COMPILER DESIGN
[R18A0512]
LECTURE NOTES

B.TECH III YEAR – I SEM
(R18)(2020-21)

DEPARTMENTOFCOMPUTER SCIENCEANDENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING&TECHNOLOGY

(AutonomousInstitution– UGC,Govt.ofIndia)

Recognizedunder 2(f)and12 (B) ofUGCACT1956

(AffiliatedtoJNTUH,Hyderabad,Approved byAICTE-Accredited byNBA&NAAC–'A'Grade-ISO9001:2015Certified)

Maisammaguda,Dhulapally(PostVia.Hakimpet),Secunderabad–500100,TelanganaState,India

MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY

IIIYearB.TechCSE-ISem L T/P/D C

3 1/0 / -4

(R18A0512)CompilerDesign

UNIT–I:

Language Translation: Basics, Necessity, Steps involved in atypical language processing system,Typesoftranslators,**Compilers:**OverviewandPhasesofaCompiler,PassandPhasesoftranslation,bootstrapping,datastructuresincompilation

Lexical Analysis (Scanning): Functions of Lexical Analyzer, **Specification of tokens:** Regularexpressionsand Regulargrammarsforcommon PL constructs.**Recognition of Tokens:**FiniteAutomata in recognition and generation of tokens.**Scanner generators:** LEX-Lexical AnalyzerGenerators. Syntax Analysis (Parsing) :Functions of a parser, Classification of parsers. Contextfreegrammarsinsyntaxspecification,benefitsandusageincompilers.

UNIT–II:

Topdownparsing–Definition,typesoftopdownparsers:Backtracking,Recursivedescent,Predictive, LL (1), Preprocessing the grammars to be used in top down parsing, Error recovery, andLimitations. Bottom up parsing: Definition, types of bottom up parsing, Handle pruning. ShiftReduce parsing, LR parsers: LR(0), SLR, CALR and LALR parsing, Error recovery, Handlingambiguous grammar,Parsergenerators:YACC-yetanothercompilercompiler..

UNIT–III:

Semantic analysis: Attributed grammars, Syntax directed definition and Translation schemes, Typechecker:functions,typeexpressions,typesystems,typescheckingofvariousconstructs.Intermediate Code Generation: Functions, different intermediate code forms- syntax tree, DAG,Polish notation, and Three address codes. Translation of different source language constructs intointermediatecode.

Symbol Tables: Definition, contents, and formats to represent names in a Symbol table. Differentapproaches used in the symbol table implementation for block structured and non block structuredlanguages,suchasLinearLists,SelfOrganizedLists,andBinarytrees,HashingbasedSTs.

UNIT–IV:

RuntimeEnvironment:Introduction,ActivationTrees,ActivationRecords,Controlstacks.Runtime storage organization: Static,Stack and Heapstorage allocation.Storage allocation forarrays,strings,andrecordsetc. Codeoptimization:goalsandConsiderationsforOptimization,ScopeofOptimization:Localoptimizations, DAGs, Loop optimization, Global Optimizations. Common optimization techniques:Folding,Copypropagation,CommonSubexpressioneliminations,Codemotion,Frequencyreduction,Strengthreductionetc.

UNIT–V:

Control flow and Data flow analysis: Flow graphs, Data flow equations, global optimization:Redundantsubexpressionelimination,Inductionvariableeliminations,LiveVariable analysis.Objectcodegeneration:Objectcodeforms,machinedependentcodeoptimization,registerallocationandassignmentgenericcodegenerationalgorithms,DAGforregisterallocation.

TEXTBOOKS:

1. Compilers,Principle,Techniques, andTools.–Alfred.VAho,Monica S.Lam,RaviSethi,Jeffrey D. Ullman;2ndEdition, PearsonEducation.
2. ModernCompilerimplementationinC,- AndrewN.AppelCambridgeUniversityPress.

REFERENCES:

1. lex&yacc,-JohnRLevine, TonyMason, DougBrown;O'reilly.
2. CompilerConstruction,-LOUDEN,Thomson.
3. Engineeringacompiler–Cooper&Linda,Elsevier
4. ModernCompilerDesign–DickGrune,HenryE.Bal,CarielTHJacobs,WileyDreatech

Outcomes:

Bytheendofthesemester,thestudentwillbeableto:

- Understandthenecessityandtypesofdifferentlanguagetranslators inuse.
- Applythetechniquesanddesigndifferentcomponents(phases)ofacompilerbyhand.
- Solveproblems,WriteAlgorithms,Programsandtest themfortheresults.
- UsethetoolsLex,Yaccincompiler componentsconstruction.

INDEX

COMPILERDESIGNNOTES IIIYEAR/ ISEM MRCET

UNIT-I

INTRODUCTIONTOLANGUAGEPROCESSING:

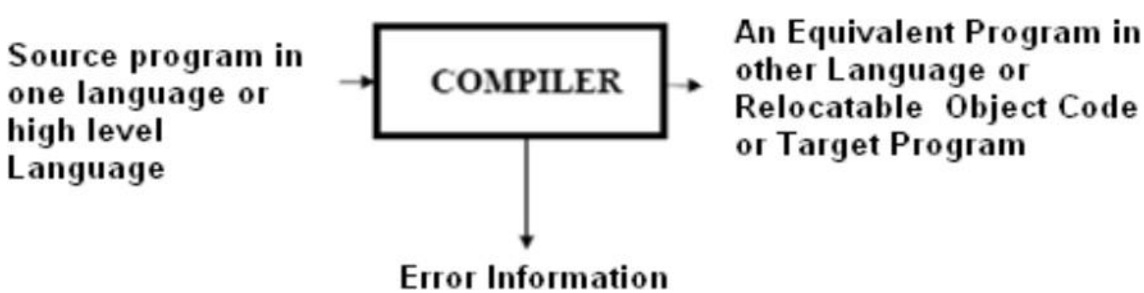
AsComputersbecameinevitableandindigenouspartofhumanlife,andseverallanguageswithdifferentandmoreadvancedfeaturesareevolvedintothisstreamtosatisfyorcomforttheuserin communicating with the machine , the development of the translators or mediator Software‘shave become essential to fill the huge gap between the human and machine understanding. Thisprocess is called Language Processing to reflect the goal and intent of the process. On the way tothis process to understand it in a better way, we have to be familiar with some key terms andconceptsexplainedinfollowing lines.

LANGUAGETRANSLATORS:

Is a computer program which translates a program written in one (Source) language to itsequivalentprograminother[Target]language.TheSourceprogramisahighlevellanguagewhereasthe Target language can be any thing from the machine language of a target machine (betweenMicroprocessortoSupercomputer)toanotherhighlevellanguageprogram.

∑TwocommonlyUsed TranslatorsareCompiler and Interpreter

1. **Compiler:** Compilerisaprogram,readsprograminonelanguagecalledSourceLanguageand translates in to its equivalent program in another Language called Target Language, inadditiontothisits presentstheerrorinformationtotheUser.



Ifthetargetprogramisanexecutablemachine-languageprogram,itcanthenbecalledbythe users toprocessinputs andproduceoutputs.

Figure1.1:RunningthetargetProgram

2. **Interpreter:** An interpreteris another commonly usedlanguage processor.Instead of producinga target program as a single translation unit, an interpreter appears to directly execute theoperationsspecifiedinthe sourceprogramoninputssuppliedbytheuser.

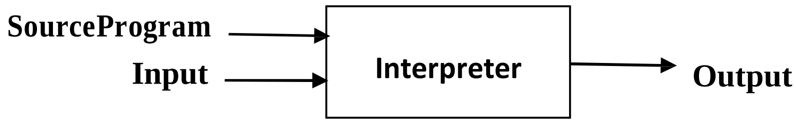


Figure1.2:RunningthetargetProgram

LANGUAGEPROCESSINGSYSTEM:

Basedontheinputthetranslatortakesandtheoutput itproduces,alanguagetranslatorcanbecalledasfollowing.

Preprocessor: Apreprocessortakestheskeletalsourceprogramasinputandproducesanextendedversion of it, which is the resultant of expanding the Macros, manifest constants if any, andincluding header files etc in the source file. For example, the C preprocessor is a macro processorthatisusedautomaticallybytheCcompilertotransformoursourcebeforeactualcompilation.O verandabove apreprocessorperforms the followingactivities:

- Collectsallthemodules,filesincaseifthesourceprogramisdividedintodifferentmodulesstoredatdif ferentfiles.
- Expandsshorthands/macrosintosourcelanguagestatements.

Compiler: Is a translator that takes as input a source program written in high level language andconvertsitintoitsequivalenttarget programinmachinelanguage.Inadditiontoabovethecompileralso

- Reportstoitsuserthepresenceoferrorsinthesourceprogram.
- Facilitatestheuserinrectifyingtheerrors,andexecutethecode.

Assembler: Isaprogramthattakesasinputanassemblylanguageprogramandconvertsitintoitsequivalentma chinelanguagecode.

Loader/Linker: Thisisaprogramthattakesasinputarelocatablecodeandcollectsthelibraryfunctions,reloc atableobjectfiles,and producesitsequivalentabsolutemachinecode.

Specifically,

- **Loading**consistsoftakingtherelocatablemachinecode,alteringtherelocatableaddresses,andplacin gthealteredinstructionsanddatainmemoryattheproperlocations.
- **Linking** allows us to make a single program from several files of relocatable machinecode. Thesefilesmay havebeen resultof several differentcompilations, one or moremay belibraryroutinesprovidedbythesystemavailableto anyprogramthatneedsthem.

In addition to these translators, programs like interpreters, text formatters etc., may be used inlanguage processing system.

To translate a program in a high-level language to anexecutableone,thecompilerperformsbydefaultthe compileandlinkingfunctions.

Normally the steps in a language processing system include: Preprocessing the skeletal Sourceprogram which produces an extended or expanded source or a ready to compile unit ofthe source program, followed by compiling the resultant code, then linking/loading, and finally itsequivalent executable code is produced. As I said earlier, not all these steps are mandatory. Insomecases,theCompiler onlyperformsthislinking andloadingfunctionsimplicitly.

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

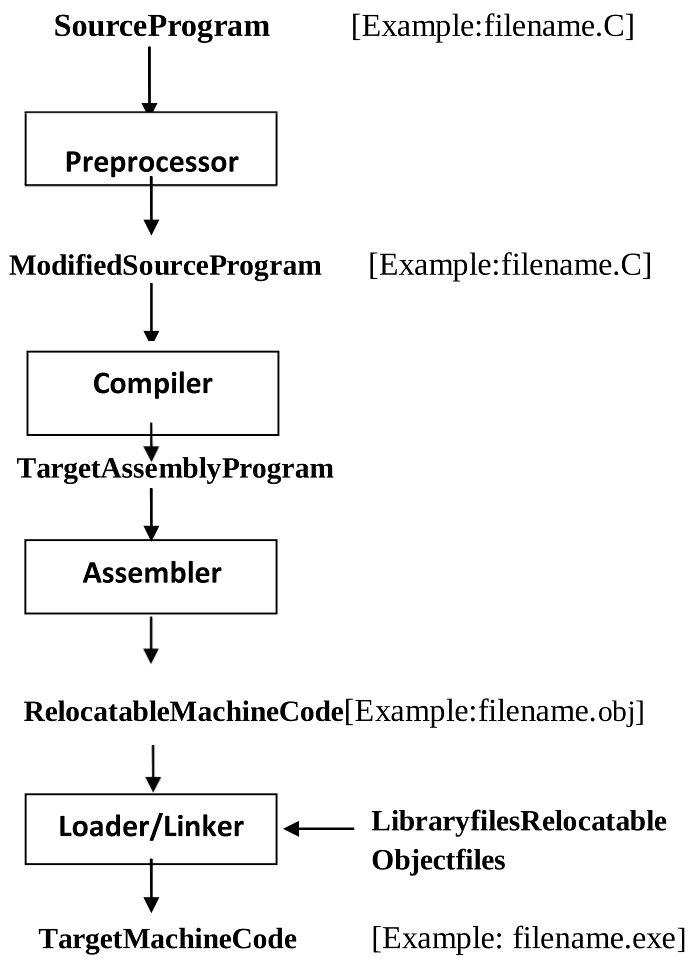


Figure1.3:ContextofaCompilerinLanguageProcessingSystem

TYPESOFCOMPILERS:

Basedonthespecificinputittakesandtheoutputitproduces,theCompilerscanbeclassifiedintothefol lowingtypes;

- **TraditionalCompilers(C,C++, Pascal):**TheseCompilersconvertasourceprograminaHLLintoitsequivalentinnativemachine codeorobjectcode.
- **Interpreters(LISP,SNOBOL,Java1.0):**TheseCompilersfirstconvertSourcecodeintointermediate code,andtheninterprets(emulates)ittoitsequivalentmachine code.
- **Cross-Compilers:**These are the compilers thatrun on one machine and produce code foranothermachine.
- **IncrementalCompilers:**Thesecompilersseparatethesourceintouserdefined–steps;Compiling/recompilingstep-by-step;interpretingstepsinagivenorder
- **Converters (e.g. COBOL to C++):** These Programs will be compiling from one high levellanguage toanother.
- **Just-In-Time (JIT) Compilers (Java, Micosoft.NET):** These are the runtime compilers fromintermediate language (byte code, MSIL) to executable code or native machine code. Theseperformtype–basedverificationwhichmakestheexecutable codemoretrustworthy
- **Ahead-of-Time (AOT) Compilers (e.g., .NET ngen):** These are the pre-compilers to the nativecode forJavaand.NET
- **Binary Compilation:** These compilers will be compiling object code of one platform into object codeofanotherplatform**.**

PHASESOFACOMPILER:

Due to the high complexity in the compilation process, a Compiler typically proceeds in a Sequence ofcompilation phases. The phases communicate with each other via clearly defined interfaces.Generally aninterface contains a Data structure (e.g., tree), Set of exported functions.Eachphase works on an abstract **intermediate representation** of the source program, not the sourceprogramtextitself(exceptthefirstphase)

Compiler Phases are the individual modules which are chronologically executed to perform theirrespectiveSub-activities,andfinallyintegratethesolutionstogive targetcode.

It is desirable to have relatively few phases, since it takes time to read and write immediate files.Following diagram (Figure1.4) depicts the phases of a compiler through which it goes during thecompilation.Therefore atypicalCompilerishavingthefollowingPhases:

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

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not allthephasesaremandatoryineveryCompiler.e.g,CodeOptimizerphaseisoptional in somecases.Thedescriptionisgiveninnextsection.

The Phases of compiler are dividedin to twoparts,first three phases are called asAnalysispartremainingthreecalledasSynthesis part.

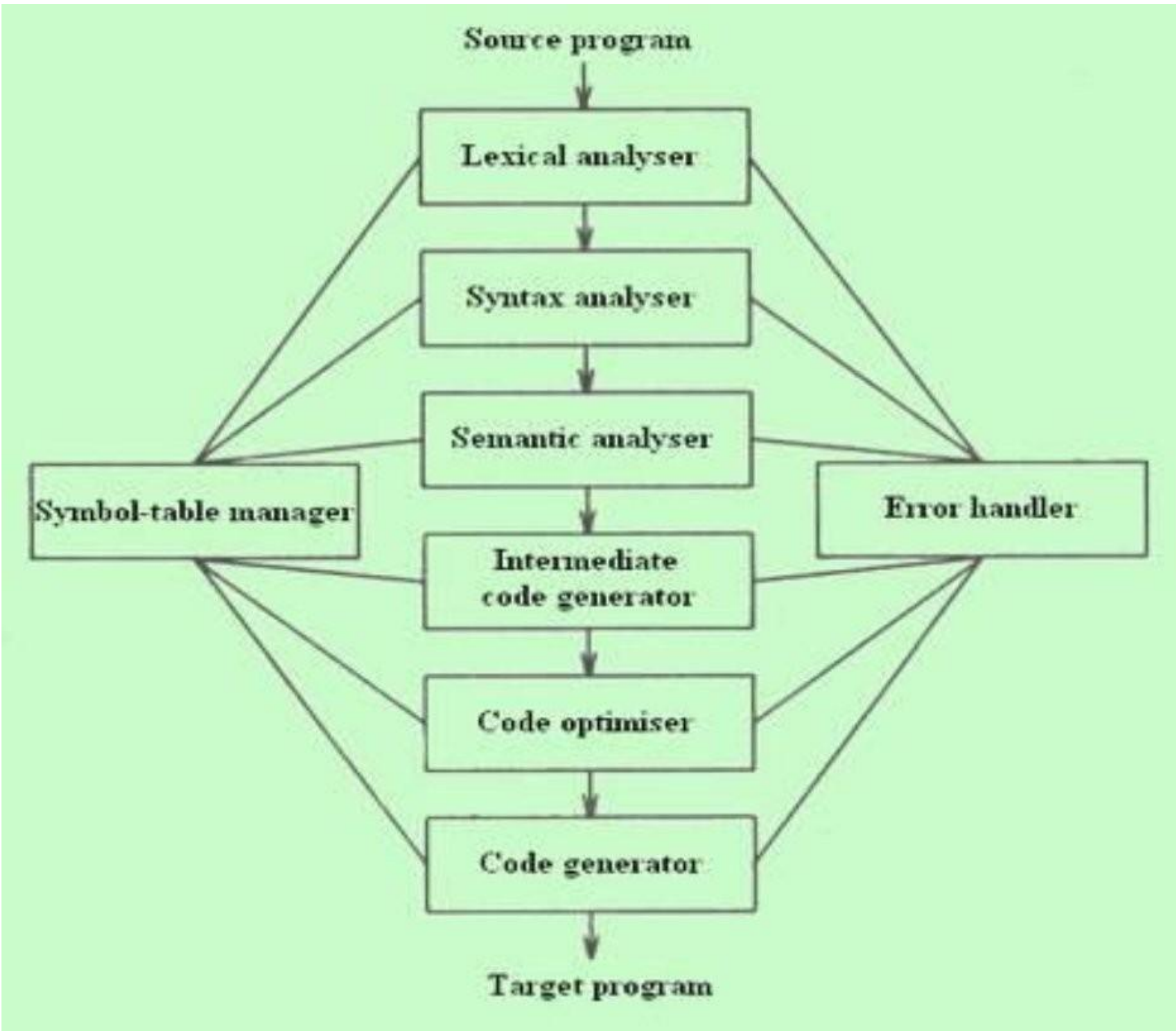


Figure1.4:PhasesofaCompiler

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 thatconvert theinputfrom one representation toacompletely deferent representation. Each pass makes a complete scan of the input and producesitsoutputto beprocessedbythesubsequentpass.Forexample atwopassAssembler.

THEFRONT-END& BACK-ENDOFACOMPILER

All of these phases of a general Compiler are conceptually divided into **The Front-end**,and**The Back-end**. This divisionis due to theirdependence on eitherthe Source Language ortheTargetmachine.ThismodeliscalledanAnalysis&Synthesismodelofacompiler.

The **Front-end** of the compiler consists of phases that depend primarily on the Sourcelanguage and are largely independent on the target machine. For example,front-end of thecompilerincludesScanner,Parser,CreationofSymboltable,SemanticAnalyzer,andtheIntermediate CodeGenerator.

The **Back-end** of the compiler consists of phases that depend on the target machine, andthose portions don't dependent on the Source language, just the Intermediate language. In this wehave different aspects of Code Optimization phase, code generation along with the necessaryErrorhandling,andSymboltableoperations.

LEXICALANALYZER(SCANNER): TheScanneristhefirstphasethatworksasinterfacebetweenthecompiler andtheSourcelanguageprogramandperformsthefollowingfunctions:

- Reads the characters in the Source program and groups them into a stream of tokens inwhicheachtokenspecifiesalogicallycohesivesequenceof characters,suchasanidentifier,a Keyword,a punctuation mark,a multicharacteroperatorlike:= .
- Thecharactersequenceforming atokeniscalled a**lexeme** ofthetoken.
- The Scanner generates a token-id, and also enters that identifiers name in the Symboltableifitdoesn'texist.
- AlsoremovestheComments,andunnecessaryspaces.

Theformat ofthetokenis**<Tokenname,Attributevalue>**

SYNTAXANALYZER(PARSER): TheParserinteractswiththeScanner, anditssubsequentphase SemanticAnalyzerandperformsthe followingfunctions:

- Groups the above received, and recorded token stream into syntactic structures, usuallyintoa structurecalled**Parse Tree**whoseleavesaretokens.
- Theinteriornodeofthistreerepresentsthe streamoftokensthatlogicallybelongstogether.
- It meansitchecksthesyntaxofprogramelements.

SEMANTICANALYZER: This phase receives the syntax tree as input, and checks thesemanticallycorrectnessoftheprogram.Thoughthetokensarevalidandsyntacticallycorrect,itmay happenthattheyarenotcorrectsemantically.

Thereforethesemanticanalyzerchecksthesemantics(meaning)ofthe statements formed.

- TheSyntacticallyand Semanticallycorrect structuresareproduced hereintheformofaSyntaxtreeorDAG orsome othersequentialrepresentationlike matrix.

INTERMEDIATECODEGENERATOR(ICG): Thisphasetakesthesyntacticallyandsemantically correct structure as input, and produces its equivalent intermediate notation of thesourceprogram. TheIntermediateCodeshouldhavetwoimportantpropertiesspecifiedbelow:

- Itshouldbeeasytoproduce,andEasytotranslateintothetargetprogram.Exampleintermediate codeformsare:
- Threeaddresscodes,
- Polishnotations,etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial interms of saving development time, effort, and cost. This phase performs the following specificfunctions:

- Attempts to improve the IC so as to have a faster machinecode.Typicalfunctionsinclude – Loop Optimization, Removal of redundant computations, Strength reduction,Frequencyreductionsetc.

- Sometimesthedatastructuresusedinrepresentingtheintermediateforms mayalsobechanged.

CODE GENERATOR: This is the final phase of the compiler and generates the target code,normally consisting of the relocatablemachinecode or Assembly code or absolutemachinecode.

- Memorylocationsareselectedfor eachvariableused,andassignmentofvariablestoregistersisdone.

- Intermediateinstructionsaretranslatedintoasequenceofmachineinstructions.

TheCompileralsoperformsthe**Symboltablemanagement**and**Errorhandling**throughoutthecompilation process. Symbol table is nothing but a data structure that stores different sourcelanguage constructs, and tokens generated during the compilation. These two modules interact with allphasesoftheCompiler.

Forexample,thesourceprogramisanassignmentstatement;thefollowingfigureshowshowthephasesof compilerconverts it gradually into thetarget program.

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

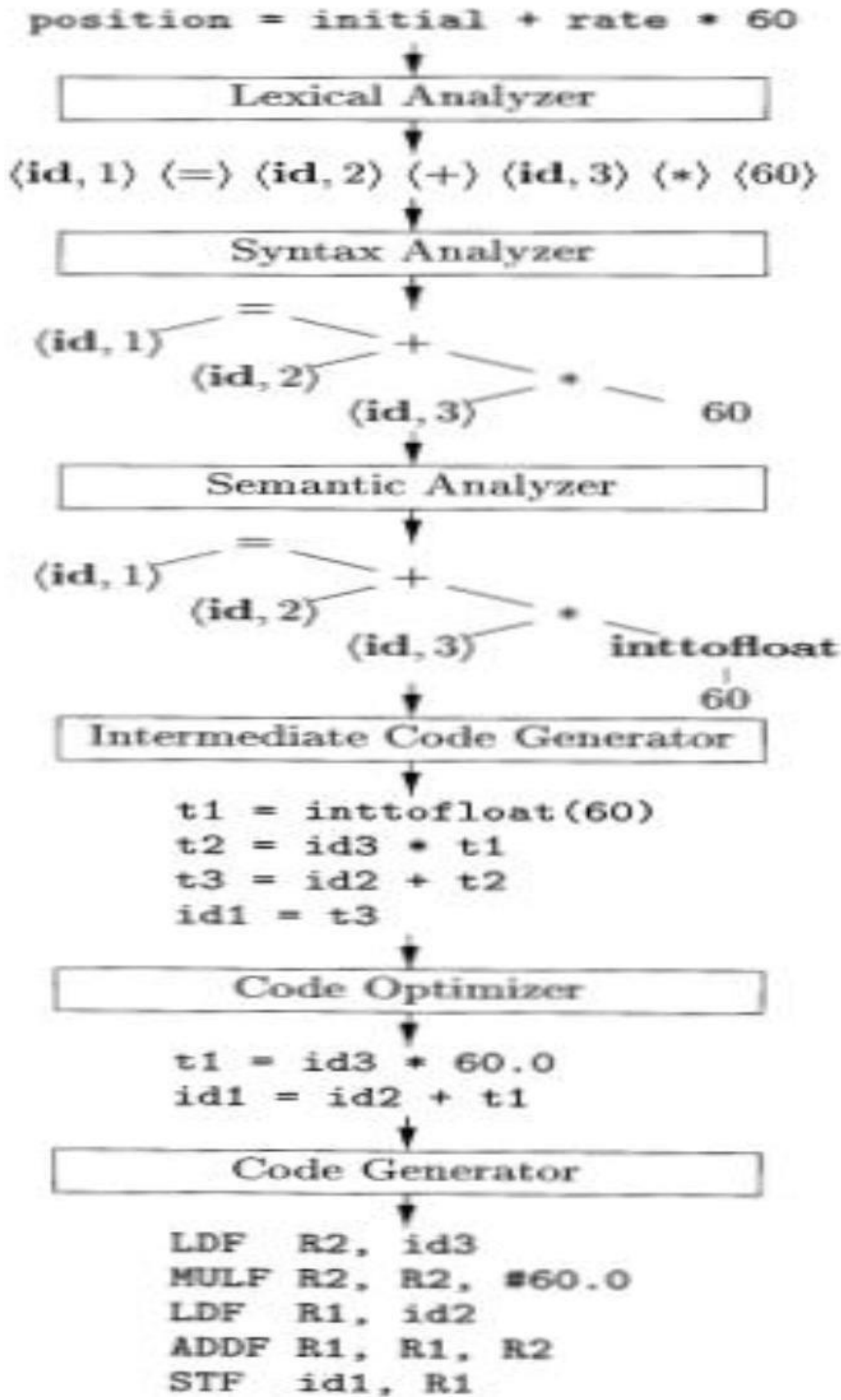


Figure1.5:TranslationofanassignmentStatement

LEXICALANALYSIS

As the first phase of a compiler, the main task of the lexical analyzer isto read theinput characters of the source program, group them into lexemes, and produce tokensfor each correct lexeme in the source program. This stream of tokens is sent to the parser for syntaxanalysis.Itiscommon forthelexicalanalyzertointeractwiththesymboltableaswell.

When the lexical analyzer discovers a lexeme constituting an a valid token, it storesthethatlexemeinto thesymboltable along with the generated token and its attributes.Apart from token generation, the scanners also performs the following

1. Escapes/removes the comments and spaces that are no interest in logic
2. Creates Symbol table
3. Reports lexical errors when a lexeme does not form a valid token

Thisprocessisshowninthefollowingfigure.

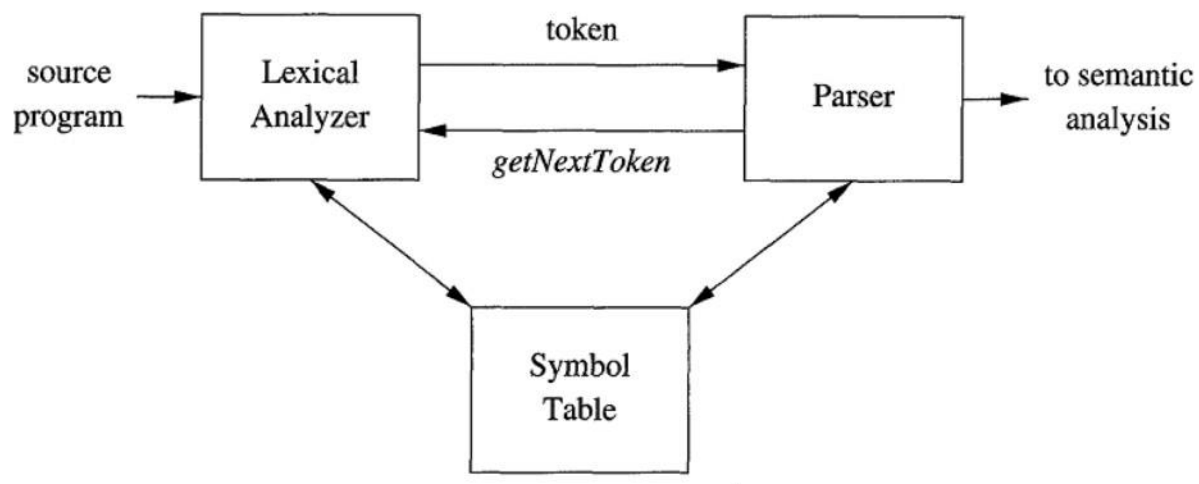


Figure1.6 :LexicalAnalyzer

. When lexical analyzer identifies the first token it will send it to the parser, the parserreceivesthetokenandcallsthelexicalanalyzertosendnexttokenbyissuingthe**getNextToken()** command. This Process continues until the lexical analyzer identifies all thetokens. During this process the lexical analyzer will neglect or discard the white spaces andcommentlines.

TOKENS,PATTERNSANDLEXEMES:

A token is a pair consistingofatokennameand an optional attribute value. The tokenname is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or asequence of input characters denoting an identifier. The token names are the input symbols thatthe parser processes. In what follows, we shall generally write the name of a token in boldface.We willoftenrefertoatokenbyits tokenname.

A pattern is a description of the form that the lexemes of a token may take [or match]. In thecaseof akeywordas a token, thepattern isjustthesequenceof characters thatform thekeyword. For identifiers and some other tokens, the pattern is a more complex structure that ismatchedbymanystrings.

Alexemeisasequence ofcharactersinthesourceprogramthatmatchesthepatternforatokenandis identifiedbythelexicalanalyzerasaninstanceofthattoken.

Example:InthefollowingC
languagestatement,printf("Total=%d\n‖,sc
ore);
both **printf**and **score** are lexemes matching the **pattern** fortoken**id**, and **"Total=%d\n**‖isalexemematching**literal[orstring]**.

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: Thereareanumber ofreasonswhytheanalysisportionofacompilerisnormallyseparatedintolexicalanalysis andparsing(syntaxanalysis)phases.

1. **Simplicity of design is the most important consideration.** The separation of LexicalandSyntacticanalysisoftenallowsustosimplifyatleastoneofthesetasks.Forexample,a parser thathad to deal with comments and whitespace as syntactic unitswouldbeconsiderablymorecomplexthanonethatcanassumecommentsandwhitespaceha ve alreadybeenremovedbythelexicalanalyzer.

2.Compiler efficiencyisimproved.A separatelexical analyzerallows us toapplyspecialized 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 compilersignificantly.

3. Compiler portability is enhanced: Input-device-specific peculiarities can berestrictedtothelexicalanalyzer.

INPUTBUFFERING:

Before discussing the problem of recognizinglexemes in the input, let us examinesomewaysthatthesimplebutimportanttaskofreadingthesourceprogramcanbespeeded up.This task is made difficult by the fact that we often have to look one or more characters beyondthe nextlexeme before we can be sure we have the rightlexeme.

There aremany situationswhere we need to look at least one additional character ahead. For instance, we cannot be surewe've seen the end of an identifier until we see a character that is not a letter or digit, andtherefore is not part of the lexeme for id.

In C,single-characteroperators like-,=,or<could alsobe the beginning of a two-character operator like ->, ==, or <=. Thus, we shallintroduce a two-bufferscheme thathandleslargelook aheadssafely.We then consideranimprovementinvolving "sentinels"thatsavestimecheckingfortheendsofbuffers.

BufferPairs

Becauseoftheamountoftimetakentoprocesscharactersandthelargenumberofcharactersthat must be processed during the compilation of a large source program, specialized bufferingtechniques have been developed to reduce the amount of overhead required to process a singleinputcharacter.Animportantschemeinvolvestwo buffersthatarealternatelyreloaded.

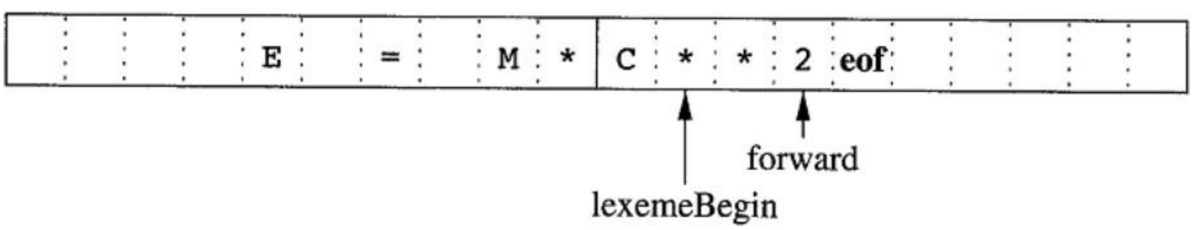


Figure1.8:UsingaPairofInput Buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096bytes.Using one system read command we can read N characters in toa buffer,rather thanusing one system call per character. If fewer than N characters remain in the input file, then aspecial character, represented by eof, marks the end of the source file and is different from anypossible characterofthesourceprogram.

- Following Twopointerstotheinputaremaintained:

1. ThePointer**lexemeBegin**,marksthebeginningofthecurrentlexeme,whoseextentweare attemptingtodetermine.
2. Pointer **forward** scans ahead until a pattern match is found; the exact strategywherebythis determinationismadewillbe coveredinthe balance ofthischapter.

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, 1exemeBeginis set to the character immediately after the lexeme just found. In Fig, we see forward has passedthe end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retractedone positiontoits left.

Advancing forward requires that we first test whether we have reached the endofoneof the buffers, andif so, we mustreload the other bufferfromthe input, and move forward tothe beginning of the newly loaded buffer.

As long as we never need to look so far ahead of theactual lexeme that the sum of the lexeme's length plus thedistance welook ahead is greaterthanN,weshallneveroverwrite thelexemein itsbufferbeforedeterminingit.

SentinelsToImproveScannersPerformance:

If we use the above scheme as described, we must check, each time we advance forward,thatwe havenotmoved off one of thebuffers; if we do, then wemustalsoreload the otherbuffer.Thus,foreachcharacterread,wemaketwotests:onefortheendofthebuffer,andoneto determine what character is read (the latter may be a multi way branch).

We can combine thebuffer-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 character that cannot be part of the source program,and a natural choice is the character **eof**.

Figure 1.8 shows the same arrangement as Figure 1.7,but with the sentinels added. Note that eof retains its use as a marker for the end of the entireinput.

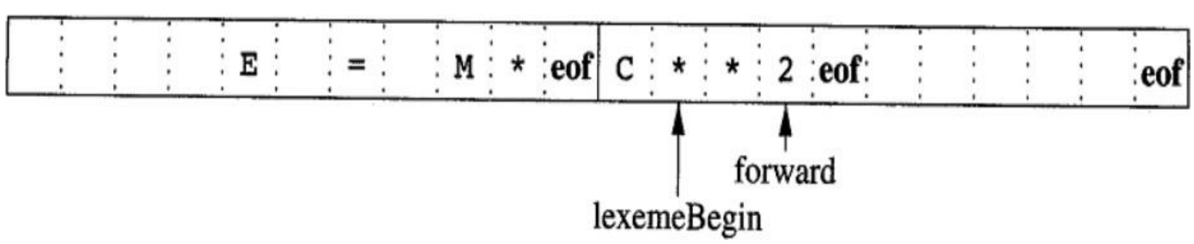


Figure1.8 :Sententialattheend ofeachbuffer

Any**eof**thatappearsotherthanattheends ofabuffermeansthat theinputisatanend.Figure1.9summarizesthealgorithmfor advancingforward.

Noticehowthefirsttest,whichcanbepartofamultiwaybranchbasedonthecharacterpointedtobyforward,isthe onlytestwemake,exceptinthe casewhere we actuallyareattheendofa bufferorthe endoftheinput.

```
switch(*forward++)
{
        caseeof:if(forwardisat endoffirstbuffer)
                  {
                    reloadsecondbuffer;
                    forward=beginningofsecondbuffer;
                   }
                  elseif(forward isatend ofsecondbuffer)
                  {
                    reloadfirstbuffer;
                    forward=beginningoffirstbuffer;
                  }
                    else
                         /*eofwithinabuffermarkstheendofinput*/termi
                      nate lexicalanalysis;
           break;
}
```

Figure1.9:useofswitch-case forthe sentential

SPECIFICATIONOFTOKENS:

Regular expressions are an important notation for specifying lexeme patterns. While they cannot expressallpossiblepatterns,theyarevery effectiveinspecifyingthosetypesofpatterns thatweactuallyneed fortokens.

