivationsofaprocedure

These are the fundamental characteristics of static allocation. Since name binding occurs during compilation, there is no need for a run-time support package. The retention oflocal name values across procedure activations means that when control returns to a procedure, the values of the localsarethesameastheywerewhencontrollastleft.Forexample,supposewehadthe following code, written in a language using static allocation:

```
functionF()
 {
     int a;
     print(a);
     a = 10;
 }
```

Aftercalling F()once, ifit wascalledasecondtime, thevalueofawould initiallybe10,andthis is what would get printed.

The type of a name determines its storage requirement. The address for this storage is an offset fromtheprocedure'sactivationrecord,andthecompilerpositionstherecordsrelativetothetarget code and to one another (on some computers, it may be possible to leave thisrelative position unspecified, and let the link editor link the activation records to the executable code). After this position has been decided, the addresses of the activation records, and hence of the storage for eachname inthe records,are fixed. Thus, at compile time, the addressesat which the target codecanfind thedatait operatesuponcanbe filled in. Theaddressesat which information is to be saved whena procedure calltakes place are also knownat compile time. Static allocation does have some limitations.

- Sizeofdataobjects,aswellasanyconstraintsontheirpositionsinmemory, must be available at compile time.
- Norecursion, becauseallactivationsofagivenprocedureusethesame bindingsfor local names.
- Nodynamicdatastructures,sincenomechanismisprovidedforruntimestorageallocation.

STACK ALLOCATION: Figure shows the activation records that are pushed onto and popped for the run time stack as the control flows through the given activation tree.

First the procedure is activated. Procedure readarray 's activation is pushed onto the stack, when thecontrolreachesthefirst line intheproceduresort.Afterthecontrolreturnsfromtheactivation ofthe readarray, its activation is popped. Inthe activation ofsort ,the controlthen reaches a call of qsort with actuals 1 and 9 and an activation of qsort is pushed onto the top of thestack. In the last stage the activations for partition (1,3) and qsort (1,0) have begun and ended during the life time of qsort (1,3), so their activation records have come and gone from the stack, leaving the activation record for qsort (1,3) on top.

CALLINGSEQUENCES:Acallsequenceallocatesanactivationrecordandentersinformation into its field. A return sequence restores the state of the machine so that calling procedure can continue execution.

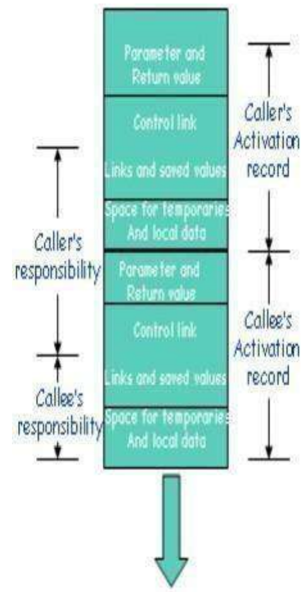
Callingsequenceandactivationrecordsdiffer,evenforthesamelanguage.Thecodeinthecalling sequence is often divided between the calling procedure and the procedure it calls.

Thereisnoexactdivisionofruntimetasksbetweenthecaller and the colleen.

Asshowninthefigure,theregisterstacktoppointstotheend of the machine status field in the activation record.

This position is known to the caller, so it can be made responsible for setting up stack top before control flows to the called procedure.

ThecodefortheCalleecanaccess itstemporariesandthe local data using offsets from stack top.



CallSequence: Inacallsequence,followingsequenceofoperationsisperformed.

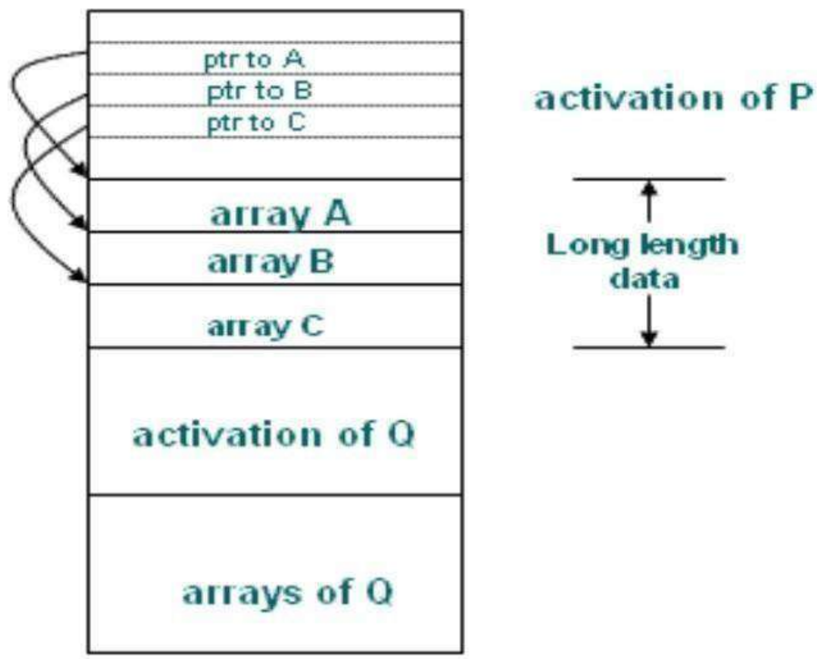
- ∑Callerevaluatestheactualparameters
- ∑Caller storesreturnaddressandothervalues(controllink)intocallee‘sactivationrecord
- ∑Calleesavesregistervaluesandother statusinformation
- ∑Calleeinitializesitslocaldataandbeginsexecution

The fields whose sizes arefixed early are placedin the middle. The decision of whether or not to usethe controland access links is part ofthe design of the compiler, so these fields can be fixed at compiler constructiontime. Ifexactlythe same amount ofmachine-status information issaved foreachactivation,thenthesamecodecandothesavingandrestoring forallactivations. Thesizeoftemporaries may not beknowntothe front end. Temporariesneeded bytheprocedure may be reduced by careful code generation or optimization. This field is shown after that for the local data. The caller usually evaluates the parameters and communicates themto the activation recordofthe callee. Inthe runtime stack, the activation recordof the calleris just below that for the callee. The fields for parameters and a potential return value are placed next to the activation record of the caller. The caller can then access these fields using offsets from the end of its own activation record. In particular, there is no reason for the caller to know about the local data or temporaries of the callee.

ReturnSequence: Inareturnsequence,followingsequenceofoperationsareperformed.

∑Calleeplacesareturnvaluenext toactivationrecordofcaller
∑Restoresregistersusinginformationinstatusfield
∑Branchtoreturnaddress
∑Callercopiesreturnvalueintoitsownactivationrecord

As described earlier, in the runtime stack, the activation record of the caller is just below that for the callee. The fields for parameters and a potential return value are placed next to the activation record of the caller.The caller can then access thesefields using offsets from the end of its own activation record. The caller copies the return value into its own activation record. In particular,thereisno reasonforthecallertoknowaboutthelocaldataortemporariesofthe callee. The given calling sequence allows the number ofarguments ofthe called procedureto depend on the call. At compile time, the target code of the caller knows the number of arguments it is supplying to the callee. The caller knows the size of the parameter field. The target code of the called must be prepared to handle other calls as well, so it waits until it is called, then examines the parameter field. Information describing the parameters must be placed next to the status field so the callee can find it.



LongLengthData:

The procedure P has three local arrays. The storage for these arrays is not part of the activation record for P; only a pointer to the beginning of each array appears in the activation record. The relative addresses ofthese pointers are known at the compile time, so the target code can access array elements through the pointers. Also shown is the procedure Q called by P . The activation record for Q begins after the arrays of P. Access to data on the stack is through two pointers, top and stack top. The first ofthese marks the actualtopofthe stack; it points to the positionat whichthe next activation record begins. The second is used to find the local data. For consistencywiththe organizationofthe figure inslide 16, supposethe stacktop pointstothe end ofthemachinestatusfield.Inthisfigurethestacktoppointstotheendofthisfield inthe activation recordfor Q. Within the field isacontrollink tothepreviousvalueofstacktopwhencontrolwas incalling activationofP. The codethat repositions top and stacktopcanbe generated at compile time, using the sizesofthe fields in the activationrecord. Whenq returns, the new value oftopis stacktopminus the lengthofthe machine statusandthe parameter fields inQ's activationrecord. This length is knownat the compile time, at least to the caller. After adjusting top,the new value of stack top can be copied from the control link of Q.

DanglingReferences: Referringto locationswhichhave beende-allocated.

```
void main()
{
   int*p;
   p=dangle();/*danglingreference*/
}

int*dangle();
{
   int i=23;
   return&i;
}
```

Theproblemofdanglingreferencesarises,wheneverstorageisde-allocated.Adanglingreference occurs when there is a reference to storage that has been de-allocated. It is a logical error to use danglingreferences,sincethevalueofde-allocatedstorageisundefinedaccordingtothesemantics of most languages. Since that storage may later be allocated to another datum, mysterious bugs can appear in the programs with dangling references.

HEAP ALLOCATION: Ifa procedure wantstoput avalue that is to be used after its activation is over then we cannot use stack for that purpose. That is language like Pascal allows data to be allocatedunderprogramcontrol.Also incertainlanguageacalledactivationmayoutlivethecaller procedure. Insucha case last-in-first-out queuewillnot workand wewillrequire a data structure likeheaptostoretheactivation.Thelast caseisnottrueforthoselanguageswhoseactivationtrees correctly depict the flow of control between procedures.

LimitationsofStackallocation:It cannotbeusedif,

- Thevaluesofthelocalvariablesmustberetainedwhenanactivationends
- Acalledactivationoutlivesthecaller

∑Insucha casede-allocationofactivationrecordcannotoccurin last-infirst-outfashion
∑Heap allocationgivesoutpiecesofcontiguousstorageforactivationrecords

Therearetwo aspectsofdynamicallocation-:

- Runtimeallocationand de-allocationofdata structures.
- Languages like Algolhavedynamicdatastructuresand it reservessomepartofmemory for it**.**

Initializing data-structures may require allocating memory but where to allocate this memory. After doingtype inferencewe haveto dostorageallocation. It willallocatesomechunk of bytes. But in language like LISP, it will try to give continuous chunk. The allocation in continuous bytes may lead to problem of fragmentation i.e. you may develop hole in process of allocation and de-allocation. Thus storage allocation of heap may lead us with many holes and fragmentedmemorywhichwillmakeithardtoallocatecontinuouschunkofmemorytorequesting program.So,wehave heap mangerswhichmanagethefreespaceandallocationandde-allocation ofmemory. It would beefficient to handle smallactivationsand activationsofpredictablesizeas a specialcase as described in the next slide. The various allocation and de- allocationtechniques used will be discussed later.

Fillarequestofsize swithblock ofsize s'wheres'isthesmallestsizegreaterthanorequaltos

- Forlargeblocksofstorageuseheapmanager
- Forlarge amount ofstoragecomputation maytakesometime to use upmemoryso that time taken by the manager may be negligible compared to the computation time

Asmentionedearlier,forefficiencyreasonswecanhandlesmallactivationsandactivationsof predictable size as a special case as follows:

1. Foreachsizeofinterest,keepalinkedlistiffreeblocksofthatsize

2. If possible, fill a request for size s with a block of size s', where s' is the smallest size greater thanorequaltos.Whentheblockiseventuallyde-allocated, itisreturnedtothelinked list it came from.

3. Forlargeblocksofstorageusetheheapmanger.

Heapmangerwilldynamicallyallocate memory. Thiswillcomewitharuntimeoverhead. Asheapmanagerwillhavetotakecareofdefragmentationandgarbagecollection. Butsinceheap manger saves space otherwise we will have to fix size of activation at compile time, runtime overhead is the price worth it.

ACCESSTONON-LOCALNAMES:

Thescoperulesofa languagedecide howtoreferencethenon-localvariables. Therearetwo methods that are commonly used:

1. StaticorLexicalscoping:Itdeterminesthedeclarationthat appliesto anamebyexamining the program text alone. E.g., Pascal, C and ADA.
2. DynamicScoping:Itdeterminesthedeclarationapplicabletoanameat runtime,by considering the current activations. E.g., Lisp

ORGANIZATIONFORBLOCKSTRUCTURES:

Ablock isaanysequenceofoperationsorinstructionsthat areusedtoperforma[sub] task.In any programming language,

∑Blockscontainits ownlocaldatastructure.

∑Blockscanbenestedandtheir starting andendsaremarkedbyadelimiter.

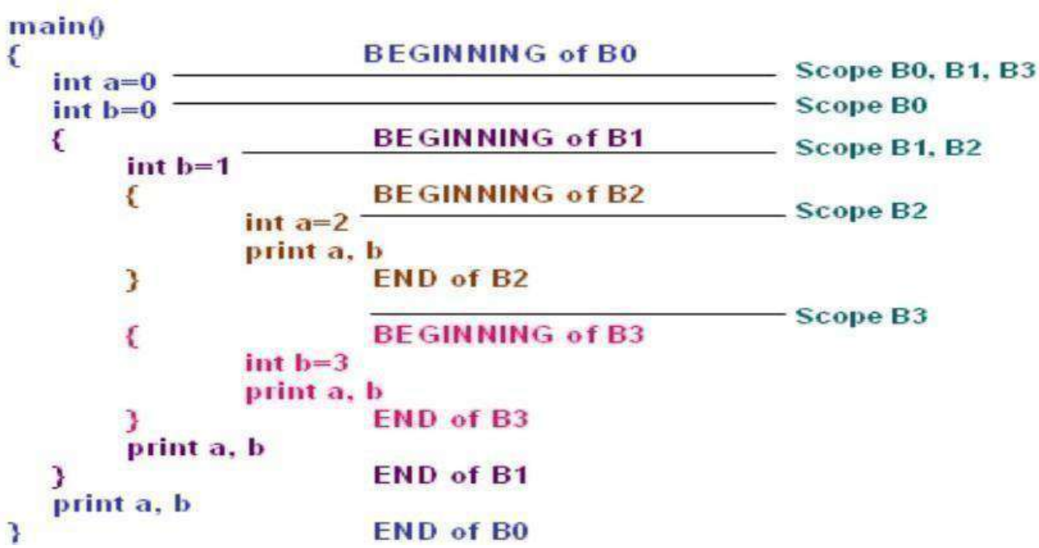
∑ They ensure that either block is independent of other or nested in another block. Thatis,it isnotpossiblefortwoblocksB1andB2tooverlapinsuchawaythatfirstblockB1begins, then B2, but B1 end before B2.

