UNIT–III
MorePowerful LR parser (LR1,LALR) Using Armigers Grammars Equal Recovery in Lr parserSyntaxDirectedTransactionsDefinition,EvolutionorderofSDTSApplicationofSDTS.SyntaxDirectedTranslationSchemes.

UNIT-3

CANONICALLRPARSING

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to buildthe CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to theSLR (1)parsing.

IntheCLR(1),weplace

thereducenodeonlyinthelookaheadsymbols.Varioussteps involved in the

CLR (1)Parsing:

1) Forthegiveninput stringwriteacontextfreegrammar
2) Checktheambiguityofthegrammar
3) AddAugmentproductionin thegivengrammar
4) CreateCanonicalcollection ofLR (0)items
5) Drawadataflowdiagram(DFA)
6) Constructa CLR(1)parsingtable

In the SLR method we were working with LR(0)) items. In CLR parsing we will be using LR(1)items. LR(k) item is defined to be an item using lookaheads of length k. So ,the LR(1) item iscomprised of two parts : the LR(0) item and the lookahead associated with the item. The look aheadis used to determine that where we place the final item. The look ahead always add $ symbol for theargumentproduction.
LR(1)parsersaremorepowerfulparser.
forLR(1)items we modify theClosureandGOTOfunction.

ClosureOperation
Closure(I)

```
repeat
  for(eachitem[A->?.B?,a]inI)for (each
    production B -> ? in
    G')for(eachterminalbinFIRST(?a))
      add [ B -> .? , b ] to set
I;untilnomoreitemsareaddedtoI;ret
urn I;
```

GotoOperation

```
Goto(I,X)
InitialiseJtobethe empty set;
for( each item A ->?.X?, a] inI)
    AdditemA->?X.?,a ] toseJ;/*movethedotone
step*/returnClosure(J);    /* apply closureto theset */
```

LR(1)items

```
Voiditems(G’)
Initialise C to { closure ({[S’ -> .S,
$]})};Repeat
    For(eachset ofitemsIinC)
        For(eachgrammar symbolX)
            if( GOTO(I, X) is not empty and not in
                C)AddGOTO(I,X)to C;
Untilnonew setof itemsareaddedto C;
```

ALGORITHMFORCONSTRUCTIONOFTHECANONICALLRPARSINGTABLE

Input:grammarG'
Output:canonicalLR parsingtablefunctions actionandgoto

1. ConstructC={I0,I1, ...,In}thecollectionofsets ofLR(1)items forG'.Stateiisconstructed fromIi.
2. if[*A*-> a.ab,b>]isinIi and goto(Ii,a)=Ij,thensetaction[i, a]to"shiftj".Hereamust beaterminal.
3. if[*A*-> a.,a] isinIi,thensetaction[i, a]to"reduce *A*->a"forallainFOLLOW(*A*). Here *A*may *not* be*S*'.
4. if[*S*'->*S*.]is inIi,thenset action[i,$]to"accept"
5. IfanyconflictingactionsaregeneratedbytheserulesthegrammarisnotLR(1)and the algorithmfails to produceaparser.
6. The goto transitions for state i are constructed for all *nonterminal*s *A* using therule:If goto(Ii, A)= Ij, then goto[i,*A*] =j.
7. Allentriesnot definedbyrules2and 3aremade"error".
8. Theinital stateoftheparseris theoneconstructed from theset ofitemscontaining[*S*' ->.S, $].

Example,

Considerthefollowinggrammar,

S‟-
>SS-
>CCC
-
>cCC-
>d

Setsof LR(1)items

I0: S‟-
>.S,$S-
>.CC,$
C-
>.Cc,c/dC-
>.d,c/d

I1: S‟->S.,$

I2: S->C.C,$
C->.Cc,$
C->.d,$

I3:C->c.C,c/dC-
>.Cc,c/dC-
>.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

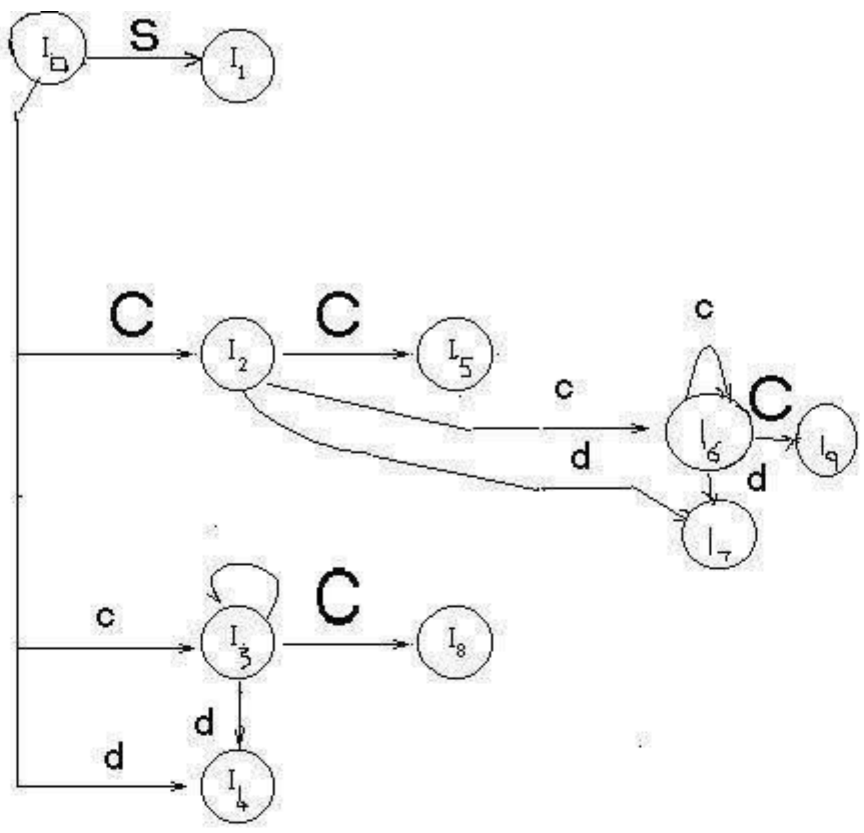
I6: C->c.C,$
C->.cC,$
C->.d,$

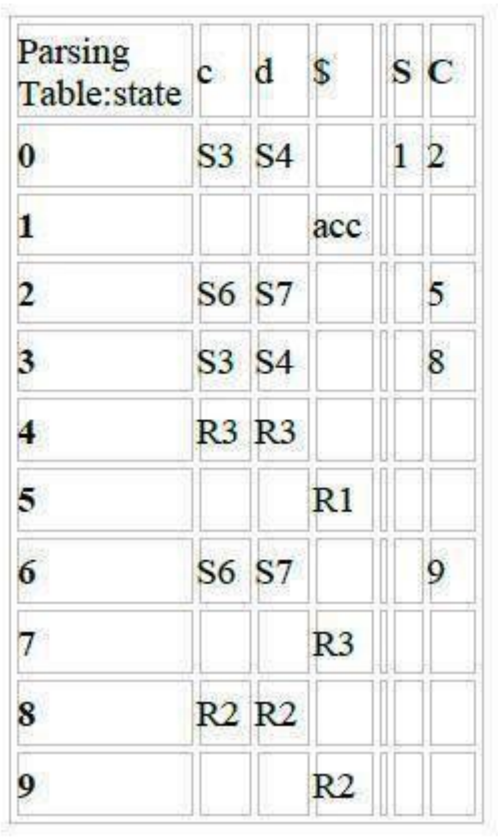
I7: C->d.,$

I8: C->cC.,c/d

I9: C->cC.,$

Hereis whatthecorresponding DFAlooks like





Parsing Table:state	c	d	$	S	C
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

.LALRPARSER:

We begin with two observations. First, some of the states generated for LR(1) parsinghavethesamesetofcore(orfirst)componentsanddifferonlyintheirsecondcomponent,thelo okaheadsymbol.Ourintuitionisthatweshouldbeabletomergethesestatesandreduce the number of states we have, getting close to the number of states that would begenerated for LR(0) parsing. This observation suggests a hybrid approach: We can constructthe canonical LR(1) sets of items and then look for sets of items having the same core. Wemerge these sets with common cores into one set of items. The merging of states withcommon cores can never produce a shift/reduce conflict that was not present in one of theoriginal states because shift actions depend only on the core, not the lookahead. But it ispossiblefor themerger to produceareduce/reduceconflict.

Our second observation is that we are really only interested in the lookahead symbolin places where there is a problem. So our next thought is to take the LR(0) set of items andadd lookaheads only where they are needed. This leads to a more efficient, but much morecomplicatedmethod.