LEXtheLexicalAnalyzergenerator

Lex is a tool used to generate lexical analyzer, the input notation for the Lex tool isreferred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, theLexcompiler transformstheinputpatternsinto atransitiondiagramandgeneratescode,ina filecalledlex.yy.c,itisacprogramgivenforCCompiler,givestheObjectcode.Hereweneedto know howtowritetheLexlanguage.The structureofthe Lexprogramisgivenbelow.

StructureofLEXProgram: ALexprogramhasthefollowingform:

Declarations
%%
Translationrules
%%
Auxiliaryfunctionsdefinitions

The declarations section : includes declarations of variables, manifest constants (identifiersdeclared to stand for a constant, e.g., the name of a token), and regular definitions. It appearsbetween%{...%}

In the **Translation rules** section, We place Pattern Action pairs where each pair have the formPattern{Action}

The auxiliary function definitions section includes the definitions of functions used to installidentifiersandnumbersintheSymboltale.

LEXProgramExample:

```
%{

/*definitionsofmanifestconstantsLT,LE,EQ,NE,GT,GE,IF,THEN,ELSE,ID,NUMBER,RELO
P*/

%}

/*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(){/*functiontoinstallthelexeme,whosefirstcharacterispointedtobyyytext,andwhoselengthisyyleng,into thesymboltableandreturnapointerthereto*/
intinstallNum(){/*similartoinstallID,butputsnumericalconstants intoaseparatetable*/}

Figure1.10 :LexProgramfortokens commontokens

SYNTAXANALYSIS(PARSER)

THEROLEOFTHEPARSER:

Inourcompilermodel,theparserobtainsastringoftokensfromthelexicalanalyzer,as shown in the below Figure, and verifiesthatthestringof token names can be generatedby the grammar forthesource language. We expect the parser to report any syntax errors inan intelligible fashion and to recover from commonly occurring errors to continue processing theremainder of the program. Conceptually, for well-formed programs, the parser constructs a parsetreeandpassesittothe rest ofthecompilerforfurtherprocessing.

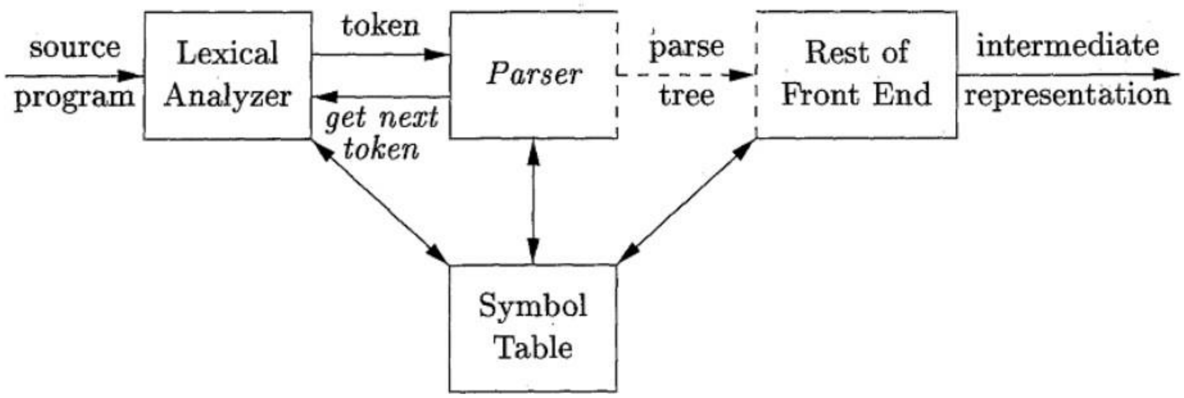


Figure2.1:ParserintheCompiler

Duringtheprocessofparsingitmayencountersomeerrorandpresenttheerrorinformationbacktotheuser

Syntacticerrorsincludemisplacedsemicolonsorextraormissingbraces;thatis, –{"or"}."Asanotherexample,inC or Java, theappearanceof a case statement without an enclosing switch is a syntactic error (however, this situationisusually allowedby theparserand caughtlaterintheprocessing,asthe compilerattemptsto generatecode).

Based on the way/order the Parse Tree is constructed, **Parsing** is basically **classified** in tofollowingtwotypes:

1. **TopDownParsing:**Parsetreeconstructionstartattheroot nodeandmovestothechildrennodes (i.e.,topdownorder).
2. **BottomupParsing:**Parsetreeconstructionbegins fromtheleafnodesandproceedstowardstherootnode(calledthebottomuporder).

IMPORTANT(OR)EXPECTEDQUESTIONS

1. WhatisaCompiler?ExplaintheworkingofaCompilerwithyour ownexample?
2. WhatistheLexicalanalyzer?DiscusstheFunctionsofLexicalAnalyzer.
3. Writeshortnotesontokens,patternandlexemes?
4. WriteshortnotesonInputbufferingscheme?Howdoyouchangethebasicinputbufferi ngalgorithmtoachievebetterperformance?
5. WhatdoyoumeanbyaLexicalanalyzer generator?ExplainLEXtool.

ASSIGNMENTQUESTIONS:

1. Writethedifferencesbetweencompilersandinterpreters?
2. Writeshortnotesontokenreorganization?
3. WritetheApplicationsoftheFiniteAutomata?
4. ExplainHowFinite automata areusefulinthelexicalanalysis?
5. ExplainDFAandNFAwithanExample?

UNIT-II

TOPDOWNPARSING:

- ∑ Top-down parsing can be viewed as the problem of constructing a parse tree for the giveninput string, starting from the root and creating the nodes of the parse tree in preorder(depth-firstlefttoright).

- ∑ Equivalently, top-down parsing can beviewedas finding aleftmostderivation foraninputstring.

ItisclassifiedintotwodifferentvariantsnamelyonewhichusesBackTrackingandtheotherisNonBackTrackinginnature.

NonBackTrackingParsing: Therearetwovariantsofthisparserasgiven below.

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

BackTracking

1.BruteForcemethod

NONBACKTRACKING:

LL(1) ParsingorPredictiveParsing

LL(1)standsfor, lefttorightscanofinput,usesaLeftmostderivation,andtheparsertakes1 symbolasthelook ahead symbolfromtheinputintaking parsing actiondecision.

A non recursive predictive parser can be built by maintaining a stack explicitly, ratherthan implicitly via recursive calls. The parser mimics a leftmost derivation. If w is the inputthat has been matchedso far, thenthestack holds a sequence ofgrammar symbols a suchthat

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

Thetable-drivenparserinthefigurehas

- ∑An input buffer that contains the string to be parsed followed by a $ Symbol, used toindicate endofinput.

- ∑A stack, containing a sequence of grammar symbols with a $ at the bottom of the stack,whichinitiallycontains thestartsymbolofthegrammarontopof$.

- ∑AparsingtablecontainingtheproductionrulestobeappliedThisisatwodimensionalarrayM

[Nonterminal,Terminal].

∑A parsing Algorithm that takes input String and determines if it is conformant

toGrammaranditusestheparsingtable and stackto take suchdecision.

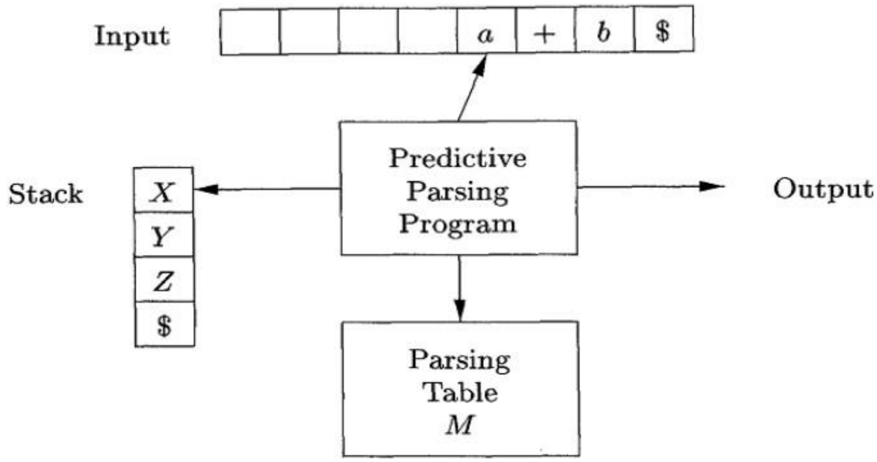


Figure2.2:Model fortabledrivenparsing

TheStepsInvolvedInconstructing anLL(1)Parser are:

1. WritetheContextFreegrammarforgiveninputString
2. CheckforAmbiguity.Ifambiguousremoveambiguityfromthegrammar
3. CheckforLeftRecursion.Removeleftrecursionifitexists.
4. CheckForLeft Factoring.Performleftfactoringifit containscommonprefixesinmore thanonealternates.
5. ComputeFIRSTandFOLLOWsets
6. Construct LL(1)Table
7. Using LL(1)Algorithmgenerate Parsetree astheOutput

Context Free Grammar (CFG): CFG used to describe or denote the syntax of theprogramming language constructs. The CFG is denoted as G, and defined using a four tuplenotation.

Let G be CFG, then G is written as, G= (V, T, P, S)Where

∑V is a finite set of Non terminal; Non terminals are syntactic variables that denote sets ofstrings. The sets of strings denoted by non terminals help define the language generatedbythe grammar.Nonterminalsimpose a hierarchicalstructureonthelanguagethat iskeytosyntaxanalysisandtranslation.

∑ T is a Finite set of Terminal; Terminals are the basic symbols from which strings areformed. The term "token name" is a synonym for "'terminal" and frequently we will usethe word "token" for terminal when it is clear that we are talking about just the tokenname. We assume that the terminals are the first components of the tokens output by thelexicalanalyzer.

∑ S is the Starting Symbol of the grammar, one non terminal is distinguished as the

startsymbol, and the set of strings it denotes isthe language generated by the grammar.
PisfinitesetofProductions;theproductionsofagrammar specifythemannerinwhichthe

terminalsand nonterminalscanbecombined toformstrings,eachproduction isinα->βform,whereαisa singlenonterminal,βis(VUT)*.Eachproductionconsistsof:

(a) A non terminal called theheadorleftsideoftheproduction; thisproductiondefines someofthe strings denotedbythehead.

(b) Thesymbol->.Sometimes: =hasbeenusedinplaceofthearrow.

(c) Abody orrightsideconsistingofzeroormoreterminalsandnon-terminals. The components of the body describe one way in which strings of the nonterminalattheheadcanbeconstructed.

∑Conventionally, theproductionsforthestartsymbolarelistedfirst.

Example:Context FreeGrammartoacceptArithmeticexpressions.

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 avoidalwayshaving to state that –these are the terminals,""these are the non terminals,"andsoon,thefollowingnotationalconventionsforgrammarswillbeusedthroughoutourdiscussions.

1. Thesesymbolsareterminals:

(a) Lowercaselettersearlyinthealphabet,such asa,b,e.

(b) Operatorsymbolssuchas**+**,*****,**andsoon.

(c) Punctuationsymbolssuchasparentheses, comma,andsoon.

(d) Thedigits0,1... 9.

(e) Boldface strings such as id or if, each of which represents a singleterminalsymbol.

2. Thesesymbolsarenonterminals:

(a) Uppercaselettersearlyinthealphabet,suchasA,B,*C*.

(b) The letterS,which,whenitappears,isusuallythestartsymbol.

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

(d) Whendiscussingprogrammingconstructs,uppercaselettersmaybeusedtorepresentNon terminals for the constructs. For example, non terminal for expressions, terms,andfactorsareoftenrepresentedbyE,T,andF,respectively.

Usingtheseconventionsthegrammar forthearithmeticexpressionscanbewrittenas

E→E+T |E–T |T
T→T* F | T / F |
F→(E)|id

DERIVATIONS:

The construction of a parse tree can be made precise by taking a derivational view, inwhich productions are treated as rewriting rules. Beginning with the start symbol, each rewritingstep replaces a Non terminal by the body of oneof its productions. This derivational viewcorresponds to the top-down construction of a parse tree as well as the bottom construction of theparse tree.

∑Derivationsareclassifiedinto**Let mostDerivation**and**RightMostDerivations.**

LeftMostDerivation(LMD):

It is the process of constructing the parse tree or accepting the given inputstring,inwhich at every time we need to rewrite the production rule it is done with left most non terminalonly.
Ex: -IftheGrammarisE->**E+E| E*E|-E| (E)|id** and theinputstringis**id +id*id**

The production **E -> - E** signifies that if E denotes an expression, then **–** E must also denote anexpression.The replacementofa single Eby-Ewillbe describedbywriting

E=>-Ewhich isread as**“Ederives_E”**

For a general definition of derivation, consider a non terminal A in the middle of asequenceof grammarsymbols,asin αAβ,whereαandβ arearbitrary strings of grammarsymbol. Suppose A ->γ is a production. Then, we write αAβ => αγβ. The symbol => means"derivesinonestep".Often,wewishtosay,"Derivesinzeroormoresteps."Forthispurpose,we can use the symbol $\stackrel{*}{\Rightarrow}$, If we wish to say, "Derives in $\stackrel{+}{\Rightarrow}$one or more steps." We cn usethe symbol $\stackrel{+}{\Rightarrow}$. If S $\stackrel{*}{\Rightarrow}$ a, where S is the start symbol of a grammar G, we say that α is asententialformofG.

TheLeftmostDerivationforthegiveninputstring**id+id*id** is

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

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

=>id+<u>E</u>*E

=>id+id*<u>E</u>

=>id+id*id

NOTE: Everytimeweneedtostartfromtherootproductiononly,theunderlineusingatNonterminal indicating that, it is the non terminal (left most one) we are choosing to rewrite theproductionstoacceptthestring.

RightMostDerivation(RMD):

Itistheprocessofconstructingtheparsetreeoracceptingthegiveninputstring, everytimeweneedtorewritetheproductionrule withRightmostNonterminalonly.

TheRightmostderivationforthegiveninputstring**id+id*id**is

E=>E+<u>E</u>
=>E+E*<u>E</u>
=>E+ <u>E</u>*id
=><u>E</u>+id*id
=>id+id*id

NOTE: Every time we need to start from the root production only, the under line using at Nonterminal indicating that, it is the non terminal (Right most one) we are choosing to rewrite theproductionstoacceptthestring.

WhatisaParseTree?

Aparse tree isa graphicalrepresentationofa derivationthatfiltersoutthe orderinwhichproductionsareappliedtoreplacenonterminals.

- ∑Each interiornodeofaparsetree representstheapplicationofaproduction.
- ∑AlltheinteriornodesareNonterminalsand alltheleafnodesterminals.
- ∑Alltheleafnodesreadingfromthelefttorightwillbethe outputofthe parsetree.
- ∑Ifanode nislabeledXand haschildrenn1,n2,n3,…nkwithlabelsX1,X2,…Xkrespectively,thentheremustbe aproduction A->X1X2…Xkinthegrammar.

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

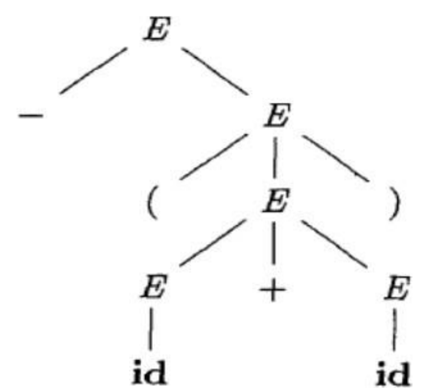


Figure2.4: ParseTreefortheinputstring**-(id+id)**

TheFollowingfigureshowsstep bystep constructionofparsetreeusing CFGfortheparsetreefortheinputstring**-(id+id).**

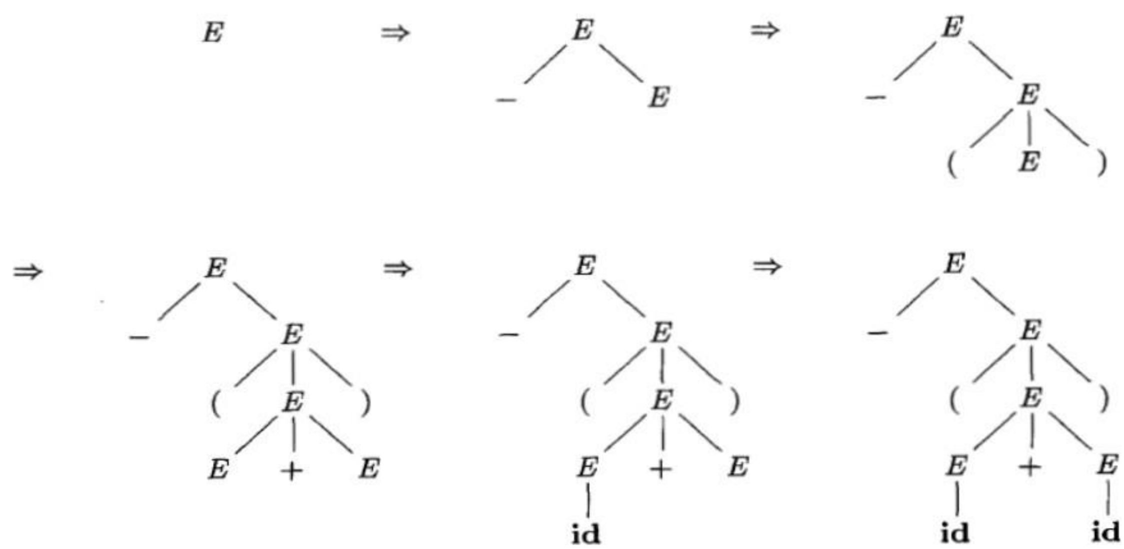


Figure2.5 : SequenceoutputsoftheParseTreeconstructionprocessfortheinputstring**–(id+id)**

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

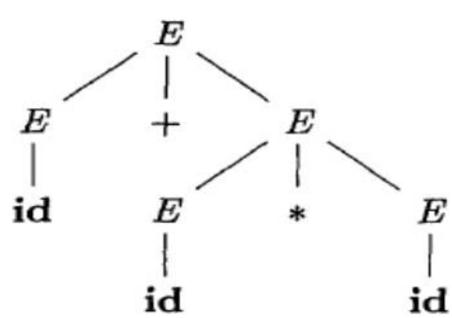


Figure2.6: Parsetreefortheinputstring id+id*id

AMBIGUITYinCFGs:

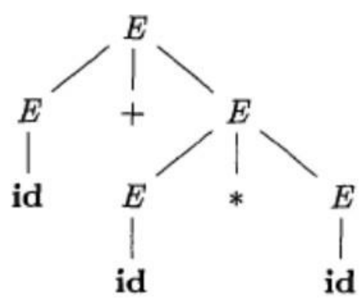
Definition:

Agrammarthatproducesmorethanoneparsetreeforsomesentence(inputstring)issaidtobeambiguous
.

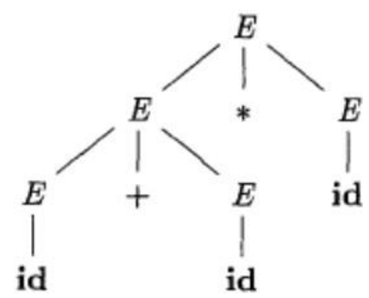
Inother words,
anambiguousgrammarisonethatproducesmorethanoneleftmostderivationormorethanone rightmostderivationforthesame sentence.

Or If the right hand production of the grammar is having two non terminals which areexactlysameas lefthandsideproductionNonterminalthenitissaidtoanambiguousgrammar.Example:
IftheGrammaris**E->E+E| E*E|-E|(E)| id**andtheInputStringis**id+id* id**

Twoparsetreesforgiveninputstringare



(a)



(b)

TwoLeftmostDerivationsforgiveninputStringare:

(a)	(b)
E=><u>E</u>+E	**E=><u>E</u>*E**
=>id+<u>E</u>	**=><u>E</u>+E*E**
=>id+<u>E</u> *E	**=>id+ <u>E</u>*E**
=>id+id*<u>E</u>	**=>id+id*<u>E</u>**
=>id+id*id	**=>id+id*id**
(a)	(b)

TheaboveGrammarisgivingtwoparsetreesortwoderivations forthegiveninputstringso,itisanambiguousGrammar

Note: LL (1) parser will not accept the ambiguous grammars or We cannot construct anLL(1) parser for the ambiguous grammars. Because such grammars may cause the TopDown parser to go into infinite loop or make it consume more time for parsing. If necessarywemustremove alltypesofambiguityfromitandthenconstruct.

ELIMINATING AMBIGUITY: SinceAmbiguous grammars may cause the top down Parsergointoinfiniteloop,consumemore time duringparsing.

Therefore, sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. Thegeneralformofambiguous productionsthatcause ambiguityingrammarsis

$$\boxed{A \rightarrow A\alpha|\beta}$$

Thiscanbewrittenas(introduceonenewnonterminalin theplaceofsecondnonterminal)

$$\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A'|\varepsilon \end{array}}$$

Example:LetthegrammarisE → **E+E|E*E|-E|(E)|id.**Itisshownthatitisambiguousthatcanbewrittenas

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

Intheabovegrammarthe1st and2nd productionsarehaving ambiguity.So,theycanbewrittenas

E->E+E|E*Ethisproductionagaincanbewrittenas
E->E+E| β,whereβisE*E
Theaboveproductionissameasthegeneralform.so,thatcanbewrittenasE->E+T|T
T->β

ThevalueofβisE*Eso,abovegrammar canbewrittenas

1) **E->E+T|T**
2) **T->E*E** **Thefirstproductionisfreefromambiguity**andsubstituteE->Tinthe 2nd productionthenitcanbewrittenas
T->T*T|-E|(E)|**id**this productionagaincanbewrittenas
T->T*T |βwhereβis -E|(E)|id,introducenewnonterminalintheRighthandsideproductionthenitbecomes
T->T*F|F
F->-E |(E)|id nowtheentiregrammarturned intoitequivalentunambiguous,

TheUnambiguousgrammarequivalenttothegivenambiguousoneis

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 leftrecursion. A grammar is left recursive if it has a non terminal A such that there is a derivationA**=>**Aα for some string α in (TUV)*. LL(1) or Top Down Parsers can not handle the

LeftRecursive grammars, so we need to remove the left recursion from the grammars before beingusedinTopDownParsing.

TheGeneralformofLeftRecursionis

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

Theaboveleftrecursiveproductioncanbewrittenasthenonleftrecursiveequivalent:

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

Example: -
Isthefollowinggrammarleftrecursive?Ifso,findanonleftrecursivegrammarequivalenttoit.

E → E+T |T

T → T * F |
F→E|(E)|id

Yes,thegrammarisleftrecursiveduetothefirsttwoproductions whicharesatisfyingthe

generalformofLeftrecursion,sotheycan berewritten afterremovingleftrecursionfrom

E→ E+T,and **T→T*F**is

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

LEFTFACTORING:

Left factoring is a grammar transformation that is useful for producing a grammar suitable forpredictive or top-down parsing. A grammar in which more than one production has commonprefixis toberewrittenbyfactoringouttheprefixes.

Forexample,inthefollowinggrammartherearenAproductions havethecommonprefix**α,** whichshouldberemovedorfactoredoutwithoutchangingthelanguagedefinedforA.

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

Wecanfactoroutthe**α**fromallnproductionsbyaddinga newAproduction**A → αA′**, andrewritingtheA′productionsgrammaras

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

FIRSTandFOLLOW:

The construction of both top-down and bottom-up parsers is aided by two functions,FIRST and FOLLOW, associated with a grammar G. During top down parsing, FIRST andFOLLOW allow us to choose which production to apply, based on the next input (look a head)symbol.

ComputationofFIRST:

FIRST function computes the set of terminal symbols with which the right hand side ofthe productions begin. To compute FIRST (A) **for all grammar symbols**, apply the followingrulesuntilnomoreterminals or€canbeaddedtoanyFIRSTset.

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

ComputationOfFOLLOW:

Follow(A)isnothingbutthesetofterminalsymbolsofthegrammarthatareimmediately following the Non terminal A. If **a** is to the immediate right of non terminal A, thenFollow(A)= {**a**}. To compute FOLLOW (A) for **all non terminals**A, apply the following rulesuntilnomore symbols canbeaddedtoanyFOLLOWset.

1. Place $ in FOLLOW(S), where *S* is the start symbol, and $ is the input right endmarker.
2. IfthereisaproductionA->αBβ, theneverythinginFIRST(β)except€isinFOLLOW(B).
3. IfthereisaproductionA->αBor aproductionA->αBβwithFIRST(β) contains €,thenFOLLOW(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.FO

LLOW (E′)={FOLLOW (E)} satisfyingthe3rdruleofFOLLOW()

= { $,)}

FOLLOW (T)={FIRSTE′} ItisSatisfyingthe2ndrule.

U{FOLLOW(E′)}

= {+,FOLLOW(E′)}

= { +,$,)}

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

= {+, $,)}

FOLLOW(F)={FIRST(T′)} Itis Satisfyingthe2ndrule.

U{FOLLOW(E′)}

={*,FOLLOW(T)}

= { *, +, $,)}

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

Table2.1:FIRST andFOLLOWvalues

ConstructingPredictiveOrLL(1)ParseTable:

Itistheprocessofplacing theallproductionsofthegrammarintheparsetablebased ontheFIRSTandFOLLOWvaluesoftheProductions.

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],wherebisallterminalsinFOLLOW(A).
   ii.If€isinFIRST(α) and $isinFOLLOW(A)thenaddproductionA->αtoM[A,$].
4. Markotherentriesintheparsingtableaserror.

	INPUTSYMBOLS

NON-TERMINALS	+	*	(	)	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:
iftherearenomultipleentriesinthetableforsingleaterminalthengrammarisacceptedbyLL(1)Parser.

LL(1)ParsingAlgorithm:

Theparseractsonbasisonthebasisoftwo symbols

i. A,thesymbolonthetopofthestack
ii. a,thecurrentinputsymbol

Therearethreeconditions forAand_a‘,thatareusedfrotheparsingprogram.

1. IfA=a=$thenparsingisSuccessful.
2. IfA=a≠$ thenparserpopsoffthestack andadvancesthecurrentinputpointertothenext.
3. If A is a Non terminal the parser consults the entry M [A, a] in the parsing table. IfM[A, a] is a Production A-> $X_1X_2..X_n$, then the program replaces the A on the top oftheStackby$X_1X_2..X_n$insuchawaythatX_1comes onthetop.

STRINGACCEPTANCEBYPARSER:

Iftheinputstringfortheparser is**id+id * id,**thebelowtableshowshowtheparseracceptthestringwiththehelpofStack.

Stack	Input	Action	Comments
$E	id+id*id$	E TE`	Eontopofthestackis replacedbyTE`
$E`T	id+ id*id$	T FT`	Tontopofthestackis replacedbyFT`
$E`T`F	id+ id*id$	F id	Fontopofthestackisreplacedbyid
$E`T`id	id+ id*id$	popandremoveid	Condition2issatisfied
$E`T`	+id*id$	T` €	T`ontopofthestackisreplacedby€
$E`	+id*id$	E` +TE`	E`ontopofthestackisreplacedby+TE`
$E`T+	+id*id$	Popandremove+	Condition2 issatisfied
$E`T	id*id$	T FT`	Tontopofthestackis replacedbyFT`
$E`T`F	id*id$	F id	Fontopofthestackisreplacedbyid
$E`T`id	id*id$	popandremoveid	Condition2 issatisfied

$E`T`	*id$	T` *FT`	T`ontopofthestackis replacedby*FT`
$E`T`F*	*id$	popandremove*	Condition2 issatisfied
$E`T`F	id$	F id	Fontopofthestackisreplacedbyid
$E`T`id	id$	Popandremoveid	Condition2 issatisfied
$E`T`	$	T` €	T`ontopofthestackisreplacedby€
$E`	$	E` €	E`ontopofthestackisreplacedby€
$	$	Parsing issuccessful	Condition1satisfied

Table2.3 : Sequenceofsteps takenbyparserinparsing theinput**tokenstreamid+id* id**

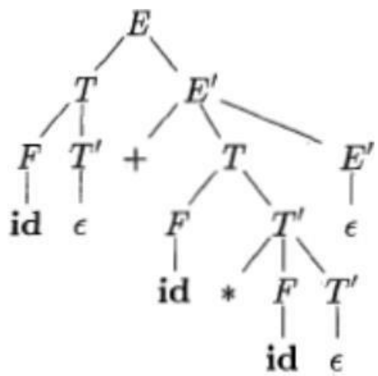


Figure2.7:Parsetreefortheinputid+id*id

ERRORHANDLING(RECOVERY)INPREDICTIVEPARSING:

In table driven predictive parsing, it is clear as to which terminal and Non terminals theparserexpectsfromthe rest ofinput.Anerrorcanbedetected inthe following situations:

1. Whentheterminalontopofthestack doesnotmatchthecurrentinputsymbol.
2. when Non terminal A is on top of the stack, a is the current inputsymbol, andM[A,a]is emptyorerror

The parser recovers from the error and continues its process. The following error recoveryschemesareuseinpredictiveparsing:

PanicmodeErrorRecovery:

It is based on the idea that when an error is detected, the parser will skips theremaining input until a synchronizing token is en countered in the input. Some examples arelistedbelow:

1. ForaNonTerminalA,placeallsymbolsinFOLLOW(A)areaddeintothesynchronizingsetof nonterminalA.ForExample,considertheassignmentstatement –c=;‖Here,theexpressionontherighthandsideismissing.SotheFollowofthisis considered. It is–;‖and istakenassynchronizingtoken.Onencounteringit,parser emitsanerrormessage–MissingExpression‖.
2. ForaNonTerminalA,placeallsymbolsinFIRST(A)areaddeintothesynchronizingsetofno nterminalA.ForExample,considertheassignmentstatement –22c=a+b;‖Here,FIRST(expr)is22.Itis–;‖andistakenassynchronizingtoken

andthenthereportstheerror as–extraneous token‖.

PhraseLevelRecovery:

It can be implemented in the predictive parsing by filling up the blank entries inthe predictive parsing table with pointers to error Handling routines. These routines caninsert,modifyordeletesymbolsintheinput.

RECURSIVEDESCENTPARSING:

A recursive-descent parsing program consists of a set of recursive procedures, one for each nonterminal. Each procedure is responsible for parsing the constructs defined by its non terminal,Execution begins with the procedure for the start symbol, which halts and announces success ifitsprocedurebodyscanstheentireinputstring.

Ifthegivengrammaris

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

Reccursiveproceduresfortherecursivedescentparser forthegivengrammar aregivenbelow.

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

```
ifinput=_*‘
{
        advance();
        F();
```

```
        T′();
    returntrue;
    }
    else returnerror;
}
procedureF()
{
    ifinput=‗(‗
    {
        advance();
        E();
        if input =
        ‗)‘advance()
        ;returntrue;
    }
    elseifinput= ―id‖
    {