∑This nestingpropertyiscalledblockstructure.Thescopeofadeclaration inablock-structured language is given by the most closely nested rule:

1. Thescopeofadeclaration inablock BincludesB.

2. Ifaname Xis notdeclaredin a block B, then an occurrence of Xin B isin the scope ofa declarationofX inanenclosing block B 'suchthat. B'has a declarationofX, and. B' is more closely nested around B then anyother block with a declaration ofX.

Forexample, considerthefollowingcodefragment.



For the example, in the above figure, the scope of declaration of b in B0 does not include B1 because b is re-declared in B1. We assume that variables are declared before the first statementin which they are accessed. The scope of the variables will be as follows:

DECLARATION	SCOPE
inta=0	B0notincludingB2
intb=0	B0notincludingB1
intb=1	B1notincludingB3
inta=2	B2 only
intb=3	B3 only

Theoutcomeoftheprintstatementwillbe,therefore:

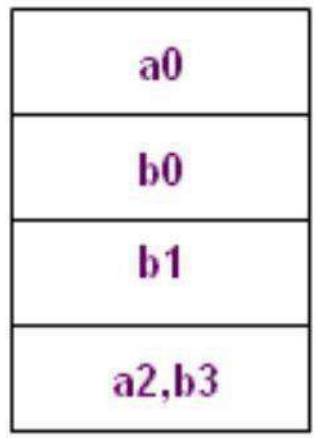
21
03
01
00

Blocks:.Blocksaresimplertohandlethanprocedures

.Blockscanbetreatedasparameterlessprocedures

.Usestackformemoryallocation

.Allocatespacefor completeprocedurebodyatonetime

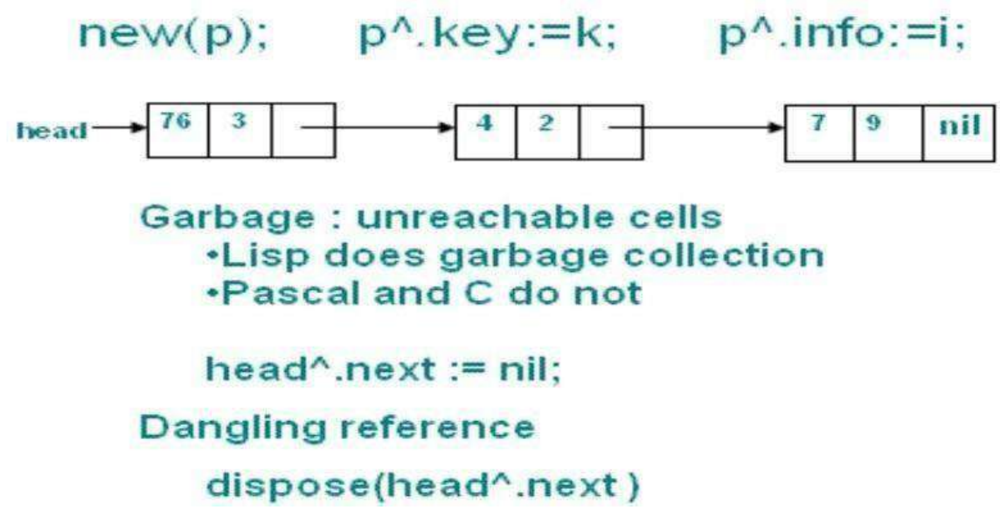


Therearetwomethodsofimplementingblockstructureincompilerconstruction:

1. **STACKALLOCATION:**Thisisbasedontheobservationthat scopeofadeclarationdoesnot extend outside the block in which it appears, the space for declared name can be allocated when the block is entered and de-allocated when controls leave the block. The view treat blockas a "parameter less procedure" called only fromthe point just before the block and returning onlyto the point just before the block.

2. **COMPLETE ALLOCATION:** Here you allocate the complete memory at one time. If there are blocks within the procedure, then allowance is made for the storage needed for declarations withinthe books.Iftwo variables are never alive at the same time and are at same depththeycan be assigned same storage.

DYNAMICSTORAGEALLOCATION:

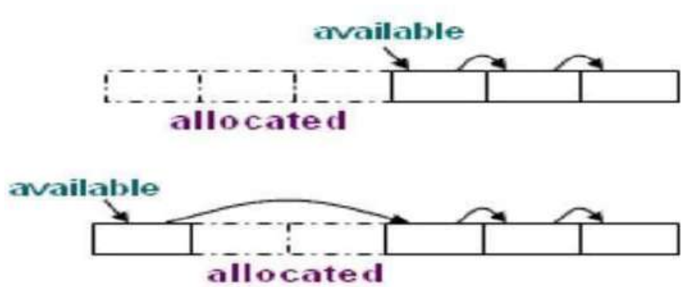


GenerallylanguageslikeLispandMLwhichdo notallow forexplicit de-allocationofmemorydo garbage collection. Areference to apointerthat isno longer valid is called a'danglingreference'. For example, consider this C code:

```
intmain(void)
  {
        int*a=fun();
  }
int* fun()
  {
        int a=3;
        int*b=&a;
        return b;
  }
```

Here, the pointer returned by fun() no longer points to a valid address in memory as the activation of fun() has ended. This kind of situation is called a 'dangling reference'. In case of explicitallocationit is more likelytohappenastheusercande-allocateanypartofmemory, even something that has to a pointer pointing to a valid piece of memory.

InExplicit AllocationofFixed Sized Blocks, Linktheblocks ina list ,and Allocationand de-allocation can be done with very little overhead.



The simplest formofdynamic allocation involves blocks ofa fixed size. By linking the blocks in a list, as shown in the figure, allocation and de-allocation can be done quickly with little or no storage overhead.

ExplicitAllocationof FixedSizedBlocks: Inthisapproach,blocksaredrawnfrom contiguous area ofstorage, and an area ofeach block is used as pointer to the next block

- ∑Thepointer availablepointstothefirstblock
- ∑Allocationmeansremovingablockfromtheavailablelist
- ∑De-allocation meansputtingtheblockintheavailablelist
- ∑Compilerroutinesneednotknowthetype ofobjectsto beheldintheblocks
- ∑Eachblockistreatedasavariantrecord

Supposethat blocksareto bedrawnfromacontiguousareaofstorage.Initializationofthe areaisdonebyusingaportionofeachblockforalinktothenext block. Apointeravailablepoints to the first block. Generally a list of free nodes and a list of allocated nodes is maintained, and whenever a new block has to be allocated, the block at the head of the free list is taken off and allocated (added tothe list ofallocated nodes). Whena node has to be de-allocated, it is removed from the list of allocated nodes by changing the pointer to it in the list to point to the block previously pointed to by it, and then the removed block is added to the head of the list of free blocks.Thecompiler routinesthatmanage blocksdo notneedtoknowthetypeofobject thatwill beheldintheblock bytheuser program. These blockscancontainanytypeofdata (i.e.,theyare used as generic memory locations by the compiler). We can treat each block as a variant record, with the compiler routines viewing the block as consisting of some other type. Thus, there is no spaceoverhead becausetheuser programcanusetheentireblock for itsownpurposes. Whenthe block is returned, then the compiler routines use some ofthe space fromthe block itselfto link it into the list ofavailable blocks, as shown in the figure in the last slide.

ExplicitAllocationofVariableSizeBlocks:

Limitations of Fixed sized block allocation: In explicit allocation of fixed size blocks, internal fragmentation canoccur,that is, the heap mayconsist ofalternate blocks that arefree and in use, as shown in the figure.

Thesituationshowncanoccur ifaprogramallocates five blocksandthende-allocatesthesecond and the fourth, for example.

Fragmentation is of no consequence if blocks are of fixed size, but if theyare of variable size, a situation like this is a problem, because we could not allocate a block larger than any one of the free blocks, even though the space is available in principle.

So, ifvariable- sized blocks are allocated,then internalfragmentationcanbe avoided, as weonly allocate as much space as we need in a block. But this creates the problem of external fragmentation, where enough space is available in total for our requirements, but not enough spaceisavailable incontinuousmemorylocations,asneeded forablockofallocatedmemory. For example, consider another case where we need to allocate 400 bytes of data for the next request, and theavailablecontinuousregionsofmemorythat wehaveareofsizes300, 200and 100bytes. So we have a total of 600 bytes, which is more than what we need. But still we are unable to allocate the memory as we do not have enough contiguous storage.

Theamountofexternalfragmentationwhileallocatingvariable-sizedblockscanbecomeveryhigh on using certain strategies for memory allocation.

Sowetrytousecertainstrategiesformemoryallocation,sothatwecanminimizememorywastage due to external fragmentation. These strategies are discussed in the next few lines.

.Storagecanbecomefragmented,Situation mayarise,Ifprogramallocatesfiveblocks
.thende-allocatessecond andfourthblock

IMPORTANT QUESTIONS:

1. Whatarecallingsequence,andReturnsequences?Explainbriefly.
2. WhatisthemaindifferencebetweenStatic&Dynamicstorageallocation?Explainthe problems associated with dynamic storage allocation schemes.
3. What istheneedofadisplayassociatedwithaprocedure?Discusstheproceduresfor maintaining the display when the procedures are not passed as parameters.
4. Writenotesonthestaticstorageallocationstrategywithexampleanddiscuss its limitations?
5. Discussaboutthestackallocationstrategyofruntimeenvironmentwithanexample?
6. Explaintheconceptofimplicitdeallocationofmemory.
7. Giveanexampleofcreating danglingreferencesandexplain howgarbageiscreated.

ASSIGNMENTQUESTIONS:

1. Whatisacallingsequence?Explain briefly.
2. Explaintheproblemsassociatedwithdynamicstorageallocationschemes.
3. ListandexplaintheentriesofActivationRecord.
4. Explainaboutparameterpassing mechanisms.

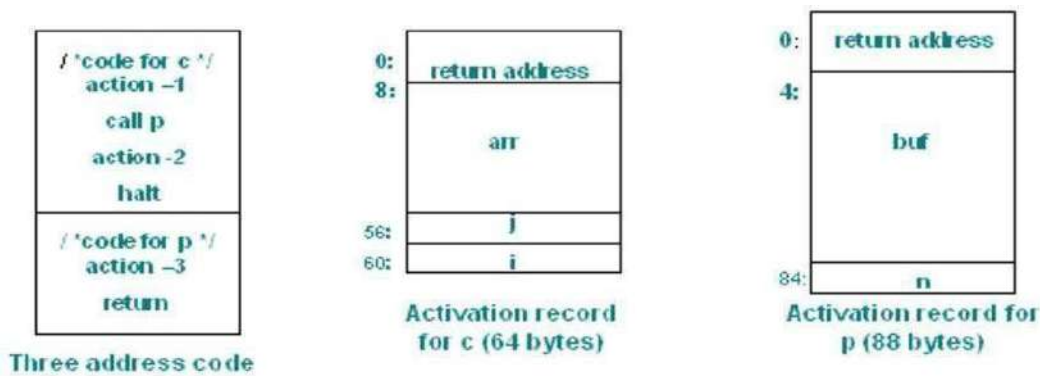
UNIT-IV

RUNTIMESTORAGEMANAGEMENT:

Tostudytherun-timestoragemanagementsystemitissufficienttofocusonthestatements:action, call,returnandhalt,becausetheybythemselvesgiveussufficient insight intothebehaviorshown by functions in calling each other and returning.

And the run-time allocation and de-allocation of activations occur on the call of functions and when they return.

There are mainly two kinds of run-time allocation systems: **Static allocation** and **Stack Allocation**. While static allocation is used bythe FORTRAN class of languages, stack allocation is used by the Ada class of languages.



<u>STATICALLOCATION</u>: Inthis,Acallstatement isimplementedbyasequenceoftwo instructions.

∑Amoveinstructionsavesthereturnaddress
∑Agototransfers controltothetargetcode.

The instruction sequence is

MOV#here+20,callee.static-area

GOTO callee.code-area

callee.static-areaandcallee.code-areaareconstantsreferringtoaddressoftheactivationrecord and the first address of called procedure respectively.

.#here+20 inthe move instructionisthereturnaddress;theaddressofthe instructionfollowing the goto instruction

.Areturnfromprocedurecallee is implementedby

GOTO *callee.static-area

Forthecallstatement, weneedto savethereturnaddresssomewhereand thenjumptothe locationofthecallee function. Andtoreturnfroma function, wehaveto accessthereturnaddress as stored byits caller, and then jump to it. So for call, we first say: MOV #here+20, callee.static- area. Here, #here refers to the location ofthe current MOV instruction, and callee.static- area is a fixed location in memory. 20 is added to #here here, as the code corresponding to the call instruction takes 20 bytes (at 4 bytes for each parameter: 4*3 for this instruction, and 8 for the next). Thenwe sayGOTO callee. code-area,totake usto the codeofthecallee,ascallee.codearea is merely the address where the code of the callee starts. Then a return from the callee is implemented by:GOTO*callee.staticarea. Notethat thisworksonlybecausecallee.static-area is a constant.

Example:

.Assumeeach	100:ACTION-l
action	120: MOV140, 364
blocktakes 20	132:GOTO200
bytesofspace	140:ACTION-2
.Startaddress	160:HALT
ofcodeforc	:
andpis	200:ACTION-3
100and200	220:GOTO*364

.The activation	:
Records	300:
arestatically	304:
allocatedstarting	:
ataddresses	364:
300and364.	368:

Thisexamplecorrespondstothecodeshowninslide57.Staticallywesaythatthecodefor c starts at 100 and that for p starts at 200. At some point, c calls p. Using the strategy discussed earlier,andassumingthatcallee.staticareaisatthememorylocation364,wegetthecodeasgiven. Here we assume that a call to 'action' corresponds to a single machine instruction which takes 20 bytes.

STACK ALLOCATION :.Positionoftheactivationrecordisnotknownuntilruntime

- ∑. Positionisstoredinaregisteratruntime, and wordsintherecordareaccessedwithan offset from the register
- ∑. Thecodeforthefirst procedureinitializesthestackbysettingupSPtothestartofthe stack area

MOV#Stackstart, SP

codeforthefirstprocedure

HALT

In stack allocation we do not need to know the position ofthe activation record until run-time. This gives us an advantage over static allocation, as we can have recursion. So this is used in many modern programming languages like C, Ada, etc. The positions of the activations are stored in the stack area, and the position for the most recent activation is pointed to bythe stack pointer. Words in a record are accessed with an offset from the register. The code for the first procedureinitializesthestackbysettingupSPtothestackareabythe followingcommand: MOV #Stackstart, SP. Here, #Stackstart is the location in memory where the stack starts.