ALGORITHMFOREASY CONSTRUCTIONOF ANLALR TABLE

Input:G'
Output:LALRparsingtablefunctionswithaction

andgotoforG'.Method:

1. Construct C={I0,I1 ,...,In}thecollectionofsetsof LR(1)itemsforG'.
2. Foreachcorepresentamong theset ofLR(1)items,find allsetshaving thatcoreandreplacethesesets bythe union.
3. Let C' = {J0, J1 , ..., Jm} be the resulting sets of LR(1) items. The parsing actionsfor state i are constructed from Ji in the same manner as in the construction of thecanonicalLR parsing table.
4. Ifthereisaconflict,the grammar isnotLALR(1)andthealgorithm fails.
5. The goto table is constructed as follows: If J is the union of one or more sets ofLR(1) items, that is, J = I0U I1 U ... U Ik, then the cores of goto(I0, X), goto(I1,X), ..., goto(Ik, X) are the same, since I0, I1 , ..., Ik all have the same core. Let Kbethe union ofallsets ofitems having thesame coreasgoto(I1,X).

6. Thengoto(J,X)= K.

Considerthe aboveexample,

I3&I6canbereplacedbytheirunion I36:C->c.C,c/d/$

C-

>.Cc,C/D/$C

->.d,c/d/$

I47:C-

>d.,c/d/$I89:C-

>Cc.,c/d/$

ParsingTable

state	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

HANDLINGERRORS

The LALR parser may continue to do reductions after the LR parser would have spotted anerror, but the LALR parser will never do a shift after the point the LR parser would havediscoveredtheerror andwill eventually find theerror.

DANGLINGELSE

The dangling else is a problem in computer programming in which an optional else clause inan If–then(–else) statement results in nested conditionals being ambiguous. Formally, thecontext-free grammar of the language is ambiguous, meaning there is more than one correctparsetree.

Inmany programminglanguagesonemay writeconditionallyexecutedcodeintwoforms:theif-then form,andtheif-then-elseform– the elseclauseis optional:

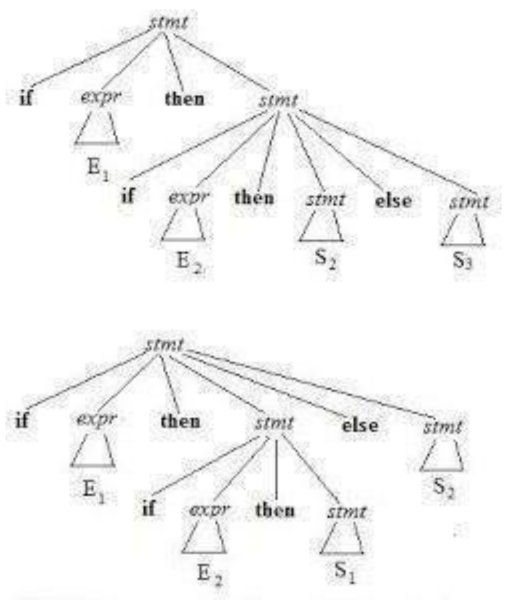


Fig 2.4 Two parse trees for an ambiguous sentence

Considerthegrammar:

S ::=E $

E ::=E +E

|E * E

|(E)

|id

|num

andfour ofits LALR(1) states:

I0:S ::=.E$	?		
E::=. E+E +*$	I1:S ::=E. $	?I2:E ::= E* .E	+*$
E::=. E* E +*$	E::=E . +E +*$	E::=.E+E	+*$
E::=. (E) +*$	E::=E . *E +*$	E ::=. E * E	+*$
E ::=. id+*$		E::=.(E)	+*$
E ::=. num +*$	I3:E ::=E*E . +*$	E::=.id	+*$
	E ::=E . +E +*$	E::=.num	+*$

-

E::=E .* E +*$

Here we have a shift-reduce error. Consider the first two items inI3.If we have a*b+c andwe parsed a*b, do we reduce using E ::= E * E or do we shift more symbols? In the formercase we get a parse tree (a*b)+c; in the latter case we get a*(b+c). To resolve this conflict, wecan specify that * has higher precedence than +. The precedence of a grammar production isequal to the precedence of the rightmost token at the rhs of the production. For example, theprecedence of the production E ::= E * E is equal to the precedence of the operator *, theprecedence of the production E ::= (E) is equal to the precedence of the token), and theprecedence of the production E ::= if E then E else E is equal to the precedence of the tokenelse. The idea is that if the look ahead has higher precedence than the production currentlyused,weshift.Forexample,ifweareparsingE+EusingtheproductionruleE::=E+Eand the look ahead is *, we shift *. If the look ahead has the same precedence as that of thecurrent production and is left associative, we reduce, otherwise we shift. The above grammaris valid if we define the precedence and associativity of all the operators. Thus, it is veryimportant when you write a parser using CUP or any other LALR(1) parser generator tospecifyassociativitiesandprecedence‟sformosttokens(especiallyforthoseusedasoperators). Note: you can explicitly define the precedence of a rule in CUP using the %precdirective:

E::=MINUS E%precUMINUS

where UMINUS is a pseudo-token that has higher precedence than TIMES, MINUS etc, sothat-1*2 is equal to (-1)*2, not to-(1*2).

AnotherthingwecandowhenspecifyinganLALR(1)grammarforaparsergenerator is error recovery. All the entries in the ACTION and GOTO tables that have nocontentcorrespondtosyntaxerrors.Thesimplestthingtodoincaseoferroristoreportitandstopthe parsing.Butwe wouldlike to continue parsingfinding more errors.Thisiscalled*error recovery*. Consider thegrammar:

```
S ::=L =E ;
   |{ SL }
   ;|error ;
SL ::= S ;
    |SLS;
```

The special token error indicates to the parser what to do in case of invalid syntax for S (aninvalid statement). In this case, it reads all the tokens from the input stream until it finds thefirst semicolon. The way the parser handles this is to first push an error state in the stack. Incase of an error, the parser pops out elements from the stack until it finds an error state whereit can proceed. Then it discards tokens from the input until a restart ispossible. Insertingerror handling productions in the proper places in a grammar to do good error recovery isconsideredvery hard.

LRERROR RECOVERY

An LR parser will detect an error when it consults the parsing action table and find ablank or error entry. Errors are never detected by consulting the goto table. An LR parser willdetectanerrorassoonasthereisnovalidcontinuationfortheportionoftheinputthusfar scanned. A canonical LR parser will not make even a single reduction before announcing theerror.SLRandLALRparsersmaymakeseveralreductionsbeforedetectinganerror,buttheywill never shift an erroneous inputsymbol onto thestack.

PANIC-MODEERRORRECOVERY

We can implementpanic-modeerror recovery by scanning down the stack until astate s with a goto on a particular nonterminal A is found. Zero or more input symbols arethendiscardeduntilasymbolaisfoundthatcanlegitimatelyfollowA.Theparserthenstacks the state GOTO(s, A) and resumes normal parsing. The situation might exist wherethere is more than one choice for the nonterminal A. Normally these would be nonterminalsrepresentingmajorprogrampieces,e.g.anexpression,astatement,orablock.Forexample,if A is the nonterminal stmt, a might be semicolon or }, which marks the end of a statementsequence. This method of error recovery attempts to eliminate the phrase containing thesyntactic error. The parser determines that a string derivable from A contains an error. Part ofthat string has already been processed, and the result of this processing is a sequence of stateson top of the stack. The remainder of the string is still in the input, and the parser attempts toskipovertheremainderofthisstringbylookingforasymbolontheinputthatcanlegitimately follow A. By removing states from the stack, skipping over the input,andpushing GOTO(s, A) on the stack, the parser pretends that if has found an instance of A andresumesnormal parsing.

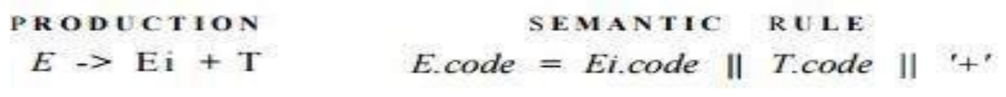
PHRASE-LEVELRECOVERY

Phrase-level recovery is implemented by examining each error entry in the LR actiontable and deciding on the basis of language usage the most likely programmer error thatwould give rise to that error. An appropriate recovery procedure can then be constructed;presumably the top of the stack and/or first input symbol would be modified in a way deemedappropriate for each error entry. In designing specific error-handling routines for an LRparser, we can fill in each blank entry in the action field with a pointer to an error routine thatwilltakethe appropriateaction selected by thecompilerdesigner.