        advance(
        );returntrue
        ;
    }
    else returnerror;
}
advance()
{
    input=nexttoken;
}
```

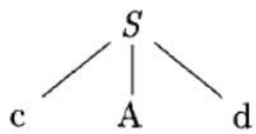
BACK TRACKING: This parsing method uses the technique called Brute Force methodduring the parse tree construction process. This allows the process to go back (back track) andredothe steps byundoingthe workdonesofarinthe pointofprocessing.

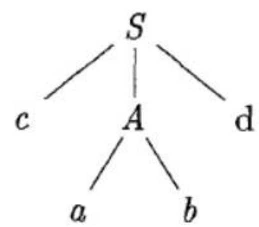
Brute force method: It is a Top down Parsing technique, occurs when there is morethan one alternative in the productions to be tried while parsing the input string. It selectsalternativesintheordertheyappearandwhenitrealizesthatsomethinggonewrongittrieswithnext alternative.
Forexample,considerthegrammarbellow.

S → cAd
A → ab|a

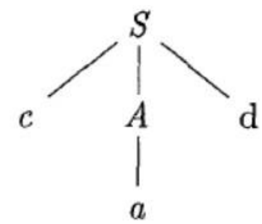
Togeneratetheinputstring―cad‖,initiallythefirstparsetreegivenbelowisgenerated.

Asthestringgeneratedisnot–cad‖,inputpointerisbacktrackedtoposition–A‖,toexaminethe next alternateof –A‖. Nowamatch to the input string occursasshown in the 2nd parsetrees givenbelow.





(1)



(2)

IMPORTANTANDEXPECTEDQUESTIONS

1. ExplainthecomponentsofworkingofaPredictiveParserwithanexample?
2. WhatdotheFIRSTandFOLLOWvaluesrepresent?GivethealgorithmforcomputingFIRST nFOLLOWofgrammarsymbolswithanexample?
3. ConstructtheLL(1)Parsingtableforthefollowinggrammar?E
   → E+T|T

   T → T*F

   F → (E)|id
4. Fortheabovegrammarconstruct, andexplaintheRecursiveDescentParser?
5. Whathappensif multipleentriesoccurringinyourLL (1)Parsingtable?Justifyyouranswer?HowdoestheParser

ASSIGNMENTQUESTIONS

1. EliminatetheLeftrecursionfromthebelowgrammar?

   A-> Aab| AcB|b

   B->Ba|d
2. Explaintheproceduretoremovetheambiguityfromthegivengrammarwithyourownexampl e?
3. Writethegrammarfortheif-elsestatementintheCprogrammingandcheckfortheleftfactoring?
4. WillthePredictiveparseraccepttheambiguousGrammarjustifyyour answer?
5. IsthegrammarG={ S->L=R,S->R,R->L,L->*R|**id**}anLL(1)grammar?

BOTTOM-UPPARSING

Bottom-up parsing corresponds to the construction of a parse tree for an input stringbeginning at the leaves (the bottom nodes) and working up towards the root (the top node). Itinvolves–reducing an input string ‗w' to the Start Symbol ofthe grammar. in eachreduction step, a perticular substring matching the right side of the production is replaced by symbol on theleft of that production and it is the Right most derivation. For example consider the followingGrammar:

E → E+T|T

T → T*F

F → (E)|id

Bottomupparsingoftheinputstring**"id *id"**isasfollows:

INPUTSTRING	**SUBSTRING**	**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 Stringisaccepted

ParseTreerepresentationisas follows:

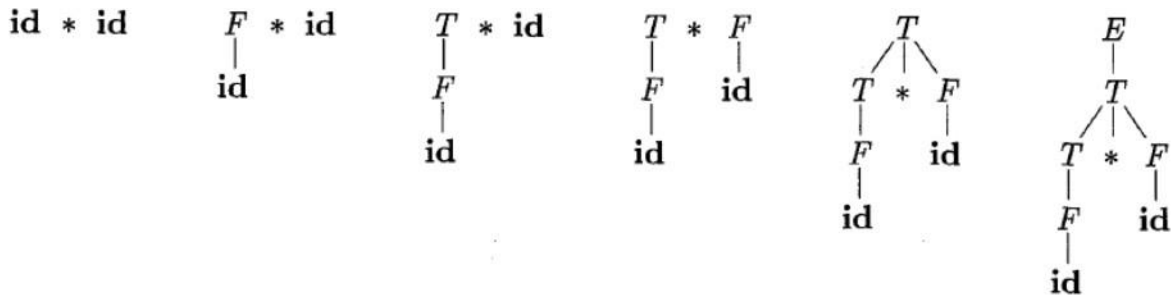


Figure3.1 :ABottom-up ParsetreefortheinputString“id*id”

Bottomupparsingisclassifiedinto1. Shift-ReduceParsing, 2. OperatorPrecedenceparsing,and3.[TableDriven]LR 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 grammarsymbols and an input buffer holds the rest of the string to be parsed, We use$ to mark thebottom of the stack and also the right end of the input. And it makes use of the process of shiftand reduce actions to accept the input string. Here, the parse tree is Constructed bottom up fromthe leafnodes towardstherootnode.

When we are parsing the given inputstring, if the match occurs the parser takes thereduceactionotherwiseitwillgoforshiftaction.Anditcanacceptambiguousgrammarsalso.

For example,considerthebelowgrammartoaccepttheinputstring–id*id–,usingS-Rparser

E → E+T|T

T → T*F|F

F → (E)|id

ActionsoftheShift-reduceparser usingStackimplementation

STACK	INPUT	ACTION
$	Id*id$	Shift
$id	*id$	Reduce withF → d
$F	*id$	Reduce withT → F
$T	*id$	Shift →
$T*	id$	Shift →
$T*id	$	Reduce withF → id
$T*F	$	Reduce withT T*F
$T	$	Reduce withE T
$E	$	Accept

Considerthefollowinggrammar:

S → aAcBe

A → Ab|b

B → d

Lettheinputstringis―abbcde‖.Theseriesofshiftandreductionstothestartsymbolareas follows.

abbcde ⇨ aAbcde ⇨ aAcde ⇨ aAcBe ⇨ S

Note:intheaboveexampletherearetwo actionspossibleinthesecond Step,theseareasfollows:

1. Shiftactiongoingto3rdStep
2. Reduceaction,thatisA->b

Iftheparseristakingthe1staction thenitcansuccessfully acceptsthegiveninputstring,ifitisgoingforsecondactionthenitcan‗tacceptgiveninputstring.Thisiscalledshiftreduceconflict. Where, S-R parser is not able take proper decision, so it not recommended for parsing.**OPERATORPRECEDENCE PARSING:**

Operatorprecedencegrammariskindsofshiftreduceparsingmethodthatcanbeappliedtoasmallclassofoperatorgrammars.Anditcanprocessambiguousgrammarsalso.

∑Anoperatorgrammarhastwoimportantcharacteristics:

1. TherearenoЄproductions.
2. Noproductionwouldhavetwo adjacentnonterminals.

∑Theoperatorgrammartoacceptexpressions isgivebelow:

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

TwomainChallenges intheoperatorprecedenceparsingare:

1. IdentificationofCorrecthandles inthereductionstep, suchthatthegiveninputshould bereducedtostartingsymbolofthegrammar.
2. Identificationofwhichproductionto useforreducinginthereductionsteps,suchthatweshouldcorrectlyreduce the giveninputtothe startingsymbolofthegrammar.

Operatorprecedenceparserconsistsof:

1. Aninputbufferthatcontainsstringtobeparsedfollowedbya$, asymbolusedtoindicate theendingofinput.
2. Astackcontaininga sequence ofgrammarsymbolswitha $atthe bottomofthestack.
3. An operator precedence relation table O, containing the precedence ralations between thepair of terminal. There are three kinds of precedence relations will exist between the pairofterminalpair_a‗and_b‗asfollows:
4. Therelationa<•bimpliesthat heterminal_a‗haslowerprecedencethanterminal_b‗.
5. Therelationa•>bimpliesthat heterminal_a‗hashigherprecedencethanterminal_b‗.
6. Therelationa=•bimpliesthatheterminal_a‗haslowerprecedencethanterminal_b‗.

7. An operator precedence parsing program takes an input string and determines whether itconforms to the grammar specifications. It uses an operator precedence parse table andstacktoarriveatthedecision.

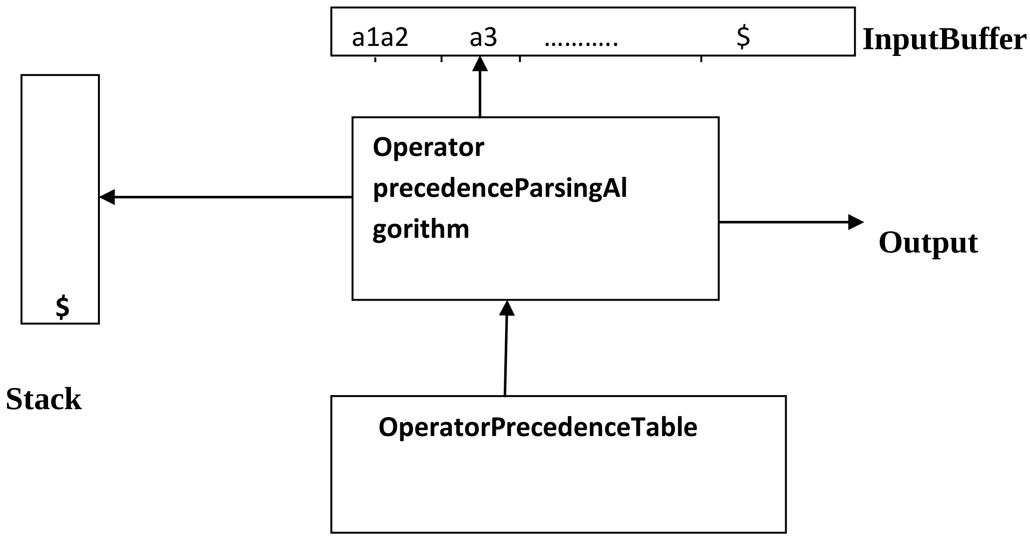


Figure3.2:Componentsofoperatorprecedenceparser

Example,Ifthegrammaris

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

Theprecedencerelationsbetweentheoperatorsare
(id)>(^)>(*/)>(+-)>$,,,^‟operatorisRightAssociativeandreamingalloperatorsare LeftAssociative

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

(	<•	<•	<•	<•	<•	<•	<•	=	**Err**
)	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	**Err**	**Err**

Theintentionoftheprecedencerelationsistodelimitthehandleofthegiveninput Stringwith<•marking theleftendofthe Handle and •>marking the rightendofthehandle.

ParsingAction:

Tolocatethehandlefollowingstepsarefollowed:

1. Add $symbolat thebothendsofthegiven inputstring.
2. Scantheinputstringfromlefttorightuntiltherightmost•>isencountered.
3. Scantowardsleftoveralltheequalprecedence‘suntilthefirst<•precedenceisencountered.
4. Everything between<•and •>isahandle.
5. $onSmeansparsingissuccess.

Example, Explain the parsing Actions of the OPParser for the input string is **“id*id”** and thegrammaris:

E → E+E

E → E*E

E → id

1.$<•id•>*<•id•>$

Thefirsthandleis_id‘andmatchforthe_id_inthegrammarisE → id .So, id is replaced with the Non terminal E. the given input string can bewrittenas

2.$<•E•>*<•id•>$

Theparserwillnotconsiderthe Nonterminalasan input.So,theyare notconsideredintheinputstring.So,thestringbecomes

3.$<•*<•id•>$

Thenext handleis_id‘andmatchforthe_id_inthegrammarisE → id. So, id is replaced with the Non terminal E. the given input string can bewrittenas

4.**$<•*<•E•>$**

Theparserwillnotconsiderthe Nonterminalasan input.So,theyare notconsideredintheinputstring.So,the stringbecomes

5.**$<• *•>$**

Thenexthandleis_*‘andmatchforthe__inthegrammarisE E * E .So, id is replaced with the Non terminal E. the given input string can bewrittenas

6.**$E $**

Theparserwillno tconsiderthe	Nonterminalasan input.So,theyare notconsideredintheinputstring.So,the stringbecomes

7.**$$**

Onmeansparsingsuccessful.

OperatorParsingAlgorithm:

TheoperatorprecedenceParser parsing programdeterminestheactionoftheparser dependingon

1. _a‘istopmostsymbolonthe Stack
2. _b‘isthecurrentinput symbol

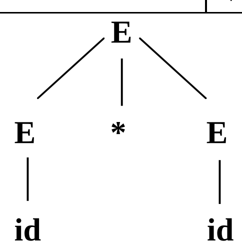
Thereare3 conditionsfor _a‘and_b‘thatareimportantfortheparsingprogram

1. a=b=$,theparsingissuccessful
2. a <• b or a = b, the parser shifts the input symbol on to the stack and advances theinputpointertothe nextinputsymbol.
3. a •>b, parser performs the reduce action. The parser pops out elements one byone from the stack until we find the current top of the stack element has lowerprecedence thanthemostrecentlypoppedoutterminal.

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‘usingE->id
$E*E	$	*•>$,reduce_*‘using E->E*E
$E	$	$=$=$,soparsingissuccessful



AdvantagesandDisadvantagesofOperatorPrecedenceParsing:

Thefollowingaretheadvantagesofoperator precedenceparsing

1. Itissimpleandeasytoimplementparsingtechnique.
2. Theoperatorprecedenceparsercanbeconstructedbyhandafterunderstandingthegram mar.Itis simple todebug.

Thefollowingarethedisadvantagesofoperatorprecedenceparsing:

1. Itisdifficulttohandletheoperatorlike_-_whichcanbeeither unaryorbinaryandhencedifferentprecedence‘sandassociativities.
2. It canparse onlya smallclassofgrammar.

3. Newadditionordeletionoftherulesrequirestheparser to berewritten.
4. Toomanyerrorentriesintheparsingtables.

LRParsing:

Most prevalent type of bottom up parsing is LR (k) parsing. Where, L is left to right scan of thegiven inputstring, R is RightMostderivation in reverse and K is no of inputsymbols as theLookahead.

∑Itisthemostgeneralnonbacktrackingshiftreduceparsingmethod

∑Theclassofgrammarsthatcanbeparsedusing theLRmethodsisaproper supersetofthe classofgrammarsthatcanbe parsedwithpredictiveparsers.

∑AnLRparser candetect asyntacticerror assoonasitispossibletodoso,onalefttorightscanoftheinput.

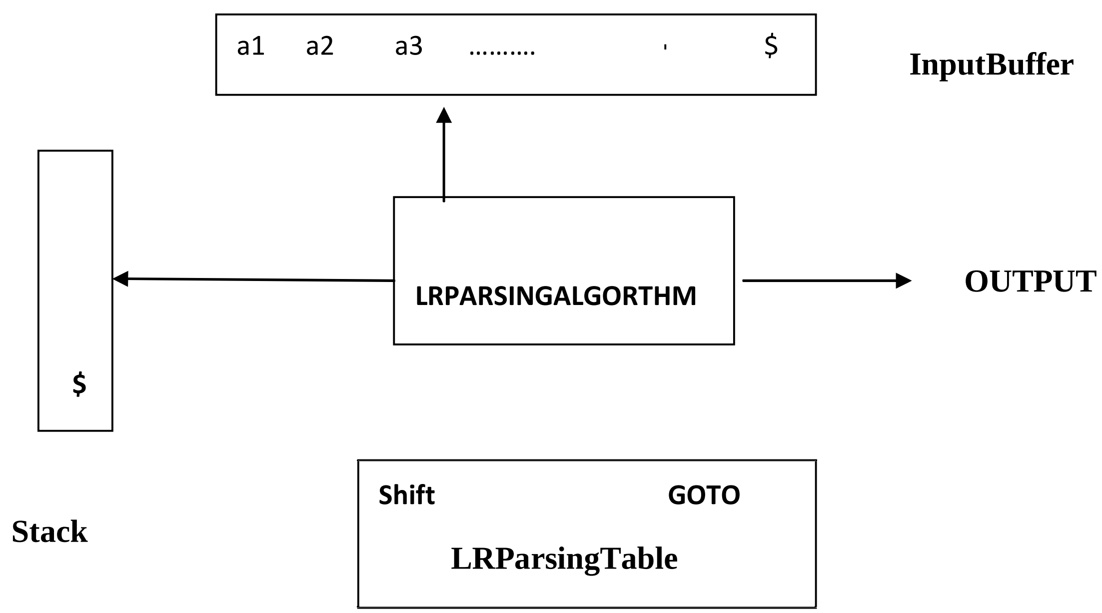


Figure3.3:ComponentsofLRParsing

LRParserConsistsof

∑Aninputbufferthatcontainsthestringtobeparsedfollowedbya$Symbol,usedtoindicate endofinput.

∑A stack containing a sequence of grammar symbols with a $ at the bottom of the stack,whichinitiallycontains theInitialstateof theparsingtableontopof$.

∑Aparsingtable(M),itisatwodimensionalarrayM[state,terminalorNonterminal]anditcontains twoparts

1. **ACTIONPart**
   The ACTION part of the table is a two dimensional array indexed by state and theinput symbol, i.e. **ACTION**[state][input], An action table entry can have one offollowingfourkinds ofvaluesinit.Theyare:
   1. ShiftX,whereXisaStatenumber.
   2. ReduceX, whereXisaProductionnumber.
   3. Accept,signifyingthecompletionofasuccessfulparse.
   4. Errorentry.
2. **GOTOPart**
   The GO TO part of the table is a two dimensional array indexed by state and aNonterminal,i.e.GOTO[state][NonTerminal].AGOTOentryhasastatenumberinthe table.

∑ A parsing Algorithm uses the current State X, the next input symbol _a' to consult theentryataction[X][a].itmakesoneofthefourfollowing actionsasgivenbelow:

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

Theparsingisclassifiedinto

1. LR(0)
2. Simple LR(1)
3. CanonicalLR(1)
4. LookaheadLR(1)

LR(1)Parsing: Varioussteps involvedintheLR(1)Parsing:

1. WritetheContextfreeGrammarforthegiveninputstring
2. CheckfortheAmbiguity
3. AddAugment production
4. CreateCanonicalcollectionofLR(0)items
5. DrawDFA
6. Constructthe LR(0)Parsingtable
7. Based ontheinformation fromthe Table,with helpofStack andParsingalgorithmgenerate theoutput.

AugmentGrammar

The Augment Grammar G`, is G with a new starting symbol S` an additional productionS` → S. this helps the parser to identify when tostop the parsing and announce the acceptance ofthe input. The input string is accepted if and only if the parser is about to reduce by S` → S. Forexampleletus considerthe Grammarbelow:

E → E+T|T
T → T*F
F → (E)| id **theAugmentgrammarG`isRepresentedby**

E`→ E
E → E+T|T
T → T*F
F → (E)| id

NOTE: Augment Grammar is simply adding one extra production by preserving the actualmeaningofthegivenGrammarG.

CanonicalcollectionofLR (0)items

LR(0) items

An LR (0) item of a Grammar is a production G with dot at some position on the rightsideoftheproduction.Anitemindicateshowmuchoftheinputhasbeenscanneduptoagivenpointintheprocessofparsing.Forexample,iftheProductionisX → YZ then, The LR (0)items are:

1. X → •AB,indicatesthattheparserexpectsastringderivablefromAB.
2. X → A•B,indicatesthattheparserhasscannedthestringderivablefromtheAandexpectingthestringfromY.
3. X → AB•,indicates thatheparserhas scannedthestringderivablefromAB.IfthegrammarisX → €the,theLR(0)itemis X→ •,indicatingthattheproductionisreducedone.

CanonicalcollectionofLR(0)Items:

Thisistheprocessofgrouping theLR(0) itemstogether based ontheclosureand Gotooperations

Closureoperation

IfIisaninitialState,thentheClosure(I)isconstructedasfollows:

1. Initially,addAugmentProductiontothestateandcheckfor the• symbolintheRighthand side production, if the • is followed by a Non terminal then Add ProductionswhichareStatingwiththatNonTerminalinthe StateI.

2. If a production X→α•Aβ is in I, then add Production which are starting with X in theState I. Rule 2 is applied until no more productions added to the State I(meaning thatthe •isfollowedbya Terminalsymbol).

Example:

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

Closure(I_0)State

Add**E`** → **•E**inI_0State

Since, the‗•'symbolintheRighthandsideproductionisfollowedbyANonterminal E. So, add productions starting with E in to Io state. So, the statebecomes

E` → •E

0. E → •E+T

1. T → •F

The1st and2nd productionsaresatisfiesthe2nd rule.So, addproductionswhicharestartingwithEandTinI_0

Note:onceproductionsareaddedinthestatethesameproductionshouldnot addedforthe 2nd timeinthe same state.So,thestate becomes

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

GOTOOperation

Go to (I_0, X), where I_0 is set of items and X is the grammar Symbol on whichwearemovingthe„•"
symbol.ItislikefindingthenextstateoftheNFAforagiveState→I_0andtheinputsymbolis X.Forexample,iftheproductionis**E•E+T**

Goto **(I_0,E)**is**E` → •E, E → E•+T**

Note: OncewecompletetheGotooperation,weneedtocomputeclosureoperationfortheoutputprod uction

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

E`->.E

E->.E+T

E.+TT->.T*F

E

E`->E.

E->

ConstructionofLR(0)parsingTable:

OncewehaveCreatedthecanonicalcollectionofLR(0) items,needtofollowthestepsmentionedbelow:

Ifthereisa transactionfromone state (**I_i**)to anotherstate(**I_j**)onaterminalvalue then,weshouldwritethe shiftentryinthe actionpartasshownbelow:

a

A->α•aβ A->αa•β

I_i **I_j**

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

Ifthereisa transactionfromone state (**I_i**)toanoth erstate(**I_j**)onaNonterminalval ue then, we should write the subscript value of **I_i**in the GO TO part as shown below: part as shownbelow:

A

A->α•Aβ A->αA•β

I_i **I_j**

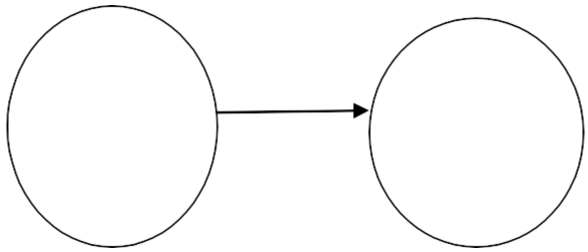
States	ACTION		GOTO
	a	$	A
I_i			**j**
I_j			

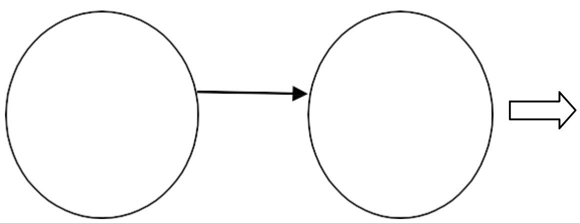
If there is one state (**I_i**), where there is one production which has no transitions. Then, theproduction is said tobe a reduced production. These productions should have reduced entry inthe Action part along with their production numbers. If the Augment production is reducing then,write acceptintheActionpart.

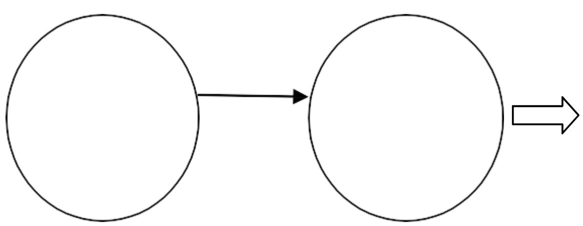
1 A->αβ•

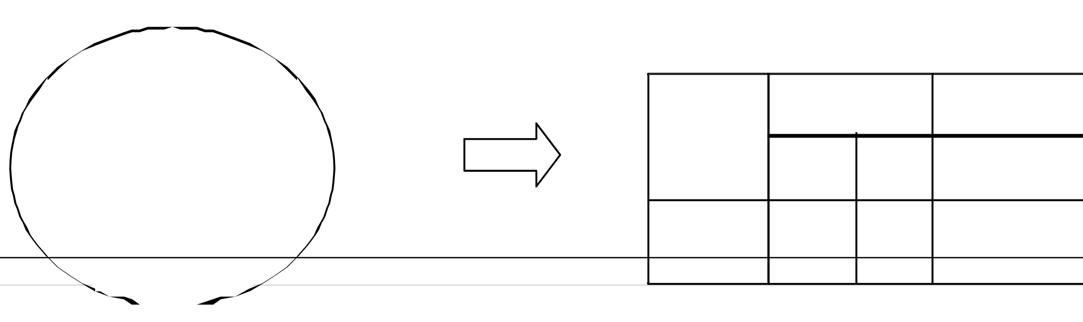
States	ACTION		GOTO
	a	$	A

l_i r_1 r_1









I_i

I_i

ForExample,Construct theLR(0)parsing Tableforthegiven Grammar(G)

S → aB

B → bB|b

Sol:1. AddAugmentProductionandinsert„•‟symbolatthefirstpositionforeveryproductioninG

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

<u>I_0State:</u>

1. AddAugmentproductiontotheI_0StateandComputetheClosure

I_0 =Closure(S′ → •S)

Since ‗•' is followed by the Non terminal, add all productions starting with S in to I_0 State.
So,the I_0Statebecomes

I_0= **S′ → •S**

S → •aB Here,intheSproduction‗.'Symbolisfollowed byaterminalvalueso closethe state.

I_1=Goto(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‗•'symbolisfollowed byTheNonterminalB.So,addthe productionswhichareStartingB.

I_2= **B → •bB**

B → •b Here,the‗•'symbolintheBproductionisfollowed bytheterminalvalue.So,Close theState.

I_2= **S → a•B**

B → •bB

B → •b

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

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

Addproductionsstarting withBinI_4.

B → •bB

B → •b TheDotSymbolisfollowedbytheterminalvalue.So,closetheState.

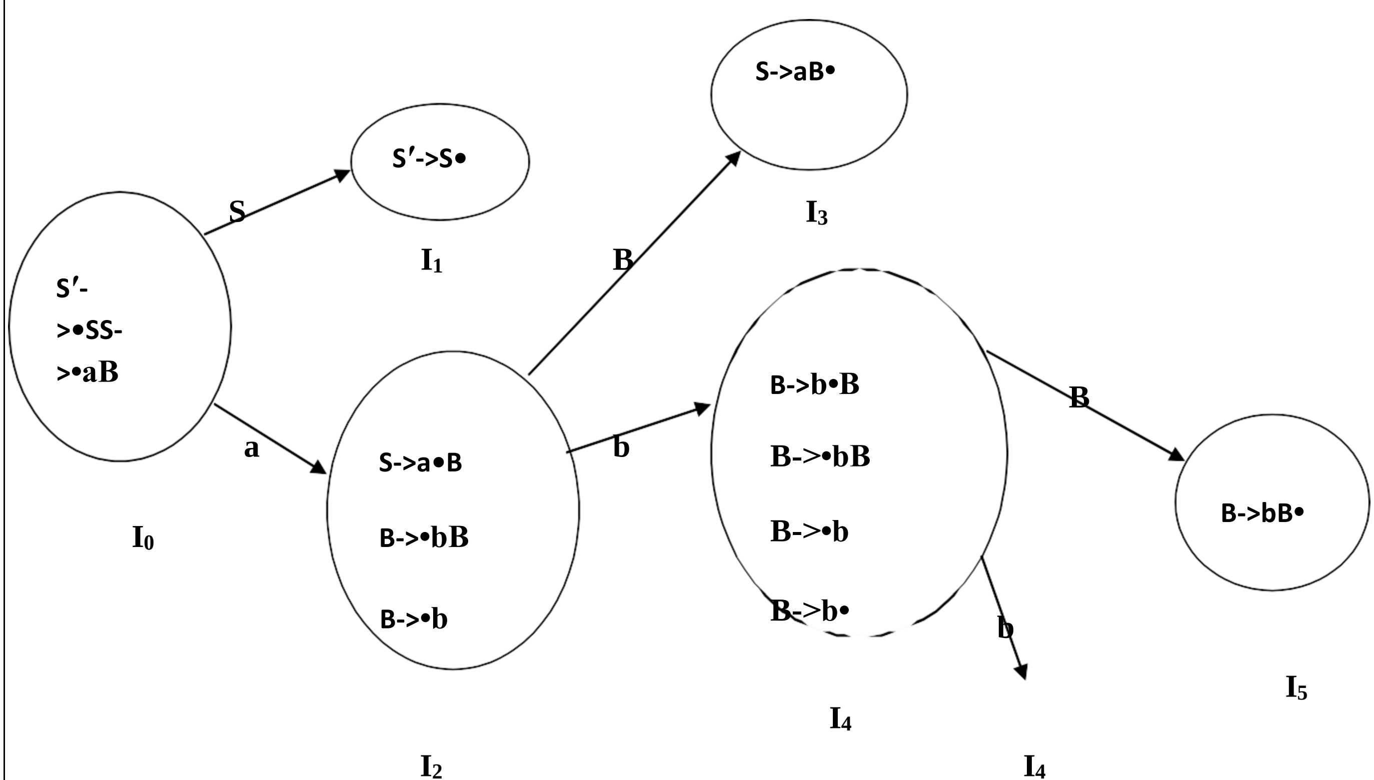
I_4= **B → b•B**
B → •bB
B → •b
B → b•

I_5= Goto(I_2,b)=Closure(B → b•)=B → b•

I_6=Goto(I_4,B)=Closure(B → bB•)=B → bB•

I_7=Goto(I_4,b)=I_4

<u>DrawingFiniteStatediagramDFA:</u>FollowingDFAgivesthestatetransitionsoftheparserandis usefulinconstructingtheLRparsingtable.



LRParsingTable:

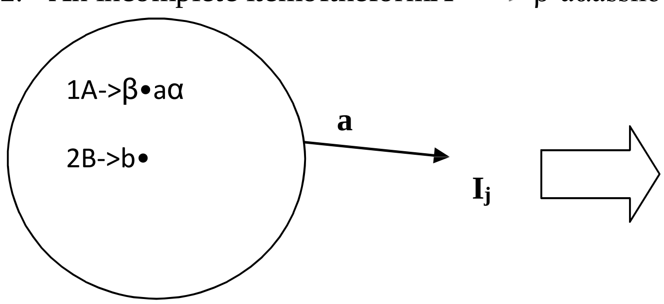
States	ACTION			GOTO	
	a	B	$	S	B
I_0	S_2			1	
I_1			ACC		
I_2		S_4			3
I_3	R_1	R_1	R_1		
I_4	R_3	S_4/R_3	R_3		5
I_5	R_2	R_2	R_2		

Note: if there are multiple entries in the LR (1) parsing table, then it will not accepted by theLR(1) parser. In the above table I_3 row is giving two entries for the single terminal value _b' anditis calledasShift-Reduceconflict.

Shift-ReduceConflictinLR(0)Parsing:

ShiftReduceConflictintheLR(0)parsingoccurswhenastatehas

1. AReduceditemoftheformA → α•and
2. An incomplete itemoftheformA → β•aαasshownbelow:



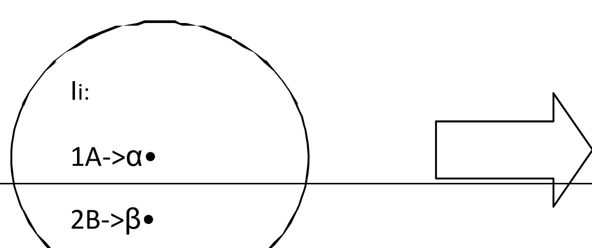
I_i

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

Reduce-ReduceConflictin LR(0)Parsing:

Reduce- Reduce Conflict in the LR (1) parsing occurs when a state has two or morereduceditems oftheform

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

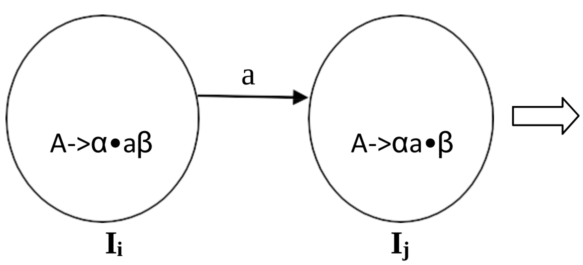
VariousstepsinvolvedintheSLR(1) Parsingare:

1. WritetheContextfreeGrammarforthegiveninputstring
2. ChecfortheAmbiguity
3. AddAugment production
4. CreateCanonicalcollectionofLR(0)items
5. DrawDFA
6. ConstructtheSLR(1)Parsingtable
7. Basedontheinformation fromtheTable,withhelpofStack andParsingalgorithmgenerate theoutput.

SLR(1)ParsingTableConstruction

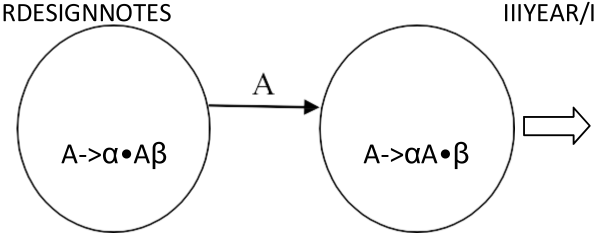
OncewehaveCreatedthecanonicalcollectionofLR(0) items,needtofollowthestepsmentionedbelow:

Ifthereisa transactionfromone state (**I_i**)to anotherstate(**I_j**)onaterminalvalue then,weshouldwritethe shiftentryinthe actionpartasshownbelow:



States	ACTION		GOTO
	a	$	A
I_i	S_j		
I_j			

If there is a transaction from one state (**I_i**) to another state (**I_j**) on a Non terminal valuethen, we should write the subscript value of **I_i**in the GO TO part as shown below: part as shownbelow:



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

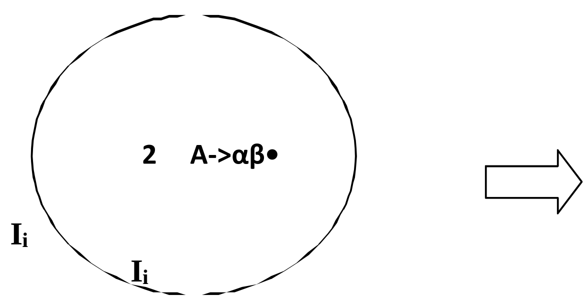
I_i **I_j**

1 Ifthere is one state (**I_i**),where there is one production(**A->αβ•**)which has no transitionstothenextState.Then,theproductionissaidtobea reducedproduction.ForallterminalsXinFOLLOW(A),writethereduceentryalongwiththeirproductionnumbers.IftheAugmentproductionisreducingthenwriteaccept.

1 *S* ->•aAb

2 A->αβ•

Follow(S)={$}
Follow(A)=(b}



States	ACTION			GOTO	
	a	b	$	S	A
I_i		r_2			

SLR(1)tablefortheGrammar

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		

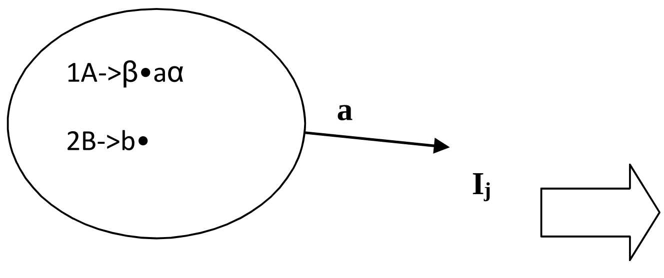
Note:WhenMultipleEntriesoccursintheSLRtable.
Then,thegrammarisnotacceptedbySLR(1)Parser.

Conflictsin theSLR(1)Parsing:

Whenmultipleentriesoccur inthetable.Then,thesituationissaidtobeaConflict.

Shift-ReduceConflict inSLR(1)Parsing: ShiftReduceConflictintheLR(1) parsingoccurswhenastatehas

1. AReduceditemoftheformA → α•and Follow(A)includestheterminalvalue ‗a‘.
2. An incomplete itemoftheformA → β•aαasshownbelow:



I_i

States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce-ReduceConflictinSLR(1)Parsing

Reduce- ReduceConflictintheLR(1) parsingoccurswhenastatehastwoor morereduceditems oftheform

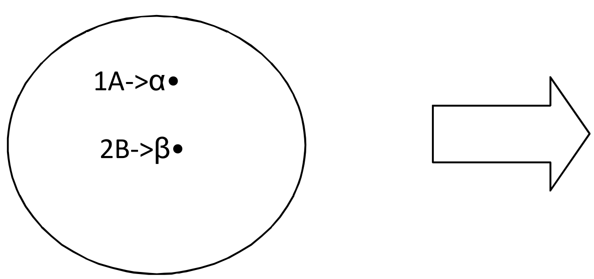
1. **A→ α•**
2. **B → β•and Follow(A) ∩ Follow(B) ≠null** asshownbelow:

IfTheGrammaris

S-
>αAaBaA-
>α
B->β

Follow(S)={$}

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



I_i

States	Action		GOTO	
	a	$	A	B
I_i	r_1/r2			

CanonicalLR(1) Parsing: VariousstepsinvolvedintheCLR(1) Parsing:

1. WritetheContextfreeGrammarforthegiveninputstring
2. ChecktheAmbiguity
3. AddAugment production

4. CreateCanonicalcollectionofLR(1)items
5. DrawDFA
6. ConstructtheCLR(1)Parsingtable
7. BasedontheinformationfromtheTable,withhelpofStack andParsingalgorithmgeneratetheoutput.

LR (1)items:

TheLR(1) itemisdefinedby**production**,**positionofdata**anda**terminalsymbol**.Theterminalis calledas***Lookaheadsymbol***.

General formofLR(1)itemis

S->α•Aβ,$
A->•γ, FIRST(β,$)

Rulestocreatecanonicalcollection:

1. EveryelementofIisadded toclosureofI
2. If an LR (1) item **[X-> A•BC, a]** exists in I, and there exists a production B->b_1b_2.....,thenadditem**[B->•b_1b_2,z]**wherezisaterminalin**FIRST(Ca),**ifitisnotalreadyin Closure(I).keep applyingthisrule untilthere arenomore elementsadde.

Forexample,ifthegrammaris

S-
>CCC
-
>cCC-
>d

TheCanonicalcollectionofLR(1)itemscanbecreatedasfollows:

0. *S′*->•S(AugmentProduction)

1. S->•CC

2. C-
>•cC3.C-
>•d

I_0 State : Add Augment production and compute the Closure, the look ahead symbol for the AugmentProductionis$.

S′->•S,$=Closure(S′->•S,$)

ThedotsymbolisfollowedbyaNonterminal S.So,addproductionsstarting withSinI_0 State.

**S->•CC,FIRST($),** using2ndrule

S->•CC,$

ThedotsymbolisfollowedbyaNonterminal C.So,addproductionsstarting with CinI_0 State.

C->•cC, FIRST(C,
$)C->•d,FIRST(C,
$)

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

C->•cC,
c/dC-
>•d,c/d

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

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

I_1=Goto(I_0,S)=S′->S•,$

I_2=Go to (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, the I_3Stateis

C->c•C,
c/dC->•cC,
c/dC->•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 ,
$C->•d,$

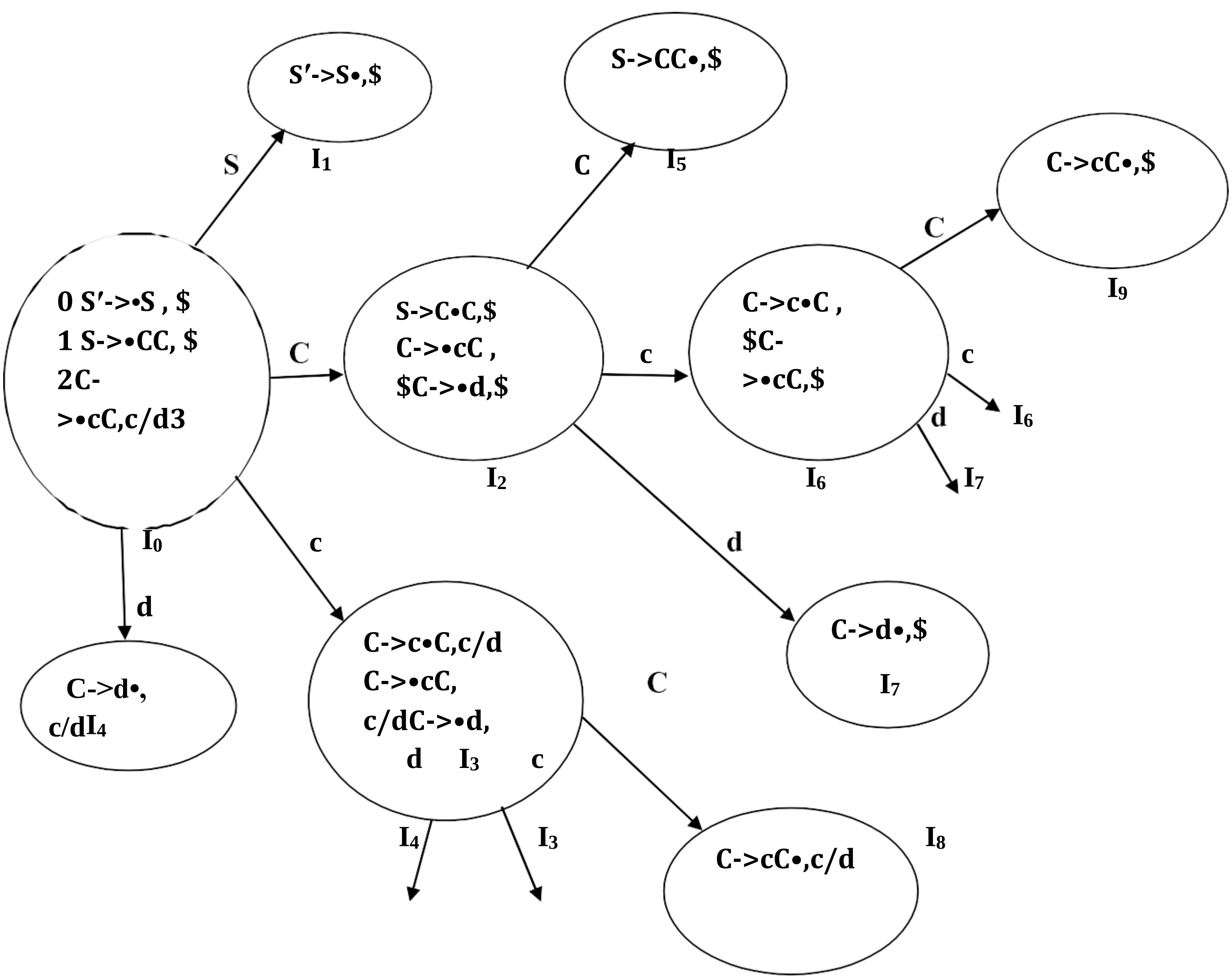
I_7 = Go to (I_2 , d)= Closure(C->d•,$) = C->d•,
$Goto($I_3$,c)= closure(C->•cC, c/d)= I_3.

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



ConstructionofCLR(1)Table

Rule1: if there is an item [A->α•Xβ,b] in $I_i$and goto(I_i,X) is in I_j then action [I_i][X]= Shiftj,WhereXisTerminal.
Rule2: if there is an item [A->α•, b] in $I_i$and (A≠S`) set action [I_i][b]= reduce along withthe productionnumber.
Rule3:ifthereisan item[S`->S•,$]in$I_i$thensetaction[$I_i$][$]= Accept.
Rule4:ifthereisanitem[A->α•Xβ,b]in$I_i$and goto(I_i,X) isin$I_j$thengoto[I_i][X]=j,Where Xis NonTerminal.

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 ofStates 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 italso as efficient as CLR parser. Here LR (1) items that have same productions but different look-aheadsarecombinedtoformasinglesetofitems.

For example, consider the grammar in the previous example. Consider the states I_4 and I_7asgivenbelow:

I_4= Goto(I_0, d)= Colsure(C->d•, c/d) = C->d•, c/dI_7= Goto(I_2,d)= Closure(C->d•,$)=C->d•,$

Thesestatesaredifferingonlyinthelook-aheads. Theyhavethesameproductions. HencethesestatesarecombinedtoformasinglestatecalledasI_{47}.

SimilarlythestatesI$_3$and I_6differing onlyintheir look-aheadsasgivenbelow:

I_3=Goto(I_0,c)=

C->c•C,
c/dC->•cC,
c/dC->•d,
c/d

I_6=Goto($I_{2,}$c)=

C->c•C ,
\$C->•cC ,
\$C->•d,\$

Thesestatesaredifferingonlyinthelook-aheads. Theyhavethesameproductions. Hencethesestatesarecombinedtoformasingle statecalledasI_{36}.

Similarly the States $I_8$and I_9 differing only in look-aheads. Hence they combined to formthestateI_{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		

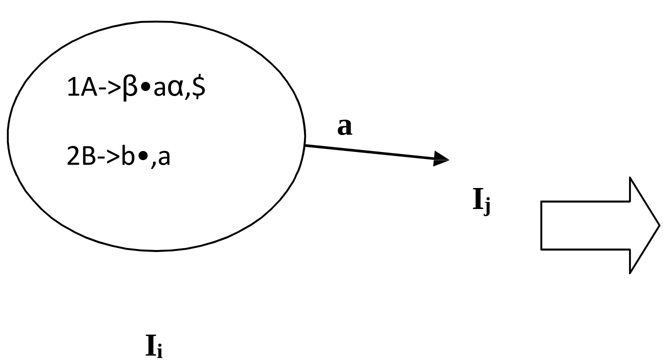
Table:LALRTable

Conflictsin theCLR(1)Parsing: Whenmultipleentriesoccurinthetable.Then,thesituationis saidtobeaConflict.

Shift-ReduceConflictinCLR(1)Parsing

ShiftReduceConflictintheCLR(1)parsingoccurswhenastatehas

3. **3.** AReduceditemoftheformA → α•,aand
4. **4.** An incomplete itemoftheformA → β•aαasshownbelow:



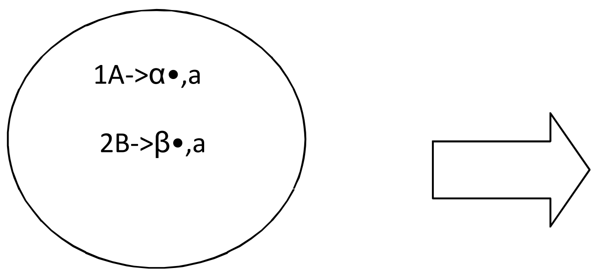
Ii

States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce/ReduceConflictinCLR(1)Parsing

Reduce-ReduceConflictintheCLR(1)parsingoccurswhenastatehastwoormorereduceditems oftheform

3. **A→ α•**
4. **B→ β•Iftwoproductionsinastate(I) reducing onsamelookahead symbolasshownbelow:**



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

3C->•d

STACK	**INPUT**	**ACTION**
$0	cdd$	ShiftS_3
$0c3	dd$	ShiftS_4
$0c3d4	d$	ReducewithR3,C->d,pop2*βsymbolsfromthestack
$0c3C	d$	Goto(I_3,C)=8ShiftS_6

$0c3C8	d$	ReducewithR_2,C->cC,pop2*βsymbolsfromthestack
$0C	d$	Goto(I_0,C)=2
$0C2	d$	ShiftS_7
$0C2d7	$	ReducewithR3,C->d,pop2*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 canhavemorethanoneparsetreefor astring.Forexample,considergrammar.

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

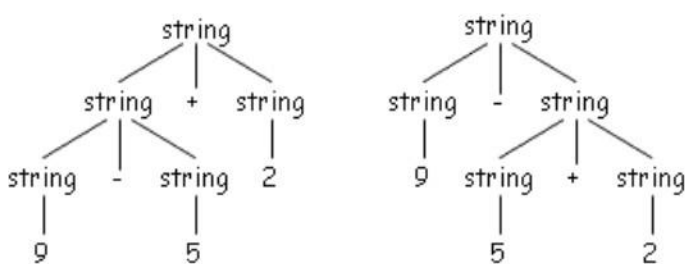
String9-5+2 hastwoparsetrees

A grammar is said to be an ambiguous grammar if there is some string that it can generate inmore than one way (i.e., the string has more than one parse tree or more than one leftmostderivation).A languageisinherently ambiguousif itcan only be generated by ambiguousgrammars.

Forexample,considerthefollowinggrammar:

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

Inthisgrammar,thestring9-5+2hastwopossibleparsetreesasshowninthenextslide.



Consider the parse trees for string 9-5+2, expression like this has more than one parse tree. Thetwo trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and

9-(5+2).The secondparenthesizationgivesthe expressionthe value 2insteadof6.

∑Ambiguityisproblematicbecausemeaning oftheprogramscanbeincorrect

∑ Ambiguitycanbehandledinseveralways

- Enforceassociativityandprecedence

- Rewritethegrammar(cleanestway)

Therearenogeneraltechniques forhandlingambiguity, but

. Itisimpossibletoconvertautomaticallyanambiguousgrammar toanunambiguousone

Ambiguity is harmful to the intent of the program. The input might be deciphered in a way whichwas not really the intention of the programmer, as shown above in the 9-5+2 example. Thoughthere is no general technique to handle ambiguity i.e., it is not possible to develop some featurewhich automatically identifies and removes ambiguity from any grammar. However, it can beremoved,broadlyspeaking,inthefollowingpossibleways:-

1) Rewritingthewholegrammarunambiguously.

2) Implementingprecedenceandassociativelyrules inthegrammar.
Weshalldiscussthistechniqueinthelater slides.

Ifanoperandhas operatoronboththesides,thesideonwhichoperatortakesthis operandistheassociativityofthatoperator

.Ina+b+cb istakenby left+
. +,-, *, /areleftassociative
.^,=arerightassociative

Grammartogeneratestringswithrightassociativeoperatorsrightàletter=right|letterlettera|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 conventionallyevaluatedfromlefttorighti.e.,operandistakenbytheoperatorontheleftside.
Forexample,
6*5*4=(6*5)*4 and not6*(5*4)
6/5/4=(6/5)/4andnot6/(5/4)

Aright-associativeoperationisanon-associativeoperationthatisconventionallyevaluatedfromrighttolefti.e.,operandis takenbytheoperatoronthe rightside.

Forexample,

6^5^4 => 6^(5^4) and not
(6^5)^4)x=y=z=5=> x=(y=(z=5))

Following is the grammar to generate strings with left associative operators. (Note that this is leftrecursive and may go into infinite loop. But we will handle this problem later on by making itrightrecursive)

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

IMPORTANTQUESTIONS

1. DiscussthetheworkingofBottomupparsingandspecificallytheOperatorPrecedenceParsin gwithanexaple?
2. WhatdoyoumeanbyanLRparser?ExplaintheLR(1)Parsingtechnique?
3. Writethedifferencesbetweencanonicalcollectionof LR(0)itemsand LR(1)items?
4. WritetheDifferencebetweenCLR(1) andLALR(1)parsing?
5. WhatisYACC?Explainhowdoyouuseitinconstructingtheparserusingit.

ASSIGNMENTQUESTIONS

1. ExplaintheconflictsintheShiftreduceParsingwithanexample?
2. E → E+T|T

   T → T*F

   F → (E)|id,constructtheLR(1) Parsingtable?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 sourceprogramintoanintermediateformprogramminglanguageconstructssuchasdeclarations,assignmentsandflow-of-controlstatements.

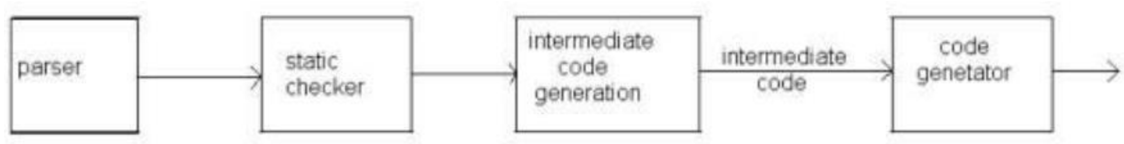


Figure4.1:IntermediateCodeGenerator

Intermediatecodeis:

- TheoutputoftheParser and theinputto theCodeGenerator.
- Relativelymachine-independentandallowsthecompilertoberetargeted.
- Relativelyeasytomanipulate(optimize).

WhataretheAdvantagesofanintermediatelanguage?

AdvantagesofUsinganIntermediateLanguageincludes:

1. Retargetingisfacilitated-Buildacompilerfor anewmachinebyattachinganewcodegeneratortoanexistingfront-end.

2. Optimization-reuseintermediatecodeoptimizersincompilersfordifferentlanguagesanddifferentmachines.

Note:theterms―intermediatecode‖,–intermediatelanguage‖,and–intermediate representation‖areallusedinterchangeably.

Types of Intermediate representations / forms: There are three types of intermediaterepresentation:-

1. SyntaxTrees
2. Postfixnotation
3. ThreeAddressCode

Semanticrulesforgeneratingthree-addresscodefromcommonprogramminglanguageconstructsaresimilar tothosefor constructingsyntaxtreesoffor generatingpostfixnotation.

GraphicalRepresentations

A syntax tree depicts the natural hierarchical structure of a source program. A DAG(Directed Acyclic Graph)gives the sameinformation butin amore compactway becausecommon sub-expressions are identified. A syntax tree for the assignment statement a:=b*-c+b*-cappearinthefollowingfigure.

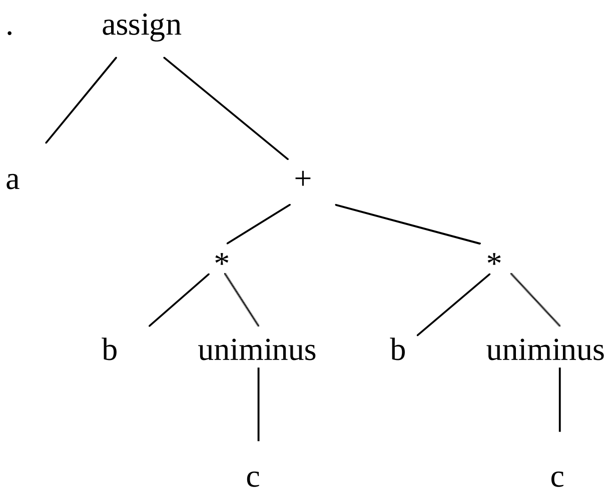


Figure4.2 : AbstractSyntaxTreeforthestatementa:=b*-c+b*-c

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the inwhich a node appears immediately after its children. The postfix notation for the syntax tree inthe figis

abc**uminus**+ bc**uminus***+**assign**

The edges in a syntax tree do not appear explicitly in postfix notation. They can berecovered in the order in which the nodes appear and the no. of operands that the operator at anode expects. The recovery of edges is similar to the evaluation, using a staff, of an expression inpostfixnotation.

WhatisThreeAddressCode?

Three-addresscodeis asequenceofstatementsofthegeneralform:X:=Y OpZ

where x, y, and z are names, constants, or compiler-generated temporaries; op stands forany operator, such as a fixed- or floating-point arithmetic operator, or a logical operator onBoolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is onlyone operator on the rightside of a statement. Thus asource language expression likex+y*zmightbetranslatedintoa sequence

t1 := y *
zt2:=x+ t1

Wheret1andt2arecompiler-generatedtemporarynames.Thisunravelingofcomplicated arithmetic expressions and of nested flow-of-control statements makes three-addresscode desirable for target code generation and optimization. The use of names for the intermediatevalues computed by a program allow- three-address code to be easily rearranged – unlike postfixnotation. Three - address code is a linearzed representation of a syntax tree or a dag in whichexplicitnames correspondtotheinteriornodesofthegraph.

Intermediate code using Syntax for the above arithmetic
expressiont1:=-c
t2:=b*t1
t3:=-c
t4 := b *
t3t5:=t2+t4
a:=t5

The reason for the term‖three-address code‖ is that each statement usually contains threeaddresses, two for the operands and one for the result. In the implementations of three-addresscode given later in this section, a programmer-defined name is replaced by a pointer tc a symbol-table entryforthatname.

TypesofThree-AddressStatements

Three-address statements are akin to assembly code. Statements can have symbolic labelsand 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 forthelabelseitherbymaking aseparate pass,orbyusing ‖back patching,‖ discussedinSection
8.6.Herearethecommonthree-addressstatementsusedintheremainderofthisbook:

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

2. **Assignment instructions** of the form x:= op y, where op is a unary operation. Essential unaryoperations include unary minus, logical negation, shift operators, and conversion operators that,forexample,converta fixed-pointnumbertoafloating-pointnumber.

3. **Copy statements**oftheformx: =ywherethe valueofyisassignedtox.

4. **Theunconditionaljump**gotoL.Thethree-addressstatementwithlabelListhenexttobeexecuted.

5. **Conditional jumps** such as if x relop y goto L. This instruction applies a relational operator(<, =, >=, etc.) to x andy, and executes the statement with label L nextif x stands in relationrelop to y. If not, the three-address statement following if x relop y goto L is executed next, as inthe usualsequence.

6. **param x and call p, n** for procedure calls and return y, where y representing a returned valueisoptional.Theirtypicaluseis asthe sequenceofthree-addressstatements

param
x1param
x2param
xncallp,n
Generatedaspartofacalloftheprocedurep(x,,x~,...,x‖).Theintegernindicatingthenumberofactualparametersin‖callp,n‖isnotredundantbecausecallscanbenested.Theimplementationofprocedurecallsisoutlinedin Section8.7.

7. **Indexed assignments** of the form x: = y[i] and x [i]: = y. The first of these sets x to thevalue in the location i memory units beyond location y. The statement x[i]:=y sets the contents ofthe location i units beyond x to the value of y. In both these instructions, x, y, and i refer to dataobjects.

8. **Address and pointer assignments** of the form x:=&y, x:= *y and *x: = y. The first of thesesets the value of x to be the location of y. Presumably y is a name, perhaps a temporary, thatdenotes an expression with an I-value such as A[i, j], and x is a pointer name or temporary. Thatis,ther-valueofxisthel-value(location)ofsomeobject!.Inthestatementx:=~y,presumablyy is a pointer or a temporary whose r- value is a location. The r-value of x is made equal to thecontents of that location. Finally, +x: = y sets the r-value of the object pointed to by x to the r-value ofy.

The choice of allowable operators is an important issue in the design of an intermediateform. The operator set must clearly be rich enough to implement the operations in the sourcelanguage. A small operator set is easier to implement on a new target machine. However, arestricted instruction set may force the front end to generate long sequences of statements forsome source, language operations. The optimizer and code generator may then have to workharderifgoodcodeistobegenerated.

SYNTAXDIRECTEDTRANSLATIONOFTHREEADDRESSCODE

When three-address code is generated, temporary names are made up for the interiornodes ofasyntaxtree.Thevalueofnon-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 codetoevaluateEintosometemporaryt,followedbytheassignmentid.place:=t.Ifanexpressionisa single identifier, say y, then y itself holds the value of the expression. For the moment, wecreate a new name every time a temporary is needed; techniquesforreusing temporaries aregiven in Section S.3. The S-attributed definition in Fig. 8.6 generates three-address code forassignment statements. Given input a: = b+ – c + b+ – c, it produces the code in Fig. 8.5(a). Thesynthesized attribute S.code represents the three- address code for the assignment S. The non-terminalEhas twoattributes:

1. E.place, thenamethatwill holdthevalueofE,and

2. E.code,the sequence ofthree-addressstatementsevaluating E.

Thefunction newtempreturns a sequence of distinctnames t1, t2,... in response tosuccessive calls. For convenience, we use the notation gen(x ‘: =‘ y ‘+‘ z) in Fig. 8.6 to representthe three-address statement x: = y + z. Expressions appearing instead of variables like x, y, and zare 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 thecodeattributes.Flow-of-controlstatementscanbeadded tothelanguageofassignmentsinFig. 8.6 by productions and semantic rules)like the ones for while statements in Fig. 8.7.In thefigure, the code for S - while E do S, is generated using‘ new attributes S.begin and S.after tomarkthefirststatementinthecodeforEandthestatementfollowingthecodeforS,respectively.

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

These attributes represent labels created by a function new label that returns a new labeleverytimeitis called.

IMPLEMENTATIONSOFTHREE-ADDRESSSTATEMENTS:

A three-address statement is an abstract form of intermediate code. In a compiler, thesestatements can be implemented as records with fields for the operator and the operands. Threesuchrepresentationsare quadruples,triples,andindirecttriples.

QUADRUPLES:

A quadrupleis a record structure with four fields, which we call op, arg l, arg 2, andresult. The op field contains an internal code for the operator. The three-address statement x:= yop z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unaryoperatorslikex:=–yorx:=ydonotusearg2.Operatorslikeparamuseneitherarg2norresult. Conditional and unconditional jumps put the target label in result. The quadruples in Fig.H.S(a)are forthe assignmenta:= b+ –c + bi–c.Theyare obtainedfromthethree-addresscode .The contents of fields arg 1, arg 2, and resultare normally pointers tothe symbol-table entriesfor the names represented by thesefields. If so, temporary names mustbe entered into thesymboltable as theyare created.

TRIPLES:

To avoid entering temporary names into the symbol table. We might refer to a temporaryvalue bi the position of the statement that computes it. If we do so, three-address statements canbe represented by records with only three fields: op, arg 1 and arg2, as Shown below. The fieldsarg l and arg2, for the arguments of op, are either pointers to the symbol table (for programmer-defined names or constants) or pointers into the triple structure (for temporary values). Sincethree fields are used, this intermediate code format is known as triples.‘ Except for the treatmentof programmer-defined names, triples correspond to the representation of a syntax tree or dag byanarrayofnodes,asin

	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-tablepointers are represented by the names themselves. In practice, the information needed to interpretthe different kinds of entries in the arg 1 and arg2 fields can be encoded into the op field or someadditionalfields.ThetriplesinFig.8.8(b) correspondtothequadruplesinFig.8.8(a).Notethat the copy statementa:= t5 is encoded in the triple representation by placing a in the arg 1 fieldand using the operator assign. A ternary operation like x[i]: = y requires two entries in the triplestructure, as shown in Fig. 8.9(a), while x: = y[i]is naturally represented as two operations inFig.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 listingpointers to triples, rather than listing the triples themselves. This implementation is naturallycalled indirect triples. For example, let us use an array statement to list pointers to triples in thedesired order.Thenthe triplesinFig.8.8(b)might be representedasinFig.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:IndirectTriplesSEMANTICANALYSIS: Thisphasefoc uses mainlyonthe

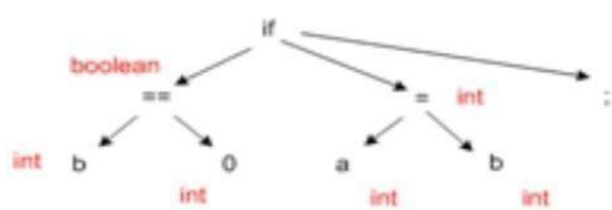
. Checkingthesemantics,

.Errorreporting

.Disambiguateoverloadedoperators

.Typecoercion,

.Staticchecking



- Typechecking
-Controlflowchecking
- Uniquenesschecking
- Namechecking aspectsoftranslation

Assume that the program has been verified to be syntactically correct and converted intosome kind of intermediate representation (a parse tree). One now has parse tree available. Thenext phase will be semantic analysis of the generated parse tree. Semantic analysis also includeserrorreportingincaseanysemanticerrorisfoundout.

Semantic analysis is a pass by a compiler that adds semantic information to the parse treeand performs certain checks based on this information. It logically follows the parsing phase, inwhich the parse tree is generated, and logically precedes the code generation phase, in which(intermediate/target) code is generated. (In a compiler implementation, it may be possible to folddifferent phases into one pass.) Typical examples of semantic information that is added andchecked is typing information (type checking) and the binding of variables and function namestotheirdefinitions(objectbinding).Sometimesalsosomeearlycodeoptimizationisdoneinthis phase. For this phase the compiler usually maintains symbol tables in which it stores whateachsymbol(variablenames,functionnames,etc.)refersto.

FOLLOWINGTHINGSAREDONEINSEMANTICANALYSIS:

Disambiguate Overloaded operators: If an operator is overloaded, one would like to specifythemeaningofthatparticularoperatorbecausefromonewillgointocodegenerationphasenext.

TYPE CHECKING: The process of verifying and enforcing the constraints of types is calledtype checking. This may occur either at compile-time (a static check) or run-time(a dynamiccheck). Static type checking is a primary task of the semantic analysis carried out by a compiler.Iftyperulesareenforcedstrongly(thatis,generallyallowingonlythoseautomatictypeconversions which do not lose information), the process is called strongly typed, if not, weaklytyped.

UNIQUENESSCHECKING: Whether avariablenameisuniqueor not,intheitsscope.

Typecoersion: Ifsomekindofmixingoftypesisallowed.Doneinlanguageswhicharenotstronglytyped.Thiscanbe donedynamicallyas wellas statically.

NAMECHECKS: Checkwhetheranyvariablehasanamewhichisnotallowed.Ex. Nameissame asanidentifier(Ex.intinjava).

- Parsercannotcatchalltheprogramerrors
- Thereisalevelofcorrectnessthatisdeeperthansyntaxanalysis
- Somelanguage featurescannot bemodeledusingcontextfreegrammarformalism

- Whetheranidentifierhasbeendeclaredbeforeuse,thisproblemisofidentifyingalanguage {wαw|wεΣ*}

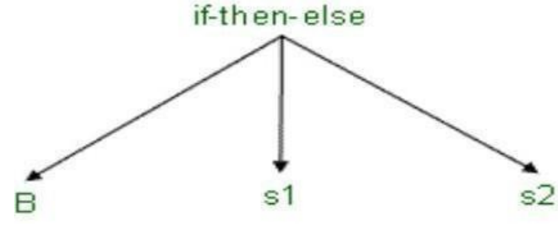
- Thislanguage isnotcontext free

Aparserhasitsownlimitationsincatchingprogramerrorsrelatedtosemantics,somethingthatis deeper than syntax analysis. Typical features of semantic analysis cannot be modeled usingcontext free grammar formalism. If one tries to incorporate those features in the definition of alanguage thenthatlanguage doesn'tremaincontextfreeanymore.
Example:instr
ingx;inty;
y=x+3
theuseofxisatypeerrorinta,
b;
a=b+ccisnot declared

An identifier may refer to differentvariables in differentparts of the program . An identifiermay be usable in one part of the program but not another These are a couple of examples whichtell us that typically what a compiler has to do beyond syntax analysis. The third point can beexplainedlike 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 tothese two different properties in the two different contexts. The fourth point can be explained inthis manner: A variable declared within one function cannot be used within the scope of thedefinitionoftheotherfunctionunlessdeclaredthereseparately.Thisisjustanexample.Probably you can think of many more examples in which a variable declared in one scope cannotbe usedinanotherscope.

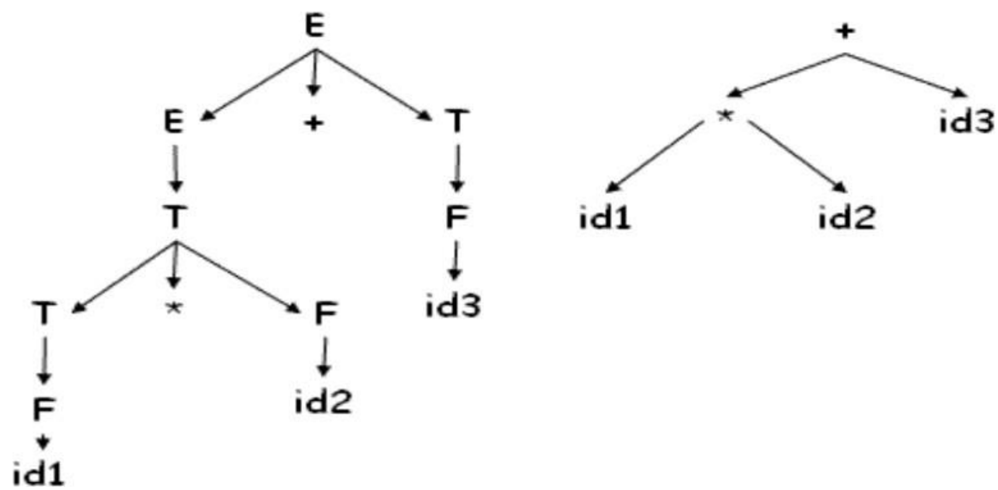
ABSTRACTSYNTAXTREE: Isnothingbutthecondensedformofaparsetree,Itis

- ∑Usefulforrepresentinglanguageconstructssonaturally.
- ∑TheproductionS ⟶ifBthens1else s2mayappearas



In the next few slides we will see how abstract syntax trees can be constructed from syntaxdirected definitions. Abstract syntax trees are condensed form of parse trees. Normally operatorsand keywords appear as leaves but in an abstract syntax tree they are associated with the interiornodes that would be the parent of those leaves in the parse tree. This is clearly indicated by theexamplesintheseslides.

Chainofsingleproductionsmaybecollapsed,and operatorsmovetotheparentnodes



Chainofsingleproductionsarecollapsedintoonenodewiththeoperatorsmovingupto becomethe node.

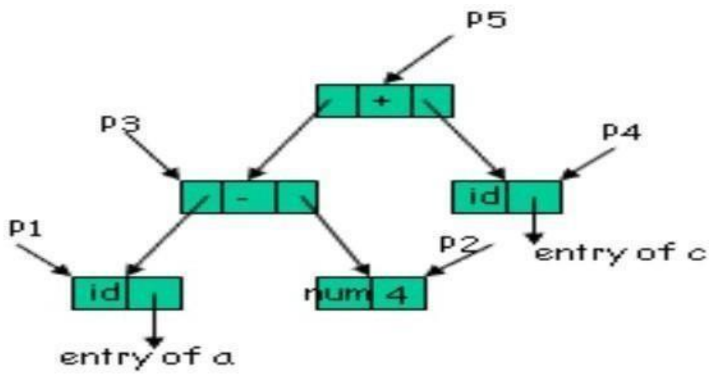
CONSTRUCTINGABSTRACTSYNTAXTREEFOREXPRESSIONS:

Inconstructing theSyntaxTree,wefollowtheconventionthat:

.Eachnodeofthetreecanberepresented asarecordconsisting ofatleasttwofieldstostoreoperatorsandoperands.
.***operators*****:** onefieldforoperator,remainingfieldsptrstooperands mknode(op,left,right)
.***identifier*****:**onefieldwithlabelidandanotherptrtosymboltablemkleaf(id,id.entry)
.***number*****:**onefieldwithlabelnumandanothertokeepthevalueofthenumbermkleaf(num,val)

Each node in an abstract syntax tree can be implemented as a record with several fields. In thenode for an operator one field identifies the operator (called the label of the node) and theremaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may haveadditional fields to hold values (or pointers to values) of attributes attached to the node. Thefunctions given in the slide are used to create the nodes of abstract syntax trees for expressions.Eachfunctionreturns apointertoanewlycreatednote.

ForExample:thefollowingsequence offunction callscreatesaparsetreeefor**w=a-4+c**



P 1 = mkleaf(id, entry.a)P2=mkleaf(num ,4)
P 3 = mknode(-, P 1 , P 2)P4 =mkleaf(id,entry.c)

P5=mknode(+,P3,P4)

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

A- Write the postfix equivalent of the expression for which we want to construct a syntax treeForabovestringw=a-4+c,itis**a4-c+**
B-Callthefunctionsinthesequence,asdefinedbythesequenceinthepostfixexpressionwhichresults inthedesiredtree.Inthecaseabove, callmkleaf()fora, mkleaf()for 4,mknode()for -,mkleaf()forc,andmknode()for+atlast.

1. P1=**mkleaf**(id, a.entry):Aleafnodemadefortheidentifiera, andanentryforais madeinthe symboltable.

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

3. P3=**mknode**(-,P1,P2):Aninternalnodeforthe-,takesthepointertopreviouslymadenodesP1,P2as argumentsandrepresents the expressiona-4.

4. P4=**mkleaf**(id, c.entry) :Aleaf nodemadefortheidentifierc,andanentryforc.entrymadeinthesymboltable.

5. P5 = **mknode**(+,P3,P4) : An internal node for the + , takes the pointer to previously madenodesP3,P4as arguments andrepresentstheexpressiona-4+c.

Followingisthesyntaxdirecteddefinitionforconstructingsyntaxtreeabove

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)

Nowwehavethesyntaxdirected definitionstoconstruct theparsetreeforagivengrammar.Alltherulesmentionedinslide 29 aretakencareofandanabstract syntaxtree isformed.

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

Inan**AG,**the**v**aluesofattributesataparsetreenodearecomputed bysemanticrules.Therearetwo different specifications of **AGs** used by the **Semantic** Analyzer in evaluating the semanticsoftheprogramconstructs.Theyare,

- **Syntaxdirected definition(SDD)s**
  - Highlevelspecifications
  - Hidesimplementationdetails
  - Explicitorderofevaluationisnotspecified
- **SyntaxdirectedTranslationschemes(SDT)s**
  - ∑Nothing but an SDD, which indicates order in which semantic rules are to beevaluatedand
  - ∑Allowsomeimplementationdetailstobeshown.

An **attribute grammar** is the formal expression of the syntax-derived semantic checksassociated with a grammar. It represents the rules of a language not explicitly imparted by thesyntax. In a practical way, it defines the information that is needed in the abstract syntax tree inorder to successfully perform semantic analysis. This information is stored as attributes of thenodesofthe abstractsyntaxtree.The valuesofthose attributesarecalculatedbysemantic rule.

Therearetwowaysforwritingattributes:

1) **SyntaxDirectedDefinition(SDD):**Isacontextfreegrammarinwhichasetofsemanticactionsa reembedded(associated)witheachproductionofG.

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

/*doesnotgiveany implementation details.Itjusttellsus.Thiskindof attributeequation we will be using, Details like at what point of time is it evaluated and in what mannerare hiddenfromtheprogrammer.*/

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 theattributes are evaluated, the order and place where they are evaluated. This is of a slightly lowerlevel.

AnSDT isanSDDinwhichsemantic actions canbe placedatanypositioninthe bodyoftheproduction.

Forexample, followingSDTprintstheprefixequivalentofanarithmeticexpressionconsistinga +and * operators.

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

Thisaction inanSDT,isexecuted assoonasitsnodeintheparsetreeisvisitedinapreordertraversalofthetree.

ConceptuallybooththeSDD and SDTschemeswill:

∑Parseinputtokenstream

∑Buildparsetree

∑Traverse the parse tree to evaluate the semantic rules at the parse tree nodesEvaluationmay:

∑Generatecode

∑Saveinformation inthesymboltable

∑Issueerrormessages

∑Performanyotheractivity

To avoidrepeatedtraversalofthe parse tree,actionsare takensimultaneouslywhena tokenisfound.Socalculationofattributesgoesalongwiththe constructionofthe 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 whilebuildingtheparsetree.

Thissavesmultiplepassesoftheparsetree.Exa
mple

Number ⟶ sign
listsign ⟶ +|-
list ⟶ list bit |bit
bit ⟶ 0|1

BuildattributegrammarthatannotatesNumberwiththevalueitrepresents

.Associateattributeswithgrammarsymbols

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

productionAttributerulenumber → sign
listlist.position 0

ifsign.negative

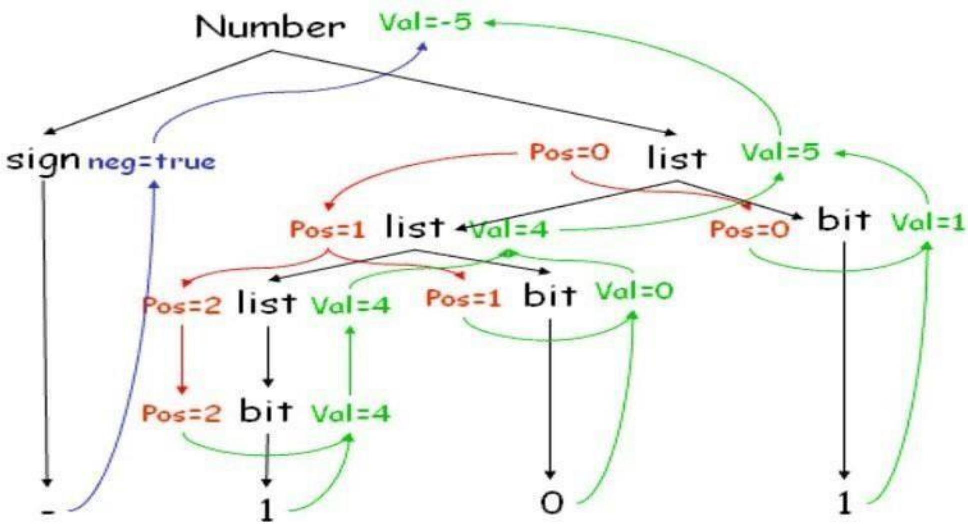
thennumber.value -
list.valueelsenumber.value
list.value
sign → +sign.negative false sign → -sign.negative truelist →
bitbit.position list.position
list.value
→ bit.value$list_0$
$list_1$bit
$list_1$.position
$list_0$.position+1bit.
position $list_0$.position
$list_0$.value $list_1$.value+bit.value
bit → 0bit.value 0bit → 1bit.value $2^{bit.position}$

Explanationofattributerules

Num->signlist /*since lististherightmostsoitisassignedposition0
*Signdetermineswhether thevalueofthenumber wouldbe
sameorthe negative ofthevalue oflist/
Sign->+|- /*SettheBooleanattribute(negative)forsign*/
List->bit /*bit positionisthesameaslistpositionbecausethisbitistherightmost
*value of the list is same as
bit.*/**List0->List1 bit**
/*positionandvaluecalculations*/**B**
it ->0|1 /*setthecorrespondingvalue*/

AttributesofRHScanbe computed fromattributesofLHSandvice versa.

TheParseTreeandtheDependencegraphareasunder



.

Dependence graph shows the dependence of attributes on other attributes, along with thesyntax tree. Top down traversal is followed by a bottom up traversal to resolve the dependencies.Number,valandnegare synthesizedattributes.Posisan inheritedattribute.

Attributes:.Attributesfallintotwoclassesnamely*synthesizedattributes*and*inheritedattributes*. Value of a synthesized attribute is computed from the values of its children nodes.Value ofaninheritedattributeis computedfromthe siblingandparentnodes.

The attributes are divided into two groups, called synthesized attributes and inheritedattributes. The synthesized attributes are the result of the attribute evaluation rules also using thevalues of the inherited attributes. The values of the inherited attributes are inherited from parentnodesandsiblings.

EachgrammarproductionA $\longrightarrow$ ahasassociatedwithitasetofsemanticrulesoftheformb=f(c_1, c_2, ..., c_k) ,Wheref isafunction,andeither,bisasynthesizedattributeofAOr
- b isaninherited attributeofoneofthegrammarsymbolsontheright

.attribute bdepends onattributesc_1,c_2,...,c_k

Dependence relation tells us what attributes we need to know before hand to calculate aparticularattribute.

Here the value of the attribute b depends on the values of the attributes $\mathbf{c_1}$ to $\mathbf{c_k}$. If c_1 to $c_k$belong to the children nodes and b to A then b will be called a synthesized attribute. And if bbelongs to one among a (child nodes) then it is an inherited attribute of one of the grammarsymbolsontheright.

SynthesizedAttributes:AsyntaxdirecteddefinitionthatusesonlysynthesizedattributesissaidtobeanS-attributeddefinition

.Aparsetreefor anS-attributeddefinitioncanbeannotatedbyevaluatingsemanticrulesforattributes

S-attributed grammars are a class of attribute grammars, comparable with L-attributed grammarsbut characterized by having no inherited attributes at all. Inherited attributes, which must bepassed down from parent nodes to children nodes of the abstract syntax tree during the semanticanalysis,poseaproblemforbottom-upparsingbecauseinbottom-upparsing,theparentnodesof the abstract syntax tree are created*after* creation of all of their children.Attribute evaluationin S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-upparsing.

SyntaxDirectedDefinitions foradeskcalculatorprogram

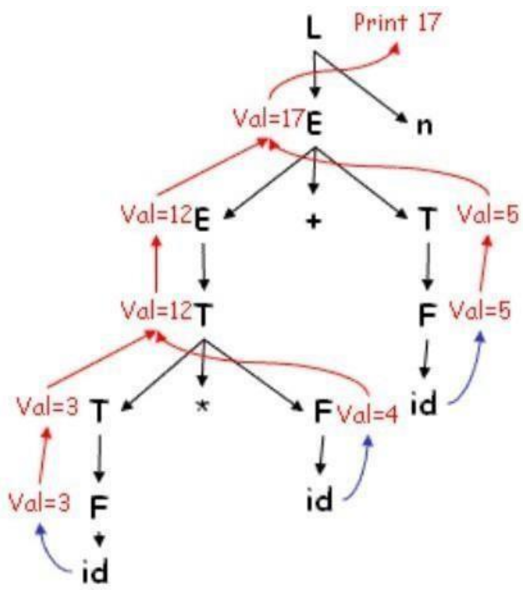
L ⟶En	Print(E.val)
E ⟶E+T	E.val=E.val+T.val
E ⟶T	E.val=T.val
T ⟶T*F	T.val=T.val*F.val
T ⟶F	T.val=F.val
F ⟶(E)	F.val=E.val
F ⟶digit	F.val=digit.lexval

.terminalsareassumedtohaveonlysynthesized attributevaluesofwhicharesuppliedbylexicalanalyzer

. startsymboldoesnothaveanyinheritedattribute

Thisisagrammarwhichusesonlysynthesizedattributes. Startsymbolhasnoparents, hencenoinheritedattributes.

Parse tree for3*4+ 5n



Usingthepreviousattributegrammar calculationshavebeenworkedoutherefor 3*4+5n.Bottomupparsinghasbeendone.

InheritedAttributes:Aninheritedattributeisonewhosevalueisdefinedintermsofattributesatt heparentand/orsiblings

. Usedforfindingoutthecontextinwhichitappears

. possibletouseonlyS-attributesbutmorenaturaltouseinheritedattributesD

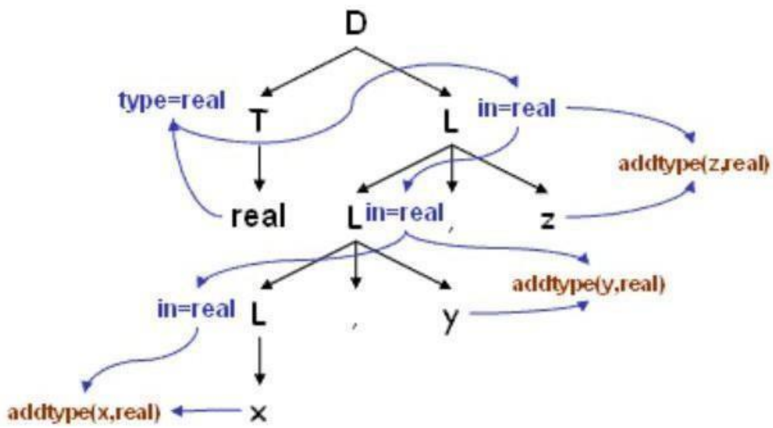
TL	L.in=T.type
T $\longrightarrow$ real	T.type=real
T $\longrightarrow$ int	T.type=int
L $\longrightarrow$ L_1,id	L_1.in=L.in;addtype(id.entry,L.in)
L $\longrightarrow$ id	addtype(id.entry,L.in)

Inheritedattributeshelptofindthecontext(type,scopeetc.)ofatokene.g.,thetypeofatokenor scope when the same variable name is used multiple times in a program in different functions.An inherited attribute system may be replaced by an S -attribute system but it is more natural touseinheritedattributesinsome caseslike the example givenabove.

Hereaddtype(a,b)functionsaddsasymboltableentryfortheid aand attachesto it thetypeofb

.

Parsetreefor realx,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 bycomputing the value of the attribute T.type at the left child of the root and then valuating L.in topdownatthethreeLnodesintherightsubtreeoftheroot.AteachLnodetheprocedureaddtypeis called which inserts the type of the identifier to its entry in the symbol table. The figure alsoshowsthedependencegraphwhichisintroducedlater.

Dependence Graph:.Ifanattribute bdependsonanattributecthenthe semantic ruleforbmustbeevaluatedafterthe semanticruleforc

.Thedependenciesamongthenodescanbedepicted byadirected graphcalleddependencygraph

DependencyGraph **:**Directedgraphindicating interdependenciesamongthesynthesizedandinheritedattributes ofvariousnodesinaparse tree.

Algorithmtoconstructdependencygraphfo
reachnode**n**inthe parsetree do
foreachattribute**a**ofthegrammarsymboldocons
tructanode inthe dependencygraph
for**a**

for each nodenintheparsetreedo

foreachsemantic ruleb=f(c_1,c_2,...,c_k)do

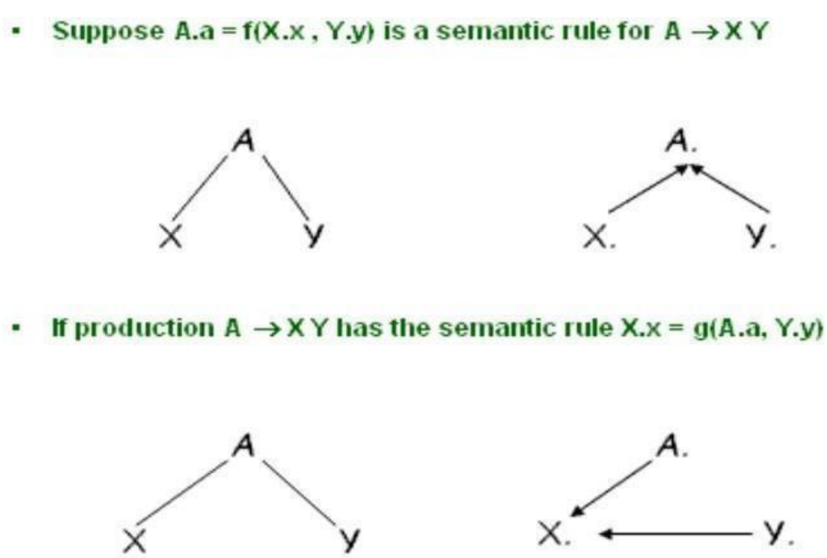
{ associatedwithproductionatn}

fori= 1tokdo

Constructanedgefrom$c_i$tob

Analgorithm toconstructthedependency graph.Aftermakingonenodeforeveryattribute of all the nodes of the parse tree, make one edge from each of the other attributes onwhichitdepends.

Forexample,



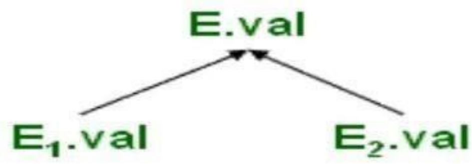
The semantic rule A.a = f(X.x , Y.y) for the production A -> XY defines the synthesizedattribute a of A to be dependent on the attribute x of X and the attribute y of Y . Thus thedependency graph will contain an edge from X.x to A.a and Y.y to A.a accounting for the twodependencies. Similarly for the semantic rule X.x = g(A.a , Y.y) for the same production therewillbeanedgefromA.atoX.xandanedgefromY.ytoX.x.

Example

.Wheneverfollowingproductionisused inaparsetreeE

$\rightarrow$ E1+E2 E.val=E1.val+E2.val

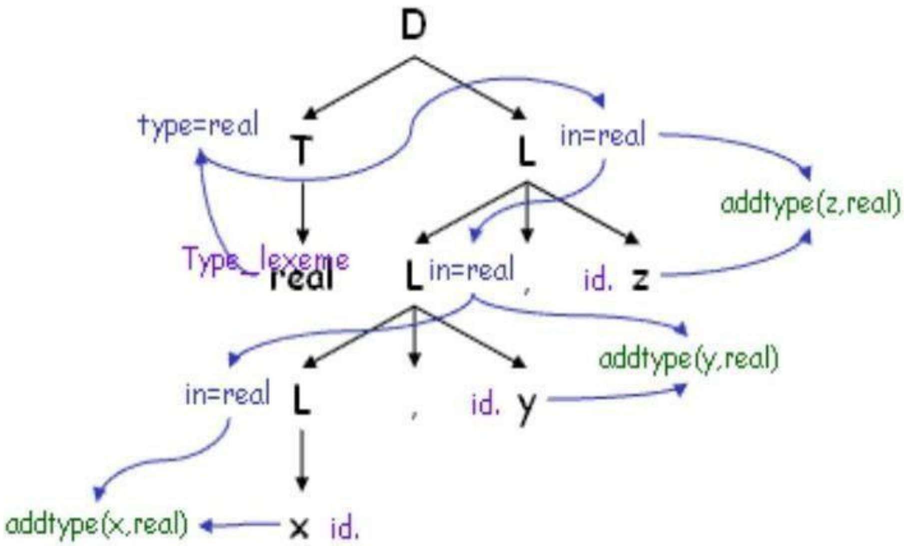
wecreate adependencygraph



The synthesized attribute E.val depends on E1.val and E2.val hence the two edges oneeachfromE1.val&E2.val

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

.Putadummysynthesized attributeb for asemanticrulethatconsistsofaprocedurecall



The figure shows the dependency graph for the statement real id1, id2, id3 along with theparse tree. Procedure calls can be thought of as rules defining the values of dummy synthesizedattributes of the nonterminal on the left side of the associated production. Blue arrows constitutethedependency graphandblacklines,theparsetree.Eachof thesemanticrulesaddtype(id.entry,L.in)associated withtheLproductionsleadsto thecreationofthedummyattribute**.**

Evaluation Order:

Anytopologicalsortofdependencygraphgivesavalidorderinwhichsemanticrulesmustbeevaluate
d

a4 =
reala5=a
4

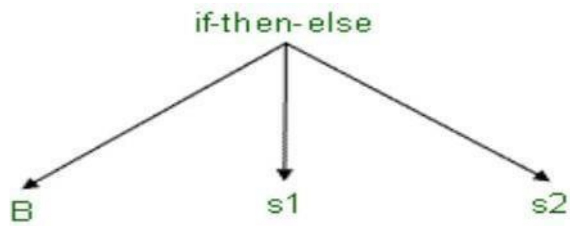
addtype(id3.entry,
a5)a7=a5addtype(id2.
entry,a7)

a9:=a7addtype(id1.entry,a9)

A topological sort of a directed acyclic graph is any ordering m1, m2, m3 ....mk of thenodes of the graph such that edges go from nodes earlierin the ordering to later nodes. Thus ifmi ->mj is an edge from mi to mj then mi appears before mj in the ordering. The order of thestatements shown in the slideis obtainedfrom the topological sort of the dependency graph inthe previous slide.'an' stands for the attribute associated with the node numbered n in thedependencygraph.Thenumberingis asshowninthepreviousslide.

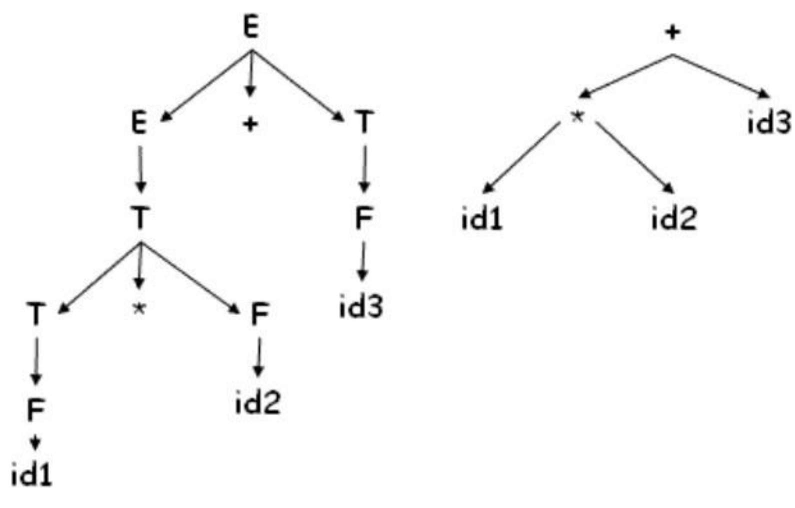
AbstractSyntaxTreeisthe condensedformoftheparsetree,whichis

.Usefulforrepresenting languageconstructs.
.Theproduction:**S** →**if**B**then**s1elses2mayappearas



In the next few slides we will see how abstract syntax trees can be constructed fromsyntax directed definitions. Abstract syntax trees are condensed form of parse trees. Normallyoperators and keywords appear as leaves but in an abstract syntax tree they are associated withthe interior nodes that would be the parent of those leaves in the parse tree. This is clearlyindicatedbytheexamplesintheseslides.

.Chainofsingleproductionsmaybecollapsed,andoperatorsmovetotheparentnodes



Chainofsingleproductionarecollapsedintoonenodewiththeoperatorsmovingupto becomethe node.

ForConstructingtheAbstractSyntaxtreeforexpressions,

.Eachnode canbe representedasa record

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

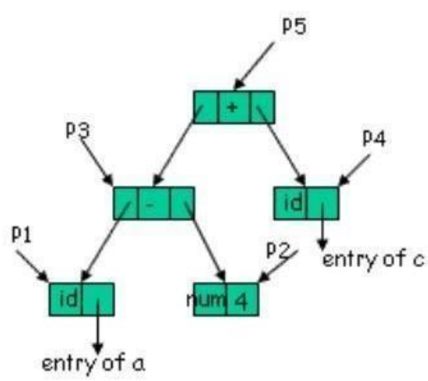
.*identifier*:onefieldwith labelidandanother ptrtosymboltablemkleaf(id,entry)

.*number*:onefield with labelnumand anothertokeep the valueofthenumbermkleaf(num,val)

Each node in an abstractsyntax 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 theremaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may haveadditional fields to hold values (or pointers to values) of attributes attached to the node. Thefunctions given in the slide are used to create the nodes of abstract syntax trees for expressions.Eachfunctionreturns apointertoanewlycreated note.

Example:**Thefollowingsequence of functioncallscreatesaparsetree fora-4+ c**

P 1 = mkleaf(id, entry.a)P2=mkleaf(num ,4)
P 3 = mknode(-, P 1 , P 2)P4 =mkleaf(id,entry.c)
P5=mknode(+,P3,P4)



Anexampleshowing theformationofanabstract syntaxtreebythegiven functioncallsfortheexpressiona-4+c.Thecallsequence canbeexplainedas:

1. P1=mkleaf(id,entry.a):AleafnodemadefortheidentifierQaRand anentryfor QaRismadeinthesymboltable.
2. P2=mkleaf(num,4):AleafnodemadeforthenumberQ4R.
3. P3 =mknode(-,P1,P2):AninternalnodefortheQ-Q.ItakesthepreviouslymadenodesasargumentsandrepresentstheexpressionQa-4R.
4. P4=mkleaf(id,entry.c):AleafnodemadefortheidentifierQcRandanentryfor QcRismadeinthesymboltable.
5. P5=mknode(+,P3,P4):An internalnodefortheQ+Q.Itakesthepreviouslymadenodesasargumentsandrepresentstheexpression Qa-4+cR.

Asyntaxdirecteddefinitionforconstructingsyntaxtree

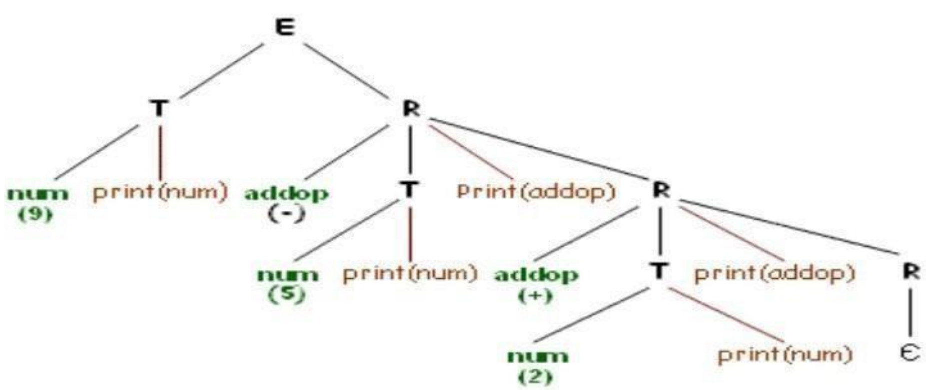
E ⟶E 1+T	E.ptr=mknode(+,E1.ptr,T.ptr)
E⟶ T	E.ptr=T.ptr
T ⟶T 1*F	T.ptr:= mknode(*,T1.ptr,F.ptr)
T ⟶F	T.ptr:=F.ptr
F ⟶(E)	F.ptr :=E.ptr
F ⟶id	F.ptr:=mkleaf(id,entry.id)
F⟶ num	F.ptr:=mkleaf(num,val)

Nowwehavethesyntaxdirected definitionstoconstruct theparsetreeforagivengrammar.Alltherulesmentionedinslide 29 aretakencareofandanabstract syntaxtree isformed.

Translationschemes: ACFGwheresemanticactionsoccur withintherighthandsideofproduction,Atranslationschemetomapinfixtopostfix.

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

Parse tree for9-5+2



We assume that the actions are terminal symbols and Perform depth first order traversal to obtain95-2+.

∑Whendesigningtranslationscheme,ensureattributevalueisavailablewhenreferredto

∑Incaseofsynthesizedattributeitistrivial(why?)

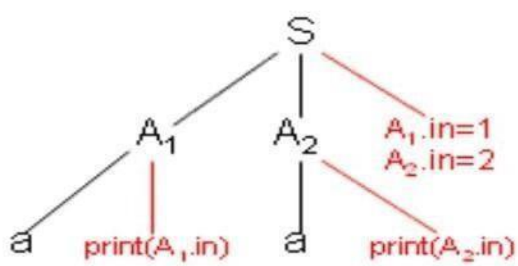
In a translation scheme, as we are dealing with implementation, we have to explicitlyworry about the order of traversal. We can now put in between the rules some actions as part ofthe RHS. We put this rules in order to control the order of traversals. In the given example, wehave twoterminals(numandaddop).Itcangenerallybe seenasa numberfollowedbyR(which necessarily has to begin with an addop). The given grammar is in infix notation and we need toconvert it into postfix notation. If we ignore all the actions, the parse tree is in black, without thered edges. If we include the red edges we get a parse tree with actions. The actions are so fartreated as a terminal. Now, if we do a depth first traversal, and whenever we encounter a actionwe execute it, we get a post-fix notation. In translation scheme, we have to take care of theevaluation order; otherwise some of the parts may beleftundefined.For differentactions,differentresultwill 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, wedo not have any evaluation order; in this case we have an explicit evaluation order. By explicitevaluation order we have to set 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 thattranslation scheme is all about. If we talk of only synthesized attribute, the translation scheme isvery trivial. This is because, when we reach we know that all the children must have beenevaluated and all their attributes must have also been dealt with. This is because finding the placeforevaluationis verysimple,itistherightmostplace.

Incaseofbothinheritedand synthesizedattributes

. An inherited attribute for a symbol on rhs of a production must be computed in an action beforethatsymbol

S→A1A 2{A1.in=1,A2.in=2}

A →a {print(A.in)}



Depthfirstordertraversalgiveserror*undefined*

. Asynthesizedattributefor nonterminalonthelhscanbecomputedafter allattributesitreferences,havebeencomputed.Theactionnormallyshouldbeplaced at theend ofrhs

We have a problem when we have both synthesized as well as inherited attributes. For the givenexample, if we place the actions as shown, we cannot evaluate it. This is because, when doing adepth first traversal, we cannot print anything for A1. This is because A1 has not yet beeninitialized. We, therefore have to find the correct places for the actions. This can be that theinherited attribute of A mustbe calculated on its left. This can beseen logically from thedefinition of L-attribute definition, which says that when we reach a node, then everything on itsleftmusthavebeencomputed.Ifwedo this,wewillalwayshavetheattributeevaluatedat the

correctplace.
Forsuchspecificcases(likethegivenexample)calculatinganywhereontheleftwillwork,butgenerallyit mustbe calculatedimmediatelyattheleft.

Example:TranslationschemeforEQN

S →B

B.pts= 10S.ht=B. ht

B →B$_1$B$_2$ B$_1$.pts = B.ptsB$_2$.pts=B. pts
B.ht=max(B$_1$.ht,B$_2$.ht)

B →B$_1$subB$_2$ 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

We now look at another example. This is the grammar for finding out how do I compose text.EQN was equation setting system which was used as an early type setting system for UNIX. Itwas earlier used as an latex equivalent for equations. We say that start symbol is a block: S ->BWecanalsohaveasubscriptandsuperscript.Here,welookatsubscript.ABlockiscomposedof several blocks: B -> B1B2 and B2 is a subscript of B1. We have to determine what is the pointsize (inherited) and height Size (synthesized). We have the relevant function for height and pointsize givenalongside.Afterputtingactionsinthe rightplace

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 all the actions at the correct places as per the rules stated. Read it from left to right,and top to bottom. We note that all inherited attribute are calculated on the left of B symbols andsynthesizedattributes areontheright.

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-attributedefinitions to construct the parse tree with a one-to-one correspondence. We first look at the top-downtranslationscheme.Thefirstmajorproblemisleftrecursion.Ifweremoveleftrecursionby our standard mechanism, we introduce new symbols, and new symbols will not work with theexistingactions.Also,we have todothe parsingina singlepass.

TYPESYSTEM ANDTYPECHECKING:

.Ifboththe operandsofarithmeticoperators+,-,xareintegers thenthe resultis oftypeinteger
.Theresultofunary&operatorisapointertotheobjectreferred tobytheoperand.
- Ifthe type ofoperandis***X***thentype ofresultis***pointertoX***

InPascal, typesareclassifiedunder:

1. *Basic*types:Theseareatomictypeswithnointernalstructure.Theyincludethetypesboolean,character,integerandreal.

2. *Sub-range*types:Asub-rangetypedefinesarangeofvalueswithin therangeofanothertype.Forexample,typeA=1..10;B=100..1000;U ='A'..'Z';

3. *Enumerated* types: An enumerated type is defined by listing all of the possible values for thetype. For example: type Colour = (Red, Yellow, Green); Country = (NZ, Aus, SL, WI, Pak, Ind,SA,Ken,Zim,Eng);Boththe sub-rangeandenumeratedtypescanbetreated asbasictypes.

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 functionscanalsobetreatedas constructedtypes.

TYPEEXPRESSION:

Itisanexpressionthatdenotesthetypeofanexpression.Thetypeofalanguageconstruct isdenotedbyatypeexpression

- ∑It is either a basic type or it is formed by applying operators called *type constructor* toothertypeexpressions
- ∑Atypeconstructorapplied toatypeexpression isatypeexpression
- ∑Abasictypeistypeexpression

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

The type of a language construct is denoted by a type expression. A type expression is either abasictypeorisformedbyapplyinganoperatorcalledatypeconstructortoothertypeexpressions.Formally,a type expressionisrecursivelydefinedas:

1. Abasictypeisatypeexpression.Amongthebasictypesare*boolean*,*char*,*integer*,and *real* .A special basic type, *type_error*, is used to signal an error during type checking. Anotherspecialbasictypeis*void*whichdenotes"theabsenceofavalue"andisusedtocheckstatements.
2. Sincetypeexpressionsmaybenamed, atypenameisatypeexpression.
3. Theresultofapplyingatypeconstructortoatypeexpressionisatypeexpression.
4. Typeexpressions maycontainvariableswhosevaluesaretypeexpressionsthemselves.

TYPECONSTRUCTORS:areusedtodefineorconstructthetypeofuser defined typesbasedontheirdependenttypes.
Arrays: IfTisatypeexpressionand Iisarangeofintegers,then*array* (*I*,*T*)isthetypeexpressiondenotingthetype ofarraywithelementsoftypeTandindexsetI.

Forexample,thePascaldeclaration,
varA:array[1..10]ofinteger;associatesthetypeexpression***array*(1..10,*integer*)**withA.

Products*:*If*T1* and*T2* aretypeexpressions,thentheir Cartesianproduct***T1 XT2***isalsoatypeexpression.

Records*:*Arecordtypeconstructorisappliedtoatupleformedfromfieldnamesandfieldtypes.Forexample,the declaration
Consider the
declarationtyperow=rec
ord
addr : integer;
lexeme:array[1..15]ofcharend;
vartable:array[1..10]ofrow;

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

Note:Includingthefieldnames inthetypeexpressionallowsustodefineanotherrecordtypewiththesamefieldsbutwithdifferentnameswithoutbeingforced to equatethetwo.

Pointers: IfTisatypeexpression,then ***pointer***(***T***)isatypeexpressiondenoting thetype"pointertoanobjectoftypeT".
Forexample,inPascal,thedeclaration
varp:row declaresvariableptohavetype***pointer(*row*)***.

Functions : Analogous to mathematical functions, functions in programming languages may bedefined as mapping a domain type D to a range type R. The type of such a function is denoted bythe type expression D R. For example, the built-in function mod of Pascal has domain type int Xint,andrangetype*int*.Thuswe saymodhasthetype:**intxint->int**

As another example, according to the Pascal declarationfunctionf(a,b:char):integer;
Here thetype offisdenotedbythetype expressionis**charxcharpointer(integer)**

SPECIFICATIONSOFATYPECHECKER: Consideralanguagewhichconsistsofasequence ofdeclarationsfollowedbya singleexpression

P $\longrightarrow$ D ;E

D $\longrightarrow$ D;D|id:T

T $\longrightarrow$ char|integer|array[num]ofT| ^TE
literal|num|EmodE|E[E]|E^

A **type checker** is a translation scheme that synthesizes the type of each expression from thetypesofitssub-expressions.Considertheabovegivengrammarthatgeneratesprogramsconsistingofasequence ofdeclarationsDfollowedbyasingle expressionE.

Specificationsofa typecheckerforthelanguageoftheabovegrammar:Aprogramgeneratedbythis grammaris

key :
integer;keymo
d1999

Assumptions:

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

2. Forsimplicity, allarraysstartat1.Forexample, thedeclarationarray[256]ofcharleadstothetype 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:

ConsidertheSyntaxDirectedDefinition,

E →E₁(E₂)

E.type=ifE_2.type==sandE_1.type==s → t
thent

elsetype-error

Therulesforthesymboltableentryarespecified above.These arebasicallythewayin whichthe symboltable entriescorrespondingtotheproductionsaredone.

Typecheckingoffunctions

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

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

This rule says that in an expression formed by applying E1 to E2, the type of E1 must be afunction *s* -> *t* from the type *s*of E2 to some range type *t* ; the type of E1 (E2) is *t* . The aboverule can be generalized to functions with more than one argument byconstructing a product typeconsistingofthearguments.Thus narguments oftype*T1*,*T2*

...*Tn*canbeviewed asasingleargumentofthetype*T1 XT2...XTn*.For
example,root:(realreal)Xrealreal
declaresafunctionrootthat takesafunction fromrealsto realsand arealasargumentsandreturnsa real.The Pascal-like syntaxforthisdeclarationis

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

TYPECHECKINGFOREXPRESSIONS:considerthefollowingSDD forexpressions

E →literal	E.type=char
E →num	E.type=integer
E →id	E.type=lookup(id.entry)
E →E_1modE_2	E.type=ifE_1.type==integerand E_2.type==integer theninteger

	elsetype_error
E ⟶E_1[E_2]	E.type= ifE_2.type==integerandE_1.type== array(s,t) thent elsetype_error
E ⟶E_1^	E.type=ifE_1.type==pointer(t)th ent elsetype_error

Toperform type checkingof expressions,followingrules are used.Where the synthesizedattribute type for E gives the type expression assigned by the type system to the expressiongeneratedbyE.

Thefollowingsemanticrulessaythatconstantsrepresentedbythetokensliteralandnumhavetype*char* and*integer* ,respectively:

E -> literal { *E.type*:= *char*
}E->num{*E.type*:= *integer* }
.The function*lookup* (*e*)isused to fetchthetypesavedinthesymbol-tableentrypointedtoby e.Whenanidentifierappears inanexpression,itsdeclaredtypeis fetchedandassignedtotheattributetype:

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

. According to the following rule, the expression formed by applying the mod operator to twosub-expressionsoftype*integer*hastype*integer*;otherwise,itstypeis*type_error*.

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

InanarrayreferenceE1 [E2],theindexexpressionE2 musthavetype*integer*,inwhichcasethe resultisthe elementtype*t*obtainedfromthe 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 yieldstheobjectpointedtobyitsoperand.ThetypeofEisthetype*t*oftheobjectpointedtobythepointerE:

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

TYPE CHECKING OF STATEMENTS: Statements typically do not have values. Specialbasic type *void* can be assigned to them. Consider the SDD for the grammar below whichgeneratesAssignmentstatementsconditional,andloopingstatements.

S ⟶id:=E	S.Type=ifid.type==E.typeth envoid elsetype_error
S ⟶ifEthenS1	S.Type=ifE.type==boolean then S1.typeelsetyp e_error
S ⟶while EdoS1	S.Type=ifE.type==boolean thenS1.type elsetype_error
S ⟶S1;S2	S.Type= ifS1.type==voidand S2.type ==void thenvoid elsetype_error

Since statements do not have values, the special basic type *void* is assigned to them, but if anerrorisdetectedwithinastatement,thetypeassigned tothe statementis*type_error*.

The statements considered below are assignment, conditional, and whilestatements. Sequencesof statements are separated by semi-colons. The productions given below can be combined withthose given before if we change the production for a complete program to P-> D; S. Theprogramnowconsistsofdeclarationsfollowedbystatements.

Rules fortypecheckingthestatementsaregivenbelow.

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

Thisrulechecksthattheleftandrightsidesofanassignmentstatementhavethesametype.

2. S ifEthenS1 {*S.type*:= if*E.type*== *boolean*then*S1.type* else *type_error*}

Thisrulespecifiesthattheexpressionsinanif-thenstatementmusthavethetype*boolean*.

3. S while EdoS1{*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*}

Errors are propagated by this last rule because a sequence of statements has type *void* only ifeachsub-statementhas type*void*.

IMPORTANT&EXPECTEDQUESTIONS

1. What do you mean by THREE ADDRESS CODE? Generate the three-address code forthe followingcode.