Aprocedurecallsequence incrementsSP,savesthereturnaddressandtransferscontroltothe called procedure

ADD#caller.recordsize,SP

MOVE #here+ 16, *SP

GOTO callee.code_area

Consider the situation when a function (caller) calls the another function(callee), then procedure call sequence increments SP by the caller record size, saves the return address and transfers control to the callee by jumping to its code area. In the MOV instruction here, we only need to add 16, as SP is a register, and so no space is needed to store *SP. The activations keep getting pushed on the stack, so #caller.recordsize needs to be added to SP, to update the value of SPtoitsnewvalue. Thisworksas#caller.recordsizeisaconstant forafunction,regardlessofthe particular activation being referred to.

DATASTRUCTURES: Followingdatastructuresareusedtoimplementsymboltables

LISTDATASTRUCTURE:Couldbeanarraybasedorpointerbased list. Butthis implementation is

- Simplesttoimplement
- Useasingle arraytostorenamesandinformation
- Searchforanameislinear
- Entryandlookupareindependentoperations
- Costofentryandsearchoperationsareveryhighandlotoftimegoesintobookkeeping

Hashtable: Hashtable isadatastructurewhichgivesO(1)performance inaccessingany element of it. It uses the features of both arrayand pointer based lists.

-Theadvantagesareobvious

REPRESENTINGSCOPEINFORMATION

Theentries inthesymboltableare for declarationofnames. Whenanoccurrenceofa nameinthe sourcetextislookedupinthesymboltable,theentryfortheappropriatedeclaration, accordingto the scoping rules of the language, must be returned. A simple approach is to maintain a separate symbol table for each scope.

Mostcloselynestedscoperulescanbe implementedbyadaptingthedatastructuresdiscussed in the previous section. Each procedure is assigned a unique number. If the language isblock-structured,theblocks must also beassigneduniquenumbers.Thename isrepresentedasa pairof a number and a name. This new name is added to the symbol table. Most scope rules can be implemented in terms of following operations:

a) Lookup-findthemostrecentlycreatedentry.
b) Insert-makeanewentry.
c) Delete-removethemostrecentlycreated entry.
d) Symboltable structure
e) .Assignvariablestostorageclassesthatprescribescope,visibility, andlifetime

f) - scoperulesprescribe the symboltablestructure
g) -scope:unitofstaticprogramstructurewithoneormore variabledeclarations
h) -scopemaybe nested
i) .Pascal:proceduresarescopingunits
j) .C:blocks,functions,filesarescopingunits
k) .Visibility,lifetimes,globalvariables
l) . Common(inFortran)
m) . Automatic orstackstorage
n) .Staticvariables
o) **storageclass:**Astorageclass isanextrakeywordatthebeginningofadeclarationwhich modifiesthedeclarationinsomeway.Generally,thestorageclass(ifany) isthe first word in the declaration, preceding the type name. Ex. static, extern etc.
p) Scope:Thescopeofavariable issimplythepartoftheprogramwhere itmaybeaccessed orwritten.It isthepartoftheprogramwherethe variable's name maybeused.Ifavariable is declared within a function, it is localtothatfunction. Variables ofthe same name may be declared and used within other functions without any conflicts. For instance,
q)

```
intfun1()
{
   inta;
   intb;
   ....
}

intfun2()
{
   inta;
   intc;
   ....
}
```

Visibility: The visibility of a variable determines how much of the rest of the program canaccessthat variable.Youcanarrangethatavariable isvisibleonlywithinonepartof one function, or in one function, or in one source file, or anywhere in the program.

r) **Local and Global variables:** A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called local variables.Ontheotherhand,avariabledeclaredoutsideofanyfunctionisaglobalvariable ,anditispotentiallyvisibleanywherewithintheprogram.

s) **Automatic Vs Static duration:** How long do variables last? By default, local variables (thosedeclaredwithinafunction)haveautomaticduration:theyspringintoexistencewhen thefunctioniscalled,andthey(andtheirvalues)disappearwhenthefunction returns. Global variables, onthe other hand, have static duration: they last, and the values storedinthempersist,foraslongastheprogramdoes.(Ofcourse,thevaluescaningeneral still be overwritten, so they don't necessarily persist forever.) By default, local variables haveautomaticduration.Togivethemstaticduration(sothat,insteadofcomingandgoing as the function is called, they persist for as long as the function does), you precede their declaration with the static keyword: static int i; By default,a declaration of a global variable (especially if it specifies an initial value) is the defining instance. To make it an externaldeclaration,ofavariablewhichisdefinedsomewhereelse, youprecedeit withthe keywordextern:externint j;Finally,to arrangethataglobalvariable isvisibleonlywithin its containing source file, you precede it with the static keyword: static int k; Notice that the static keyword can do two different things: it adjuststhe duration of a local variable fromautomatic to static, orit adjusts the visibilityofa global variable fromtrulyglobalto private-to-the-file.

t) Symbolattributesandsymboltableentries
u) Symbolshaveassociatedattributes
v) Typicalattributesarename,type,scope,size,addressingmodeetc.
w) Asymboltableentrycollectstogether attributessuchthattheycanbeeasilyset and retrieved
x) Exampleoftypicalnamesinsymboltable

Name	Type
name	characterstring
class	enumeration
size	integer
type	enumeration

LOCALSYMBOLTABLEMANAGEMENT:

Followingareprototypesoftypicalfunctiondeclarationsused formanaging localsymboltable. The right hand side ofthe arrows is the output ofthe procedure and the left side has the input.

NewSymTab : SymTab ⟶SymTab
DestSymTab : SymTab ⟶SymTab
InsertSym : SymTab X Symbol ⟶boolean
LocateSym:SymTabXSymbol⟶boolean
GetSymAttr : SymTab X Symbol X Attr ⟶boolean
SetSymAttr:SymTabXSymbolXAttrXvalue⟶boolean
NextSym : SymTab X Symbol ⟶Symbol
MoreSyms:SymTabXSymbol⟶boolean

Amajorconsiderationindesigningasymboltable isthat insertionandretrievalshouldbeasfast as possible
.Onedimensionaltable:searchisveryslow

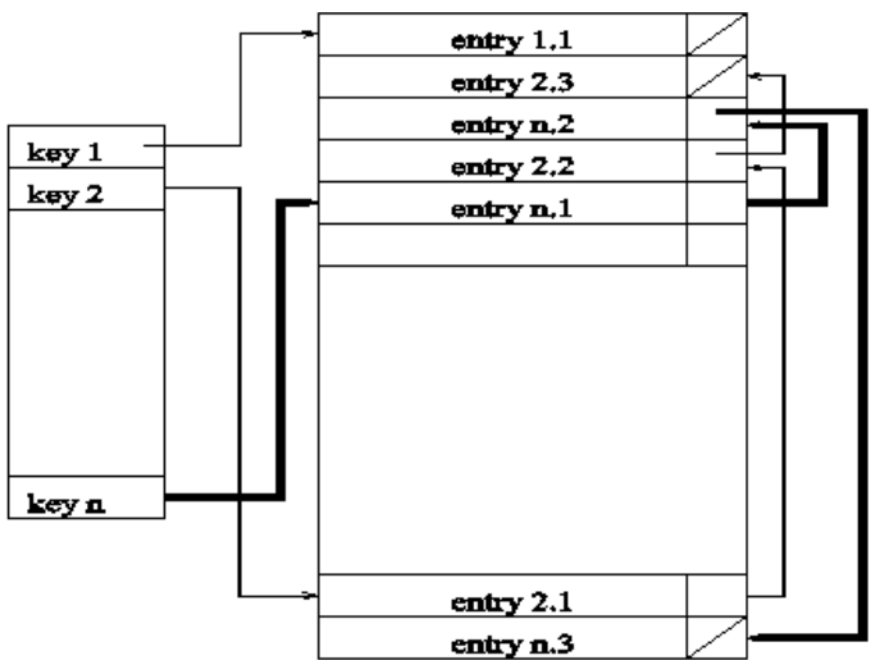
.Balancedbinarytree:quick insertion, searchingandretrieval;extraworkrequiredtokeepthe tree balanced

.Hashtables:quickinsertion,searchingandretrieval;extraworktocomputehashkeys

.Hashing withachainofentriesisgenerallyagood approach

Amajor considerationindesigningasymboltable isthat insertionandretrievalshould be as fast as possible. We talked about theone dimensionaland hashtables a few slides back. Apart fromthese balanced binarytrees can be used too. Hashing is the most common approach.

HASHEDLOCALSYMBOLTABLE



Hash tables can clearly implement 'lookup' and 'insert' operations. For implementing the 'delete', we do not want to scan the entire hash table looking for lists containing entries to be deleted. Each entry should have two links:

a) Ahashlinkthat chainstheentrytoother entrieswhosenameshashtothesame value-the usual link in the hash table.

b) A scope link that chains all entries in the same scope - an extra link. If the scope link is left undisturbedwhenanentryisdeletedfromthehashtable,thenthechainformedbythescope links will constitute an inactive symbol table for the scope in question.

NestingstructureofanexamplePascalprogram

```
program e;
  var a, b, c: integer;

  procedure f;
    var a, b, c: integer;
    begin
      a := b+c
    end;

  procedure g;
    var a, b: integer;

    procedure h;
      var c, d: integer;
      begin
        c := a+d
      end;

    procedure i;
      var b, d: integer;
      begin
        b:= a+c
      end;

  procedure j;
    var b, d: integer;
    begin
      b := a+d
    end;

  begin
    a := b+c
  end.
```

Lookatthenestingstructureofthisprogram. Variablesa,bandcappearinglobalaswell as localscopes. Localscopeofa variable overrides the globalscopeoftheother variable withthe same name within its own scope. The next slide will show the global as well as the localsymbol tables for this structure. Here procedure I and h lie within the scope of g (are nested within g).

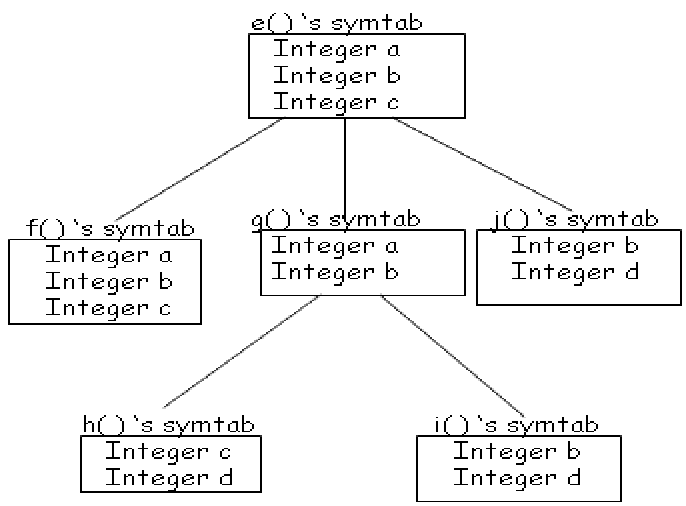
GLOBALSYMBOLTABLESTRUCTURETheglobalsymboltablewill beacollectionof symbol tables connected with pointers.

. Scope and visibility rules determine the structure of global symbol table

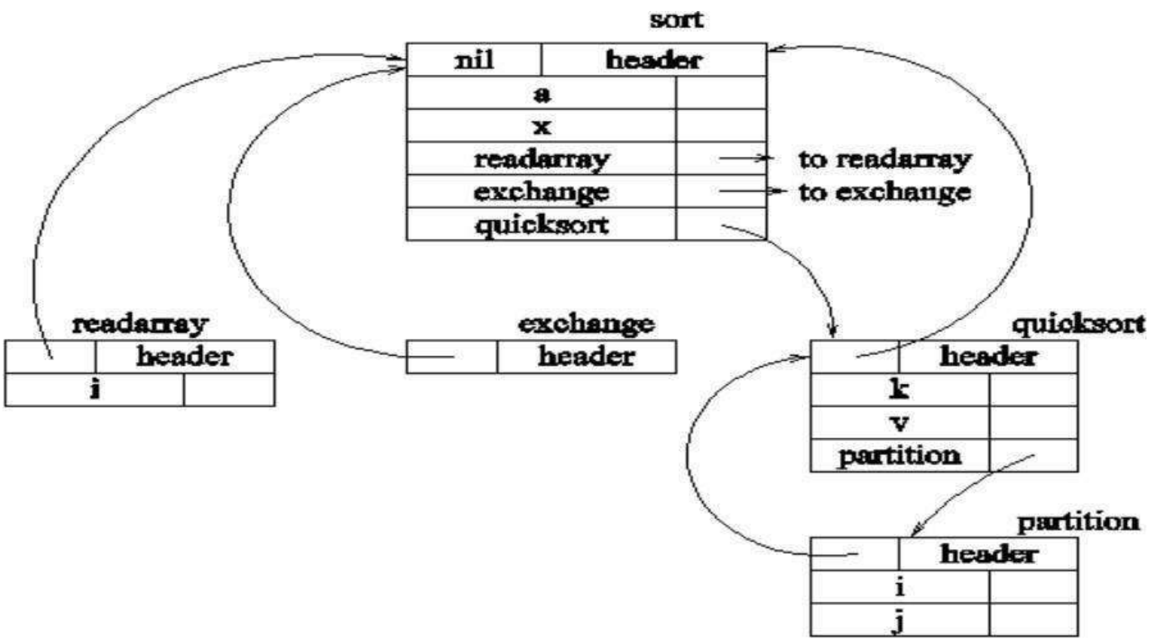
. For ALGOL class of languages scoping rules structure the symbol table as tree of local tables

- Globalscopeasroot

- Tables for nested scope as children of the table for the scope they are nested in



Theexactstructurewillbedeterminedbythescopeandvisibilityrulesofthelanguage.The global symbol table will be a collection of symbol tables connected with pointers. The exact structure will be determined by the scope and visibility rules of the language. Whenever a new scope isencountered a new symboltable is created. This new table containsa pointer back tothe enclosing scope's symbol table and the enclosing one also contains a pointerto this new symbol table. Anyvariable used inside the new scope should either be present in its own symboltable or inside the enclosing scope's symbol table and all the way up to the root symbol table. A sample global symbol table is shown in the below figure.