Theactionsmayincludeinsertionordeletionofsymbolsfromthestackortheinputor both, or alteration and transposition of input symbols. We must make our choices so thatthe LR parser will not get into an infinite loop. A safe strategy will assure that at least oneinput symbol will be removed or shifted eventually, or that the stack will eventually shrink ifthe end of the input has been reached. Popping a stack state that covers a non terminal shouldbe avoided, because this modification eliminates from the stack a construct that has alreadybeensuccessfully parsed.

SyntaxDirectedTranslations

Weassociateinformationwithalanguageconstruct byattaching attributes tothegrammar symbol(s)representingtheconstruct, A syntax-directed definition specifies the values of attributes by associatingsemantic rules with the grammar productions. For example, an infix-to-postfix translator might have aproductionandrule



PRODUCTION	SEMANTIC RULE
E -> Ei + T	$E.code$ = $Ei.code$ || $T.code$ || '+'

This production has two nonterminals, E and T; the subscript in E1distinguishes the occurrence of E inthe production body from the occurrence of E as the head. Both E and T have a string-valued attributecode. The semantic rule specifies that the string E.code is formed by concatenating Ei.code, T.code, andthecharacter'+'. Whiletherulemakesitexplicit thatthetranslationofE isbuiltup fromthetranslationsofE1, T,and '+',it maybeinefficientto implementthetranslationdirectlybymanipulating strings.

asyntax-
directedtranslationschemeembedsprogramfragmentscalledsemanticactionswithinproductionbod
ies
Therearetwonotationsforattachingsemantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementationdetails(also called **AttributeGrammars**).

2. **Translation Schemes.** More implementation oriented: Indicate the order in whichsemanticrules areto beevaluated.

SyntaxDirectedDefinitions

SyntaxDirectedDefinitionsareageneralizationof context-freegrammars inwhich:

1. Grammarsymbolshavean associatedsetof**Attributes**;
2. Productions are associated with **Semantic Rules** for computing the values of attributesSuch formalism generates **Annotated Parse-Trees** where each node of the tree is arecord with a field for each attribute (e.g.,X.a indicates the attribute a of the grammarsymbol X).

Thevalue ofan attribute of agrammar symbol atagivenparse-treenodeisdefined byasemantic ruleassociated with theproduction usedat that node.

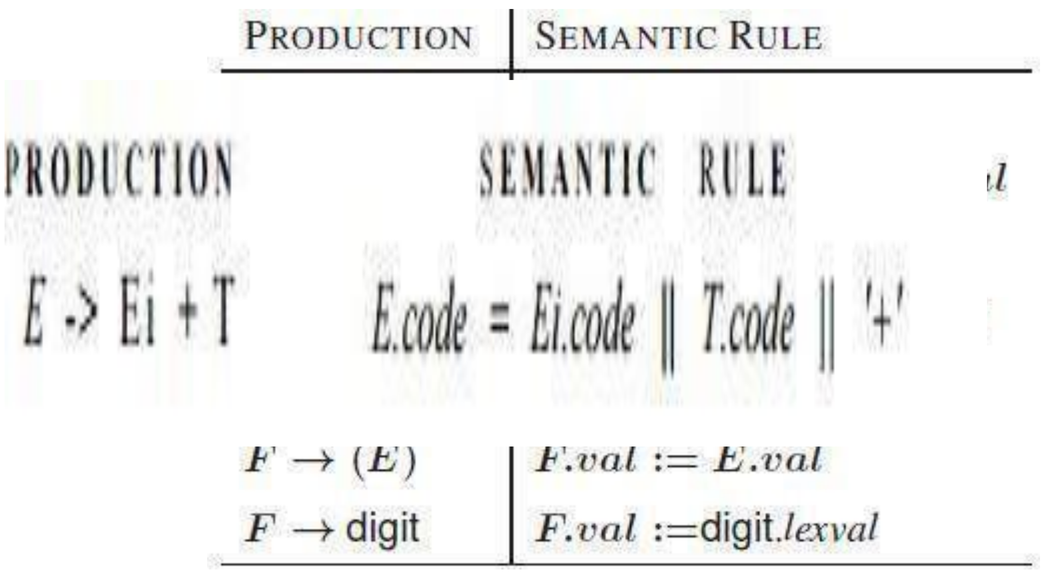
Wedistinguishbetweentwo kindsofattributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of thechildrennodes.

2. **Inherited Attributes.** They are computed from the values of the attributes of both thesiblingsand theparent nodes

SyntaxDirectedDefinitions:AnExample

LetusconsidertheGrammarforarithmeticexpressions.TheSyntaxDirectedDefinitiona ssociates to eachnon terminal asynthesized attributecalled*val*.



PRODUCTION	SEMANTIC RULE
PRODUCTION	SEMANTIC RULE
$E \to Ei + T$	$E.code = Ei.code \parallel T.code \parallel '+'$
$F \to (E)$	$F.val := E.val$
$F \to$ digit	$F.val :=$ digit.*lexval*

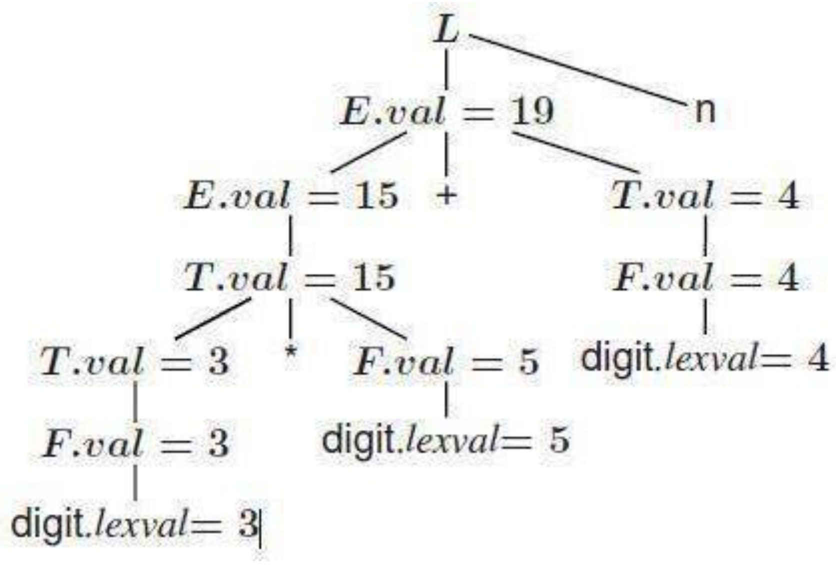
SDDofasimpledeskcalculator

S-ATTRIBUTEDDEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that usesonlysynthesized attributes.

• **Evaluation Order.** Semantic rules in a S-Attributed Definition canbeevaluatedby abottom-up,orPostOrder,traversal oftheparse-tree.

• **Example.** The above arithmetic grammar is an example of an S-AttributedDefinition.Theannotatedparse-treefortheinput3*5+4nis:



L-attributeddefinition

Definition:ASDDits*L-attributed* ifeachinheritedattributeofXiinthe RHSofA! X1:

:Xndepends onlyon

1. attributesof X1;X2;: : :;Xi 1 (symbolsto theleft of Xiin the RHS)
2. inheritedattributesofA.

Restrictionsfortranslation schemes:

1. InheritedattributeofXimustbecomputedbyanaction beforeXi.
2. Anactionmustnotrefer to synthesizedattributeofanysymbol totherightofthataction.
3. Synthesized attribute for A can only be computed after all attributes it references havebeencompleted(usually at end ofRHS).

EvaluationorderofSDTS

1 DependencyGraphs

2 OrderingtheEvaluationofAttributes3

S-Attributed Definitions

4L-AttributedDefinitions

"Dependencygraphs"areausefultoolfordeterminingan evaluationorderfortheattributeinstances in a given parse tree. While an annotated parse tree shows the values of attributes, adependencygraph helps us determinehow thosevalues can becomputed.

1DependencyGraphs

A *dependency graph* depicts the flow of information among the attribute in-stances in aparticular parse tree; an edge from one attribute instance to an-other means that the value of the first isneededtocomputethesecond.Edgesexpress constraintsimpliedby thesemanticrules.Inmoredetail:

Suppose that a semantic rule associated with a production *p* defines the value of inheritedattribute *B.c* in terms of the value of *X.a*. Then, the dependency graph has an edge from *X.a* to *B.c*. Foreach node *N* labeled *B* that corresponds to an occurrence of this *B* in the body of production *p*, create anedge to attribute c at *N* from the attribute *a* at the node *M* that corresponds to this occurrence of *X*. Notethat*M*could be either theparent or asibling of*N*.