```
begin
        PROD: =0;
        I:=1;
dobe
gin
        PROD:=PROD + A[I]
        B[I];I:=I+1;
End
```

whileI<=20e
nd

2. Writeashort noteonAttributed grammar& Annotated parsetree.
3. Defineanintermediatecodeform.Explainvariousintermediatecodeforms?
4. WhatisSyntaxDirectedTranslation?ConstructSyntaxDirectedTranslationschemetoconve rta givenarithmetic expressionintothreeaddresscode.
5. WhatareSynthesizedandInheritedattributes?Explainwithexamples?
6. ExplainSDTforSimpleTypechecker?
7. Defineandconstructtriples, quadruplesandindirecttriplenotationsofanexpression:a* -(b+ c).

ASSIGNMENTQUESTIONS:

1. WriteThreeaddresscodeforthebelowexample

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

2. What is a Syntax Directed Definition? Write Syntax Directed definition to convert binaryvalueintodecimal?

SYMBOLTABLE

SymbolTable(ST): Isadatastructure usedbythe**c**ompilerto keeptrackofscopeandbindinginformationaboutnames

- Symboltableischanged everytimeanameisencountered inthesource;

Changes to table occur when ever a new name is discovered; new information about an existingnameis discovered

As we know the compiler uses a symbol table to keep track of scope and binding informationabout names. It is filled after the AST is made by walking through the tree, discovering andassimilating information about the names. There should be two basic operations - to insert a newname or information into the symbol table as and when discovered and to efficiently lookup anameinthe symboltable toretrieveitsinformation.

Two commondata structuresusedforthe symboltable organizationare-

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

performance.Ideallyacompilershouldbeabletogrowthesymboltabledynamically,i.e.,insertnewentriesorinformationas andwhenneeded.

Butifthesizeofthetableisfixedinadvancethen(anarrayimplementationfor example),thenthesizemustbebig enough inadvanceto accommodatethelargestpossibleprogram.

Foreachentryindeclarationofa name

- Theformatneednotbeuniformbecauseinformationdependsupontheusageofthename
- Eachentryis a recordconsistingofconsecutivewords
- TokeeprecordsuniformsomeentriesmaybeoutsidethesymboltableInfor

mationisenteredintosymboltableatvarioustimes.Forexample,

- keywordsareenteredinitially,
- identifierlexemesareenteredbythelexicalanalyzer.

. Symboltableentrymaybesetupwhenroleofnamebecomesclear ,attributevaluesarefilledinasinformationis available duringthetranslationprocess.

For each declaration of a name, there is an entry in the symbol table. Different entriesneed to store different information because of the different contexts in which a name can occur.An entry corresponding to a particular name can be inserted into the symbol table at differentstages depending on when the role of the namebecomes clear.The various attributes thatanentry in the symbol table can have are lexeme, type of name, size of storage and in case offunctions -theparameterlistetc.

Anamemaydenote severalobjectsinthesame block

- intx;structx{floaty,z;}

The lexical analyzer returns the name itself and not pointer to symbol table entry. A record in thesymbol table is created when role of the name becomes clear. In this case two symbol tableentriesarecreated.

∑Aattributesofa nameareentered inresponse to declarations

- ∑Labelsareoften identifiedbycolon

The syntax of procedure / function specifies that certain identifiers are formals, characters in aname.Thereisa distinctionbetweentoken id,lexemeandattributesofthenames. Itisdifficulttoworkwithlexemes

- ∑ifthereismodestupperboundonlengththen lexemescan be storedinsymboltable
- ∑iflimitis largestorelexemesseparately

There might be multiple entries in the symbol table for the same name, all of them havingdifferent roles. It is quite intuitive that the symbol table entries have to be made only when therole of a particular name becomes clear. The lexical analyzer therefore just returns the name andnotthesymboltableentryasitcannotdeterminethecontextofthatname.Attributescorresponding to the symbol table are entered for a name in response to the correspondingdeclaration. There has to be an upper limit for the length of the lexemes for them to be stored inthe symboltable.

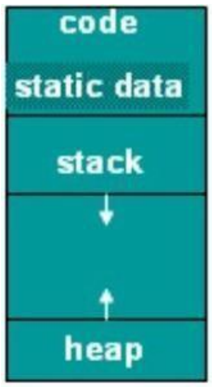
STORAGEALLOCATIONINFORMATION: Informationaboutstoragelocationsiskeptinthe symboltable.

Iftargetcodeisassemblycode,thenassemblercantakecareofstorage forvariousnamesandthecompilerneedstogeneratedata definitionstobe appended toassemblycode

If target code is machine code, then compiler does the allocation. No storage allocation is donefornames whose storageis allocatedatruntime

Information about the storage locations that will be bound to names at run time is kept inthesymbol table.If thetargetis assembly code, theassemblercan takecare of storageforvarious names. All the compiler has to do is to scan the symbol table, after generating assemblycode, and generate assembly language data definitions to be appended to the assembly languageprogram for each name. If machine code is to be generated by the compiler, then the position ofeach data object relative to a fixed origin must be ascertained. The compiler has to do theallocation in this case. In the case of names whose storage is allocated on a stack or heap, thecompiler doesnotallocatestorageat all,itplansouttheactivationrecord for eachprocedure.

STORAGEORGANIZATION:



Theruntimestoragemightbesu bdividedinto:

- ∑Targetcode,
- ∑Dataobjects,
- ∑Stackto keeptrackofprocedure activation,and
- ∑Heapto keep allotherinformation

This kind of organization of run-time storage is used for languages such asFortran, Pascal and C. The size of the generated target code, as well as that ofsomeofthedataobjects,isknownatcompiletime.Thus,these canbe stored instaticallydeterminedareas inthememory.

STORAGEALLOCATIONPROCEDURECALLS: PascalandCusethe stack for procedure activations. Whenever a procedure is called, execution ofactivationgetsinterrupted,andinformationaboutthemachinestate(likeregisterva lues)is storedonthestack.

When the called procedure returns, the interrupted activation can be restarted after restoring thesaved machine state. The heap may be used to store dynamically allocated data objects, and alsoother stuff such as activation information (in the case of languages where an activation treecannot be used to represent lifetimes). Both the stack and the heap change in size during programexecution,sothey cannotbe allocated a fixedamountof space.Generally they startfromopposite ends of the memory and can grow as required, towards each other, until the spaceavailablehasfilledup.

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

- ∑Temporaries:usedinexpressionevaluation
- ∑Localdata:fieldforlocaldata
- ∑Saved machinestatus:holdsinfo aboutmachinestatus beforeprocedurecall
- ∑Accesslink : to accessnon localdata
- ∑Controllink :pointsto activationrecordofcaller
- ∑Actualparameters: field toholdactualparameters
- ∑Returnedvalue:fieldforholdingvaluetobereturned

The activation record is used to store the information required by asingle procedure call. Not all the fields shown in the figure may beneeded for all languages. The record structure can be modified as perthe language/compilerrequirements.

For Pascal and C, the activation record is generally stored on the run-timestack during theperiodwhentheprocedureisexecuting.

Of the fields shown in the figure, access link and control link are optional (e.g. FORTRANdoesn't need access links). Also, actual parameters and return values are often stored in registersinsteadoftheactivationrecord,forgreaterefficiency.

- ∑ The activation record for a procedure call is generated by the compiler. Generally, allfieldsizescanbe determinedatcompiletime.

However, this is not possible in the case of a procedure which has a local array whose sizedepends on a parameter. The strategies used for storage allocation in such cases will be discussedinforthcominglines.

STORAGEALLOCATIONSTRATEGIES: ThestorageisallocatedbasicallyinthefollowingTHREEways,

∑Staticallocation:laysoutstorageatcompiletimeforalldataobjects

∑Stackallocation:managestheruntimestorageasastack

∑Heapallocation:allocatesand de-allocatesstorageasneededat runtimefromheap

These represent the different storage-allocation strategies used in the distinct parts of therun-time memory organization (as shown in slide 8). We will now look at the possibility of usingthese strategies to allocate memory foractivation records. Differentlanguages use differentstrategiesforthispurpose.Forexample,oldFORTRANusedstaticallocation,Algol typelanguagesuse stack allocation,and LISPtypelanguagesuse heap allocation.

STATIC ALLOCATION: In this approach memory is allocated statically. So,Names are boundtostorageastheprogramis compiled

∑Noruntimesupportisrequired
∑Bindingsdonot changeatruntime
∑Oneveryinvocationofprocedure namesareboundtothe samestorage
∑Valuesoflocalnamesare retained acrossactivationsofaprocedure

These are the fundamental characteristics of static allocation. Since name binding occurs duringcompilation, there is no need for a run-time support package. The retention of local name valuesacross procedure activations means that when control returns to a procedure, the values of thelocals are the same as they were when control lastleft. For example, suppose we had thefollowingcode,writteninalanguageusingstaticallocation:

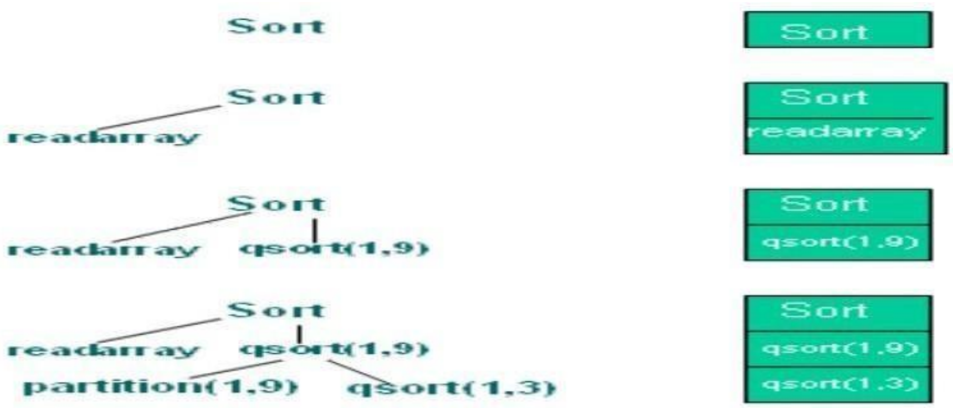
```
functionF()
 {
       inta;pri
       nt(a);a=
       10;
 }
```

After calling F( ) once,if itwas called a second time,the value of a wouldinitially be 10,andthisis whatwouldgetprinted.
The type of a name determines its storage requirement. The address for this storage is an offsetfrom the procedure's activation record, and the compilerpositions the records relative to thetargetcodeandtooneanother(onsomecomputers,itmaybepossibletoleavethisrelative

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 thestorage for each name in the records, are fixed. Thus, at compile time, the addresses at which thetarget code can find the data it operates upon can be filled in. The addresses at which informationis to be saved when a procedure call takes place are also known at compile time. Static allocationdoeshavesomelimitations.

- Sizeofdataobjects,aswellasanyconstraintsontheir positionsinmemory,mustbeavailable atcompiletime.
- Norecursion, becauseallactivationsofagivenprocedureusethesamebindingsforlocalnames.
- Nodynamicdatastructures,sincenomechanismisprovidedforruntimestorageallocation.

STACK ALLOCATION: Figure shows the activation records that are pushed onto and poppedfortheruntime stackas the controlflowsthroughthe givenactivationtree.



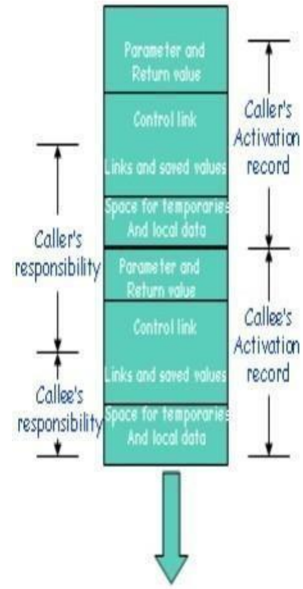
First the procedure is activated. Procedure readarray 's activation is pushed onto the stack, whenthe control reaches thefirstlinein the procedure sort.After the control returnsfrom theactivation of the readarray , its activation is popped. In the activation of sort , the control thenreaches a call of 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 endedduring the life time of qsort (1,3), so their activation records have come and gone from the stack,leavingtheactivationrecordforqsort(1,3)ontop.

CALLINGSEQUENCES:Acallsequenceallocatesanactivationrecordandentersinformation into its field. A return sequence restores the state of the machine so that callingprocedurecancontinueexecution.

Calling sequence and activation records differ, even for the same language. The code in thecallingsequenceisoftendividedbetweenthe callingprocedureandtheprocedureitcalls.

Thereisnoexactdivisionofruntimetasksbetweenthecallerandth ecolleen.

Asshownin thefigure,theregisterstack toppointstotheendofthemachine status fieldinthe activationrecord.



Thispositionisknowntothecaller,soitcanbemaderesponsible for setting up stack top before control flows to thecalledprocedure.

The code for the Callee can access its temporaries and thelocaldatausingoffsetsfromstacktop.

CallSequence: Inacallsequence,followingsequenceofoperationsisperformed.

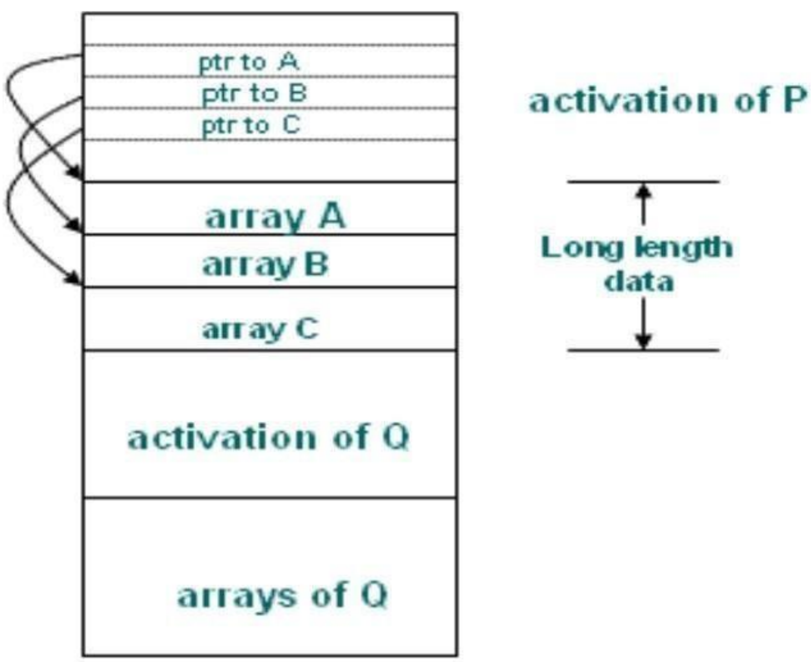
- ∑Callerevaluatesthe actualparameters
- ∑Callerstoresreturnaddressandother values(controllink)intocallee‘sactivationrecord
- ∑Calleesavesregister valuesandother statusinformation
- ∑Calleeinitializes its localdataandbeginsexecution

Thefieldswhosesizesarefixedearlyareplacedinthemiddle.Thedecisionofwhetheror not to use the control and access links is part of the design of the compiler, so these fields canbe fixed at compiler construction time. If exactly the same amount of machine-status informationissavedforeachactivation,thenthesamecodecandothesavingandrestoringforallactivatio ns. The size of temporaries may not be known to the front end. Temporaries needed bythe procedure may be reduced by careful code generation or optimization. This field is shownafterthatforthelocaldata.Thecallerusuallyevaluatestheparametersandcommunicatesthemto the activation record of the callee. In the runtime stack, the activation record of the callerisjust below that for the callee. The fields for parameters and a potential return value are placednext to the activation record of the caller. The caller can then access these fields using offsetsfrom the end of its own activation record. In particular, there is no reason for the caller to knowaboutthelocaldataortemporariesofthecallee.

ReturnSequence: Inareturnsequence,followingsequenceofoperationsareperformed.

∑Calleeplacesareturnvaluenext toactivationrecordofcaller
∑Restoresregistersusinginformation instatusfield
∑Branchtoreturnaddress
∑Callercopiesreturnvalueintoitsownactivationrecord

As described earlier, in the runtime stack, the activation record of the caller is just belowthat for the callee. The fields for parameters and a potential return value are placed next to theactivationrecordofthecaller.Thecallercanthenaccessthesefieldsusingoffsetsfromtheendof its own activation record. The caller copies the return value into its own activation record. Inparticular, there is no reason for the caller to know about the local data or temporaries of thecallee. The given calling sequence allows the number of arguments of the called procedure todepend on the call. At compile time, the target code of the caller knows the number of argumentsit is supplying to the callee. The caller knows the size of the parameter field. The target code ofthe called must be prepared to handle other calls as well, so it waits until it is called, thenexamines the parameter field. Information describing the parameters must be placed next to thestatusfieldsothe calleecanfindit.



LongLengthData:

The procedure P has three local arrays. The storage for these arrays is not part of theactivation record for P; only a pointer to the beginning of each array appears in the activationrecord. The relative addresses of these pointers are known at the compile time, so the target codecan access array elements through the pointers. Also shown is the procedure Q called by P . Theactivation record for Q begins after the arrays of P. Access to data on the stack is through twopointers,topandstacktop.Thefirstofthesemarkstheactualtopofthe stack;itpointstothe position at which the next activation record begins. The second is used to find the local data. Forconsistency with the organization of the figure in slide 16, suppose the stack top points to the endof the machine status field. In this figure the stack top points to the end of this field in theactivation record for Q. Within the field is a control link to the previous value of stack top whencontrol wasin callingactivation of P. Thecodethatrepositions top andstack topcan begenerated at compile time, using the sizes of the fields in the activation record. When q returns,the new value of top is stack top minus the length of the machine status and the parameter fieldsin Q's activation record. This length is known at the compile time, at least to the caller. Afteradjusting top,thenew value ofstacktopcanbe copiedfromthe controllinkofQ.

Dangling References: Referring to locations which have been de-allocated.voidmain()

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

int*dangle();
{
   inti=23;re
   turn&i;
}
```