BLOCK STRUCTURESANDNONBLOCKSTRUCTURESTORAGEALLOCATION

Storage bindingand symbolicregisters : Translatesvariablenamesintoaddressesandthe process must occur before or during code generation

- .Eachvariableisassigned anaddressoraddressingmethod
- .Eachvariable isassignedanoffset withrespecttobasewhichchangeswithevery invocation
- .Variablesfallinfourclasses:global,globalstatic,stack,local(non-stack)static
- Thevariablenameshavetobetranslatedintoaddressesbeforeorduringcodegeneration.

There isa baseaddressand everyname isgivenanoffset withrespecttothisbasewhichchanges with every invocation. The variables can be divided into four categories**:**

a) **GlobalVariables:**fixedrelocatableaddressoroffsetwithrespect tobaseasglobalpointer

b) **GlobalStaticVariables:**.Globalvariables, ontheotherhand,havestaticduration(hencealso called static variables): theylast, andthe values stored inthempersist, for as long asthe program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) Therefore they have fixed relocatable address or offset with respect to base as global pointer.

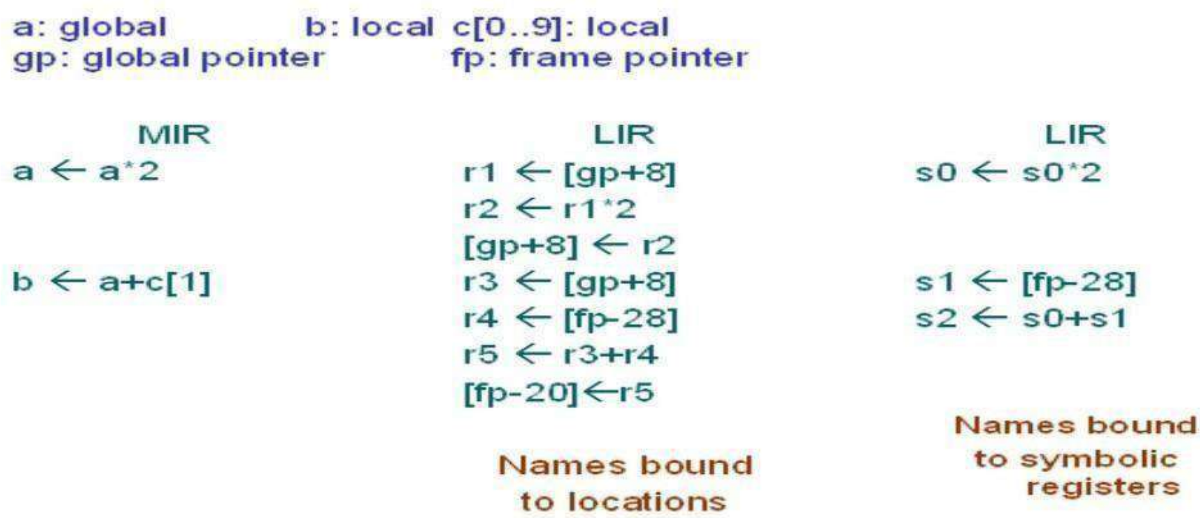
c) **Stack Variables :** allocate stack/global in registers and registers are not indexable, therefore, arrays cannot be in registers

.Assignsymbolicregisterstoscalar variables

.Usedforgraphcoloringfor globalregister allocation

d) **Stack Static Variables :** Bydefault, local variables (stack variables) (those declared within a function)haveautomaticduration:theyspring intoexistencewhenthefunctioniscalled,andthey (and their values) disappear when the function returns. This is why they are stored in stacks and have offset from stack/frame pointer.

Registerallocationisusuallydoneforglobalvariables.Sinceregistersarenotindexable,therefore, arrays cannot be in registers as they are indexed data structures. Graph coloring is a simple techniqueforallocatingregisterandminimizingregisterspillsthat workswellinpractice.Register spills occur when a register is needed for a computation but allavailable registers are inuse. The contents of one of the registers must be stored in memory to free itup for immediate use. We assign symbolic registers to scalar variables which are used in the graph coloring.



LocalVariablesinFrame

- ∑Assigntoconsecutivelocations;allowenoughspaceforeach
- ∑Mayputwordsizeobjectinhalfwordboundaries
- ∑Requirestwohalfwordloads
- ∑Requiresshift,or,and
- ∑Alignondoubleword boundaries
- ∑Wastesspace
- ∑AndMachinemayallowsmalloffsets

wordboundaries-themostsignificant byteoftheobject must be locatedatanaddresswhose two least significant bits are zero relative to the frame pointer

half-wordboundaries-themostsignificant byteoftheobject beinglocatedatanaddress whose least significant bit is zero relative to the frame pointer **.**

Sortvariablesbythealignmenttheyneed

- Storelargestvariablesfirst
- Utomaticallyalignsallthevariables
- Doesnotrequirepadding
- Storesmallestvariablesfirst
- Requiresmorespace(padding)
- Forlargestackframemakesmorevariablesaccessiblewithsmalloffsets

Whileallocatingmemorytothevariables, sort variablesbythealignmenttheyneed.Youmay:

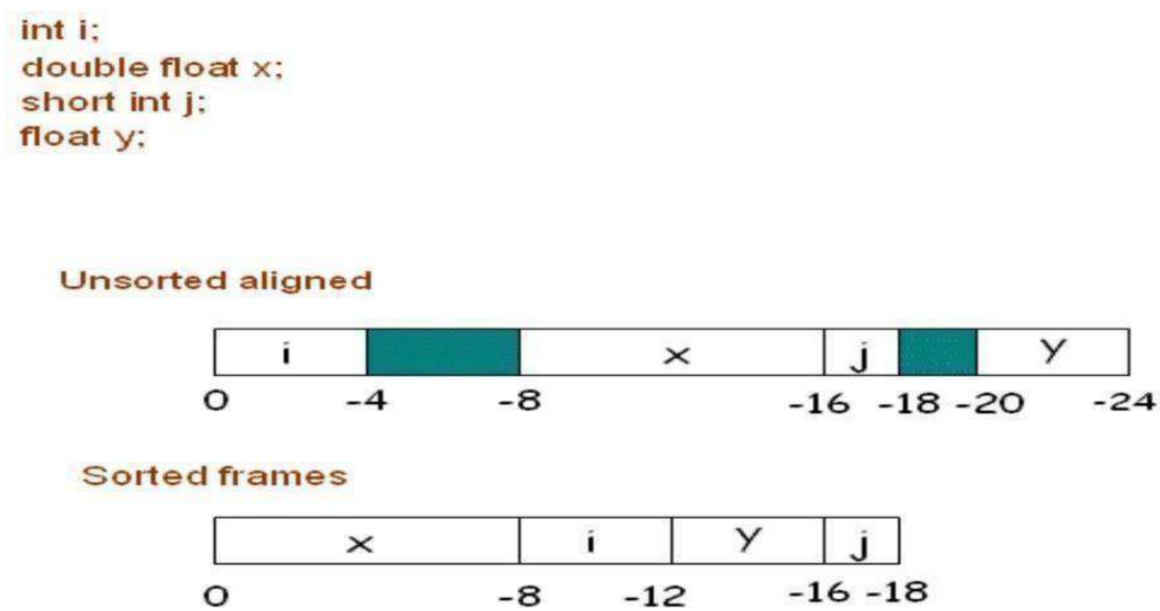
Storelargestvariablesfirst:Itautomaticallyalignsallthevariablesanddoesnotrequirepadding since the next variable's memory allocation starts at the end ofthat ofthe earlier variable

. Store smallest variables first: It requires more space (padding) since you have to accommodate forthebiggest possible lengthofanyvariabledatastructure.Theadvantage isthat for largestack frame, more variables become accessible within small offsets

Howtostorelargelocaldatastructures?BecausetheyRequires largespace inlocalframesand therefore large offsets

- Iflargeobjectisput neartheboundaryotherobjectsrequire largeoffset either fromfp(if put near beginning) or sp (if put near end)
- Allocateanother baseregistertoaccesslargeobjects
- Allocatespaceinthe middleorelsewhere;storepointertothese locations fromat asmall offset from fp
- Requiresextraloads

Large local data structures require large space in local frames and therefore large offsets. Astoldinthepreviousslide'snotes,iflargeobjectsareputneartheboundarythentheotherobjects require large offset. You can either allocate another base register to access large objectsor you can allocate space in the middle or elsewhere and then store pointers to these locations starting from at a small offset from the frame pointer, fp.



Intheunsortedallocationyoucanseethewasteofspace ingreen. Insortedframethere isno waste of space.

STORAGEALLOCATIONFORARRAYS

Elementsofanarrayarestoredinablockofconsecutive locations. Forasingledimensionalarray, if low is the lower bound of the index and base is the relative address of the storage allocated to thearrayi.e.,therelativeaddressofA[low],thentheithElementsofanarrayare storedinablock of consecutive locations

Forasingledimensionalarray,iflowisthelowerboundoftheindexandbaseistherelative address of the storage allocated to the array i.e., the relative address of A[low], then the i th elementsbeginsatthe location: *base+(I-low)*w*.Thisexpressioncanbereorganizedas *i*w+ (base - low*w)* . The sub-expression *base-low*w* is calculated and stored in the symbol table at compile time when the array declaration is processed, so that the relative address of A[i] can be obtained by just adding *i*w* to it.

- AddressingArrayElements
- Arraysare storedinablockofconsecutivelocations
- Assumewidthofeachelementisw
- ithelementofarrayAbeginsinlocationbase+(i-low)xwwherebase isrelative address of A[low]
- Theexpressionisequivalentto
- ixw+(base-lowxw)

$\longrightarrow$ i x w + const

2-DIMENSIONALARRAY:For arowmajortwodimensionalarraytheaddressofA[i][j] can be calculated by the formula :

base+((i-low_i)*n2+j- low_j)*wwhere $low_i$and $low_j$are lowervaluesofIand jand n2 is number of values jcan take i.e. n2 = high2 - low2 + 1.

Thiscanagainbewrittenas:

((i*n2)+j)*w+(base-((low_i*n2)+low_j)*w)andthesecondtermcanbecalculatedatcompile time.

In the same manner, the expression for the location of an element in column major two-dimensionalarraycanbeobtained.Thisaddressing canbegeneralizedtomultidimensionalarrays. **S**torage can be either row major or column major approach.

Example: Let Abea10x20 arraytherefore, n1=10 and n2=20and assume w=4 The

Three address code to access A[y,z] is

t_1 = y* 20 t

$_1$ = t $_1$ + z

t_2= 4 * t_1

t_3=A-84{((low_1Xn_2)+low_2)Xw)=(1*20+1)*4=84}

t_4=t_2+t_3

x=t_4
LetAbea10x20array n1
= 10 and n2 = 20

Assumewidthofthetypestoredinthearrayis4. Thethreeaddresscodetoaccess A[y,z] is t1 = y *
20
t1=t1+z
t2=4*t1
t3=baseA-84{((low1*n2)+low2)*w)=(1*20+1)*4=84} t4
=t2 +t3
x=t4

Thefollowingoperationsaredesigned:1.mktable(previous):createsanewsymboltableand returns a pointer to this table. Previous is pointer to the symbol table ofparent procedure.

2. entire(table,name,type,offset):createsanewentryfor*name*inthesymboltablepointed toby *table*.

3. addwidth(table,width):recordscumulativewidthofentriesofatablein itsheader.

4. enterproc(table,name,newtable):createsanentryforprocedure*name*inthesymboltable pointed to by*table* . *newtable* is a pointer to symboltable for *name*.

P $\longrightarrow$ {t=mktable(nil);
push(t,tblptr);
push(0,offset)}

D
{addwidth(top(tblptr),top(offset));
pop(tblptr);
pop(offset)}

D$\longrightarrow$**D;** D

The symboltablesare created using two stacks: *tblptr*to hold pointersto symboltablesof the enclosing procedures and *offset* whose top element is the next available relative address for a local of the current procedure. Declarations in nested procedures can be processed by the syntax directed definitions given below. Note that they are basically same as those given above but we have separatelydealt with the epsilon productions. Go to the next page for the explanation**.**

```
P -> MD        {     addwidth(top(tblptr) ,top(offset));
                     pop(tblptr); pop(offset);
               }
M ->           {     t= mktable(nil);
                     push(t,tblptr);
                     push(0,offset);
               }
D -> D1 ; D2
D -> proc id ; ND1 ; S         {     t = top(tblptr);
                                     addwidth(t, top(offset));
                                     pop(tblptr); pop(offset);
                                     enterproc(top(tblptr), id.name , t)
                               }
D -> id:T                      { enter(top(tblptr), id.name, T.type , top(offset));
                               top(offset) = top(offset) + T.width
                               }
N ->                           { t = mktable ( top(tblptr));
                               push(t,tblptr); push(0,offset);
                               }
```

D→proc id;
 { t = mktable(top(tblptr));
 push(t,tblptr);push(0,offset)}
D1;S
 { t = top(tblptr);
 addwidth(t,top(offset));
 pop(tblptr);pop(offset);;
 enterproc(top(tblptr),id.name,t)}
D→id:T
 {enter(top(tblptr),id.name,T.type,top(offset));
 top(offset) = top (offset) + T.width }

The action for M creates a symboltable for the outermost scope and hence a nilpointer is passed in place of previous. When the declaration, D proc id ; ND1 ; S is processed, the action corresponding to N causes the creation ofa symboltable for the procedure;the pointerto symbol table of enclosing procedure is given by top(tblptr). The pointer to the new table is pushed on to the stack tblptr and 0 is pushed as the initial offset on the offset stack. When the actions corresponding to the subtrees ofN, D1and S have been executed, theoffset corresponding to the currentprocedurei.e.,top(offset)containsthetotalwidthofentriesinit.Hencetop(offset)isadded to the header of symbol table of the current procedure. The top entries of *tblptr* and *offset* are popped so that the pointer and offset of the enclosing procedure are now on top of these stacks. Theentryfor id isaddedtothesymboltableofthe enclosingprocedure. Whenthe declarationD->id:T isprocessed entryfor id iscreated inthesymboltableofcurrent procedure. Pointer to the symbol tableof currentprocedure is again obtainedfrom top(tblptr).

Offsetcorrespondingtothecurrentprocedurei.e.top(offset)isincrementedbythewidth required by type T to point to the next available location.