Since a node ***N*** can have several children labeled ***X*,** we again assume that subscripts distinguishamonguses of thesame symbol atdifferent placesin theproduction.

Example: Considerthefollowing productionandrule:

PRODUCTION	SEMANTIC RULE
$E \to E_1 + T$	$E.val = E_1.val + T.val$

At every node *N* labeled *E*, with children corresponding to the body of this production, the synthesizedattribute *val* at *N* iscomputedusingthevaluesof *val* atthetwochildren,labeled *E* and *T*. Thus,aportion of the dependency graph for every parse tree in which this production is used looks like Fig. 5.6.As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependencygapharesolid.

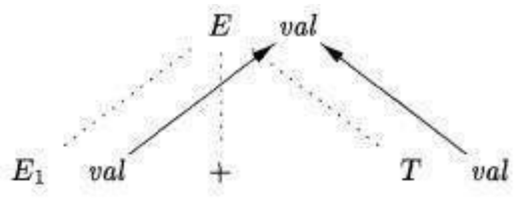


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

2. OrderingtheEvaluationofAttributes

The dependency graph characterizes the possible orders in which we can evalu-ate the attributesat the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, thenthe attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowableorders of evaluation are those sequences of nodes N1, N2,... ,Nk such that if there is an edge of thedependency graph from Ni to Nj, then i < j. Such an ordering embeds a directed graph into a linearorder,and is called atopological sort of thegraph.

Ifthereisanycyclein thegraph, thentherearenotopological sorts;thatis,thereis nowaytoevaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least onetopologicalsort

3. S-AttributedDefinitions

AnSDD is*S-attributed* if every attributeis synthesized.When anSDDisS-attributed, wecanevaluate its attributes in any bottom-up order of the nodes of the parse tree. It is often especially simpleto evaluate the attributes by performing a postorder traversal of the parse tree and evaluating theattributesat a node*N* when the traversalleaves*N* forthe last time.

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parsecorrespondstoapostordertraversal.Specifically,postordercorrespondsexactlytotheorderinwhichanLR parser reduces aproduction body to its head.

4L-AttributedDefinitions

The idea behind this class is that, between the attributes associated with a production body,dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). Moreprecisely,eachattributemust beeither

1. Synthesized,or
2. Inherited,butwith the rules limitedas follows.Supposethatthereisaproduction A->X1X2 .... Xn,andthat thereisan inheritedattributeXi.acomputedbyaruleassociatedwith thisproduction.

Thenthe rule mayuseonly:
Inheritedattributesassociatedwiththehead A.
Either inherited or synthesized attributes associated with the occurrences of symbols X1, X2,... , X(i-1)locatedto the left ofXi.
Inheritedorsynthesizedattributesassociatedwiththisoccurrenceof Xi itself,butonlyinsuchawaythattherereareno cycles in adependency graph formed by theattributes of this X i

ApplicationofSDTS

1ConstructionofSyntaxTrees2
TheStructureofa Type

Themainapplicationistheconstructionofsyntaxtrees.Sincesomecompilersusesyntax treesas an intermediate representation, a common form of SDD turns its input string into a tree. To completethe translation to intermediate code, the compiler may then walk the syntax tree, using another set ofrulesthat areineffect anSDDonthe syntax treeratherthan theparsetree.

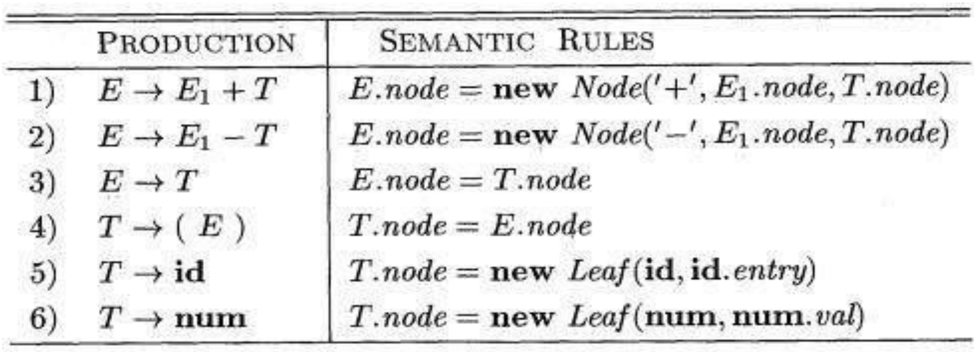
1 ConstructionofSyntaxTrees

Each node in a syntax tree represents a construct; the children of the node represent themeaningfulcomponentsoftheconstruct. Asyntax-treenoderepresentinganexpression*E1*+ E_2haslabel + and twochildrenrepresenting the subexpressions*E1* and E_2
implement the nodes of a syntax tree by objects with a suitable number of fields. Each objectwillhavean *op* field thatis thelabel of thenode.
Theobjects willhave additional fieldsas follows:

• If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function Leaf(op,val)createsaleafobject.Alternatively,ifnodesareviewedasrecords,thenLeafreturnsapointertoa new record for aleaf.

• Ifthenodeisaninteriornode,thereareasmanyadditionalfieldsasthenodehaschildreninthesyntax tree. A constructor function Node takes two or more arguments: Node(op,ci,c2,... ,ck) creates anobjectwith first field opand k additionalfields for thek childrenc1,... ,.

Example



	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

Figure 5.1 1 shows the construction of a syntax tree for the input *a* — 4 + *c*. The nodes of thesyntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines.Theunderlying parsetree, which need not actuallybeconstructed, is shown with dottededges. The third type of line, shown dashed, represents the values of *E.node* and *T-node;* each line points to theappropriatesyntax-treenode.

.

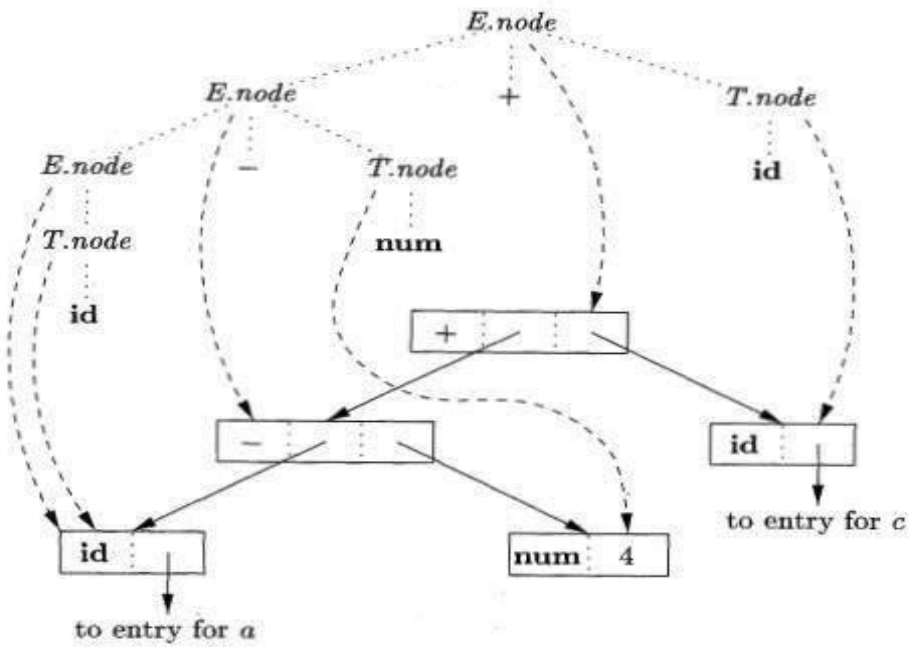


Figure 5.11: Syntax tree for $a - 4 + c$

1) p_1 = **new** *Leaf*(**id**, *entry-a*);
2) p_2 = **new** *Leaf*(**num**, 4);
3) p_3 = **new** *Node*('−', p_1, p_2);
4) p_4 = **new** *Leaf*(**id**, *entry-c*);
5) p_5 = **new** *Node*('+', p_3, p_4);

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

2 TheStructureof aType

The type int [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding typeexpression array(2, array(3, integer)) is represented by the tree in Fig. 5.15. The operator array takestwoparameters,anumberandatype.Iftypes arerepresentedby trees,then thisoperator returnsatreenodelabeledarray with two children for anumber andatype.