Theproblemofdanglingreferencesarises,wheneverstorageisde-allocated.Adanglingreference occurs when there is a reference to storage that has been de-allocated. It is a logicalerror to use dangling references, since the value of de-allocated storage is undefined according tothe semantics of most languages. Since that storage may later be allocated to another datum,mysteriousbugscanappearinthe programs withdanglingreferences.

HEAP ALLOCATION: If a procedure wants to put a value that is to be used after its activationis over then we cannot use stack for that purpose. That is language like Pascal allows data to beallocated under program control. Also in certain language a called activation may outlive thecaller procedure. In such a case last-in-first-out queue will not work and we will require a datastructure like heap to store the activation. The last case is not true for those languages whoseactivationtrees correctlydepicttheflowofcontrolbetweenprocedures.

LimitationsofStackallocation:Itcannotbeusedif,

- Thevaluesofthelocalvariablesmustberetainedwhenanactivationends
- Acalled activationoutlivesthecaller

∑Insucha case de-allocationofactivationrecordcannotoccurin last-infirst-outfashion

∑Heap allocationgivesoutpiecesofcontiguousstorageforactivationrecords

There aretwoaspectsofdynamicallocation-:

- Runtimeallocationand de-allocationofdatastructures.
- Languages like Algol have dynamic data structures and it reserves some part of memoryforit**.**

Initializing data-structures may require allocating memory butwhere toallocate thismemory. After doing type inference we have to do storage allocation. It will allocate some chunkof bytes. But in language like LISP, it will try to give continuous chunk. The allocation incontinuous bytes may lead to problem of fragmentation i.e. you may develop hole in process ofallocation and de-allocation. Thus storage allocation of heap may lead us with many holes andfragmentedmemorywhichwillmakeithardtoallocatecontinuouschunkofmemorytorequesting program. So,we have heap mangers which manage the free space and allocation andde-allocation of memory. It would be efficient to handle small activations and activations ofpredictable size as a special case as described in the next slide. The various allocation and de-allocationtechniques usedwillbediscussedlater.

Filla requestofsize swithblockofsize s'wheres'isthe smallestsize greaterthanorequaltos

- Forlargeblocksofstorageuseheapmanager
- For largeamountofstoragecomputation maytakesometimetouseupmemorysothattimetakenbythemanagermaybe negligiblecomparedto the computationtime

As mentioned earlier, for efficiency reasons we can handle small activations and activations ofpredictablesizeasaspecialcase asfollows:

1. Foreachsizeofinterest,keepalinkedlistiffreeblocksofthatsize

2. If possible, fill a request for size s with a block of size s', where s' is the smallest size greaterthan or equal to s. When the block is eventually de-allocated, it is returned to the linked list itcamefrom.

3. For largeblocksofstorageusetheheapmanger.

Heapmangerwilldynamicallyallocatememory.Thiswillcomewitharuntimeoverhead.As heapmanager will have to take care of defragmentation and garbage collection.But since heap manger saves space otherwise we will have to fix size of activation at compiletime,runtimeoverheadisthepriceworthit.

ACCESS TO NON-LOCALNAMES:

Thescoperulesofalanguagedecidehowtoreferencethenon-localvariables.
Therearetwomethodsthatarecommonlyused:

1. StaticorLexicalscoping:Itdeterminesthedeclarationthatappliestoanamebyexaminingthe programtextalone.E.g.,Pascal,C andADA.
2. DynamicScoping:Itdeterminesthedeclarationapplicabletoanameatruntime,byconside

ringthe currentactivations.E.g.,Lisp

ORGANIZATIONFORBLOCKSTRUCTURES:

Ablockisaanysequenceofoperationsorinstructionsthatareusedto performa[sub]task.Inanyprogramminglanguage,

∑Blockscontainitsownlocaldata structure.

∑Blockscanbenestedandtheirstartingandendsaremarkedbyadelimiter.

∑Theyensurethateitherblockisindependentofotherornestedinanotherblock.Thatis,it is not possible for two blocks B1 and B2 to overlap in such a way that first block B1begins,thenB2,butB1endbeforeB2.

∑This nesting property is called block structure. The scope of a declaration in a block-structuredlanguageisgivenbythemostcloselynestedrule:

1. ThescopeofadeclarationinablockBincludesB.

2. IfanameXisnotdeclaredinablockB,thenanoccurrenceofXinBisinthescopeof a declaration of X in an enclosing block B ' such that. B ' has a declaration of X, and. B'ismore closelynestedaroundBthenanyotherblockwitha declarationofX.

Forexample,considerthefollowingcodefragment.

For the example, in the above figure, the scope of declaration of b in B0 does not include B1becausebisre-declaredinB1.Weassumethatvariablesaredeclaredbeforethefirststatementinwhichtheyare accessed.The scopeofthevariableswillbe asfollows:

DECLARATION	SCOPE
inta=0	B0notincludingB2
intb=0	B0notincludingB1
intb=1	B1notincludingB3
inta=2	B2only
intb=3	B3only

Theoutcomeoftheprintstatementwillbe, therefore:

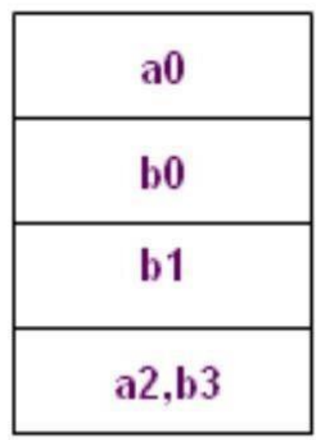
21
03
01
00

Blocks:.Blocksaresimpler tohandlethanprocedures

.Blockscanbetreatedasparameterlessprocedures

. Usestackfor memoryallocation

. Allocatespaceforcompleteprocedurebodyatonetime

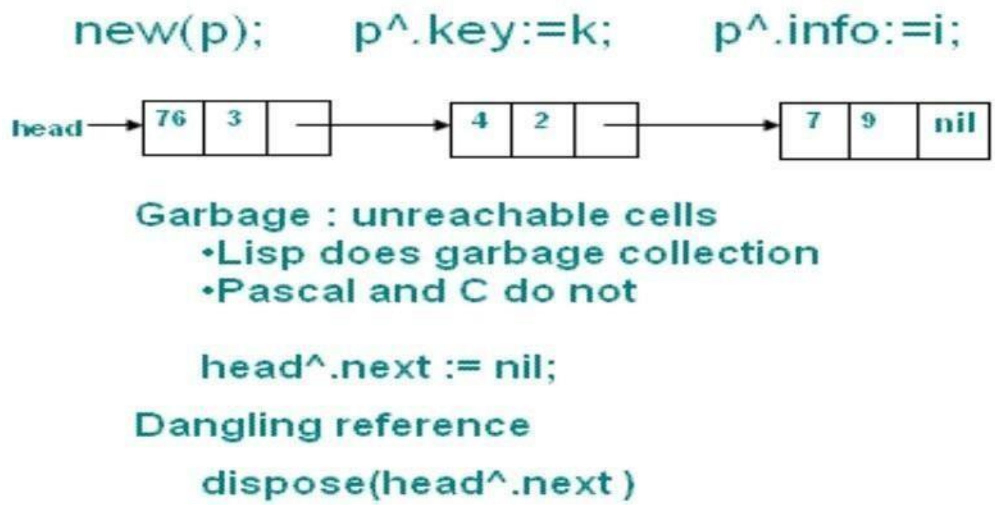


Therearetwomethodsofimplementingblockstructureincompilerconstruction:

1. **STACK ALLOCATION:** This is based on the observation that scope of a declaration doesnot extend outside the block in which it appears, the space for declared name can be allocatedwhentheblockisenteredandde-allocatedwhencontrolsleavetheblock.Theviewtreatblockas a "parameter less procedure" called only from the point just before the block and returningonlytothepointjustbeforetheblock.

2. **COMPLETE ALLOCATION:** Here you allocate the complete memory at one time. If thereare blocks within the procedure, then allowance is made for the storage needed for declarationswithin the books. If two variables are never alive at the same time and are at same depth they canbe assignedsamestorage.

DYNAMICSTORAGEALLOCATION:

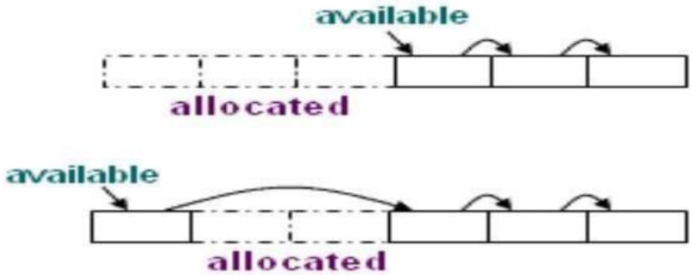


Generally languages like Lisp and ML which do not allow for explicit de-allocation of memorydo garbage collection. A reference to a pointer that is no longer valid is called a 'danglingreference'.Forexample,considerthis Ccode:

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

Here, the pointer returned by fun() no longer points to a valid address in memory as theactivation of fun() has ended. This kind of situation is called a 'dangling reference'. In case ofexplicitallocation itis morelikely tohappen as the user can de-allocate any part of memory,evensomethingthathastoapointerpointingtoavalidpiece ofmemory.

In Explicit Allocation of Fixed Sized Blocks , Link the blocks in a list , and Allocation and de-allocationcanbedone withverylittleoverhead.



The simplest form of dynamic allocation involves blocks of a fixed size. By linking the blocks ina list, as shown in the figure, allocation and de-allocation can be done quickly with little or nostorageoverhead.

Explicit Allocationof FixedSizedBlocks: Inthisapproach, blocksaredrawnfromcontiguousarea ofstorage,andanarea ofeachblockisusedaspointertothenextblock

- ∑Thepointer availablepointstothefirstblock
- ∑Allocationmeansremovingablockfromtheavailablelist
- ∑De-allocationmeansputtingtheblockintheavailablelist
- ∑Compiler routinesneednot knowthetypeofobjectstobeheldintheblocks
- ∑Eachblockistreatedasa variantrecord

Suppose thatblocks are tobe drawn from a contiguous area of storage.Initialization ofthe area is done by using a portion of each block for a link to the next block. A pointer availablepoints 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 offand allocated (added to the list of allocated nodes). When a node has to be de-allocated, it isremoved from the list of allocated nodes by changing the pointer to it in the list to point to theblock previously pointed to by it, and then the removed block is added to the head of the list offree blocks. The compiler routines thatmanage blocks do not need to know the type of objectthat will beheldin the block by the user program. These blocks can contain any type of data(i.e., they are used as generic memory locations by the compiler). We can treat each block as avariant record, with the compiler routines viewing the block as consisting of some other type.Thus, there is no space overhead because the user program can use the entire block for its ownpurposes. When the block is returned, then the compiler routines use some of the space from theblockitselftolinkitintothelistofavailableblocks,asshowninthefigureinthelastslide.

ExplicitAllocationofVariableSizeBlocks:

Limitations of Fixed sized block allocation: In explicit allocation of fixed size blocks, internalfragmentation can occur, that is, the heap may consist of alternate blocks that are free and in use,asshowninthefigure.

Thesituation shown can occur if aprogram allocates fiveblocks andthen de-allocates thesecondandthefourth,forexample.

Fragmentation is of no consequence if blocks are of fixed size, but if they are of variable size, asituation like this is a problem, because we could not allocate a block larger than any one of thefree blocks,eventhoughthespaceis availableinprinciple.

So, if variable- sized blocks are allocated, then internal fragmentation can be avoided, as we onlyallocateasmuchspaceasweneedinablock.Butthiscreatestheproblemofexternalfragmentation,w hereenoughspaceisavailablein totalforourrequirements,butnot enough space is available in continuous memory locations, as needed for a block of allocated memory.For example, consider another case where we need to allocate 400 bytes of data for the nextrequest, and the available continuous regions of memory that we have are of sizes 300, 200 and100 bytes. So we have a total of 600 bytes, which is more than what we need. But still we areunable toallocate thememoryaswedonothaveenoughcontiguousstorage.

The amount of external fragmentation while allocating variable-sized blocks can become veryhighonusingcertainstrategiesformemoryallocation.

So we try to use certain strategies for memory allocation, so that we can minimize memorywastageduetoexternalfragmentation.Thesestrategiesarediscussedinthenextfewlines.

.Storagecanbecomefragmented,Situationmayarise,Ifprogramallocatesfiveblocks
.thende-allocatessecond andfourthblock

IMPORTANTQUESTIONS:

1. Whatarecallingsequence,andReturnsequences?Explainbriefly.
2. WhatisthemaindifferencebetweenStatic&Dynamicstorageallocation?Explaintheproblemsassociatedwithdynamic storage allocationschemes.
3. Whatistheneed ofadisplayassociatedwithaprocedure?Discusstheproceduresformaintainingthe displaywhenthe proceduresarenotpassedasparameters.
4. Writenotesonthestaticstorageallocationstrategywithexampleanddiscussitslimitations?
5. Discussabout thestackallocationstrategyofruntimeenvironmentwithanexample?
6. Explaintheconceptofimplicitdeallocationofmemory.
7. Giveanexampleofcreatingdanglingreferencesandexplainhowgarbageiscreated.

ASSIGNMENTQUESTIONS:

1. Whatisacallingsequence?Explainbriefly.
2. Explaintheproblemsassociatedwithdynamicstorageallocationschemes.
3. Listand explaintheentriesofActivationRecord.
4. Explainaboutparameterpassingmechanisms.

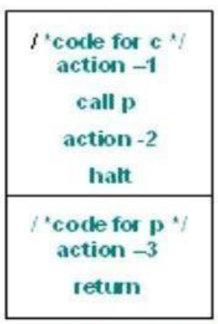
UNIT-IV

RUNTIMESTORAGEMANAGEMENT:

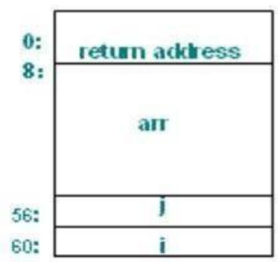
To study the run-time storage management system it is sufficient to focus on the statements:action,call,return and halt,because they by themselves give us sufficientinsightinto thebehaviorshownbyfunctionsincallingeachotherandreturning.

And the run-time allocation and de-allocation of activations occur on the call of functions andwhentheyreturn.

Therearemainlytwokindsofrun-timeallocationsystems:**Staticallocation**and**StackAllocation**.
While static allocation is used by the FORTRAN class of languages, stack allocationisusedbytheAdaclass oflanguages.



Three address code



Activation record for c (64 bytes)

Activation record for p (88 bytes)

STATICALLOCATION: In this, A call statement is implemented by a sequence of twoinstructions.

- ∑Amoveinstructionsavesthereturn address
- ∑Agototransfers controltothe targetcode.

TheinstructionsequenceisMOV# here+20,callee.static-areaGOTO callee.code-area
callee.static-areaandcallee.code-areaareconstantsreferringtoaddressoftheactivationrecordandthefirstaddressofcalledprocedurerespectively.

. #here+20inthemoveinstructionisthereturnaddress;theaddressoftheinstructionfollowingthe gotoinstruction

. A return from procedure callee is implemented byGOTO *callee.static-area

For the call statement, we need tosave the return address somewhere andthenjumptothe location of the callee function. And to return from a function, we have to access the returnaddress as stored by its caller, and then jump to it. So for call, we first say: MOV #here+20,callee.static-area.Here,#herereferstothelocationofthecurrentMOVinstruction,andcallee.static-areaisafixedlocationinmemory.20isaddedto#herehere,asthecodecorresponding to the call instruction takes 20 bytes (at 4 bytes for each parameter: 4*3 for thisinstruction,and8forthenext).ThenwesayGOTOcallee.code-area,totakeustothecodeofthe callee, as callee.codearea is merely the address where the code of the callee starts. Then areturn from the callee is implemented by: GOTO *callee.static area. Note that this works onlybecause callee.static-areais aconstant.

Example:

.Assumeeach	100:ACTION-l
action	120:MOV140,364
blocktakes20	132:GOTO200
bytes ofspace	140:ACTION-2
.Startaddress	160:HALT
ofcodeforc	:
andpis	200: ACTION-3
100 and200	220:GOTO*364

.Theactivation	:
Records	300:
arestatically	304:
allocatedstarting	:
at addresses	364:
300 and 364.	368:

This example corresponds to the code shown in slide 57. Statically we say that the codefor c starts at 100 and that for p starts at 200. At some point, c calls p. Using the strategydiscussed earlier, and assuming that callee.staticarea is at the memory location 364, we get thecode as given. Here we assume that a call to 'action' corresponds to a single machine instructionwhichtakes 20bytes.

<u>STACKALLOCATION :</u>.Positionoftheactivationrecordisnotknownuntilruntime

- ∑.Position isstored inaregister atruntime,and wordsintherecord areaccessed withanoffsetfromtheregister
- ∑. The code for the first procedure initializes the stack by setting up SP to the start of thestackarea

MOV #Stackstart,SP

code for the first
procedureHALT

In stack allocation we do not need to know the position of the activation record until run-time. This gives us an advantage over static allocation, as we can have recursion. So this is usedin many modern programming languages like C, Ada, etc. The positions of the activations arestored in the stack area, and the position for the most recent activation is pointed to by the stackpointer. Words in a record are accessed with an offset from the register. The code for the firstprocedure initializes the stack by setting up SP to the stack area by the following command:MOV#Stackstart,SP.Here,#Stackstartisthelocation in memorywherethestack starts.

AprocedurecallsequenceincrementsSP,savesthereturnaddressand transferscontrolto thecalledprocedure

ADD #caller.recordsize,
SPMOVE#here+
16,*SPGOTOcallee.code_a
rea

Consider the situation when a function (caller) calls the another function(callee), thenprocedure call sequence increments SP by the caller record size, saves the return address andtransfers control to the callee by jumping to its code area. In the MOV instruction here, we onlyneed to add 16, as SP is a register, and so no space is needed to store *SP. The activations keepgetting pushed on the stack, so #caller.recordsize needs to be added to SP, to update the value ofSP to its new value.This works as#caller.recordsizeis a constant for a function,regardless ofthe particularactivationbeingreferredto.

DATASTRUCTURES: Followingdatastructuresareusedtoimplementsymboltables

LISTDATASTRUCTURE :Couldbeanarraybasedorpointerbasedlist.Butthisimplementationis

- Simplesttoimplement
- Useasinglearraytostorenamesandinformation
- Searchforanameislinear
- Entryandlookupareindependentoperations
- Costofentryandsearchoperationsareveryhighandlotoftimegoesintobookkeeping

Hashtable: HashtableisadatastructurewhichgivesO(1)performanceinaccessinganyelementofit. Itusesthe featuresofbotharrayandpointerbasedlists.

- Theadvantagesareobvious

REPRESENTINGSCOPEINFORMATION

The entriesin thesymbol table are fordeclaration of names.When an occurrence of a nameinthe source textislooked up in the symbol table,the entry for the appropriate declaration,according to the scoping rules of thelanguage,mustbe returned. A simple approach is tomaintainaseparatesymboltableforeachscope.

Mostcloselynestedscoperulescanbeimplementedbyadaptingthedatastructuresdiscussed in the previous section. Each procedure is assigned a unique number. If the language isblock-structured, the blocks must also be assigned unique numbers. The name is represented as apair of a number and a name. This new name is added to the symbol table. Most scope rules canbeimplementedinterms offollowingoperations:

a) Lookup-findthemostrecentlycreatedentry.
b) Insert-makeanewentry.
c) Delete- removethemostrecentlycreatedentry.
d) Symboltable structure
e) .Assignvariablesto storageclassesthatprescribescope,visibility,andlifetime

f) -scoperulesprescribe the symboltablestructure
g) -scope:unitofstaticprogramstructurewithoneor morevariabledeclarations
h) -scopemaybenested
i) .Pascal:proceduresarescopingunits
j) .C:blocks,functions,filesarescopingunits
k) .Visibility, lifetimes,globalvariables
l) .Common(inFortran)
m) .Automaticor stackstorage
n) .Staticvariables
o) **storage class :**A storage class is an extra keyword at the beginning of a declarationwhich modifies the declaration in some way. Generally, the storage class (if any) is thefirstwordinthe declaration,precedingthetype name.Ex.static,externetc.
p) Scope: The scope of a variableis simply the part of the program whereitmay beaccessedorwritten.Itisthepartoftheprogramwherethevariable'snamemaybeused.If a variable is declared within a function, itis local to thatfunction. Variables of thesame name may be declared and used within other functions without any conflicts. Forinstance,
q) intfun1()

```
{
   int
   a;int
   b;
   ....
}

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

Visibility: The visibility of a variable determines how much of the rest of the programcanaccessthatvariable.You canarrangethatavariableisvisibleonlywithinonepartofonefunction,orinonefunction,orinon esourcefile,oranywhereintheprogram.

r) **Local and Global variables:** A variable declared within the braces {} of a function isvisible only within that function; variables declared within functions are called localvariables. On the other hand, a variable declared outside of any function is a globalvariable,anditispotentiallyvisible anywhere withintheprogram.

s) **Automatic Vs Static duration:** How long do variables last? By default, local variables(those declared within a function) have automatic duration: they spring into existencewhenthefunctioniscalled,andthey(andtheirvalues)disappearwhenthefunction returns. Global variables, on the other hand, have static duration: they last, and the valuesstored in them persist, for as long as the program does. (Of course, the values can ingeneral still be overwritten, so they don't necessarily persist forever.) By default, localvariableshaveautomaticduration.Togivethemstaticduration (sothat,insteadofcoming and going as the function is called, they persist for as long as the function does),youprecedetheirdeclarationwiththestatickeyword:staticinti;Bydefault,adeclaration of a global variable (especially if it specifies an initial value) is the defininginstance. To make it an external declaration, of a variable which is defined somewhereelse, you precede it with the keyword extern: extern int j; Finally, to arrange that a globalvariable is visible only within its containing source file, you precede it with the statickeyword: static int k; Notice that the static keyword can do two different things: it adjuststhe duration of a local variable from automatic to static, or it adjusts the visibility of aglobalvariablefromtrulyglobaltoprivate-to-the-file.

t) Symbolattributesand symboltableentries
u) Symbolshaveassociated attributes
v) Typicalattributesarename, type,scope,size,addressingmodeetc.
w) Asymboltable entrycollectstogetherattributes suchthattheycanbe easilysetandretrieved
x) Exampleoftypical namesinsymboltable

Name	Type
name	characterstring
class	enumeration
size	integer
type	enumeration

LOCALSYMBOLTABLEMANAGEMENT:

Followingareprototypesoftypicalfunctiondeclarationsusedformanaginglocalsymboltable.Theright handsideofthearrowsistheoutputoftheprocedureandtheleftsidehastheinput.

NewSymTab:SymTab →

→SymTabDestSy

mTab:SymTab →

SymTab→InsertS

ym:SymTabXSymbol

booleanLocateS

ym:SymTabXSymbol boolean

GetSymAttr:SymTabXSymbolXAttr →

booleanSetSym→Attr

:SymTabXSymbolXAttrXvalue→

booleanNextSym:SymTa
bXSymbol Symbol
MoreSyms:SymTabXSymbol → boolean

Amajorconsiderationindesigningasymboltableisthatinsertionandretrievalshouldbeas fastaspossible

. Onedimensionaltable:searchisveryslow

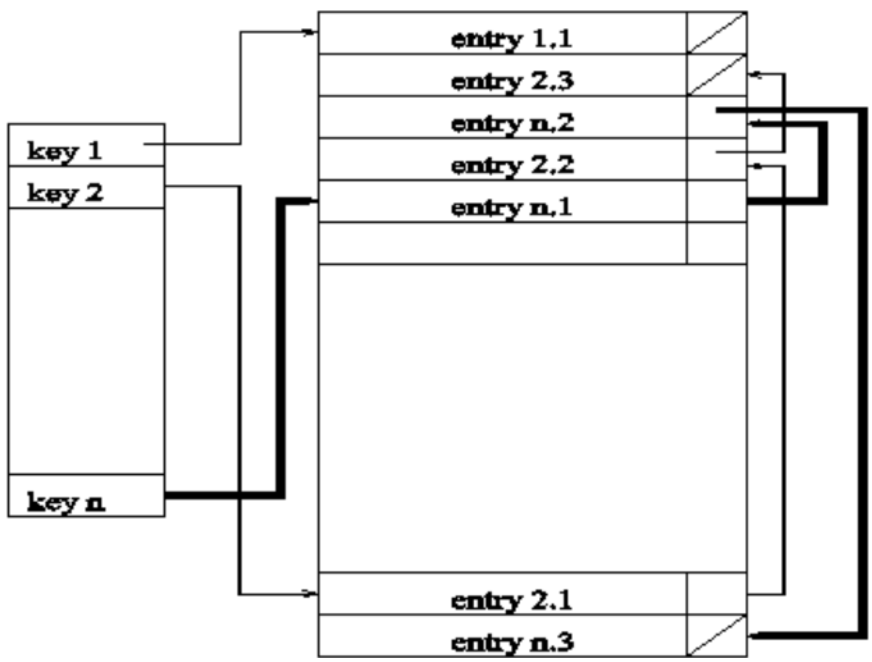
.Balancedbinarytree:quickinsertion, searchingandretrieval;extraworkrequiredtokeepthetreebalanced

. Hashtables:quickinsertion,searchingandretrieval;extraworktocomputehashkeys

.Hashingwitha chainofentriesisgenerallya goodapproach

A major consideration in designing a symbol table is that insertion and retrieval should beas fast as possible. We talked about the one dimensional and hash tables a few slides back. Apartfromthesebalanced binarytreescanbeused too.Hashingisthemostcommonapproach.

<u>HASHEDLOCALSYMBOLTABLE</u>



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 bedeleted.Eachentryshouldhavetwolinks:

a) A hash link that chains the entry to other entries whose names hash to the same value - theusuallinkinthehash table.

b) A scope link that chains all entries in the same scope - an extra link. If the scope link is leftundisturbed when an entry is deleted from the hash table, then the chain formed by the scopelinkswillconstitute aninactive symboltableforthescope inquestion.

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.
```

Look at the nesting structure of this program.Variables a,b and c appearin global aswell as local scopes. Local scope of a variable overrides the global scope of the other variablewith the 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 nestedwithing).