STORAGEALLOCATIONFORRECORDS

Fieldnamesinrecords

T → record

{t=mktable(nil);

push(t,tblptr);push(0,offset)} D

end

{T.type=record(top(tblptr));

T.width = top(offset);

pop(tblptr); pop(offset)}

T->recordLDend	{t=mktable(nil); push(t,tblptr);push(0,offset) }
L ->	{T.type=record(top(tblptr)); T.width = top(offset); pop(tblptr); pop(offset) }

The processing done corresponding to records is similar to that done for procedures.AfterthekeywordrecordisseenthemarkerLcreatesanewsymboltable. Pointertothistable and offset 0 are pushed on the respective stacks. The action for the declaration D-> id :T push the information about the field names on the table created. At the end the top of the offset stack containsthetotalwidthofthedataobjectswithintherecord.This isstoredintheattribute T.width. The constructor *record* is applied to the pointer to the symbol table to obtainT.type.

NamesintheSymboltable:

S→id := E
{p=lookup(id.place);
ifp<>nilthenemit(p:=E.place) else
error}
E→id
{p=lookup(id.name);
ifp<>nilthenE.place=p

elseerror}

The operation lookup in the translation scheme above checks if there is an entry for this occurrence of the name in the symbol table. If an entry is found, pointer to the entry is returned else nilis returned. Lookup first checks whether the name appears inthe current symboltable. If notthenit looksforthename inthesymboltableoftheenclosingprocedureandsoon.Thepointer to the symbol table of the enclosing procedure is obtained from the header of the symbol table.

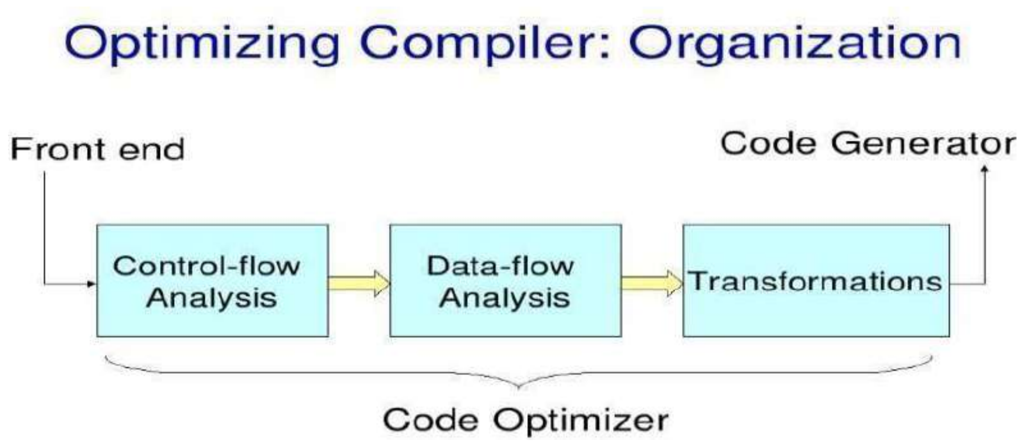
CODEOPTIMIZATION

Considerations for optimization : The code produced by the straight forward compiling algorithmscanoftenbemadetorunfasterortakelessspace,orboth.Thisimprovementisachieved by program transformations that are traditionally called optimizations. Machine independent optimizations are program transformations that improve the target code without taking into considerationanypropertiesofthetargetmachine. Machinedependantoptimizationsarebasedon register allocation and utilization of special machine-instruction sequences.

Criteriaforcodeimprovementtransformations

- Simplystated,thebest programtransformationsarethosethatyieldthemost benefit for the least effort.
- First,thetransformationmustpreservethemeaningofprograms.Thatis,theoptimization must not change the output produced by a program for a given input, or cause an error.
- Second,atransformationmust,ontheaverage,speedupprogramsbyameasurable amount.
- Third,thetransformationmustbeworththeeffort.

Some transformations can only be applied after detailed, often time-consuming analysis of the source program, so there is little point in applying them to programs that will be run only a few times.



OBJECTIVESOFOPTIMIZATION: Themainobjectivesoftheoptimizationtechniquesare as follows

1. Exploitthefastpathincaseofmultiplepaths froagivensituation.
2. Reduceredundantinstructions.
3. Produceminimumcodeformaximumwork.
4. Tradeoffbetweenthe size ofthe codeandthe speedwithwhichitgetsexecuted.
5. Placecodeanddatatogetherwhenever it isrequiredto avoidunnecessarysearchingof data/code

Duringcodetransformationintheprocessofoptimization,thebasicrequirementsareasfollows:

1. Retainthesemanticsofthesourcecode.
2. Reducetimeand/orspace.
3. Reducetheoverheadinvolvedintheoptimizationprocess.

ScopeofOptimization:Control-FlowAnalysis

Consider all that has happened up to this point in the compiling process—lexical analysis, syntactic analysis, semantic analysis and finally intermediate-code generation. The compiler has done an enormous amount of analysis, but it still doesn‘t really know how the program does what it does. In control-flow analysis, the compiler figures out even more information about how the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

Control-flow analysisbegins by constructing a control-flow graph, which is a graph ofthe different possible paths program flow could take through a function. To build the graph, we first dividethecodeintobasic blocks. Abasic block isasegmentofthecodethat aprogrammust enter at the beginning and exit only at the end. This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all statements are executed consecutively after the first one is (no branches or halts until the exit). Thus a basic block has exactly one entrypoint and one exit point. If a programexecutes the first instruction ina basic block, it must execute every instruction in the block sequentiallyafter it.

Abasicblockbeginsinoneofseveralways:

• Theentrypointintothefunction

• Thetargetofabranch(inourexample,anylabel)
• Theinstructionimmediatelyfollowingabranchorareturn

Abasicblock endsinanyofthefollowingways:
• Ajumpstatement
• Aconditionalorunconditionalbranch
• Areturnstatement

Now we can construct the control-flow graph between the blocks. Each basic block is a node inthe graph, and the possible different routes a program might take arethe connections, i.e. ifablockendswitha branch, therewillbeapathleading fromthat blocktothebranchtarget. The blocksthat can follow a block are called its successors. There may be multiple successorsor just one. Similarly the block may have many, one, or no predecessors. Connect up the flow graphfor Fibonacci basic blocks given above. What does an if then-else look likein a flow graph? What aboutaloop?Youprobablyhaveallseenthegccwarningorjavacerrorabout:"Unreachablecode at line XXX." How can the compiler tell when code is unreachable?

LOCALOPTIMIZATIONS

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information; we just work with the statements within the block. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform. We'll consider some of the more common local optimizations as examples.

FUNCTIONPRESERVINGTRANSFORMATIONS

∑Commonsubexpressionelimination

∑Constantfolding

∑Variablepropagation

∑DeadCodeElimination

∑Codemotion

∑StrengthReduction

1. <u>CommonSubExpressionElimination:</u>

Two operations are common if they produce the same result. In such a case, it is likely more efficienttocomputetheresultonceandreferenceitthesecondtimeratherthanre-evaluateit.An expressionisalive iftheoperandsusedto computetheexpressionhavenot beenchanged.An expression that is no longer alive is dead.

Example:

```
a=b*c;
d=b*c+x-y;
```

Wecaneliminatethesecondevaluationofb*c fromthiscodeifnoneoftheintervening statements has changed its value. We can thus rewrite the code as

```
t1=b*c;
a=t1;
d=t1+x-y;
```

Letusconsiderthefollowingcode

```
a=b*c;
b=x;
d=b*c+x-y;
```

inthiscode, wecannoteliminatethesecondevaluationofb*cbecausethe valueofbischanged due to the assignment b=x before it is used in calculating d.

Wecansaythetwoexpressionsarecommonif

- ∑Theylexicallyequivalent i.e.,theyconsist ofidenticaloperandsconnectedtoeachother by identical operator.
- ∑Theyevaluatetheidenticalvalues i.e.,no assignment statements foranyoftheiroperands exist between the evaluations of these expressions.
- ∑Thevalueofanyoftheoperandsuse intheexpressionshouldnot be changedevendueto the procedure call.

Example:

```
c=a*b;
x=a;
d=x*b;
```

We maynotethateventhoughexpressionsa*band x*barecommonintheabovecode, they can not be treated as common sub expressions.

2. VariablePropagation:

Letusconsidertheabovecodeonceagain c=a*b;

```
x=a;
d=x*b+4;
```

if we replace x by a in the last statement, we can identify a*b and x*b as common sub expressions.Thistechniqueiscalledvariablepropagationwheretheuseofonevariableisreplaced by another variable if it has been assigned the value of same

CompileTimeevaluation

The execution efficiency of the program can be improved by shifting execution time actions to compile time so that they are not performed repeatedly during the program execution. Wecanevaluateanexpressionwithconstantsoperandsatcompiletimeandreplacethatexpression bya single value. This is called folding. Consider the following statement:

a= 2*(22.0/7.0)*r;

Here,wecanperformthecomputation2*(22.0/7.0)atcompiletimeitself.

3. DeadCodeElimination:

If the value contained in the variable at a point is not used anywhere in the program subsequently, the variable is said to be dead at that place. If an assignment is made to a dead variable,thenthatassignmentisadeadassignmentanditcanbesafelyremovedfromtheprogram.

Similarly,apiece ofcodeissaid to bedead, which computesvaluethat arenever used anywhere in the program.

c=a*b;

x=a;

d=x*b+4;

Usingvariablepropagation,thecodecanbewrittenasfollows:

c=a*b;

x=a;

d=a*b+4;

UsingCommonSubexpressionelimination,the codecanbewrittenasfollows:

t1=a*b;
c=t1;
x=a;
d=t1+4;

Here,x=awillconsideredasdeadcode.Henceitiseliminated. t1=
a*b;
c=t1;
d=t1+4;

4. CodeMovement:

The motivation for performing code movement in a program is to improve the execution time of theprogrambyreducingtheevaluationfrequencyofexpressions. Thiscanbedonebymovingthe evaluation ofan expression to other parts ofthe program. Let us consider the bellow code:

```
If(a<10)
{
b=x^2-y^2;
}
else
{
b=5;
a=(x^2-y^2)*10;
}
```

Atthetimeofexecutionoftheconditiona<10, x^2-y^2 isevaluatedtwice. So,wecanoptimize the code by moving the out side to the block as follows:

```
t=x^2-y^2;
If(a<10)
{
b=t;
}
else
{
b=5;
a=t*10;
}
```

5. **StrengthReduction:**

Inthefrequencyreductiontransformationwetriedtoreducetheexecutionfrequencyofthe expressionsbymovingthecode.Thereisother classoftransformationswhichperformequivalent actions indicated in the source program by reducing the strength of operators. By strength reduction, we mean replacing the high strength operator with low strength operator with out affecting the program meaning. Let us consider the bellow example:

```
i=1;
while(i<10)
{
y=i*4;
}
```

Theabovecanwrittenasfollows: i=1;

```
t=4;
```

```
while(i<10)
{
y=t;
t=t+4;
}
```

Herethehighstrengthoperator*isreplacedwith+.

GLOBALOPTIMIZATIONS,DATA-FLOW ANALYSIS:

So far we were only considering making changes within one basic block. With some Additional analysis, we can apply similar optimizations across basic blocks, making them global optimizations. It‘s worth pointing out that global in this case does not mean across the entire program. We usually optimize only one function at a time. Inter procedural analysis is an even larger task, one not even attempted by some compilers.

The additionalanalysis the optimizer doesto performoptimizations across basic blocks is called **data-flow analysis**. Data-flow analysis is much more complicated than control-flow analysis, and we can only scratch the surface here.

Let‘s consider a global common sub expression elimination optimization as our example. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be **available at thatpoint**. Once the set ofavailable expressions is known, commonsub-expressionscanbeeliminatedonaglobalbasis. Eachblock isanodeinthe flow graph of a program. The **successor** set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. Anexpression is defined at the point where it is assigned a value and killed when oneofitsoperandsissubsequentlyassignedanewvalue. Anexpressionisavailableat some point p in a flow graph if everypath leading to p contains a prior definition ofthat expression which is not subsequently killed. Lets define such useful functions in DF analysis in following lines.

avail[B] =setofexpressions availableonentrytoblockB
exit[B]=setofexpressionsavailableonexitfromB
avail[B]=∩**exit[x]: x∈pred[B]**(i.e. Bhasavailablethe intersectionoftheexit ofits predecessors)
killed[B]=setoftheexpressionskilled inB
defined[B]=setofexpressionsdefined inB
exit[B] = avail[B]- killed[B] + defined[B]

avail[B]=∩**(avail[x]-killed[x]+defined[x])**:x∈**pred[B]**

Hereisan**AlgorithmforGlobalCommonSub-expressionElimination**:

1) First,computedefinedandkilledsetsforeachbasicblock(thisdoesnotinvolveanyofits predecessors or successors).

2) Iterativelycomputetheavailandexit setsforeachblock byrunningthefollowingalgorithm until you hit a stable fixed point:

a) Identifyeachstatement **s**oftheform**a=bopc**insomeblockBsuchthat **bopc**is available at the entryto B and neither **b** nor **c** is redefined in B prior to **s**.

b) Followflowofcontrolbackward inthegraphpassingbacktobutnotthrougheach blockthat defines**bopc**.The last computationof**bopc**insuchablockreaches**s**.

c) After eachcomputation**d=bopc**identified instep2a,addstatement **t =d**tothat block where t is a new temp.

d) Replace**s**by**a=t**.

Tryanexampletomakethingsclearer:

```
main:
    BeginFunc28;
          b=a+2;
        c = 4 * b ;
        tmp1=b<c;
        ifNZtmp1gotoL1; b
        = 1 ;
        L1:
        d=a+2;
EndFunc ;
```

First, divide the code above into basic blocks. Now calculate the available expressions for each block.Thenfindanexpressionavailableinablockandperformstep2cabove.Whatcommonsub-expression can you share between the two blocks? What if the above code were:

```
main:
    BeginFunc28;
          b=a+2;
        c = 4 * b ;
        tmp1=b<c;
        IfNZtmp1GotoL1; b
        = 1 ;
        z=a+2;<========= anadditionallinehere
  L1:
        d=a+2;
EndFunc;
```

MACHINEOPTIMIZATIONS

Infinalcodegeneration, there isa lotofopportunityforcleverness ingeneratingefficient target code. In this pass, specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for this particular architecture.

REGISTERALLOCATION:

Onemachineoptimizationofparticular importanceisregisterallocation,whichisperhaps thesinglemosteffectiveoptimizationforallarchitectures.Registersarethefastestkindofmemory available, but as a resource, they can be scarce.