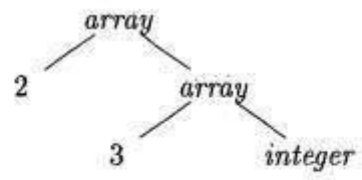


Figure 5.15: Type expression for **int**[2][3]

Nonterminal *B* generates one of the basic types **int** and **float.** *T* generates a basic type when *T* derives *BC* and *C* derives e. Otherwise, *C* generates array components consisting of a sequence of integers, eachintegersurrounded by brackets.

PRODUCTION	SEMANTIC RULES
$T \to B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \to$ **int**	$B.t = integer$
$B \to$ **float**	$B.t = float$
$C \to [$ **num** $]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \to \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

An annotated parse tree for the input string **int [** 2] [3] is shown in Fig. 5.17. The corresponding typeexpression in Fig. 5.15 is constructed by passing the type *integer* from *B*, down the chain of C's throughtheinherited attributes*b.*Thearray typeis synthesized upthe chainof C'sthrough theattributes*t.*

In more detail, at the root for *T* -» *B C*, nonterminal *C* inherits the type from *B*, using the inheritedattribute *C.b.* At the rightmost node for C, the production is C e, so C.t equals C.6.The semantic rulesfor the production C [num] C1 form C.t by applying the operator array to the operands num.ua/ andC1.t.

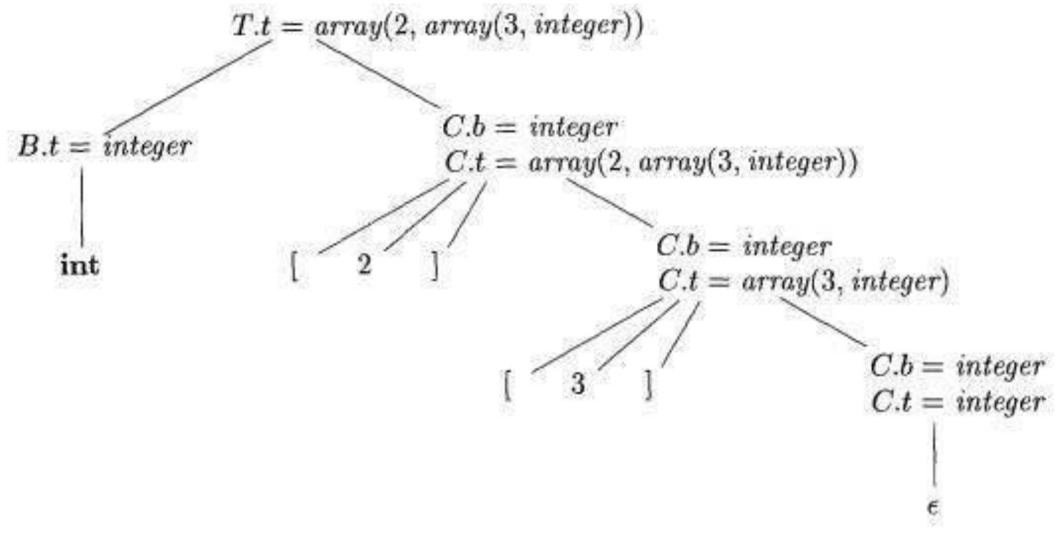


Figure 5.17: Syntax-directed translation of array types

SyntaxDirectedTranslationSchemes.

1Postfix Translation Schemes
2 Parser-Stack Implementation of Postfix
SDT's3SDT's WithActions InsideProductions
4EliminatingLeftRecursion FromSDT's

syntax-directed translation scheme (SDT) is a context-free grammar with program fragmentsembedded within production bodies. The program fragments are called *semantic actions* and can appearat any position within a production body. By convention, we place curly braces around actions; if bracesare needed as grammar symbols, then we quote them.SDT's are implemented during parsing, withoutbuildinga parsetree.
Twoimportantclassesof SDD'sare

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.2.TheunderlyinggrammarisLL-parsable, andtheSDDisL-attributed.

1PostfixTranslationSchemes

simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDDis S-attributed. In that case, we can construct an SDT in which each action is placed at the end of theproduction and is executed along with the reduction of the body to the head of that production. SDT'swithall actions at therightends ofthe productionbodiesare called postfixSDT's.

Example 5.14 : The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with onechange: the action for the first production prints a value. The remaining actions are exact counterparts ofthe semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions canbecorrectly performedalong with thereduction steps of theparser.

$$\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \text{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}$$

Figure 5.18: Postfix SDT implementing the desk calculator

2Parser-StackImplementationofPostfixSDT's

The attribute(s) of each grammar symbol can be put on the stack in a place where they can befoundduringthereduction. Thebestplan istoplacetheattributesalongwith thegrammarsymbols (ortheLR states that represent thesesymbols) in recordson thestack itself.
InFig.5.19,theparserstackcontainsrecordswithafieldforagrammarsymbol(orparserstate)and,belowit,afieldforan attribute.ThethreegrammarsymbolsXYZ areontopofthestack;perhapsthey

are about to be reduced according to a production like *A* —> *X YZ*. Here, we show *X.x* as the oneattribute of *X*, and so on. In general, we can allow for more attributes, either by making the records largeenough or by putting pointers to records on the stack. With small attributes, it may be simpler to makethe records large enough, even if some fields go unused some of the time. However, if one or moreattributes are of unbounded size — say, they are character strings — then it would be better to put apointer to the attribute's value in the stack record and store the actual value in some larger, sharedstorageareathat is not part ofthe stack.

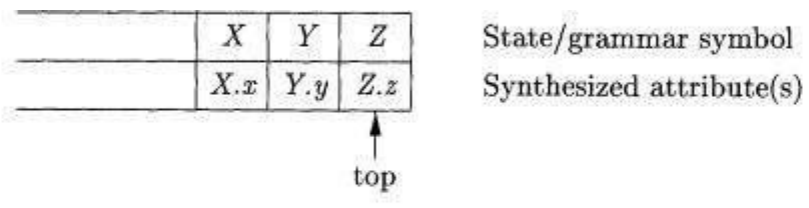


Figure 5.19: Parser stack with a field for synthesized attributes

3 SDT'sWithActionsInsideProductions

An action may be placed at any position within the body of a production.It is performed immediatelyafter all symbols to its left are processed. Thus,if we have a production B -» X {a} Y, the action a isdone after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is anonterminal).

Moreprecisely,

• If the parse is bottom-up, then we perform action a as soon as this occurrence of X appears on the topoftheparsing stack.

• If the parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y anonterminal)orcheck for Yon the input (if Yis aterminal).

4 EliminatingLeftRecursion FromSDT's

First, consider the simple case, in which the only thing we care about is the order in which theactionsin anSDTareperformed.For example,if each actionsimply printsastring, wecareonlyabouttheorder in whichthestrings areprinted.In this case,the following principlecan guide us:

Whentransforming thegrammar,treat theactionsas if theywereterminal symbols.

This principle is based on the idea that the grammar transformation preserves the order of the terminalsin the generated string. The actions are therefore executed in the same order in any left-to-right parse,top-downorbottom-up.

The"trick" foreliminating leftrecursionis totaketwoproductions

A-> *A*a|b

that generate strings consisting of a *j3* and any number of en's, and replace them by productions thatgeneratethe same stringsusing anewnonterminal*R* (for"remainder")of thefirst production:

A->b*R*

R—»•a*R*|e

If*(3*doesnotbeginwith *A*,then*A*nolongerhasaleft-recursiveproduction.Inregular-definitionterms,withboth sets of productions, *A*is defined by***0****(a)**.

Example**5** .17:ConsiderthefollowingE-productionsfromanSDTfortranslatinginfixexpressionsinto postfix notation:

E -> Ei+T{print('+');}E -

> T

Ifweapply thestandard transformationtoE,theremainder oftheleft-recursiveproduction isa

= +T {print('-r'); }

and thebodyoftheotherproductionisT.IfweintroduceRfortheremainderofE,wegetthesetofproductions:

E -->TR

R --> + T { printC-h'); }

RR->e

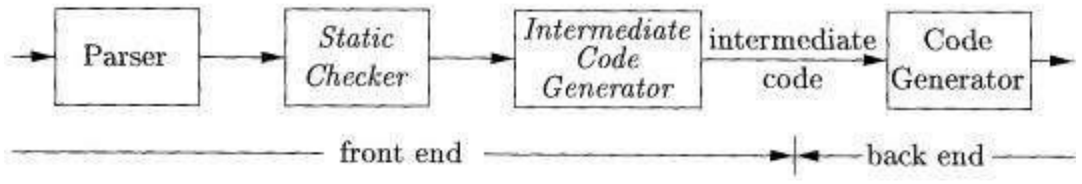
When the actions of an SDD compute attributes rather than merely printing output, we must be morecarefulabouthowweeliminateleftrecursionfromagrammar.However,iftheSDDisS-attributed,then we can always construct an SDT by placing attribute-computing actions at appropriate positions inthenew productions.