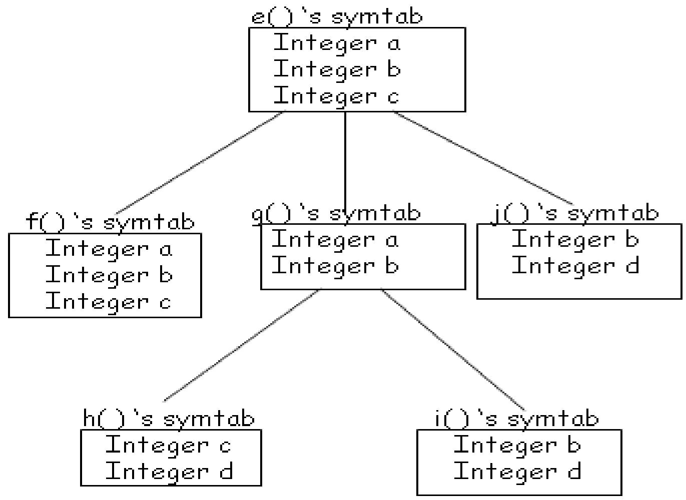
<u>GLOBALSYMBOLTABLESTRUCTURE</u>Theglobalsymboltablewillbeacollectionofsymboltables connectedwithpointers.

.Scopeandvisibilityrulesdeterminethestructureofglobalsymboltable

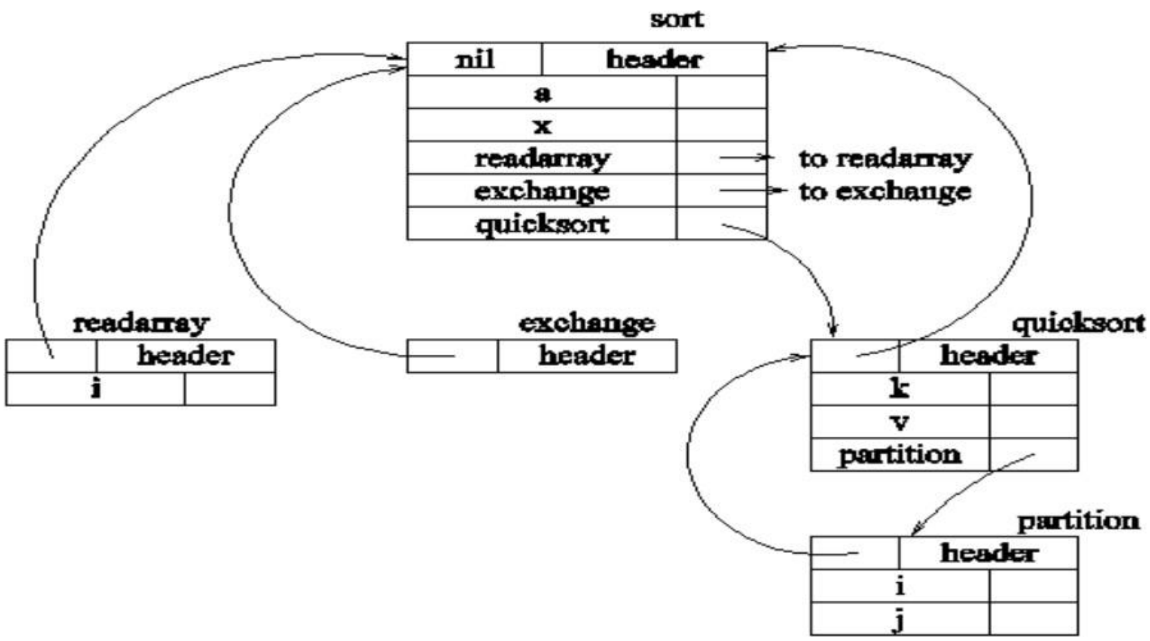
.ForALGOLclassoflanguages scopingrulesstructure the symbol table astreeoflocaltables

- Globalscopeasroot

- Tables for nested scope aschildren of the tableforthescopetheyare nestedin



Theexactstructurewillbedeterminedbythescopeandvisibilityrulesofthelanguage.The global symbol table will be a collection of symbol tables connected with pointers.Theexactstructurewillbedeterminedbythescopeandvisibilityrulesofthelanguage.Wheneve r a new scope is encountered a new symbol table is created. This new table contains apointerbacktotheenclosingscope'ssymboltableandtheenclosingonealsocontainsapointerto this new symbol table. Any variable used inside the new scope should either be present in itsown symbol table or inside the enclosing scope's symbol table and all the way up to the rootsymboltable.Asample globalsymboltableisshown inthebelowfigure.



BLOCKSTRUCTURESANDNONBLOCKSTRUCTURESTORAGEALLOCATION

Storagebindingandsymbolicregisters: Translatesvariablenamesintoaddressesand theprocessmustoccurbeforeorduringcodegeneration

- . Eachvariableisassignedanaddressor addressing method
- . Each variable is assigned an offset with respect to base which changes with everyinvocation
- .Variables fallinfourclasses:global,globalstatic, stack, local(non-stack)static
- Thevariablenameshavetobetranslatedintoaddressesbeforeorduringcodegeneration.

There is a base address and every name is given an offset with respect to this base which changeswitheveryinvocation.The variables canbe dividedintofourcategories**:**

a) GlobalVariables:fixedrelocatableaddressoroffsetwithrespecttobaseasglobalpointer

b) Global Static Variables :.Global variables, on the other hand, have static duration (hencealso called static variables): they last, and the values stored in them persist, for as long as theprogram does. (Of course, the values can in general still be overwritten, so they don't necessarilypersist forever.) Therefore they have fixed relocatable address or offset with respect to base asglobalpointer.

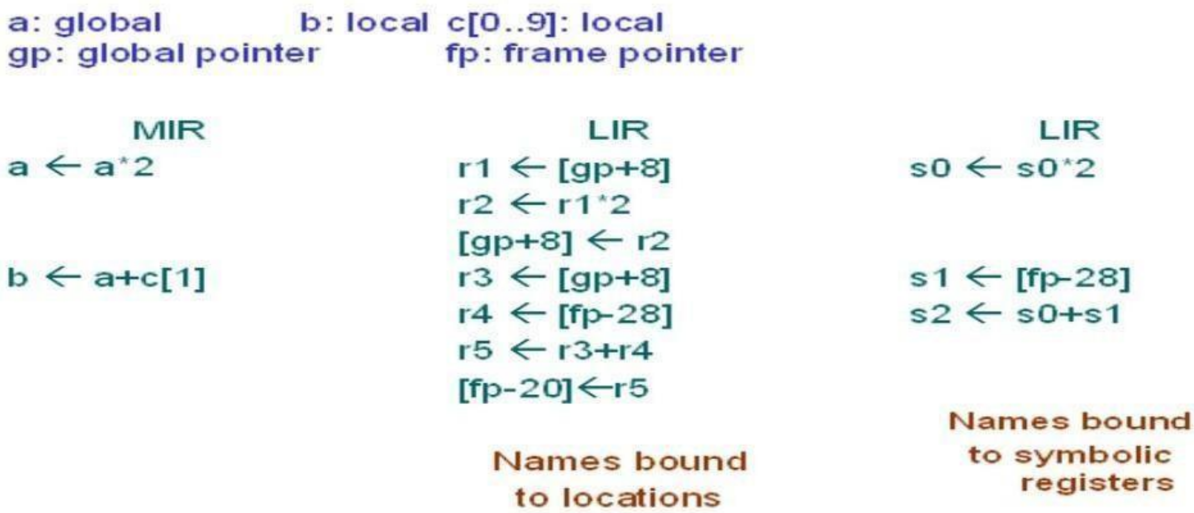
c) Stack Variables :allocate stack/global in registers and registers are not indexable, therefore,arrayscannotbein registers

.Assignsymbolicregisterstoscalarvariables

.Usedforgraphcoloringforglobalregisterallocation

d) Stack Static Variables : By default, local variables (stack variables) (those declared within afunction) have automatic duration: they spring into existence when the function is called, andthey (and their values) disappear when the function returns. This is why they are stored in stacksandhaveoffsetfromstack/framepointer.

Registerallocationisusuallydoneforglobalvariables.Sinceregistersarenotindexable,therefore, arrays cannot be in registers as they are indexed data structures. Graph coloring is asimple technique for allocating register and minimizing register spills that works well in practice.Register spills occur when a register is needed for a computation but all available registers are inuse. The contents of one of the registers mustbe stored in memory to free itup for immediateuse.Weassignsymbolicregisterstoscalarvariableswhichareusedinthegraphcoloring.



LocalVariables inFrame

- ∑Assigntoconsecutivelocations;allowenoughspaceforeach
- ∑Mayputword sizeobjectinhalfwordboundaries
- ∑Requirestwohalfwordloads
- ∑Requiresshift,or, and
- ∑Alignondouble wordboundaries
- ∑Wastesspace
- ∑AndMachinemayallowsmalloffsets

wordboundaries-
themostsignificantbyteoftheobjectmustbelocatedatanaddresswhosetwoleastsignificantbitsare zerorelative tothe frame pointer

half-wordboundaries-
themostsignificantbyteoftheobjectbeinglocatedatanaddresswhoseleastsignificantbitiszerorelative totheframe pointer**.**

Sortvariablesbythealignment theyneed

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

Whileallocatingmemorytothevariables,sortvariablesbythealignmenttheyneed.Youmay:

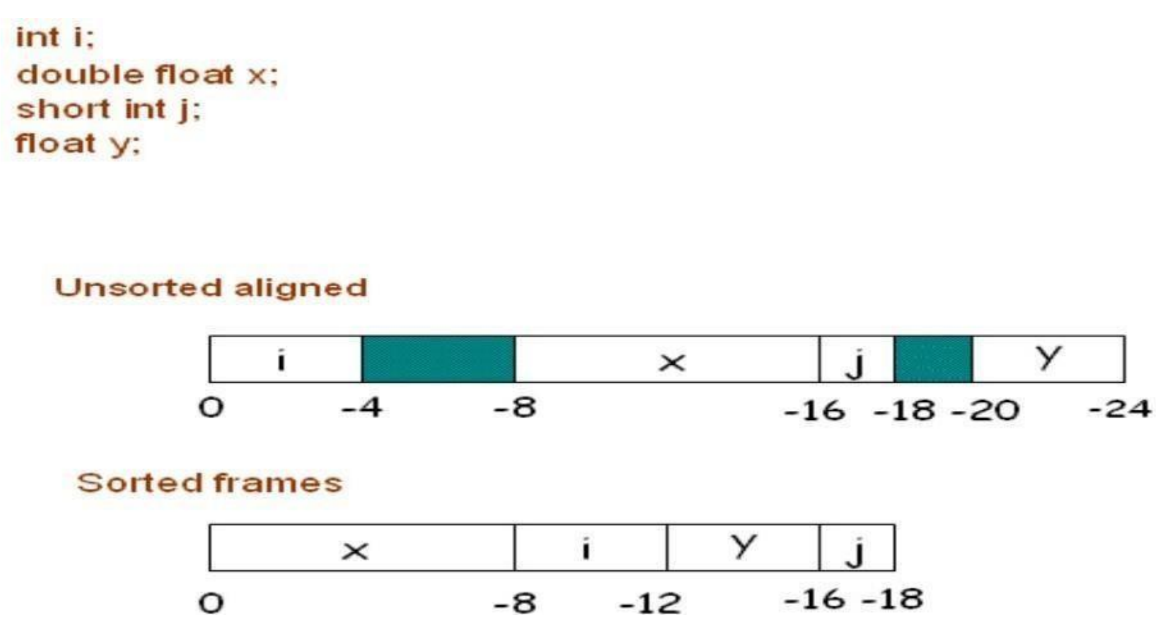
Storelargestvariablesfirst:Itautomaticallyalignsallthevariablesanddoesnotrequirepaddingsincethen extvariable'smemoryallocationstartsatthe end ofthatoftheearliervariable

. Store smallest variables first: It requires more space (padding) since you have to accommodatefor the biggest possible length of any variable data structure. The advantage is that for large stackframe,more variablesbecome accessible withinsmalloffsets

How to store large local data structures? Because they Requires large space in local framesandthereforelargeoffsets

- Iflargeobjectisputneartheboundaryotherobjectsrequirelargeoffseteitherfromfp(ifputnearbeginning)orsp(ifputnearend)
- Allocateanother baseregistertoaccesslargeobjects
- Allocatespaceinthemiddleorelsewhere;storepointertotheselocations fromatasmalloffsetfromfp
- Requiresextraloads

Large local data structures require large space in local frames and therefore large offsets.As told in the previous slide's notes, if large objects are put near the boundary then the otherobjectsrequirelargeoffset.Youcaneitherallocateanotherbaseregistertoaccesslargeobjectsor you can allocate space in the middle or elsewhere and then store pointers to these locationsstartingfromatasmalloffsetfromtheframe pointer,fp.



Intheunsortedallocation youcanseethewasteofspaceingreen.Insortedframethereisnowasteofspace.

STORAGEALLOCATIONFORARRAYS

Elements of an array are stored in a block of consecutive locations. For a single dimensionalarray, if low is the lower bound of the index and base is the relative address of the storageallocated to the array i.e., the relative address of A[low], then the ith Elements of an array arestoredinablockofconsecutivelocations

For a single dimensional array, if low is the lower bound of the index and base is therelative address of the storage allocated to the array i.e., the relative address of A[low], then the ith elements begins atthe location:*base + (I - low)*w* . This expression can be reorganized as*i*w + (base -low*w)* . The sub-expression *base-low*w* is calculated and stored in the symboltable at compile time when the array declaration is processed, so that the relative address of A[i]canbeobtainedbyjustadding*i*w*toit.

- AddressingArrayElements
- Arraysarestoredinablockofconsecutivelocations
- Assumewidthofeachelementisw
- ithelementofarrayAbeginsinlocation base+(i-low) xwwherebaseisrelativeaddressofA[low]
- Theexpressionisequivalentto
- ixw+(base-
  lowxw)ixw+const

2-DIMENSIONAL ARRAY:For a row majortwodimensional array the address of A[i][j]canbecalculatedbytheformula :

base + ((i-low_i)*n2 +j - low_j)*w where low $_i$and $low_j$are lower values of I and j and n2 isnumberofvaluesjcantakei.e.n2=high2-low2+ 1.

Thiscanagainbe written as:

((i * n2) + j) *w + (base - ((low_i*n2) + low_j) * w) and the second term can be calculated atcompile time.

Inthesamemanner,theexpressionforthelocationof anelementincolumnmajortwo-dimensional array can be obtained.This addressing can be generalized to multidimensionalarrays.**S**torage canbe eitherrowmajororcolumnmajorapproach.

Example: Let A be a 10x20 array therefore, n 1 = 10 and n 2 = 20 and assume w =

4The Three addresscodetoaccessA[y,z]is

t_1 = y *

20t_1 = t_1 +

zt_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

Let A be a 10x20

arrayn1= 10andn2= 20

Assumewidthofthetypestoredinthearrayis4.ThethreeaddresscodetoaccessA[y,z]ist1=y*20

t1 = t1 +

zt2=4*t1

t3=baseA -

84{((low1*n2)+low2)*w)=(1*20+1)*4=84}t4=t2+t3

x=t4

Thefollowingoperationsaredesigned:1. mktable(previous):createsanewsymboltableandreturnsa pointertothistable.Previousispointertothe symboltableofparentprocedure.

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

3. addwidth(table,width):recordscumulativewidthofentriesofatablein its header.

4. enterproc(table,name,newtable):createsanentryforprocedure*name* inthe symboltablepointedtoby*table*.*newtable*isapointer tosymboltablefor*name*.

P $\longrightarrow$

{t=mktable
(nil);push(t,tblp
tr);push(0,offset
)}

D

{addwidth(top(tblptr),top(offset));pop(t
blptr);
pop(offset)}

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

The symbol tables are created using two stacks: *tblptr*to hold pointers to symbol tables ofthe enclosing procedures and *offset* whose top element is the next available relative address for alocal of the current procedure. Declarations in nested procedures can be processed by the syntaxdirected definitions given below. Note that they are basically same as those given above but wehaveseparatelydealtwiththe epsilonproductions.Gotothe nextpage fortheexplanation.

```
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 → procid;

{t=
mktable(top(tblptr));push(t,tbl
ptr);push(0,offset)}

D 1;S

{ t =
top(tblptr);addwidth(t,
top(offset));pop(tblptr);
pop(offset);;
enterproc(top(tblptr),id.name,t)}

Did:T

{enter(top(tblptr),id.name,T.type,top(offset));to
p(offset)=top(offset)+T.width}

The action for M creates a symbol table for the outermost scope and hence a nil pointer is passedinplaceof previous.When thedeclaration,Dprocid; ND1; Sis processed,theactioncorresponding to N causes the creation of a symbol table for the procedure; the pointer to symboltable of enclosing procedure is given by top(tblptr). The pointer to the new table is pushed on tothe stack tblptrand0 is pushedas the initial offseton the offsetstack. When the actionscorresponding to the subtrees of N, D1 and S have been executed, the offset corresponding to thecurrent procedure i.e., top(offset) contains the total width of entries in it. Hence top(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 thesestacks. The entry for id is added to the symbol table of the enclosing procedure. When thedeclarationD->id:Tisprocessedentryforidiscreatedinthesymbol tableof currentprocedure.Pointertothesymboltableofcurrentprocedureisagainobtainedfromtop(tblptr).

Offsetcorrespondingtothecurrentprocedurei.e.top(offset)is incrementedbythewidthrequiredbytype Ttopointtothe nextavailablelocation.

STORAGEALLOCATIONFORRECORDS

FieldnamesinrecordsT

→ record
{t=mktable(nil);

push(t,tblptr);push(0,offset)}D
end
{T.type =
record(top(tblptr));T.width =
top(offset);pop(tblptr);pop(of
fset)}

T->recordLD end {t=mktable(nil);
push(t,tblptr);push(0,offset)
}

L->
{T.type=record(top(tbl
ptr));T.width =
top(offset);pop(tblptr);pop(of
fset)
}

Theprocessingdonecorrespondingtorecordsissimilartothatdoneforprocedures.After the keyword record is seen the marker L creates a new symbol table. Pointer to this tableand offset0 are pushed on the respective stacks. The action for the declaration D->id :T pushthe information about the field names on the table created. At the end the top of the offset stackcontains the total width of the data objects within the record. This is stored in the attributeT.width.Theconstructor*record*isappliedtothepointertothesymboltabletoobtainT.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 thisoccurrence of the name in the symbol table. If an entry is found, pointer to the entry is returnedelse nil is returned. Look up first checks whether the name appears in the current symbol table. Ifnot then it looks for the name in the symbol table of the enclosing procedure and so on. Thepointer to the symbol table of the enclosing procedure is obtained from the header of the symboltable.

CODEOPTIMIZATION

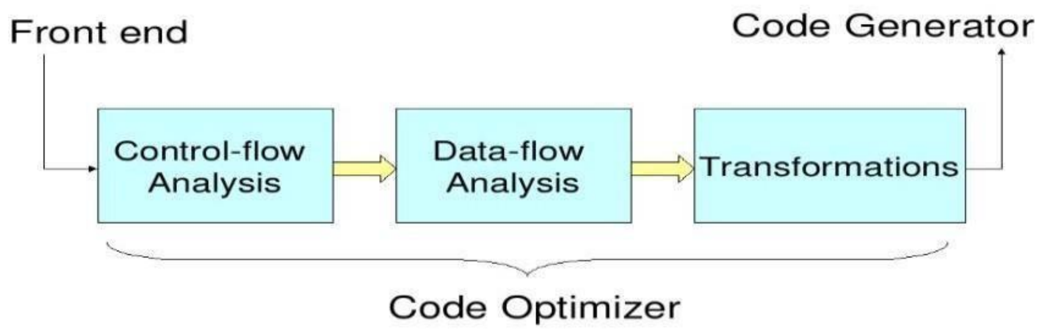
Considerations for optimization : The code produced by the straight forward compilingalgorithms can often be made to run faster or take less space,or both. This improvement isachievedbyprogramtransformationsthataretraditionallycalledoptimizations.Machineindependent optimizations are program transformations that improve the target code withouttaking into consideration any properties of the target machine. Machine dependant optimizationsare basedonregisterallocationandutilizationofspecialmachine-instructionsequences.

Criteriaforcodeimprovementtransformations

- Simply stated, the best program transformations are those that yield the most benefit forthe leasteffort.
- First,thetransformationmustpreservethemeaningofprograms.Thatis,theoptimization must not change the output produced by a program for a given input, orcauseanerror.
- Second, a transformation must, on the average, speed up programs by a measurableamount.
- Third, thetransformationmustbeworththeeffort.

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

Optimizing Compiler: Organization



OBJECTIVESOFOPTIMIZATION: Themainobjectivesoftheoptimizationtechniquesareasfollows

1. Exploitthefastpathincaseof multiplepaths froagivensituation.
2. Reduceredundant instructions.
3. Produceminimumcodeformaximumwork.
4. Tradeoffbetweenthe sizeofthecode and the speed withwhich itgetsexecuted.
5. Placecodeanddatatogetherwheneveritisrequiredtoavoidunnecessarysearchingofdata/code

Duringcodetransformationintheprocessofoptimization, thebasicrequirementsareasfollows:

1. Retainthe semanticsofthe source code.
2. Reducetimeand/orspace.
3. Reducetheoverheadinvolvedintheoptimizationprocess.

ScopeofOptimization:Control-FlowAnalysis

Consider all that has happened up to this point in the compiling process—lexicalanalysis,syntactic analysis,semantic analysis andfinally intermediate-code generation.Thecompiler has done an enormous amount of analysis, but it still doesn‘t really know how theprogramdoeswhatitdoes.Incontrol-flowanalysis,thecompilerfiguresoutevenmoreinformation about how the program does its work, only now it can assume that there are nosyntactic orsemanticerrors inthecode.

Control-flowanalysisbeginsbyconstructingacontrol-flowgraph,whichisagraphofthe different possible paths program flow could take through a function. To build the graph, wefirstdivide the code into basic blocks. Abasic block is a segmentof the code that a programmust enter at the beginning and exit only at the end. This means that only the first statement canbe reached from outside the block (there are no branches into the middle of the block) and allstatements are executed consecutively after the first one is (no branches or halts until the exit).Thus a basic block has exactly one entry point and one exit point. If a program executes the firstinstructioninabasicblock,itmustexecuteeveryinstructionintheblocksequentiallyafterit.

Abasicblockbeginsinoneofseveralways:

• Theentrypointintothefunction

- Thetarget ofabranch(inour example,anylabel)
- Theinstructionimmediatelyfollowingabranchorareturn

Abasicblockendsinanyofthefollowingways:

- Ajumpstatement
- Aconditionalorunconditional branch
- Areturnstatement

Now we can construct the control-flow graph between the blocks. Each basic block is anode in the graph, and the possible different routes a program might take are the connections, i.e.if a block ends with a branch, there will be a path leading from that block to the branch target.The blocks that can follow a block are called its successors. There may be multiple successors orjust one. Similarly the block may have many, one, or no predecessors. Connect up the flow graphfor Fibonacci basic blocks given above. Whatdoes an if then-elselook likein a flow graph?Whataboutaloop?Youprobablyhaveallseenthegccwarningorjavacerrorabout:"Unreachablecode atline XXX."How canthe compilertellwhencode isunreachable?

LOCALOPTIMIZATIONS

Optimizationsperformedexclusivelywithinabasicblockarecalled"localoptimizations". These are typically the easiest to perform since we do not consider any controlflowinformation;wejustworkwiththestatementswithintheblock.Manyofthelocaloptimizations we will discuss have corresponding global optimizations that operate on the sameprinciple, but require additional analysis to perform. We'll consider some of the more commonlocaloptimizations as examples.

FUNCTIONPRESERVINGTRANSFORMATIONS

∑Commonsubexpressionelimination

∑Constantfolding

∑Variablepropagation

∑Dead CodeElimination

∑Code motion

∑StrengthReduction

1. <u>CommonSubExpressionElimination:</u>

Two operations are common if they produce the same result. In such a case, it is likely moreefficienttocomputetheresultonceandreferenceitthesecondtimeratherthanre-evaluateit. An expressionisaliveiftheoperandsusedtocomputetheexpressionhavenotbeenchanged.Anexpressio nthatisnolongeraliveis dead.

Example:

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

Wecaneliminatethesecondevaluationofb*cfromthiscodeifnoneoftheinterveningstatementshas changeditsvalue.We canthusrewritethe codeas

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

Letusconsiderthefollowingcodea=
b*c;
b=x;d=b*c
+ x-y;

inthiscode,wecannoteliminatethesecondevaluationofb*cbecausethevalueofbischangeddue tothe assignmentb=xbeforeitisusedincalculatingd.

Wecansaythetwoexpressionsarecommonif

- Theylexicallyequivalenti.e.,theyconsistofidenticaloperands connectedtoeachotherbyidenticaloperator.
- Theyevaluatetheidenticalvalues i.e.,noassignmentstatements for anyoftheiroperandsexistbetweenthe evaluations oftheseexpressions.
- Thevalueofanyoftheoperandsuseintheexpressionshouldnot be changed evenduetothe procedurecall.

Example:

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

Wemaynotethat eventhoughexpressionsa*band x*barecommoninthe abovecode,theycannotbetreatedas commonsubexpressions.

2. **VariablePropagation:**

Letusconsider theabovecodeonceagain

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 subexpressions. This technique is calledvariable propagation where the use of one variable isreplacedbyanothervariableifithasbeenassignedthe value ofsame

CompileTimeevaluation

The execution efficiency of the program can be improved by shifting execution timeactions to compile time so that they are not performed repeatedly during the program execution.Wecanevaluateanexpressionwithconstantsoperandsatcompiletimeandreplacethatexpressionbyasinglevalue.Thisiscalledfolding.Considerthefollowing statement:

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

Here, wecanperformthecomputation2*(22.0/7.0)atcompiletimeitself.

3. <u>DeadCodeElimination:</u>

If the value contained in the variable at a point is not used anywhere in the programsubsequently, the variable is said to be dead at that place. If an assignment is made to a deadvariable, then that assignment is a dead assignment and itcan be safely removed from theprogram.

Similarly,
apieceofcodeissaidtobedead,whichcomputesvaluethatareneverusedanywhereintheprogram.

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

Usingvariablepropagation,thecodecanbewrittenas
follows:c=a*b;

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

UsingCommonSubexpressionelimination,thecodecanbewrittenasfollows:

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

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

4. <u>CodeMovement:</u>

The motivation for performing code movement in a program is to improve the execution time ofthe program by reducing the evaluation frequency of expressions. This can be done by movingtheevaluationofanexpressionto otherpartsofthe program.Letusconsiderthebellow code:

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

At thetimeofexecutionoftheconditiona<10,x^2-y^2isevaluatedtwice.So,wecanoptimizethe codebymovingtheoutside totheblockas follows:

```
t=x^2-
y^2;If(a<1
0)
{
b=t;
}
else
{b=
5;
a=t*10;
}
```

5. **StrengthReduction:**

In the frequency reduction transformation we tried to reduce the execution frequency ofthe expressions by moving the code. There is other class of transformations which performequivalent actions indicated in the source program by reducing the strength of operators. Bystrength reduction, we mean replacing the high strength operator with low strength operator withoutaffectingthe programmeaning.Letusconsiderthe bellow example:

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

Theabovecanwrittenasfollows:i=

```
1;
t=4;
```

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

Herethehighstrengthoperator*isreplaced with +.

GLOBALOPTIMIZATIONS,DATA-FLOWANALYSIS:

So far we were only considering making changes within one basic block. With someAdditional analysis, we can apply similar optimizations across basic blocks, making them globaloptimizations. It‛s worth pointing out that global in this case does not mean across the entireprogram. We usually optimize only one function at a time. Inter procedural analysis is an evenlargertask,one notevenattemptedbysomecompilers.

The additional analysis the optimizer does to perform optimizations across basic blocks iscalled**data-flowanalysis**.Data-flowanalysisismuchmorecomplicated than control-flowanalysis,andwe canonlyscratchthesurface 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 that point**. Once the set of available expressions isknown, common sub-expressions can be eliminated on a global basis. Each blockis anodeinthe flow graph of a program. The **successor** set (succ(x)) for a node x is the set of all nodes that xdirectly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flowdirectly into x. An expression is defined at the point where it is assigned a value and killed whenone of its operands is subsequently assigned a new value. An expression is available at somepoint p in a flow graph if every path leading to p contains a prior definition of that expressionwhich is not subsequently killed. Lets define such useful functions in DF analysis in followinglines.

avail[B]=setofexpressionsavailableonentryto block B

exit[B]=setofexpressionsavailableonexitfromB

avail[B] =∩**exit[x]: x∈pred[B]**(i.e.Bhasavailabletheintersectionoftheexitofitspredecessors)

killed[B] = set of the expressions killed in

B**defined[B]** = set of expressions defined in

B**exit[B]** =**avail[B]-killed[B]**+**defined[B]**

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

Hereisan**AlgorithmforGlobalCommon Sub-expression Elimination**:

1) First, computedefinedandkilledsetsfor eachbasicblock(thisdoesnotinvolveanyofitspredecessorsorsuccessors).
2) Iterativelycomputetheavailandexitsetsfor eachblockbyrunningthefollowingalgorithmuntilyouhitastablefixedpoint:
   a) Identifyeachstatement**s**ofthe form**a =bopc**insome block Bsuchthat**bopc**isavailableattheentrytoBand neither**b**nor**c**isredefinedinBpriorto**s**.
   b) Followflowofcontrolbackwardinthegraphpassingback tobut notthrougheachblockthatdefines**bopc**. Thelastcomputationof**bopc**insuchablockreaches**s**.
   c) After each computation **d = b op c** identified in step 2a, add statement **t = d** to thatblockwheretisanewtemp.
   d) Replace **s**by**a=t**.

Tryanexampletomakethingsclearer:mai

```
n:
   BeginFunc28;
         b = a + 2
         ;c=4*b;
         tmp1 = b< c;
         ifNZ tmp1 goto L1
         ;b=1;
         L1:
         d = a + 2
;EndFunc ;
```

First, divide the code above into basic blocks. Now calculate the available expressions for eachblock. Then find an expression available in a block and perform step 2c above. What commonsub-expressioncanyousharebetweenthe twoblocks? Whatiftheabove code were:

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

MACHINEOPTIMIZATIONS

Infinalcodegeneration,thereisalotofopportunityforclevernessingeneratingefficient target code. In this pass, specific machines features (specialized instructions, hardwarepipeline abilities, register details) are taken into account to produce code optimized for thisparticulararchitecture.

REGISTERALLOCATION:

Onemachineoptimizationofparticularimportanceisregisterallocation,whichisperhaps the single most effective optimization for all architectures. Registers are the fastest kindofmemoryavailable,butas aresource,theycanbescarce.

The problem is how to minimize traffic between the registers and what lies beyond themin the memory hierarchy to eliminate time wasted sending data back and forth across the bus andthe different levels of caches. Your Decaf back-end uses a very naïve and inefficient means ofassigningregisters,itjustfillsthembeforeperforminganoperationandspillsthemrightafterwards.

A much more effective strategy wouldbe to considerwhich variables are more heavilyin demand and keep thosein registers andspill those thatare nolongerneeded orwon'tbeneededuntilmuchlater.

One common register allocation technique is called "register coloring", after the centralidea to view register allocation as a graph coloring problem. If we have 8 registers, then we try tocolor a graph with eight different colors. The graph‘s nodes are made of "webs" and the arcs aredeterminedbycalculatinginterferencebetweenthewebs.Awebrepresentsavariable‘sdefinitions, places where it is assigned a value (as in **x = …**), and the possible different uses ofthosedefinitions(asin**y=x+2**).Thisproblem,infact,canbeapproachedasanothergraph.The definition and uses of a variable are nodes, and if a definition reaches a use, there is an arcbetween the two nodes. If two portions of a variable‘s definition-use graph are unconnected, thenwe have two separate webs for a variable. In the interference graph for the routine, each node is aweb. We seek to determine which webs don't interfere with one another, so we know we can usethesame registerforthosetwovariables.Forexample,considerthe followingcode:

```
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 isseparated by a definition or use of **j**, thus, **i**and **j** are "alive" at the same time. A variable is alivebetween the time it has been defined and that definition‘s last use, after which the variable isdead. If two variables interfere, then we cannot use the same register for each. But two variablesthat don't interfere can since there is no overlap in the liveness and can occupy the same register.Once we have the interference graph constructed, we r-color it so that no two adjacent nodesshare the same color (r is the number of registers we have, each color represents a differentregister).

We may recall that graph-coloring is NP-complete, so we employ a heuristic rather thananoptimalalgorithm.Hereisasimplifiedversionofsomething thatmightbeused:

1. Findthenodewiththeleastneighbors.(Breaktiesarbitrarily.)
2. Removeitfromtheinterferencegraphandpushitontoastack
3. Repeatsteps1and 2untilthegraphisempty.
4. Now,rebuildthegraphasfollows:
    a. Takethetopnodeoffthestack and reinsertitintothegraph
    b. Chooseacolorfor itbased onthecolorofanyofitsneighborspresentlyin thegraph,rotatingcolorsincase thereismorethanonechoice.
    c. Repeata,and buntilthegraphiseithercompletelyrebuilt,orthereisnocoloravailable tocolorthenode.

If we get stuck, then the graph may not be r-colorable, we could try again with a differentheuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable tomemory.

INSTRUCTIONSCHEDULING:

Anotherextremelyimportantoptimizationof thefinalcodegeneratorisinstructionscheduling. Because many machines, including most RISC architectures, have some sort ofpipeliningcapability,effectivelyharnessingthatcapabilityrequiresjudiciousorderingofinstructions .

In MIPS,each instructionisissuedin onecycle,butsometakemultiplecyclestocomplete. It takes an additional cycle before the value of a load is available and two cycles for abranch to reach its destination, but an instruction can be placed in the "delay slot" after a branchand executed in that slack time. On the left is one arrangement of a set of instructions thatrequires 7 cycles. It assumes no hardware interlock and thus explicitly stalls between the secondand third slots while the load completes and has a Dead cycle after the branch because the delayslot holds a noop. On the right, a more favorable rearrangement of the same instructions willexecute in5cycleswithnodeadCycles.

lw $t2,
4($fp)lw $t3,
8($fp)noop
add $t4, $t2,
$t3subi$t5,$t5,1
gotoL1
noop
lw $t2,
4($fp)lw $t3,
8($fp)**subi $t5,**
$t5, 1gotoL1
add $t4,$t2,$t3

PEEPHOLEOPTIMIZATIONS:

Peephole optimization is a pass that operates on the target assembly and only considers afewinstructionsatatime(througha"peephole")andattemptstodosimple,machinedependent codeimprovements.Forexample,peepholeoptimizationsmightincludeeliminationofmultiplication by 1, elimination of load of a value into a register when the previous instructionstored that value from the register to a memory location, or replacing a sequence of instructionsby a single instruction with the same effect. Because of its myopic view, a peephole optimizerdoes not have the potential payoff of a full-scale optimizer, but it can significantly improve codeat a very local level and can be useful for cleaning up the final code that resulted from morecomplex optimizations. Much of the work done in peephole optimization can be though of asfind-replace activity, looking for certain idiomatic patterns in a single or sequence of two to threeInstructionsthancanbereplacedbymoreefficientalternatives.