The problem is how to minimize traffic between the registers and what lies beyond them in the memoryhierarchyto eliminate time wasted sending data back and forthacross the bus and the different levels of caches. Your Decaf back-end uses a very naïve and inefficient means of assigning registers, it just fills them before performing an operation and spills them right afterwards.

Amuchmoreeffectivestrategywould betoconsiderwhichvariablesare moreheavilyin demand and keep those in registers and spill those that are no longer needed or won'tbe needed until much later.

One common register allocation technique is called "register coloring", after the central idea to view register allocation as a graph coloring problem. Ifwe have 8 registers, then wetryto color a graph with eight different colors. The graph‘s nodes are made of "webs" and the arcs are determined by calculating interference between the webs. A web represents a variable‘s definitions, places where it is assigned a value (as in **x = …**), and the possible different uses of those definitions (asin **y = x + 2**). This problem,in fact,can be approached as anothergraph. The definition and uses of a variable are nodes, and if a definition reaches a use, there is an arc between the two nodes. Iftwo portions ofa variable‘s definition-use graph are unconnected, then we have two separate websfor a variable. Inthe interference graphforthe routine, each node isa web. We seek to determine which webs don't interfere with one another, so we know we can use the same register for those two variables. For example, consider the following code:

```
i=10;
j=20;
x = i+ j;
y= j+k;
```

We say that **i** interferes with **j** because at least one pair of **i**‘s definitions and uses is separated by a definition or use of **j**, thus, **i** and **j** are "alive" at the same time. A variable is alive betweenthetimeit hasbeendefinedandthatdefinition‘slast use,afterwhichthevariable isdead. If two variables interfere, then we cannot use the same register for each. But two variables that don't interferecansincethere isnooverlap inthelivenessandcanoccupythesameregister. Once we have the interference graph constructed, we r-color it so that no two adjacent nodes share the same color (r is the number of registers we have, each color represents a different register).

Wemayrecallthat graph-coloring isNP-complete,so weemployaheuristicratherthanan optimalalgorithm. Here is a simplified version of something that might be used:

1. Findthenodewiththeleastneighbors.(Breaktiesarbitrarily.)
2. Removeitfromtheinterferencegraphandpushitontoastack
3. Repeatsteps1and 2untilthe graph isempty.
4. Now,rebuildthegraphasfollows:
    a. Takethetopnodeoffthestackand reinsertitintothe graph
    b. Chooseacolorforit based onthecolorofanyofitsneighborspresentlyinthegraph, rotating colors in case there is more than one choice.
    c. Repeata,andbuntilthegraphiseithercompletelyrebuilt,orthereisno color available to color the node.

Ifwegetstuck,thenthegraphmaynotber-colorable,wecouldtryagainwithadifferentheuristic, sayreusing colors as often as possible. Ifno otherchoice, we have to spilla variable to memory.

INSTRUCTIONSCHEDULING:

Another extremely important optimization of the final code generator is instruction scheduling. Because many machines, including most RISC architectures, have some sort of pipelining capability, effectively harnessing that capability requires judicious ordering of instructions.

InMIPS,eachinstructionisissuedinonecycle,butsometakemultiplecyclestocomplete. It takes an additional cycle before the value of a load is available and two cycles for a branch to reachitsdestination,butaninstructioncanbeplacedinthe"delayslot"afterabranchandexecuted in that slack time. On the left is one arrangement of a set of instructions that requires 7 cycles. It assumes no hardware interlock and thus explicitly stalls between the second and third slots while the load completes and has a Dead cycle after thebranchbecause the delayslot holds a noop. On theright, amorefavorablerearrangementofthesame instructionswillexecutein5 cycleswithno dead Cycles.

```
lw$t2,4($fp)
lw$t3,8($fp)
noop
add$t4,$t2,$t3
subi $t5, $t5, 1
goto L1
noop
lw $t2, 4($fp)
lw $t3, 8($fp)
subi$t5,$t5,1
goto L1
add $t4,$t2,$t3
```

PEEPHOLEOPTIMIZATIONS:

Peephole optimization is a pass that operates onthe target assembly and onlyconsiders a few instructions at atime (through a "peephole") and attemptsto do simple, machine dependent code improvements. For example, peephole optimizations might include elimination of multiplication by 1, elimination of load of a value into a register when the previous instruction storedthatvalue fromtheregistertoamemorylocation, orreplacingasequenceofinstructionsby a single instruction with the same effect. Because of its myopic view, a peephole optimizer does not have the potential payoff of a full-scale optimizer, but it can significantly improve code at a very local level and can be useful for cleaning up the finalcode that resulted from more complex optimizations. Much of the work done in peephole optimization can be though of as find-replace activity, looking for certain idiomatic patterns in a single or sequence of two to threeInstructions than can be replaced by more efficient alternatives.

For example, MIPS has instructions that canadd asmallinteger constant tothe value ina registerwithoutloadingtheconstantintoaregisterfirst,sothesequenceontheleftcanbereplaced with that on the right:

```
li$t0,10
lw $t1, -8($fp)
add$t2,$t1,$t0
sw $t1, -8($fp)
lw $t1, -8($fp)
addi$t2,$t1,10
sw $t1, -8($fp)
```

Whatwouldyoureplacethefollowingsequencewith? lw

```
$t0, -8($fp)
sw $t0, -
8($fp)Whataboutthi
sone? mul $t1, $t0,
2
```

AbstractSyntaxTree/DAG: Isnothingbut thecondensedformofaparsetreeandis

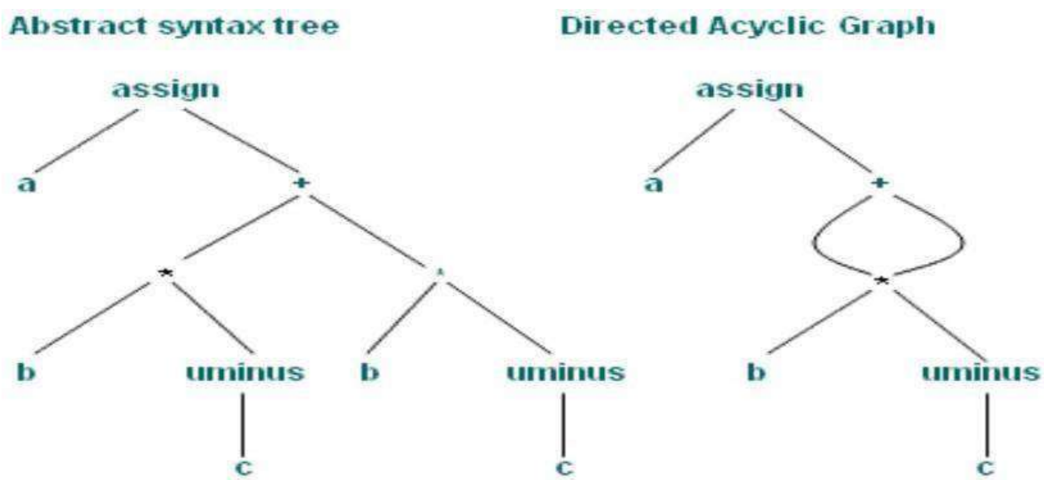
- ∑.Usefulfor representinglanguageconstructs
- ∑.Depictsthenaturalhierarchicalstructureofthesourceprogram

- Eachinternalnoderepresentsanoperator
- Childrenofthe nodesrepresentoperands
- Leafnodesrepresentoperands

.DAG is more compact thanabstract syntaxtreebecause commonsubexpressions are eliminated Asyntaxtreedepictsthenaturalhierarchicalstructureofasourceprogram.Itsstructurehasalready beendiscussedinearlier lectures. DAGsaregeneratedasacombinationoftrees:operandsthatare being reused are linked together, and nodes may be annotated with variable names (to denote assignments). This way, DAGs are highly compact, since they eliminate local common sub-expressions. Ontheother hand, theyare not so easytooptimize, since theyare more specific tree forms. However, it can be seen that proper building ofDAG for a given sequenceofinstructionscancompactlyrepresenttheoutcomeofthecalculation. An

example ofa syntax tree and DAG has been given in the next slide .

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



Youcanseethatthenode"*"comesonlyonce intheDAGaswellasthe leaf"b", but the meaningconveyedbyboththerepresentations(ASTaswellastheDAG)remainsthesame.

IMPORTANT QUESTIONS:

1. WhatisCodeoptimization?Explaintheobjectivesofit.Also discussFunctionpreserving transformations with your own examples?
2. Explainthefollowingoptimizationtechniques
   (a) CopyPropagation
   (b) Dead-CodeElimination
   (c) CodeMotion
   (d) ReductioninStrength.
4. Explaintheprinciplesourcesofcode-improvingtransformations.
5. Whatdoyoumeanbymachinedependentandmachineindependentcodeoptimization? Explain about machine dependent code optimization with examples.

ASSIGNMENTQUESTIONS:

1. ExplainLocalOptimizationtechniqueswithyourownExamples?
2. Explainindetailtheprocedurethateliminatingglobalcommonsubexpression?
3. Whatistheneed ofcodeoptimization?Justifyyouranswer?

UNIT-V

CONTROL/DATAFLOWANALYSIS:

FLOWGRAPHS:

We can add flow control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes ofa flow graph are the basic nodes. One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is, if

- Thereisconditionalorunconditionaljump fromthe last statement ofB_1tothefirst statement of B_2, or
- B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditionaljump. Wesaythat B_1isthepredecessorofB_2,and B_2isasuccessorofB_1.

Forregisterandtemporaryallocation

- Removevariablesfromregistersifnotused
- StatementX=YopZdefinesXand usesYand Z
- Scaneachbasic blocksbackwards
- Assumealltemporariesaredeadonexitandalluservariablesareliveonexit

Theuseofanameinathree-addressstatementisdefinedasfollows.Supposethree-address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement ito jalong a paththat has no intervening assignments to x,thenwe saystatementjuses the value of x computed at i.

We wish to determine for each three-address statement x := y op z, what the next uses of x, y and z are. We collect next-use information about names in basic blocks. If the name in a register is no longer needed, then the register can be assigned to some other name. This idea of keeping a name in storage only if it will be used subsequently can be applied in a number of contexts. It is used to assign space for attribute values.

Thesimplecodegenerator applies it to register assignment. Ouralgorithmis to determine next uses makes a backward pass over each basic block, recording (in the symbol table) for each name xwhether xhasa next use inthe block and ifnot, whether it is liveonexit fromthat block. We can assume that all non-temporary variables are live on exit and all temporary variables are dead on exit.

Algorithmtocomputenextuse information

- Supposewearescanningi:X:= YopZ inbackwardscan

- Attachtoi,informationinsymboltableaboutX,Y,Z
- SetXtonotliveandnonextuseinsymboltable
- SetYandZtobeliveandnextuseiniinsymboltable

Asanapplication, weconsidertheassignment ofstoragefortemporarynames. Supposewe reachthree-addressstatementi:x:=yop zinourbackwardscan.Wethendothefollowing:

1. Attachtostatementithe informationcurrentlyfoundinthesymboltableregardingthe next use and live ness of x, yand z.

2. Inthesymboltable,setxto"notlive"and"nonextuse".

3. Inthesymboltable, set yandzto "live"andthenext usesofyand ztoi. Notethatthe order ofsteps (2) and (3) may not be interchanged because x may be y or z.

Ifthree-addressstatementiisofthe formx:= yorx:=opy, thestepsarethesameasabove, ignoring z. consider the below example:

1: t_1 = a * a
2:t_2=a*b 3:
t_3 = 2 *
$t_2$4:t_4=t_1+$t_3$5:
t_5 = b * b
6:t_6=t_4+$t_5$7:
X = t $_6$

Example:

STATEMENT

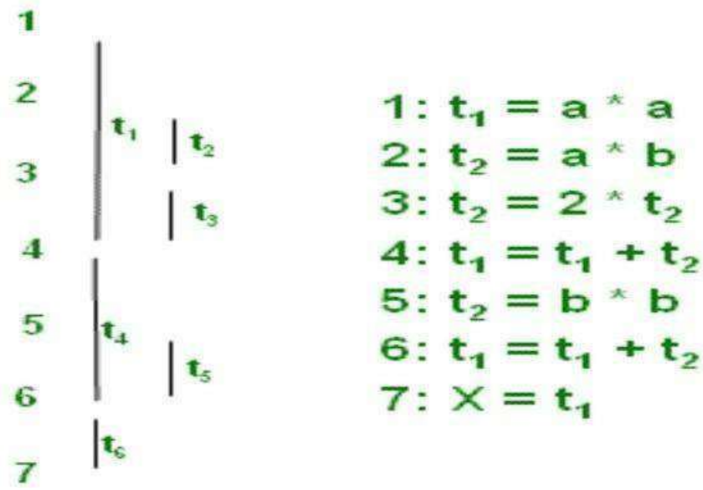
7: no temporary is live
6: t_6:use(7), t_4 t_5 not live
5: t_5:use(6)
4: t_4:use(6), t_1 t_3 not live
3: t_3:use(4), t_2 not live
2: t_2:use(3)
1: t_1:use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

Wecanallocatestoragelocations fortemporariesbyexaminingeachinturnandassigning atemporarytothefirst locationinthe field fortemporariesthat doesnot containa live temporary. If a temporary cannot be assigned to any previously created location, add a new location to the dataareaforthe current procedure. Inmanycases,temporaries canbe packed intoregisters rather than memory locations, as in the next section.

Example.



Thesixtemporariesinthebasicblockcanbepackedintotwolocations.Theselocations correspond to t 1 and t 2 in:

1:t_1=a*a,2:t_2=a*b,3:t_2=2*t_2,4:t_1=t_1+t_2,5:t_2=b*b

6:t_1=t_1+t_2,7:X=t_1

DATAFLOWEQUATIONS:

Dataanalysisisneeded forglobalcodeoptimization,e.g.:Isavariable liveonexit fromablock? Does a definition reach a certain point in the code? Data flow equations are used to collect dataflow information A typical dataflow equation has the form

Out[s]=Gen[s]U(in[s]-kill[s])

Thenotionofgenerationandkillingdependsonthe dataflowanalysisproblemtobe solved Let'sfirst considerReachingDefinitionsanalysisforstructuredprogramsAdefinitionofavariable x is a statement that assigns or may assign a value to x An assignment to x is an unambiguous definitionofxAnambiguous assignment to xcanbe anassignment to a pointer or a functioncall where x is passed by reference When x is defined, we say the definition is generated An unambiguous definition of x kills all otherdefinitions of x When all definitions ofx are the same at a certain point, we can use this information to do some optimizations Example: all definitions of x define x to be 1. Now, by performing constant folding, we can do strength reduction if x is used in z=x*y.