UNIT–III
IntermediatedCode:GenerationVariantsofSyntaxtrees3Addresscode,TypesandDeceleration,Translationof Expressions,TypeChecking.CantedFlowBackpatching?

INTERMEDIATECODE

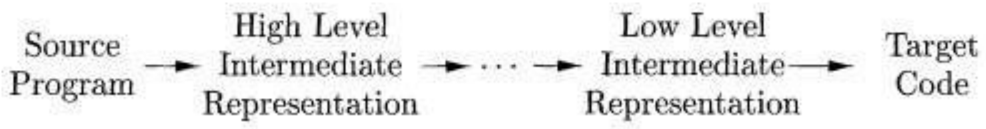
Intheanalysis-synthesismodelofacompiler,thefrontendanalyzesasourceprogram and creates an intermediate representation, from which the back end generates targetcode. This facilitates *retargeting*: enables attaching a back end for the new machine to anexistingfront end.

LogicalStructureofa CompilerFrontEnd



Acompilerfrontendisorganizedasinfigureabove,whereparsing,staticchecking, and intermediate-code generation are done sequentially; sometimes they can becombined and folded into parsing. All schemes can be implemented by creating a syntaxtreeand traversing the tree.

Static checking includes *type checking,* which ensures that operators are applied to compatibleoperands.Intheprocessoftranslatingaprogram inagiven sourcelanguageintocodefor agiventargetmachine,a compilerconstruct a sequenceofintermediate representations


Source Program
High Level Intermediate Representation
Low Level Intermediate Representation
Target Code

Sequenceofintermediaterepresentations

High-levelrepresentationsareclosetothesourcelanguageandlow-levelrepresentationsareclosetothe target machine. A low-level representation is suitable for machine-dependent tasks like registerallocationand instruction selection.

JSVGKrishna,AssociateProfessor.

VariantsofSyntax Trees

1 DirectedAcyclicGraphsforExpressions

2 TheValue-NumberMethod forConstructing DAG's

1.DirectedAcyclicGraphsforExpressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomicoperands and interior codes corresponding to operators. The difference is that a node *N* in a DAG hasmore than one parent if *N* represents a com-mon subexpression; in a syntax tree, the tree for thecommon subexpression would be replicated as many times as the subexpression appears in the originalexpression.

Example: Considerexpression

a +a*(b-c)   +   (b- c)*d

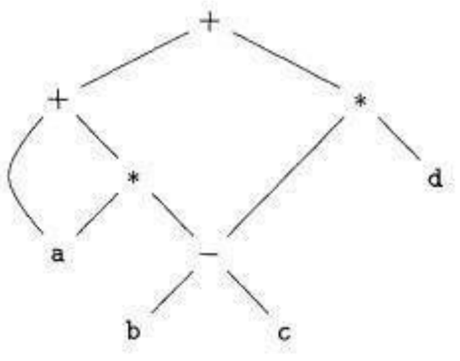
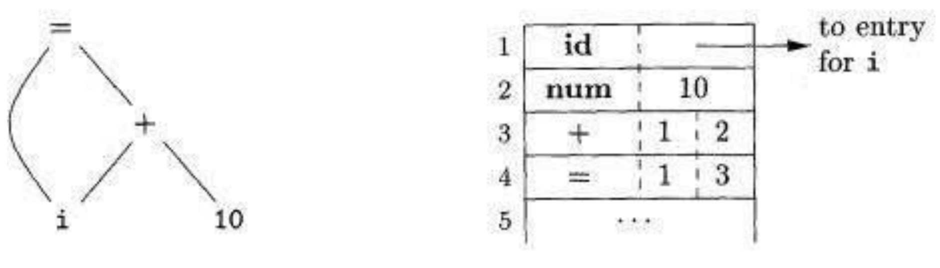


Figure 6.3: Dag for the expression a + a * (b - c) + (b - c) * d

2TheValue-NumberMethodforConstructingDAG's

The nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6.Each row of the array represents one record, and therefore one node. In each record, the first field is anoperation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, whichholds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes havetwoadditional fields indicating the left andright children.



(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

Inthisarray,werefertonodesbygivingtheintegerindex oftherecordfor thatnodewithinthearray. This integer iscalled the value numberfor thenode.

Algorithm: Thevalue-numbermethod forconstructingthe nodesof

aDAG.INPUT: Label op, node /, and noder.

OUTPUT: Thevalue numberof anodein thearray with signature(op,l,r).

METHOD : Search the array for a node M with label op, left child I, and right child r. If there is such

anode,returnthevaluenumberofM.Ifnot,createinthearrayanewnodeNwithlabelop, leftchildI,andrightchildr,andreturn its valuenumber.

Three-AddressCode

1 Addresses and

Instructions2Quadruples

3Triples

In three-address code, there is at most one operator on the right side of an instruction; that is, nobuilt-up arithmetic expressions are permitted. Thus a source-language expression like x+y*z might betranslatedinto the sequenceof three-address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

wheretiand$t_2$arecompiler-generatedtemporary names.

1AddressesandInstructions

Anaddresscanbeoneof thefollowing:

- *A name. S*ource-program names to appear as addresses in three-address code. In animplementation, a source name is replaced by a pointer to its symbol-table entry,whereall information about the name is kept.
- *Aconstant.*Acompilermustdealwithmanydifferenttypesof constantsandvariables.
- *A compiler-generated temporary*. Useful in optimizing com-pilers, to create a distinct name eachtime a temporary is needed. These temporaries can be combined, if possible, when registers areallocatedto variables.

commonthree-addressinstruction

1. **AssignmentStatement:** x=yopz andx=opy
   Here,x, yandzaretheoperands.oprepresentstheoperator.
2. **CopyStatement:** x=y
3. **ConditionalJump:** IfxrelopygotoX

Ifthe condition“x relop y”gets**satisfied,** then-

Thecontrolissentdirectlytothelocationspecifiedbylabel X.Allthe statements in between areskipped.

Ifthecondition“x relopy”**fails**,then-

Thecontrolis notsentto thelocationspecified bylabelX.
Thenext statement appearing intheusual sequenceis executed.

4. Unconditional Jump-gotoX

Onexecutingthestatement,Thecontrolissentdirectlytothelocationspecifiedbylabel X.

Allthestatementsin betweenareskipped.

5. Procedure Call-paramxcallpreturny

Here,pisafunctionwhichtakes xasaparameterandreturnsy.
Foraprocedurecallp(x1, …, xn)

```
                paramx1
                   …
                 paramxn
                 callp, n
```

6.Indexedcopyinstructions:x=y[i]and x[i]=y
Left: sets x to the value in the location i memory units beyond yRight:setsthecontentsofthelocationimemoryunitsbeyondxtoy

7.Addressandpointerinstructions:

x=&y setsthevalue of xto be thelocation(address) ofy.

x = *y, presumably y is a pointer or temporary whose value is alocation.Thevalueofxissettothecontentsofthatlocation.

*x=y setsthevalue of theobject pointed toby x to thevalue ofy.

DataStructure

Threeaddresscodeisrepresentedasrecordstructurewithfieldsforoperatorandoperands.Theserecordscanbestoredasarrayorlinkedlist.Mostcommonimplementationsofthreeaddresscodeare Quadruples,Triples andIndirecttriples.

2. Quadruples

Quadruples consists of four fields in the record structure. One field to store operator op, twofieldsto storeoperands or argumentsarg1and arg2 andonefield tostore result res.

res = arg1 op

arg2Example:a =b +c

bisrepresented asarg1, cisrepresentedasarg2, +as opandaasres.

Unary operators like „-„do not use agr2. Operators like param do not use agr2 nor result. Forconditional and unconditional statements res is label. Arg1, arg2 and res are pointers tosymboltable orliteral table for thenames.