For example, MIPS has instructions that can add a small integer constant to the value in aregister without loading the constant into a register first, so the sequence on the left can bereplacedwiththatontheright:

```
li$t0,10
lw $t1, -
8($fp)add$t2,$t
1,$t0sw$t1,-
8($fp)
lw $t1, -
8($fp)addi$t2,$
t1,10sw$t1,-
8($fp)
```

Whatwouldyoureplacethefollowingsequencewith?lw

```
$t0, -8($fp)
sw$t0,-8($fp)
```

Whataboutthisone?

```
mul$t1,$t0,2
```

Abstract SyntaxTree/DAG: Isnothingbutthecondensedformofaparsetreeandis

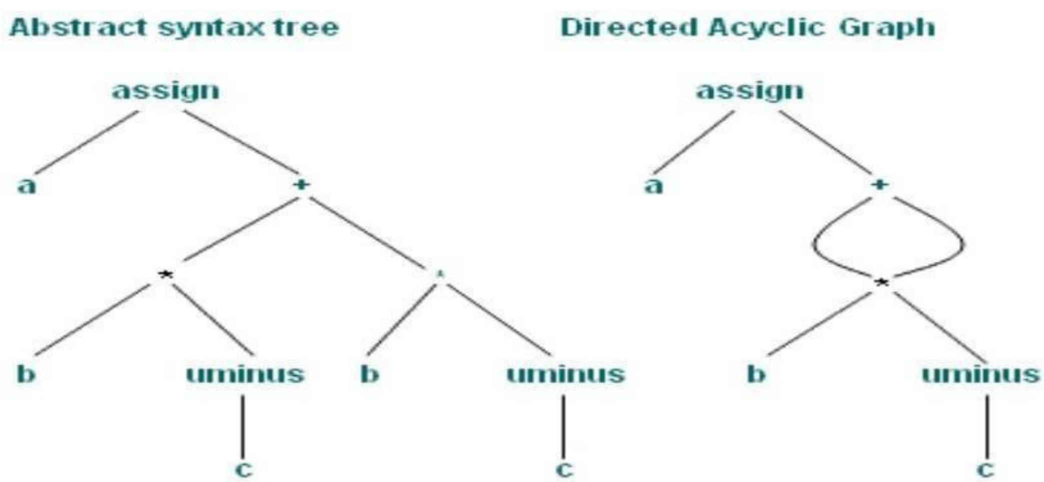
∑. Usefulforrepresentinglanguageconstructs

∑.Depicts the naturalhierarchicalstructureofthesourceprogram

- Eachinternalnoderepresentsanoperator
- Childrenofthe nodesrepresentoperands
- Leafnodesrepresentoperands

.DAG is more compact than abstract syntax tree because common sub expressions are eliminatedA syntax tree depicts the natural hierarchical structure of a source program. Its structure hasalreadybeendiscussedinearlierlectures.DAGsaregeneratedasacombinationof trees:operands that are being reused are linked together, and nodes may be annotated with variablenames (to denote assignments). This way, DAGs are highly compact, since they eliminate localcommon sub-expressions. On the other hand, they are not soeasy to optimize, since they aremorespecifictreeforms.However,itcanbeseenthatproperbuilding ofDAGforagiven

sequenceofinstructionscancompactlyrepresenttheoutcomeofthecalculation.Anexam ple ofa syntaxtree andDAG hasbeengiven inthenextslide .
a:=b*-c+b*-c



You canseethatthe node"* "comesonlyonceintheDAG aswellastheleaf"b",but themeaningconveyedbyboththerepresentations(ASTaswellastheDAG)remainsthesame.

IMPORTANTQUESTIONS:

1. WhatisCodeoptimization?Explaintheobjectivesofit.AlsodiscussFunctionpreservingtransfo rmationswithyourownexamples?
2. Explainthefollowingoptimizationtechniques
   (a) CopyPropagation
   (b) Dead-CodeElimination
   (c) CodeMotion
   (d) ReductioninStrength.
4. Explaintheprinciplesourcesofcode-improvingtransformations.
5. Whatdoyoumeanbymachinedependentandmachineindependentcodeoptimization?Explai naboutmachine dependentcodeoptimizationwithexamples.

ASSIGNMENTQUESTIONS:

1. ExplainLocalOptimizationtechniqueswith yourownExamples?
2. Explain indetailtheprocedurethat eliminating globalcommonsubexpression?
3. Whatistheneedofcodeoptimization?Justifyyouranswer?

UNIT-V

CONTROL/DATAFLOWANALYSIS:

FLOWGRAPHS:

Wecanaddflowcontrol information tothesetof basicblocksmakingupa program byconstructing a directed graph called a flow graph. The nodes of a flow graph are the basic nodes.One node is distinguished as initial; it is the block whose leader is the first statement. There is adirected edge from block $B_1$to block B_2if B_2can immediately follow B_1in some executionsequence;thatis,if

- There is conditional or unconditional jump from the last statement of B_1 to the firststatementofB_2, or
- B_2immediately follows B_1in the order of the program, and B_1does not end in anunconditionaljump.We saythat$B_1$is the predecessorofB_2,andB_2isa successorofB_1.

Forregister andtemporaryallocation

- Removevariables fromregistersif notused
- StatementX=Yop ZdefinesXand usesYand Z
- Scaneachbasicblocksbackwards
- Assumealltemporariesaredeadonexitand alluser variablesareliveon exit

The use of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can flowfrom statement i to j along a path that has no intervening assignments to x, then we say statementjuses thevalueofxcomputedati.

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

The simple code generator applies it to register assignment. Our algorithm is to determinenext uses makes a backward pass over each basic block, recording (in the symbol table) for eachname x whether x has a next use in the block and if not, whether it is live on exit from that block.We can assume that all non-temporary variables are live on exit and all temporary variables aredeadonexit.

Algorithmtocomputenextuseinformation

- Supposewe arescanningi:X:=YopZ inbackwardscan

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

As an application, we consider the assignment of storage for temporary names. Suppose wereachthree-addressstatementi:x:=yop zinourbackward scan.Wethendo thefollowing:

1. Attachtostatementitheinformationcurrentlyfoundinthesymboltableregardingthenextuse andlivenessofx,yandz.

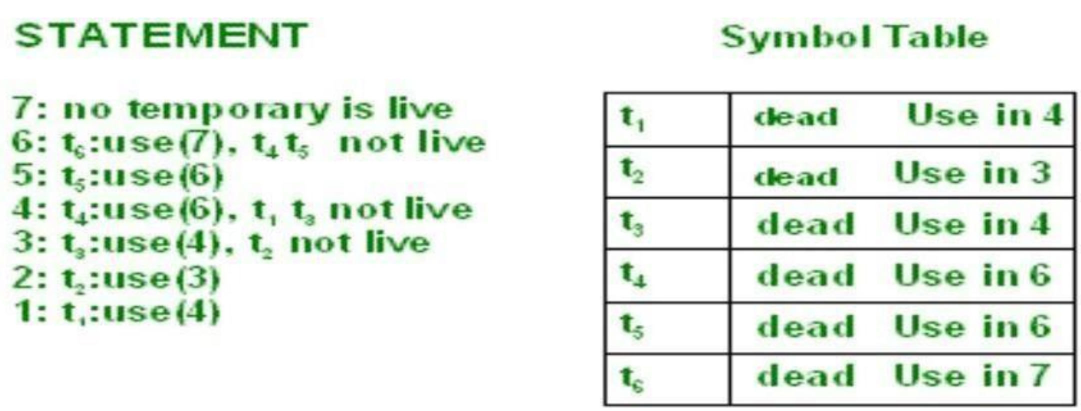
2. Inthesymboltable,setxto "notlive"and "nonextuse".

3. Inthesymboltable,setyandzto"live"and thenextusesofyand zto i.Notethattheorderofsteps(2)and(3)maynotbe interchangedbecausexmaybeyorz.

Ifthree-addressstatementiisofthe formx:=yorx:=opy,the stepsarethe same asabove,ignoringz.considerthebelow example:

1: t_1= a * a2:
t $_2$= a * b3:
t_3= 2 * $t_2$4: t_4
= t $_1$+ $t_3$5:
t_5= b * b6: t_6
= t $_4$+
$t_5$7:X=t_6

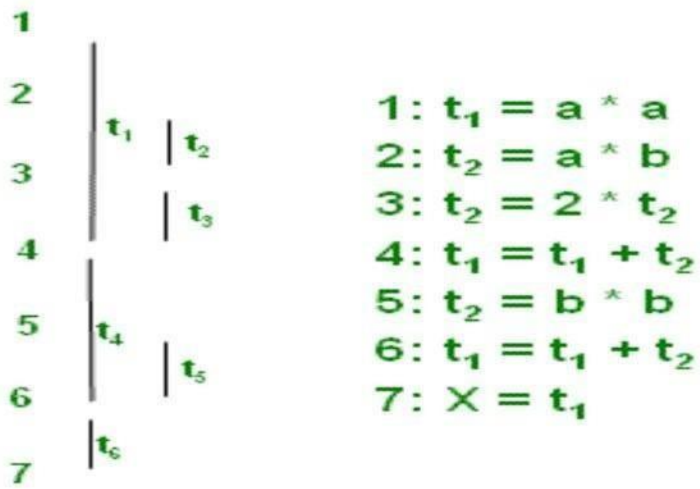
Example:



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

Wecanallocatestoragelocationsfortemporariesbyexaminingeachinturnandassigning a temporary to the first location in the field for temporaries that does not contain a livetemporary. If a temporary cannot be assigned to any previously created location, add a newlocation to the data area for the current procedure. In many cases, temporaries can be packed intoregistersratherthanmemorylocations,asinthe nextsection.

Example.



Thesixtemporariesinthebasicblockcanbepackedintotwolocations.Theselocationscorrespondtot 1andt2in:

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:

Dataanalysis is neededforglobalcodeoptimization, e.g.:Isavariableliveonexitfromablock?Does a definition reach a certain point in the code? Data flow equations are used to collectdataflowinformationAtypicaldataflowequationhastheform

Out[s]=Gen[s]U(in[s]-kill[s])

The notion of generation and killing depends on the dataflow analysis problem to besolvedLet'sfirstconsiderReachingDefinitionsanalysisforstructuredprogramsAdefinitionofa variable x is a statement that assigns or may assign a value to x An assignment to x is anunambiguous definition of x An ambiguous assignment to x can be an assignment to a pointer ora function call where x is passed by reference When x is defined, we say the definition isgeneratedAnunambiguousdefinitionofxkillsallotherdefinitionsofxWhenalldefinitionsofx 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 strengthreductionifxis usedinz=x*y.

GLOBALOPTIMIZATIONS,DATA-FLOWANALYSIS

So far we were only considering making changes within one basic block. With someadditional analysis, we can apply similar optimizations across basic blocks, making them globaloptimizations. It's worth pointing out that global in this case does not mean across the entireprogram. We usually only optimize one function at a time. Interprocedural analysis is an evenlarger task, one not even attempted by some compilers. The additional analysis the optimizermust do to perform optimizations across basic blocks is called data-flow analysis. Data-flowanalysisismuchmorecomplicatedthancontrol-flow analysis.

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.Suchanexpressionis saidtobe availableatthatpoint.

Oncethesetofavailableexpressionsisknown,commonsub-expressionscanbeeliminated on a global basis. Each block is a node in the flow graph of a program. The successorset (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 atthe point where it is assigned a value and killed when one of its operands is subsequentlyassigned a new value. An expression is available at some point p in a flow graph if every pathleadingtopcontains a priordefinitionofthatexpressionwhichisnot subsequentlykilled.

avail[B]=setofexpressionsavailableonentryto block B
exit[B]=set ofexpressionsavailableonexitfromB
avail[B]=∩**exit[x]: x∈pred[B]**(i.e.Bhasavailabletheintersectionoftheexitofits predecessors)
killed[B] = set of the expressions killed in B**defined[B]** = set of expressions defined in B**exit[B]**=**avail[B]- killed[B]**+**defined[B]**
avail[B]=∩**(avail[x]-killed[x] +defined[x])**:x∈**pred[B]**

Hereisanalgorithmfor globalcommonsub-expressionelimination:

1) First, computedefinedandkilledsetsfor eachbasicblock(thisdoesnotinvolveanyofitsredecessorsorsuccessors).
2) Iterativelycomputetheavailandexitsetsfor eachblockbyrunningthefollowingalgorithmuntilyouhitastablefixedpoint:
    a) Identifyeachstatement**s**ofthe form**a =bopc** insome block Bsuchthat**bopc**isavailableatthe entrytoBandneither**b**nor**c** isredefinedinBpriorto**s**.
    b) Followflowofcontrolbackwardinthegraphpassingback tobut notthrougheachblockthatdefines **bop c**.Thelastcomputationof**b opc**insuchablock reaches**s**.
    c) After each computation **d = b op c** identified in step 2a, add statement **t = d** to thatblockwheretisanewtemp.
    d) Replace**s**by**a=t**.

Letstryanexample tomake thingsclearer:main:

```
BeginFunc
28;b=a+2;
c = 4 * b
;tmp1=b<c;
ifNZ tmp1 goto L1
;b=1;
L1:
d = a + 2
;EndFunc;
```

First,dividethecodeaboveintobasic blocks.Nowcalculatetheavailableexpressionsforeach block.Then find anexpressionavailableinablock andperformstep2cabove. Whatcommonsubexpressioncanyousharebetweenthetwoblocks?Whatiftheabovecodewere:

```
main:
BeginFunc
28;b=a+2;
c = 4 * b
;tmp1=b<c;
IfNZ tmp1 Goto L1
;b=1;
z=a +2;<========= anadditionallinehereL1:
d = a + 2
;EndFunc;
```

CommonSubexpressionElimination

Two operations are common if they produce the same result. In such a case, it is likely moreefficient to compute the result once and reference it the second time rather than re-evaluate it. Anexpression is alive if the operands used to compute the expression have not been changed. Anexpressionthatisnolongeraliveisdead.

```
main()
{
int x,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;tm
p3 = 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) bedetermined? Here is an optimized version, after constant folding and propagation and eliminationofcommonsub-expressions:

tmp2=-x;
x=21*tmp2;tm
p3 = x * x
;tmp4= x/y;
y=tmp3+tmp4;tm
p5=x/y;
z=tmp5/tmp3;y=
z;

InductionVariableElimination

Constantfoldingreferstotheevaluationatcompile-timeofexpressionswhoseoperands are known to be constant. In its simplest form, it involves determining that all of theoperands in an expression are constant-valued, performing the evaluation of the expression atcompile-time, and then replacing the expression by its value. If an expression such as **10 + 2 * 3**is encountered, the compiler can compute the result at compile-time (**16**) and emit code as if theinput contained the result rather than the original expression. Similarly, constant conditions, suchas a conditional branch **if a < b goto L1 else goto L2** where **a** and **b** are constant can be replacedby a **Goto L1** or **Goto L2** depending on the truth of the expression evaluated at compile-time.The constantexpression has to be evaluated at least once, but if the compiler does it, it meansyou don‘t have to do it again as needed during runtime. One thing to be careful about is that thecompiler must obey the grammar and semantic rules from the source language that apply toexpressionevaluation,whichmaynotnecessarilymatchthelanguageyouarewritingthecompilerin. (For example,if you were writing an APL compiler,you would need to take carethat you were respecting its Iversonian precedence rules). It should also respect the expectedtreatment of any exceptional conditions (divide by zero, over/underflow). Consider the DecafcodeonthefarleftanditsunoptimizedTACtranslationinthemiddle,whichisthentransformedbyconstant-foldingonthe farright:

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-foldingiswhat allowsa languagetoacceptconstantexpressionswhereaconstantisrequired (suchasa caselabelorarraysize)asinthese Clanguage examples:

```
int arr[20 * 4 +
3];switch(i){
case10*5:...
}
```

In both snippets shown above, the expression can be resolved to an integer constant at compiletime and thus, we have the information needed to generate code. If either expression involved avariable, though, there would be an error. How could you rewrite the grammar to allow thegrammar to do constant folding in case statements? This situation is a classic example of the grayarea betweensyntactic andsemanticanalysis.

LiveVariableAnalysis

Avariableis liveatacertainpointinthecodeifitholdsavaluethatmaybeneededinthefuture.
Solvebackwards:
FinduseofavariableThisvariableis
livebetweenstatementsthathavefounduseasnextstatementRecursive untilyoufinda
definitionofthevariable

Using the sets *use[B]*and *def[B]*

de f[B] is the set of variables assigned values in *B* prior to any use of that variable in *B* *use* [*B]*istheset ofvariableswhosevaluesmay beusedin [*B]*priortoanydefinitionofthevariable.

A variable comes live into a block (in *in[B]*), if it is either used before redefinition of it islive coming out of the block and is not redefined in the block .A variable comes live out of ablock(in*out[B]*)ifandonlyifitislive comingintoone ofitssuccessors

In[B]=use[B] U *(out[B]-de*
f[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-constantvalues. If we have an assignment **a = b** in our instruction stream, we can replace lateroccurrencesof**a**with**b** (assumingtherearenochangestoeithervariablein-between). GiventhewaywegenerateTACcode,thisisaparticularlyvaluableoptimizationsinceitisable to eliminate a large number of instructions that only serve to copy values from one variable toanother. The code on the left makes a copy of **tmp1** in **tmp2** and a copy of **tmp3** in **tmp4**. In theoptimized version on the right, we eliminated those unnecessary copies and propagated theoriginalvariableintothe lateruses:

```
tmp2=tmp1 ;
tmp3=tmp2*tmp1;t
mp4= tmp3;
tmp5=tmp3*tmp2;c =
tmp5 + tmp4
;tmp3=tmp1*tmp1;t
mp5=tmp3*tmp1;c
=tmp5+ tmp3;
```

We can also drive this optimization "backwards", where we can recognize that the originalassignment made to a temporary can be eliminated in favor of direct assignment to the final goal:tmp1=LCall_Binky;

```
a =tmp1;
tmp2 = LCall _Winky
;b=tmp2;
tmp3 = a * b
;c =tmp3;
a = LCall
_Binky;b=LCall_
Winky;c=a*b;
```

IMPORTANTQUESTIONS:

1. WhatisDAG?ExplaintheapplicationsofDAG.
2. Explainbrieflyaboutcodeoptimizationanditsscopeinimprovingthecode.
3. ConstructtheDAGforthefollowingbasicblock:D
   :=B*C
   E:=A+B
   B :=
   B+CA:=
   E-D.
3. ExplainDetectionofLoop InvariantComputation
4. ExplainCodeMotion.

ASSIGNMENTQUESTIONS:

1. Whatisloops?Explainaboutthefollowingtermsin
   loops:(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 efficienttarget code. In this pass, specific machines features (specialized instructions, hardware pipelineabilities, register details) are taken into account to produce code optimized for this particulararchitecture.

RegisterAllocation

Onemachine optimization of particularimportanceis register allocation, which isperhaps the single most effective optimization for all architectures. Registers are the fastest kindof memory available, but as a resource, they can be scarce. The problem is how to minimizetraffic between the registers and whatlies beyond them in the memory hierarchy to eliminatetime wasted sending data back and forth across the bus and the different levels of caches. YourDecaf back-end uses a very naïve and inefficient means of assigning registers, it just fills thembefore performing an operation and spills them right afterwards. A much more effective strategywould be to consider which variables are more heavily in demand and keep those in registers andspill those that are no longer needed or won't be needed until much later. One common registerallocation technique is called "register coloring", after the central idea to view register allocationas a graph coloring problem. If we have 8 registers, then we try to color a graph with eightdifferentcolors.Thegraph‘snodesaremadeof"webs"andthearcsaredeterminedbycalculatinginterference between the webs.A web represents a variable‘s definitions,placeswhere 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 another graph. The definition and usesof a variable are nodes, and if a definition reaches a use, there is an arc between the two nodes. Iftwo portions of a variable‘s definition-use graph are unconnected, then we have two separatewebs for a variable. In the interference graph for the routine, each node is a web. We seek todetermine which webs don't interfere with one another, so we know we can use the same registerforthosetwovariables.Forexample,considerthefollowingcode:

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 isseparated by a definition or use of **j**, thus, **i**and **j** are "alive" at the same time. A variable is alivebetween the time it has been defined and that definition‘s last use, after which the variable isdead. If two variables interfere, then we cannot use the same register for each. But two variablesthatdon'tinterferecansincethereisnooverlapintheIivenessandcanoccupythesameregister.

Once we have the interference graph constructed, we r-color it so that no two adjacent nodesshare the same color (r is the number of registers we have, each color represents a differentregister). You may recall that graph-coloring is NP-complete, so we employ a heuristic ratherthananoptimalalgorithm.Hereisasimplified versionofsomethingthatmightbeused:

1. Findthenodewiththeleastneighbors.(Breaktiesarbitrarily.)
2. Removeitfromtheinterferencegraphandpushitontoastack
3. Repeatsteps1and2untilthegraph is empty.
4. Now,rebuildthegraphasfollows:
   a. Takethetopnodeoffthestack and reinsertitintothegraph
   b. Chooseacolor foritbased onthecolorofanyofitsneighborspresentlyinthegraph,rotatingcolorsincase thereismorethanonechoice.
   c. Repeataandbuntilthegraphiseither completelyrebuilt,orthereis no coloravailable tocolorthenode.

If we get stuck, then the graph may not be r-colorable, we could try again with a differentheuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable tomemory.

InstructionScheduling:

Anotherextremelyimportantoptimizationofthefinalcodegeneratorisinstructionscheduling. Because many machines,including most RISC architectures,have some sort ofpipeliningcapability,effectivelyharnessingthatcapabilityrequiresjudiciousorderingofinstructions. In MIPS, each instruction is issued in one cycle, but some take multiple cycles tocomplete. It takes an additional cycle before the value of a load is available and two cycles for abranch to reach its destination, but an instruction can be placed in the "delay slot" after a branchand executed in that slack time. On the leftis one arrangement of a set of instructions thatrequires 7 cycles. It assumes no hardware interlock and thus explicitly stalls between the secondand third slots while the load completes and has a Dead cycle after the branch because the delayslot holds a noop. On the right, a more Favorable rearrangement of the same instructions willexecute in5cycleswithnodeadCycles.

lw$t2,
4($fp)lw $t3,
8($fp)noop
add $t4, $t2,
$t3subi$t5,$t5,1
gotoL1
noop
lw $t2,
4($fp)lw $t3,
8($fp)**subi $t5,**
$t5, 1gotoL1
add $t4,$t2,$t3

RegisterAllocation

Onemachine optimization of particularimportanceis register allocation, which isperhaps the single most effective optimization for all architectures. Registers are the fastest kindof memory available, but as a resource, they can be scarce. The problem is how to minimizetraffic between the registers and whatlies beyond them in the memory hierarchy to eliminatetime wasted sending data back and forth across the bus and the different levels of caches. YourDecaf back-end uses a very naïve and inefficient means of assigning registers, it just fills thembefore performing an operation and spills them right afterwards. A much more effective strategywould be to consider which variables are more heavily in demand and keep those in registers andspill those that are no longer needed or won't be needed until much later. One common registerallocation technique is called "register coloring", after the central idea to view register allocationas a graph coloring problem. If we have 8 registers, then we try to color a graph with eightdifferentcolors.Thegraph‘snodesaremadeof"webs"andthearcsaredeterminedbycalculatinginterference between the webs.A web represents a variable‘s definitions,placeswhere 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 another graph. The definition and usesof a variable are nodes, and if a definition reaches a use, there is an arc between the two nodes. Iftwo portions of a variable‘s definition-use graph are unconnected, then we have two separatewebs for a variable. In the interference graph for the routine, each node is a web. We seek todetermine which webs don't interfere with one another, so we know we can use the same registerforthosetwovariables.Forexample,considerthefollowingcode:

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 isseparated by a definition or use of **j**, thus, **i**and **j** are "alive" at the same time. A variable is alivebetween the time it has been defined and that definition‘s last use, after which the variable isdead. If two variables interfere, then we cannot use the same register for each. But two variablesthatdon'tinterferecansincethereis nooverlapinthelivenessandcanoccupythesameregister.Once we have the interference graph constructed, we r-color it so that no two adjacent nodesshare the same color (r is the number of registers we have, each color represents a differentregister). You may recall that graph-coloring is NP-complete, so we employ a heuristic ratherthananoptimalalgorithm.Hereisasimplified versionofsomething thatmightbeused:

1. Findthenodewiththeleastneighbors.(Breaktiesarbitrarily.)
2. Removeitfromtheinterferencegraphandpushitontoastack
3. Repeatsteps1and 2untilthegraphisempty.
4. Now,rebuildthegraphasfollows:
    a. Takethetopnodeoffthestack and reinsertitintothegraph

b. Chooseacolorfor itbased onthecolorofanyofitsneighborspresentlyinthegraph,rotatingcolorsincase thereismorethanonechoice.

c. Repeataandbuntilthegraphiseithercompletelyrebuilt,orthereis nocoloravailabletocolorthenode.

If we get stuck, then the graph may not be r-colorable, we could try again with a differentheuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable tomemory.

CODEGENERATION:

The code generator generates target code for a sequence of three-address statement. Itconsiders each statementin turn, rememberingif any of theoperands ofthestatementarecurrently in registers, and taking advantage of that fact, if possible. The code-generation usesdescriptorstokeeptrackofregistercontents andaddressesfornames.

1. A register descriptor keeps track of what is currently in each register. It is consulted whenevera new register is needed. We assume that initially the register descriptor shows that all registersare empty. (If registers are assigned across blocks, this would not be the case). As the codegeneration for the block progresses, each register will hold the value of zero or more names atanygiventime.

2. An address descriptor keeps track of the location (or locations) where the current value of thename can be found at run time. The location might be a register, a stack location, a memoryaddress,orsomesetofthese,sincewhencopied,avaluealsostayswhereitwas.Thisinformationc anbestoredinthesymboltableandisusedtodeterminetheaccessingmethodfora name.

CODEGENERATIONALGORITHM:

foreachX=YopZdo

- InvokeafunctiongetregtodeterminelocationL whereXmustbestored.UsuallyLisaregister.
- ConsultaddressdescriptorofYtodetermineY'.Prefer aregister for Y'.IfvalueofYnotalreadyinLgenerate

  MovY',L

- Generate
  opZ',L

Again prefer a register for Z. Update address descriptor of X to indicate X is in L. If L is aregister update its descriptor to indicate that it contains X and remove X from all other registerdescriptors.

. IfcurrentvalueofYand/or Zhasnonextuseandaredeadonexitfromblockandareinregisters,changeregisterdescriptortoindicat ethattheynolongercontainYand/or Z.

The code generation algorithm takes as input a sequence of three-address statements constitutinga basic block. For each three-address statement of the form x := y op z we perform the followingactions:

1. InvokeafunctiongetregtodeterminethelocationLwheretheresultofthecomputation y op z should be stored. L will usually be a register, but it could also be amemorylocation.We shalldescribe getregshortly.

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. Ifthe value of u is not already in L, generate the instruction MOV y', L to place a copy of yinL.

3. Generate the instruction OP z', L where z' is a current location of z. Again, prefer aregister to a memory location if z is in both. Update the address descriptor to indicate thatx is in location L. If L is a register, update its descriptor to indicate that it contains thevalue ofx,andremovexfromallotherregisterdescriptors.

4. If the current values of y and/or y have no next uses, are not live on exitfrom theblock,andareinregisters,altertheregisterdescriptortoindicatethat,afterexecutionofx:=yop z,those registersnolongerwillcontain yand/orz,respectively.

FUNCTIONgetreg:

1. IfYisinregister(thatholdsnoothervalues)andYisnotliveandhasnonextuseafterX=YopZ thenreturnregisterofYforL.
2. Failing (1)returnanemptyregister
3. Failing(2) ifXhasanextuseintheblockoroprequiresregisterthengetaregister R,storeitscontentintoM(byMovR,M)anduseit.
4. ElseselectmemorylocationXasL

Thefunction**getreg**returnsthelocationLtoholdthevalueofxfortheassignmentx:=yopz.

1. If the name y is in a register that holds the value of no other names (recall that copyinstructionssuchasx:=ycouldcausearegistertoholdthevalueoftwoormorevariables

simultaneously),andyisnotliveandhasnonextuseafter executionofx:=yopz,thenreturntheregisterofyforL.Updatetheaddressdescriptorofytoindicatethat yisnolongerinL.

2. Failing (1),returnanemptyregisterforLifthereisone.

3. Failing(2),ifxhasanextuseintheblock,oropisanoperatorsuchasindexing,thatrequiresa register, find an occupied register R. Store the value of R into memory location (by MOVR,M)ifitisnotalreadyinthepropermemorylocationM,updatetheaddressdescriptorM,andreturn R. IfRholds thevalueofseveralvariables,aMOVinstructionmustbegeneratedforeachvariablethatneedstobestored .Asuitableoccupiedregistermightbeonewhosedatumisreferencedfurthestinthefuture,orone whose valueisalsoinmemory.

4. Ifxisnotusedintheblock,ornosuitableoccupiedregistercanbefound,selectthememorylocationofxa sL.

Example:

Stmt	code	reg desc	addr desc
t_1=a-b	mova,R_0	R_0containst$_1$	$t_1$inR_0
	subb,R_0		
t_2=a-c	mova,R_1	R_0containst$_1$	$t_1$inR_0
	subc,R_1	R_1containst$_2$	$t_2$inR_1
t_3=t_1+t_2	addR_1,R_0	R_0containst$_3$	$t_3$inR_0
		R_1containst$_2$	$t_2$inR_1
d=t_3+t_2	addR $_1$,R_0	R_0containsd	dinR_0
	movR_0,d		din$R_0$and memory

Forexample,theassignmentd:=(a-b) +(a-c)+(a-c)mightbetranslatedintothefollowingthree-address codesequence:

t_1= a -

bt_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.Shownalongsideare thevaluesof theregisterandaddressdescriptorsascodegenerationprogresses.

DAGforRegisterallocation:

DAG(DirectedAcyclicGraphs)areusefuldatastructuresforimplementingtransformations on basic blocks.A DAGgivesa picture of how the value computedby astatement in a basic block is used in subsequent statements of the block. Constructing a DAGfromthree-addressstatementsisagoodwayofdeterminingcommonsub-expressions(expressions computed more than once) within a block, determining which names are used insidethe block but evaluated outside the block, and determining which statements of the block couldhave theircomputedvalueusedoutsidetheblock.

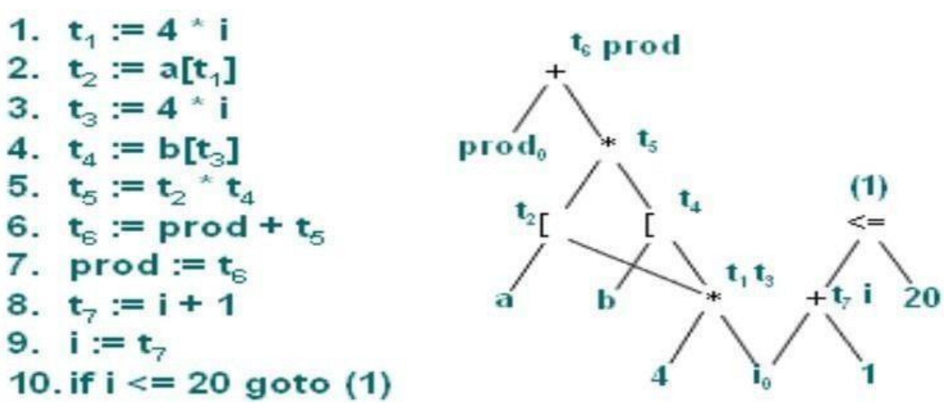
ADAGforabasicblockisadirectedcyclicgraphwiththefollowinglabelsonnodes:

1. Leaves are labeled by unique identifiers, either variable names or constants. From theoperator applied to a name we determine whether the l-value or r-value of a nameis needed;most leaves represent r- values. The leaves represent initial values of names, and we subscriptthemwith0 toavoidconfusionwith labelsdenoting"current"valuesofnamesasin(3)below.

2. Interiornodesarelabeledbyanoperatorsymbol.

3. Nodes are also optionally given a sequence of identifiers for labels. The intention isthat interior nodes represent computed values, and the identifiers labeling a node are deemed tohave thatvalue.

DAGrepresentationExample:



For example, the slide shows a three-address code. The corresponding DAG is shown.Weobserve that each node of the DAG represents a formula in terms of the leaves, that is, the valuespossessed by variables and constants upon entering the block. For example, the node labeled t 4representstheformula

b[4*i]

thatis,thevalueofthewordwhoseaddressis4*ibytesoffsetfromaddressb,which istheintendedvalueoft4.

CodeGenerationfromDAG

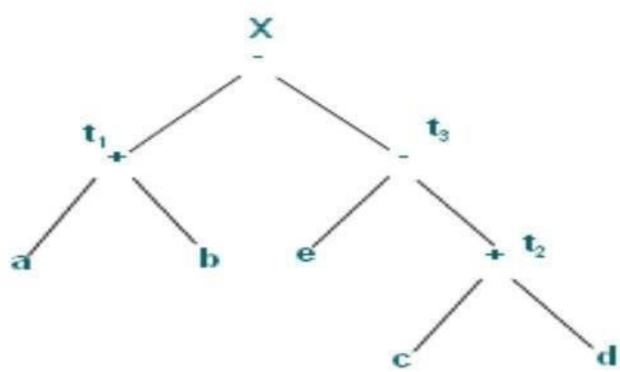
S_1=4*i	S_1=4*i
S_2=addr(A)-4	S_2=addr(A)-4
S_3=S_2[S_1]	S_3=S_2[S_1]
S_4=4*i	
S_5=addr(B)-4	S_5=addr(B)-4
S_6=S_5[S_4]	S_6=S_5[S_4]
S_7=S_3*S_6	S_7=S_3*S_6
S_8=prod+S_7	
prod=S_8	prod=prod+S $_7$
S_9=I+1	
I=S_9	I=I+1
IfI<=20goto(1)	IfI<= 20goto(1)

Weseehowtogeneratecodefora basicblockfromitsDAGrepresentation.Theadvantage of doing so is that from a DAG we can more easily see how to rearrange the order ofthe final computation sequence than we can starting from a linear sequence of three-addressstatements 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 algorithmforoptimalcodegeneration froma treeisalsousefulwhentheintermediatecodeisa parsetree.

Rearrangingorderofthe code

Considerfollowingbasic block:

t 1 = a + bt 2 = c + dt3=e-t2 X=t1-t3



andits**DAG**givenhere.

Here, we briefly consider how the order in which computations are done can affect thecost of resulting object code. Consider the basic block and its corresponding DAG representationasshownintheslide.

Rearranging order.

Threeadresscodefor theDAG(assumingo nlytworegisters are available)		Rearrangingthecodeast_2 =c+d t_3=e-t2 t_1=a+b X=t_1-t_3 gives
MOVa, R_0		MOV c,R_0
ADDb,R_0		ADDd,R_0
MOVc, R_1		MOV e,R_1
ADD d,R_1		SUBR_0,R_1
MOVR_0,t_1	Registerspilling	MOV a,R_0
MOVe,R_0		ADDb,R_0
SUBR_1,R_0		SUBR $_1$,R_0
MOVt_1,R_1	Registerreloading	MOV R_1,X
SUBR_0,R_1		
MOVR_1,X		

Ifwegeneratecodeforthethree-addressstatementsusingthecodegenerationalgorithmdescribed before, we get the code sequence as shown (assuming two registers R0 and R1 areavailable, and only X is live on exit). On the other hand suppose we rearranged the order of thestatementssothatthecomputationoft1occurs immediatelybeforethatofX as:

t_2 = c + d
t_3 = e -t2
t_1 = a + b
X =t_1-t_3

Then, using the code generation algorithm, we get the new code sequence as shown (again onlyR0 and R1 are available).By performing the computation in this order, we have been able tosave two instructions; MOV R0, t 1 (which stores the value of R0 in memory location t 1) andMOV t1,R1(whichreloads the valueoft1intheregisterR1).

IMPORTANT&EXPECTEDQUESTIONS:

ConstructtheDAGforthefollowingbasicblock:D
:=B*C
E:=A+B
B :=
B+CA:=
E-D.

1. WhatisObjectcode?Explainaboutthefollowingobjectcodeforms:
   (a) Absolutemachine-language
   (b) Relocatablemachine-language
   (c) Assembly-language.
2. ExplainaboutGenericcodegenerationalgorithm?
3. Writeandexplainaboutobjectcodeforms?
4. ExplainPeepholeOptimization

ASSIGNMENTQUESTIONS:

1. ExplainaboutGenericcodegenerationalgorithm?
2. ExplainaboutData-Flowanalysisofstructuredflowgraphs.
3. WhatisDAG?ExplaintheapplicationsofDAG.