GLOBALOPTIMIZATIONS,DATA-FLOW ANALYSIS

So far we were only considering making changes within one basic block. With some additional analysis, we can apply similar optimizations across basic blocks, making them global optimizations. It‘s worth pointing out that global in this case does not mean across the entire program. We usually only optimize one function at a time. Interprocedural analysis is an even largertask,onenot evenattemptedbysomecompilers.Theadditionalanalysistheoptimizer must dotoperformoptimizationsacrossbasicblocksiscalleddata-flowanalysis.Data-flowanalysis is much more complicated than control-flow analysis.

Let‘s consider a global commonsub-expression elimination optimization as ourexample. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be available at that point.

Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis. Each block is a node in the flow graph of a program. The successor set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. An expression is defined at thepoint where it isassignedavalueandkilledwhenoneofitsoperands issubsequentlyassigned a new value. Anexpression is available at some point p ina flow graph ifeverypath leading to p contains a prior definition of that expression which is not
subsequentlykilled.

avail[B]=setofexpressionsavailableonentrytoblockB
exit[B]=setofexpressionsavailable onexitfromB
avail[B]=∩**exit[x]: x∈pred[B]**(i.e.Bhasavailablethe intersectionofthe exit of its predecessors)
killed[B] =setoftheexpressionskilled inB
defined[B]=setofexpressionsdefined inB
exit[B] = **avail[B] - killed[B]** + **defined[B]**
avail[B]=∩**(avail[x]-killed[x]+defined[x])**:**x∈pred[B]**

Hereisanalgorithmfor globalcommonsub-expressionelimination:

1) First,computedefinedandkilledsetsforeachbasicblock(thisdoesnotinvolveanyofits redecessors or successors).
2) Iterativelycomputetheavailandexit setsforeachblock byrunningthefollowingalgorithm until you hit a stable fixed point:
   a) Identifyeachstatement **s**oftheform**a=bopc**insomeblock Bsuchthat **bopc**is available at the entryto B and neither **b** nor **c** is redefined in B prior to **s**.
   b) Followflowofcontrolbackward inthegraphpassingbacktobutnotthrougheach block that defines **b op c**. The last computation of**b op c** insuch a block reaches**s**.
   c) After eachcomputation**d=bopc**identified instep2a,addstatement **t =d**tothat block where t is a new temp.
   d) Replace**s**by**a=t**.

Letstryanexampletomakethingsclearer: main:

```
BeginFunc28;
b=a+2;
c = 4 * b ;
tmp1=b<c;
ifNZtmp1gotoL1; b
= 1 ;
L1:
d=a+2;
EndFunc ;
```

First,dividethecodeaboveintobasicblocks.Nowcalculatetheavailableexpressions for each block. Then find an expression available in a block and performstep 2c above. Whatcommonsubexpressioncanyousharebetweenthetwoblocks?What iftheabove code were:

```
main:
BeginFunc28;
b=a+2;
c = 4 * b ;
tmp1=b<c;
IfNZtmp1GotoL1; b
= 1 ;
z=a+2;<=========anadditionalline here L1:
d=a+2;
EndFunc ;
```

CommonSubexpression Elimination

Twooperations are common iftheyproducethe same result. Insucha case, it is likely more efficient to computethe result once and reference itthe secondtime ratherthanre-evaluate it. An expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

```
main()
{
intx,y,z;
x=(1+20)*-x;
y=x*x+(x/y);
y=z=(x/y)/(x*x);
}
straighttranslation:
tmp1 = 1 + 20 ;
tmp2 = -x ;
x=tmp1*tmp2;
tmp3 = x * x ;
tmp4 = x / y ;
y=tmp3+tmp4;
```

tmp5 = x/ y;
tmp6=x* x;
z=tmp5/tmp6; y
= z ;

What sub-expressions can be eliminated? How can valid common sub-expressions (live ones) be determined?Here isanoptimized version, afterconstant foldingandpropagationandelimination of common sub-expressions:

tmp2= -x;
x=21*tmp2;
tmp3 = x * x ;
tmp4 = x / y ;
y=tmp3+tmp4;
tmp5 = x / y ;
z=tmp5/tmp3; y
= z ;

InductionVariableElimination

Constantfoldingreferstotheevaluationatcompile-timeofexpressionswhoseoperands are knownto be constant. In its simplest form, it involves determining that all of the operands in an expression are constant-valued, performing the evaluation of the expression at compile-time, and thenreplacing the expressionbyits value. Ifanexpressionsuchas **10 + 2 *3**is encountered, the compiler can compute the result at compile-time (**16**) and emit code as if the input contained the result rather thantheoriginalexpression. Similarly, constant conditions, suchas a conditional branch**ifa <b goto L1else goto L2** where**a**and**b** areconstant canbe replaced bya **Goto L1**or **Goto L2** depending on the truth of the expression evaluated at compile-time. The constant expressionhasto beevaluatedat least once,but ifthecompilerdoesit, it means youdon‘t haveto do it againasneeded during runtime. Onething tobecarefulabout isthatthe compiler mustobey the grammar and semantic rules from the source language that apply to expression evaluation, which may not necessarily match the language you are writing the compiler in. (For example, if you were writing an APL compiler,you would need to take care that you were respecting its Iversonian precedence rules). It should also respect the expected treatment of any exceptional conditions (divide by zero, over/underflow). Consider the Decaf code on the far left and its un optimizedTACtranslationinthe middle,whichisthentransformedbyconstant-foldingonthefar right:

a = 10*5+6-b;_tmp0= 10;
_tmp1=5;
_tmp2=_tmp0*_tmp1;
_tmp3=6;
_tmp4=_tmp2+_tmp3 ;
_tmp5=_tmp4–b; a
= _tmp5 ;
_tmp0 = 56;_tmp1=_tmp0–b;a =_tmp1;

Constant-foldingiswhatallowsalanguagetoacceptconstantexpressionswhereaconstantis required (such as a case label or arraysize) as in these C language examples:

```
intarr[20*4+3];
switch (i) {
case10*5:...
}
```

In both snippets shown above, the expression can be resolved to an integer constant at compile time and thus, we have the information needed to generate code. If either expression involved a variable, though, there would be an error. How could you rewrite the grammar to allow the grammar to do constant folding incase statements?Thissituation isa classic exampleofthe gray area between syntactic and semantic analysis.

LiveVariableAnalysis

Avariableisliveat acertainpoint inthecodeifit holdsa valuethat maybe needed inthe future. Solvebackwards:
FinduseofavariableThisvariable is livebetweenstatementsthathave founduseasnext statement Recursive until you find a definition of the variable

Usingthesets*use[B]*and*def[B]*

*def[B]*isthesetofvariablesassigned values in*B* priortoanyuseofthat variable in*B* *use*[*B]* is the set ofvariables whose values may be used in [*B]* prior to anydefinition ofthe variable.

A variable comes live into a block (in *in[B]*), if it is either used before redefinition of it is livecomingoutoftheblockand isnotredefined intheblock.Avariablecomes liveoutofablock (in *out[B]*) ifand only if itis live coming into one of its successors

*In[B]=use[B]*U*(out[B]-def[B])*

Out[B]= Uin[s]
Ssucc[B]

Notetherelationbetweenreaching-definitionsequations: therolesof*in* and*out* areinterchanged

CopyPropagation

This optimization is similar to constant propagation, but generalized to non-constant values. If we have an assignment **a = b** in our instruction stream, we can replace later occurrencesof**a**with**b**(assumingtherearenо changesto eithervariable in-between).Giventhe waywe generate TAC code, this is a particularly valuable optimization since it is able to eliminate a large number of instructions that only serve to copy values from one variable to another.Thecodeonthe left makesacopyof**tmp1**in**tmp2** andacopyof**tmp3** in**tmp4**. Inthe optimized version on the right, we eliminated those unnecessary copies and propagated the original variable into the later uses:

```
tmp2=tmp1;
tmp3=tmp2*tmp1; tmp4
= tmp3 ;
tmp5=tmp3*tmp2; c
= tmp5 + tmp4 ;
tmp3=tmp1*tmp1;
tmp5=tmp3*tmp1; c
= tmp5 + tmp3 ;
```

We can also drive this optimization "backwards", where we can recognize that the original assignment made to atemporarycanbe eliminated in favorofdirect assignment tothe finalgoal: tmp1 = LCall _Binky ;

```
a=tmp1;
tmp2=LCall_Winky; b
= tmp2 ;
tmp3=a*b; c
= tmp3 ;
a=LCall_Binky;
b= LCall_Winky;
c=a*b;
```

IMPORTANT QUESTIONS:

1. WhatisDAG?ExplaintheapplicationsofDAG.
2. Explainbrieflyaboutcodeoptimizationanditsscopeinimprovingthecode.
3. ConstructtheDAG forthefollowingbasicblock:

    D:=B*C
    E :=A+B
    B:=B+C
    A:=E-D.

3. ExplainDetectionofLoopInvariantComputation
4. ExplainCode Motion.

ASSIGNMENTQUESTIONS:

1. Whatisloops?Explainaboutthefollowingtermsinloops:
    (a)Dominators
    (b) Naturalloops
    (c) Innerloops
    (d) pre-headers.
2. WriteshortnotesonGlobaloptimization?

OBJECTCODEGENERATION

Machinedependentcodeoptimization:

In final code generation, there is a lot of opportunity for cleverness in generating efficient target code. In this pass, specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for this particular architecture.

RegisterAllocation

One machine optimization of particular importance is register allocation, which is perhaps the single most effective optimization for all architectures. Registers are the fastest kind ofmemoryavailable,but asaresource,theycanbescarce.Theproblemis howtominimize traffic betweentheregistersandwhatliesbeyondtheminthememoryhierarchytoeliminatetimewasted sendingdatabackand forthacrossthebusandthedifferent levelsofcaches. YourDecafback-end uses a verynaïve and inefficient means ofassigning registers, it just fills thembefore performing anoperationandspillsthemright afterwards.Amuchmoreeffectivestrategywouldbetoconsider which variables are more heavily indemand and keep those inregisters andspillthose that are no longer needed or won't be needed until much later. One common register allocation technique is called "register coloring", after the central idea to view register allocation as a graph coloring problem.Ifwehave8registers,thenwetrytocoloragraphwitheight differentcolors.Thegraph‘s nodes are made of "webs" and the arcs are determinedby calculating interference between the webs. Awebrepresentsavariable‘sdefinitions,placeswhere it isassignedavalue(as in**x=…**), and the possible different uses ofthose definitions (as in **y = x + 2**). This problem, in fact, can be approached as another graph. The definition and uses of a variable are nodes, and if a definition reaches a use, there is anarc betweenthe two nodes. Iftwo portions of a variable‘s definition-use graph are unconnected, then we have two separate webs for a variable. In the interference graph for the routine, each node is a web. We seek to determine which webs don't interfere with one another, so we know we can usethe same register for thosetwo variables. For example, consider the following code:

```
i=10;
j=20;
x= i+ j;
y=j+k;
```

We say that **i** interferes with **j** because at least one pair of **i**‘s definitions and uses is separated by a definition or use of**j**, thus, **i** and **j** are "alive" at the same time. A variable is alive betweenthetimeit hasbeendefinedandthat definition‘slast use,afterwhichthevariableisdead. If two variables interfere, then we cannot use the same register for each. But two variables that don't interfere can since there is no overlap in the liveness and can occupythe same register.

Oncewehavetheinterferencegraphconstructed,wer-colorit sothatnotwo adjacent nodesshare the same color (r is the number of registers we have, each color represents a different register). You may recall that graph-coloring is NP-complete, so we employ a heuristic rather than an optimalalgorithm. Here is a simplified version ofsomething that might be used:

1. Findthenodewiththeleastneighbors.(Breaktiesarbitrarily.)
2. Removeitfromtheinterferencegraphandpushitontoastack
3. Repeatsteps1and2untilthegraph isempty.
4. Now,rebuildthe graphasfollows:
    a. Takethetopnodeoffthestackand reinsertitintothegraph
    b. Chooseacolorforit based onthecolorofanyofitsneighborspresentlyinthe graph, rotating colors in case there is more than one choice.
    c. Repeataandbuntilthegraphiseithercompletelyrebuilt,orthereisno color available to color the node.

Ifwegetstuck,thenthegraphmaynotber-colorable,wecouldtryagainwithadifferentheuristic, sayreusing colors as oftenas possible. Ifno other choice, we have to spill a variable tomemory.