Example:a =-b *d+c +(-b)* d

Three address code for the above statement is as followst1 =-b

t2 = t1 *

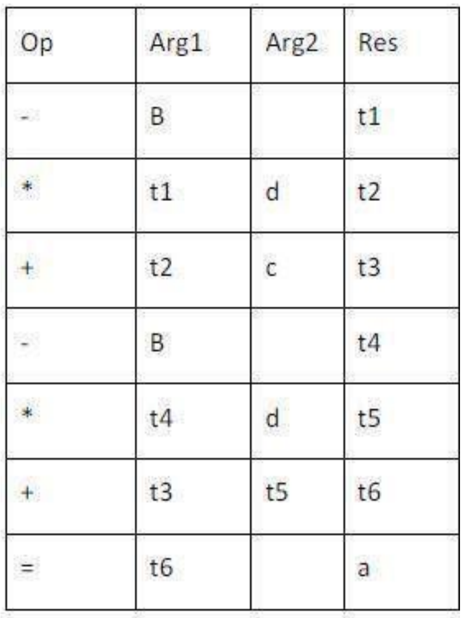
dt3 = t2 +

ct4 =-b

t5 = t4 * d

t6 = t3 +
t5a=t6

Quadruplesfortheabove exampleis asfollows



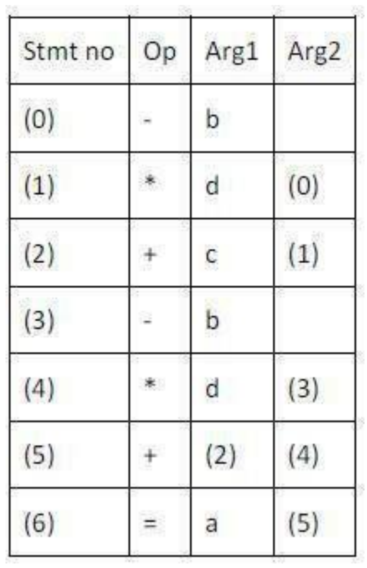
Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

3TRIPLES

Triples uses only three fields in the record structure. One field for operator, two fields foroperands named as arg1 and arg2. Value of temporary variable can be accessed by thepositionofthe statement thecomputes itand not by location as inquadruples.

Example:a =-b *d+c +(-b)* d

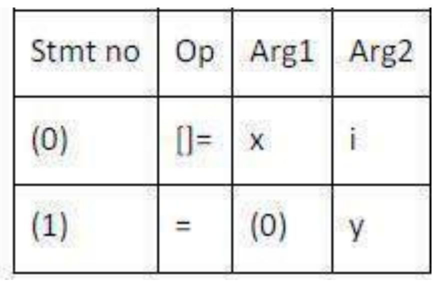
Triplesforthe aboveexampleis as follows



Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

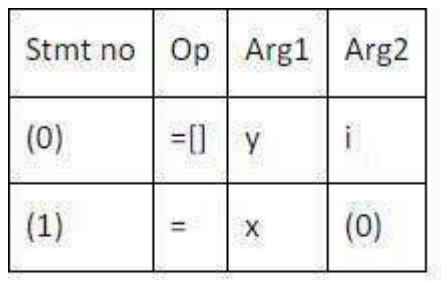
Arg1andarg2maybepointerstosymboltableforprogramvariablesorliteraltableforconst ant or pointers into triplestructureforintermediate results.

Example:Triples forstatement x[i] =y which generatestwo recordsis as follows



Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

Triplesforstatementx=y[i]whichgeneratestworecordsisasfollows



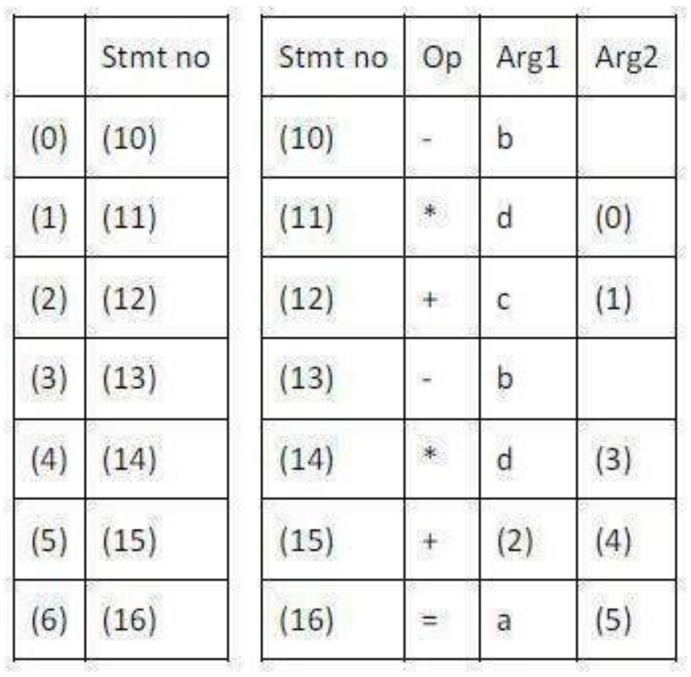
Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

Triplesarealternative waysforrepresentingsyntaxtreeorDirectedacyclic graphforprogram defined names.

IndirectTriples

Indirecttriplesareusedtoachieveindirectionin listingofpointers.Thatis,itusespointerstotriples than listing of triples themselves.

Example:a =-b *d+c +(-b)* d



	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

TypesandDeclarations

1 TypeExpressions
2 TypeEquivalence
3 Declarations
4 StorageLayoutforLocalNames

1TypeExpressions

Types have structure, which we shall represent using *type expressions:* a type expression is either abasictypeoris formed byapplying an operatorcalled a *typeconstructor*toatypeexpression.

Definition

- Abasictypeisatypeexpression.Typicalbasictypesforalanguageinclude*boolean,char,integer,float,*a nd *void;*the latterdenotes "theabsenceofavalue."
- Atypename isatypeexpression.
- Atypeexpressioncanbe formed byapplying the arraytypeconstructortoanumberandatype
- expression.
- Arecordisadatastructurewithnamedfields.Atypeexpressioncanbeformedbyapplyingthe*record* type constructor to the field names andtheirtypes.
- If *s* and *t* are type expressions, then their Cartesian product *s* x *t* is a type expression. Productsare introduced for completeness; they can be used to represent a list or tuple of types (e.g., forfunctionparameters).
- Typeexpressions may containvariables whosevaluesaretypeexpressions

2TypeEquivalence

Many type-checking rules have the form, "if two type expressions are equal then return a certaintype else error." Potential ambiguities arise when names are given to type expressions. The key issue iswhether a name in a type expression stands for itself or whether it is an abbreviation for another typeexpression.

Since type names denote type expressions, they can set up implicit cycles; see the box on "TypeNames and Recursive Types." If edges to type names are redirected to the type expressions denoted bythenames,then theresulting graph can havecycles dueto recursivetypes.

When type expressions are represented by graphs, two types are *structurally equivalent* if and only ifoneof thefollowing conditions is true:

Theyarethe samebasic type.
They are formed by applying the same constructor to structurally equivalent types.Oneis a typename that denotes the other.

Iftypenamesaretreatedasstandingforthemselves,thenthefirsttwoconditionsintheabovedefinitionlead to *nameequivalence* of typeexpressions.

Name-equivalent expressions are assigned thesame value number,. Structural equivalence can betestedusing theunification algorithm .

3. Declarations

Understand types and declarations using a simplified grammar that declares just one name at a time;Thegrammar is

$$\begin{array}{lll} D & \rightarrow & T\ \textbf{id}\ ;\ D\ \mid\ \epsilon \\ T & \rightarrow & B\ C\ \mid\ \textbf{record}\ '\{'\ D\ '\}' \\ B & \rightarrow & \textbf{int}\ \mid\ \textbf{float} \\ C & \rightarrow & \epsilon\ \mid\ [\ \textbf{num}\ ]\ C \end{array}$$

The fragment of the above grammar that deals with basic and array types.Consider storage layout aswellastypes.Nonterminal *D* generatesasequenceofdeclarations.Nonterminal *T* generatesbasic,array, or record types. Nonterminal *B* generates one of the basic types int and float. Nonterminal C, for"component," generates strings of zero or more integers, each integer surrounded by brackets. An arraytype consists of a basic type specified by *B*, followed by array components specified by nonterminal *C.A* record type (the second production for *T)* is a sequence of declarations for the fields of the record, allsurroundedby curly braces.

4. StorageLayoutfor LocalNames

From the type of a name, we can determine the amount of storage that will be needed for thename at run time. At compile time, we can use these amounts to assign each name a relative address.The type and relative address are saved in the symbol-table entry for the name. Data of varying length,such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, ishandledby reserving aknown fixed amount of storagefor apointer to thedata.