InstructionScheduling:

Another extremely important optimization of the final code generator is instruction scheduling. Because many machines, including most RISC architectures, have some sort of pipelining capability, effectively harnessing that capability requires judicious ordering of instructions. In MIPS, each instruction is issued in one cycle, but some take multiple cycles to complete. It takes an additional cycle before the value of a load is available and two cycles for a branch to reach its destination, but an instruction can be placed in the "delay slot" after a branch andexecutedinthat slacktime.Ontheleftisonearrangementofasetofinstructionsthat requires 7 cycles. It assumesno hardware interlock and thusexplicitly stalls betweenthe second and third slots while the load completes and has a Dead cycle after the branch because the delay slot holds a noop. On the right, a more Favorable rearrangement of the same instructions will execute in 5 cycles with no dead Cycles.

lw$t2,4($fp)
lw$t3,8($fp)
noop
add$t4,$t2,$t3
subi $t5, $t5, 1
goto L1
noop
lw $t2, 4($fp)
lw $t3, 8($fp)
subi$t5,$t5,1
goto L1
add $t4,$t2,$t3

RegisterAllocation

One machine optimization of particular importance is register allocation, which is perhaps the single most effective optimization for all architectures. Registers are the fastest kind ofmemoryavailable,but asaresource,theycanbe scarce.Theproblemishowtominimize traffic betweentheregistersandwhatliesbeyondtheminthememoryhierarchytoeliminatetimewasted sendingdatabackand forthacrossthebusandthedifferent levelsofcaches. YourDecafback-end uses a verynaïve and inefficient means ofassigning registers, it just fills thembefore performing anoperationandspillsthemright afterwards.Amuchmoreeffectivestrategywouldbetoconsider which variables are more heavilyin demand and keep those inregisters andspillthose that are no longer needed or won't be needed until much later. One common register allocation technique is called "register coloring", after the central idea to view register allocation as a graph coloring problem.Ifwehave8registers,thenwetrytocoloragraphwitheight differentcolors.Thegraph‘s nodes are made of "webs" and the arcs are determinedby calculating interference between the webs. Awebrepresentsavariable‘sdefinitions,placeswhere it isassignedavalue(as in**x=…**), and the possible different uses ofthose definitions (as in **y = x + 2**). This problem, in fact, canbe approached as another graph. The definition and uses of a variable are nodes, and if a definition reaches a use, there is anarc betweenthe two nodes. Iftwo portions of a variable‘s definition-use graph are unconnected, then we have two separate webs for a variable. In the interference graph for the routine, each node is a web. We seek to determine which webs don't interfere with one another, so we know we can usethe same register for thosetwo variables. For example, consider the following code:

```
i=10;
j=20;
x= i+ j;
y=j+k;
```

We saythat **i** interferes with **j** because at least one pair of **i**‘s definitions and uses is separatedbyadefinitionoruseof**j**,thus, **i**and**j** are"alive"atthesametime. A variable isalive between the time it has been defined and that definition‘s last use, after which the variable is dead.Iftwo variablesinterfere,thenwecannot usethesameregisterforeach.Buttwovariables thatdon't interferecansincethere isno overlap inthelivenessandcanoccupythesameregister. Once we have the interference graph constructed, we r-color it so that no two adjacent nodes share the same color (r is the number of registers we have, each color represents a different register). You may recall that graph-coloring is NP-complete, so we employ a heuristic rather than anoptimal algorithm. Here is a simplified version of something that might be used:

1. Findthenodewiththeleastneighbors.(Breaktiesarbitrarily.)
2. Removeitfromtheinterferencegraphandpushitonto astack
3. Repeatsteps1and 2untilthe graph isempty.
4. Now,rebuildthegraphasfollows:
    a. Takethetopnodeoffthestackand reinsertitintothe graph

b. Chooseacolorforit based onthecolorofanyofitsneighborspresentlyinthegraph, rotating colors in case there is more than one choice.

c. Repeataandbuntilthegraphiseither completelyrebuilt,orthereisno coloravailable to color the node.

Ifwegetstuck,thenthegraphmaynotber-colorable,wecouldtryagainwithadifferentheuristic, sayreusing colors as oftenas possible. Ifno other choice, we have to spill a variable tomemory.

CODEGENERATION:

The code generator generates target code for a sequence of three-address statement. It considerseachstatementinturn,remembering ifanyoftheoperandsofthestatement arecurrently inregisters, and taking advantageofthat fact, ifpossible. The code-generationuses descriptorsto keep track of register contents and addresses for names.

1. A register descriptor keeps track ofwhat is currently in each register. It is consulted whenever a new register is needed. We assume that initially the register descriptor shows that all registers are empty. (If registers are assigned across blocks, this would not be the case). As the code generationfortheblockprogresses, eachregisterwillholdthevalueofzeroormorenamesat any given time.

2. An address descriptor keeps track of the location (or locations) where the current value of the namecanbefoundatruntime.Thelocationmightbearegister, astacklocation,amemoryaddress, or some set ofthese, since when copied, a value also stays where it was. This informationcanbe stored in the symboltable andis used to determine the accessingmethod fora name.

CODEGENERATIONALGORITHM:

foreachX=YopZdo

- Invokeafunctiongetregtodetermine locationLwhereX must bestored.UsuallyLisa register.
- ConsultaddressdescriptorofYtodetermineY'.Prefer aregister forY'.IfvalueofYnot already in L generate

  MovY',L

- Generate

  op Z', L

AgainpreferaregisterforZ.UpdateaddressdescriptorofXtoindicateXisinL.IfLisaregister updateitsdescriptortoindicatethatitcontainsXandremoveXfromallotherregisterdescriptors.

.Ifcurrent valueofYand/or Zhasno next useandaredeadonexit fromblockandarein registers, change register descriptor to indicate that they no longer contain Y and/or Z.

The code generation algorithmtakes as input a sequence ofthree-address statements constituting a basic block. For each three-address statement ofthe formx := yop z we performthe following actions:

1. InvokeafunctiongetregtodeterminethelocationLwheretheresultofthecomputation yopzshouldbestored.Lwillusuallybearegister,butit couldalso beamemorylocation. We shall describe getreg shortly.

2. Consulttheaddressdescriptorforutodeterminey',(oneof)thecurrentlocation(s)of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value ofu is not already in L, generatethe instruction MOV y', L to place a copyof y in L.

3. Generate the instruction OP z', L where z' is a current location of z. Again, prefer a registerto amemorylocation ifz is inboth. Updatethe addressdescriptorto indicatethat xisinlocationL.IfLisaregister,updateitsdescriptortoindicatethatitcontainsthevalue of x, and remove x from all other register descriptors.

4. Ifthecurrent valuesofyand/or yhave no next uses, arenotliveonexit fromthe block, and are in registers,alter the register descriptor to indicate that, after execution ofx := y op z, those registers no longer will contain y and/or z, respectively.

FUNCTIONgetreg:

1. IfYisinregister(thatholdsnoothervalues)andYisnotliveandhasnonext useafter X = Y op Z thenreturnregisterofYforL.
2. Failing(1)returnanemptyregister
3. Failing(2) ifXhasanext useintheblockoroprequiresregisterthenget aregister R, storeits content into M (by Mov R, M) and use it.
4. ElseselectmemorylocationXasL

Thefunction**getreg** returnsthelocationLtohold thevalue ofxfortheassignmentx:=yop z.

1. Ifthe name y is in a register that holds the value of no other names (recall that copy instructionssuchasx:=ycouldcausearegistertoholdthevalueof twoormorevariables

simultaneously),and yisnotliveandhasno next useafter executionofx:= yopz,thenreturn the register of yfor L. Updatethe address descriptorof yto indicate that y is no longer in L.

2. Failing(1),returnanemptyregisterforLifthereisone.

3. Failing(2),ifxhasanextuseintheblock, oropisanoperatorsuchas indexing, thatrequires a register, find an occupied register R. Storethe value ofR into memory location (by MOVR, M)if itis notalreadyinthe proper memorylocationM,updatethe addressdescriptorM, and returnR.IfRholdsthevalueofseveralvariables,aMOV instructionmust begeneratedforeach variablethatneedstobestored.Asuitableoccupiedregistermightbeonewhosedatumis referenced furthest in the future, orone whose value is also in memory.

4. Ifxisnotusedinthe block,ornosuitableoccupiedregistercanbe found,select thememory location of x as L.

Example:

Stmt	code	regdesc	addrdesc
t_1=a-b	mova,R_0 subb,R_0	R_0contains t_1	$t_1$inR_0
t_2=a-c	mova,R_1 subc,R_1	R_0containst$_1$ R_1containst$_2$	$t_1$inR_0 $t_2$inR_1
t_3=t_1+t_2	addR_1,R_0	R_0contains t_3 R_1contains t_2	$t_3$inR_0 $t_2$inR_1
d=t_3+t_2	addR $_1$,R $_0$ movR_0,d	R_0containsd	dinR_0 din$R_0$and memory

Forexample,theassignment d:=(a-b)+(a-c)+(a-c)might betranslated intothefollowing three-address code sequence:

t_1=a- b

t_2=a-c

t_3=t_1+t_2d=t

$_3$+t_2

The code generation algorithm that we discussed would produce the code sequence as shown. Shown alongside are the values of the register and address descriptors as code generation progresses.

DAGforRegisterallocation:

DAG (Directed Acyclic Graphs) are useful data structures for implementing transformationsonbasicblocks. ADAGgivesapictureofhowthevaluecomputedbyastatement in a basic block is used in subsequent statements of the block. Constructing a DAG from three-addressstatements isagoodwayofdeterminingcommonsub-expressions(expressionscomputed more thanonce) withina block, determining whichnames are used insidethe block but evaluated outsidetheblock,anddeterminingwhichstatementsoftheblockcould havetheir computedvalue used outside the block.

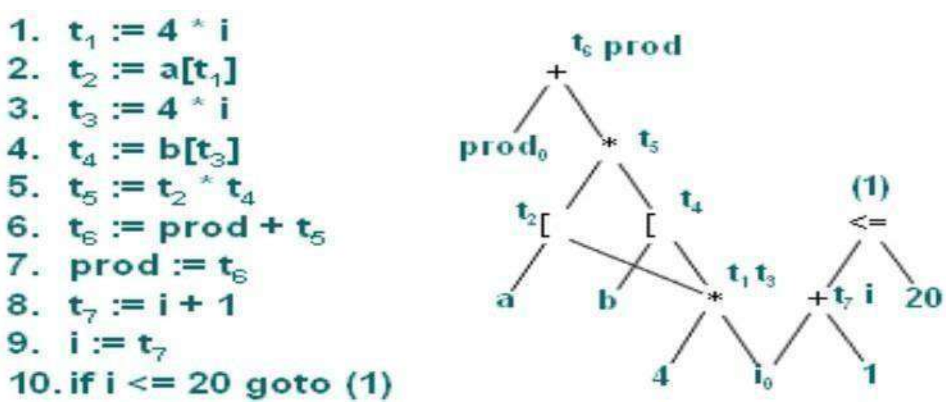
ADAGforabasicblockisadirectedcyclicgraphwiththefollowinglabelsonnodes:

1. Leaves are labeled by unique identifiers, either variable names or constants. From the operatorappliedtoanamewedeterminewhetherthe l-valueorr-valueofanameisneeded;most leavesrepresentr-values.Theleavesrepresent initialvaluesofnames,andwesubscriptthemwith 0 to avoid confusion with labels denoting "current" values of names as in (3) below.

2. Interiornodesarelabeledbyanoperator symbol.

3. Nodesarealsooptionallygivenasequenceofidentifiersforlabels.Theintentionisthat interior nodes represent computed values, and the identifiers labeling a node are deemed to have that value.

DAGrepresentationExample:



Forexample,theslideshowsathree-addresscode.ThecorrespondingDAG isshown. Weobserve thateachnodeoftheDAGrepresentsaformula intermsoftheleaves,thatis,thevaluespossessed by variables and constants upon entering the block. For example, the node labeled t 4 represents the formula

b[4*i]

thatis,thevalueofthewordwhoseaddress is4*ibytesoffset fromaddressb, whichisthe intended value of t 4 .

CodeGenerationfromDAG

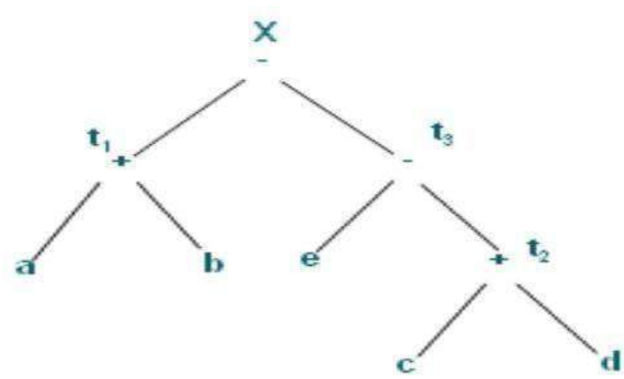
S_1=4*i	S_1=4*i
S_2=addr(A)-4	S_2=addr(A)-4
S_3=S_2[S_1]	S_3= S_2[S_1]
S_4= 4*i	
S_5=addr(B)-4	S_5=addr(B)-4
S_6= S_5[S_4]	S_6=S_5[S_4]
S_7= S_3*S_6	S_7=S_3*S_6
S_8=prod+S_7	
prod=S_8	prod=prod+S_7
S_9= I+1	
I= S_9	I=I+1
IfI<=20 goto(1)	IfI<=20goto(1)

WeseehowtogeneratecodeforabasicblockfromitsDAGrepresentation.Theadvantage of doing so is that from a DAG we can more easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence ofthree-address statements or quadruples. If the DAG is a tree, we can generate code that we can prove is optimalunder such criteria as program length or the fewest number of temporaries used. The algorithm for optimal code generation froma tree is also useful when the intermediate code is a parse tree.

Rearrangingorderofthecode

Considerfollowingbasic block :

t 1 =a +b t
2 = c +d t 3
=e-t 2
X=t1-t 3



and its**DAG**givenhere.

Here,webrieflyconsiderhowtheorderinwhichcomputationsaredonecanaffectthe cost of resulting object code. Consider the basic block and its corresponding DAG representationas shown in the slide.

Rearrangingorder.

Three adress code for the DAG (assuming only two registers are available)		Rearrangingthecodeas t_2= c + d t_3=e-t2 t_1=a+b
MOVa,R_0		X=t_1-t_3
ADDb,R_0		gives
MOVc,R_1		MOVc,R_0
ADDd,R_1		ADDd,R_0
MOVR_0,t_1	Registerspilling	MOVe,R_1
MOVe,R_0		SUBR $_0$,R_1
SUBR_1,R_0		MOVa,R $_0$
MOVt_1,R_1	Registerreloading	ADDb, R_0
SUBR_0,R_1		SUBR $_1$, R_0
MOVR_1,X		MOV R_1,X

Ifwegeneratecodeforthethree-addressstatementsusingthecodegenerationalgorithmdescribed before, we get the code sequence as shown (assuming two registers R0 and R1 are available, and onlyXisliveonexit).Ontheotherhandsupposewerearrangedtheorderofthe statementssothat the computation of t 1 occurs immediately before that of X as:

t_2 = c + d
t_3 = e -t 2
t_1 = a + b
X=t_1-t_3

Then, using the code generation algorithm, we get the new code sequence as shown (again only R0andR1areavailable).Byperformingthecomputationinthisorder,wehave beenableto save two instructions;MOV R0, t 1(whichstoresthe value ofR0 in memorylocationt 1)and MOVt 1 , R1 (which reloads the value of t 1 in the register R1).

IMPORTANT&EXPECTEDQUESTIONS:

ConstructtheDAG forthefollowingbasicblock:
D:=B*C
E :=A+B
B:=B+C
A:=E-D.

1. WhatisObjectcode?Explainaboutthefollowingobjectcodeforms:
   (a) Absolutemachine-language
   (b) Relocatablemachine-language
   (c) Assembly-language.
2. Explainabout Genericcodegenerationalgorithm?
3. Writeandexplainaboutobjectcodeforms?
4. ExplainPeepholeOptimization

ASSIGNMENTQUESTIONS:

1. Explainabout Genericcodegenerationalgorithm?
2. Explainabout Data-Flowanalysisofstructuredflowgraphs.
3. WhatisDAG?ExplaintheapplicationsofDAG.