AddressAlignment

The storage layout for data objects is strongly influenced by the address-ing constraints of the targetmachine. For example, instructions to add integers may expect integers to be *aligned*, that is, placed atcertain positions in memory such as an address divisible by 4. Although an array of ten characters needsonly enough bytes to hold ten characters, a compiler may therefore allocate 12 bytes— the nextmultiple of 4 — leaving 2 bytes unused. Space left unused due to alignment considerations is referred toas *padding.* When space is at a premium, a compiler may *pack* data so that no padding is left; additionalinstructions may then need to be executed at run time to position packed data so that it can be operatedonas if it wereproperlyaligned.

Suppose that storage comes in blocks of contiguous bytes, where a byte is the smallest unit ofaddressable memory. The *width* of a type is the number of storage units needed for objects of that type.A basic type, such as a character, integer, or float, requires an integral number of bytes. For easy access,storagefor aggregatessuch as arraysandclasses isallocated in onecontiguous block ofbytes.

The translation scheme (SDT) computes types and their widths for basic and array types; The SDTuses synthesized attributes *type* and *width* for each nonterminal and two variables *t* and *w* to pass typeand width information from a *B* node in a parse tree to the node for the production *C* —> e. In a syntax-directeddefinition, *t* and *w* would beinherited attributes for*C*.

The body of the T-production consists of nonterminal B, an action, and nonterminal C, whichappears on the next line. The action between B and C sets t to B.type and w to B. width. If B —>• intthen B.type is set to integer and B.width is set to 4, the width of an integer. Similarly, if B -+ float thenB.typeis float and B.width is 8, the width of afloat.

The productions for C determine whether T generates a basic type or an array type. If C —>• e,then t becomes C.type and w becomes C. width. Otherwise, C specifies an array component. The actionfor C —> [n u m] C1 forms C.type by applying the type constructor array to the operands num.valueand C1.type.

$T \rightarrow B$	$\{ t = B.type; w = B.width; \}$
$\quad\quad C$	
$B \rightarrow$ **int**	$\{ B.type = integer; B.width = 4; \}$
$B \rightarrow$ **float**	$\{ B.type = float; B.width = 8; \}$
$C \rightarrow \epsilon$	$\{ C.type = t; C.width = w; \}$
$C \rightarrow$ [**num**] C_1	$\{ array(\mathbf{num}.value,\ C_1.type);$ $C.width = \mathbf{num}.value \times C_1.width; \}$

Figure 6.15: Computing types and their widths

The width of an array is obtained by multiplying the width of an element by the number ofelements in the array. If addresses of consecutive integers differ by 4, then address calculations for anarrayofintegerswillincludemultiplicationsby4.Suchmultiplicationsprovideopportunitiesforoptimization ,so it is helpful for thefront endto makethem explicit.

ExampleThe parse tree for the type i n t [2] [3] is shown by dotted lines in Fig. 6.16. The solid linesshow how the type and width are passed from B, down the chain of C's through variables t and w, andthen back up the chain as synthesized attributes type and width. The variables t and w are assigned thevalues of B.type and B.width, respectively, before the subtree with the C nodes is examined. The valuesof t and w are used at the node for C —> e to start the evaluation of the synthesized attributes up thechainof C nodes.

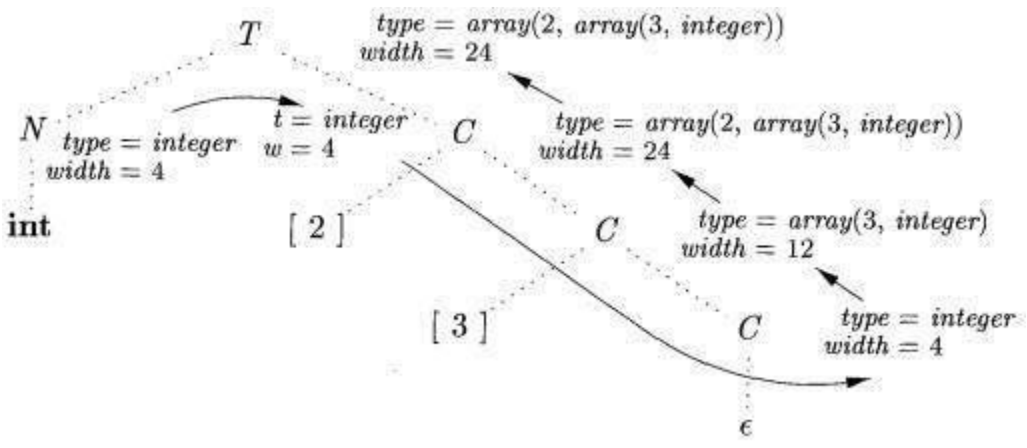


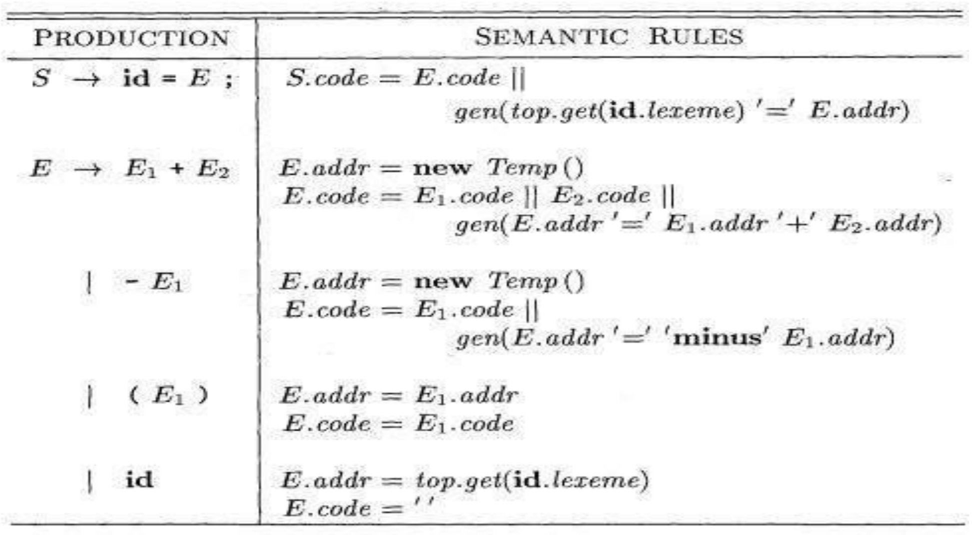
Figure 6.16: Syntax-directed translation of array types

TranslationsofExpressions

1OperationsWithinExpressions2
Incremental Translation
3 AddressingArrayElements
4 TranslationofArrayReferences

1OperationsWithinExpressions

The syntax-directed definition builds up the three-address code for an assignment statement *S* usingattribute *code* for *S* and attributes *addr* and *code* for anexpression *E.* Attributes *S.code* and *E.code*denote the three-address code for *S* and *E*, respectively. Attribute *E.addr* denotes the address that willhold thevalue ofE.



PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code$ || $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code$ || $E_2.code$ || $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ -E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code$ || $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Figure 6.19: Three-address code for expressions

Example The syntax-directed definition in Fig. 6.19 translates the assignment statementa=b+-c;intotheTAC

```
t1 = minus c
t2 = b + t1
a = t2
```

2IncrementalTranslation

Code attributes can be long strings, so they are generated incrementally In the incrementalapproach, *gen* not only constructs a three-address instruction, it appends the instruction to the sequenceof instructions generated so far. The sequence may either be retained in memory for further processing,or it may be output incrementally.attribute addr represents the address of a node rather than a variable orconstant.

$S \rightarrow \textbf{id} = E\ ;$	$\{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \}$
$E \rightarrow E_1 + E_2$	$\{\ E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$
$\mid\ -E_1$	$\{\ E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \}$
$\mid\ (\ E_1\ )$	$\{\ E.addr = E_1.addr;\ \}$
$\mid\ \textbf{id}$	$\{\ E.addr = top.get(\textbf{id}.lexeme);\ \}$

Figure 6.20: Generating three-address code for expressions incrementally

3.AddressingArrayElements

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutivelocation.Arraycan beonedimensional ortwo dimensional.

Foronedimensionalarray:

A:array[low..high]oftheithelementsisat:

base+(i-low)*width=i*width +(base-low*width)

Multi-dimensionalarrays:

Rowmajororcolumnmajorforms

- oRow major:a[1,1], a[1,2],a[1,3], a[2,1],a[2,2], a[2,3]
- oColumnmajor:a[1,1],a[2,1],a[1,2],a[2,2],a[1, 3],a[2,3]
- o Inrowmajor form,theaddressofa[i1, i2]is
- o Base+((i1-low1)*(high2-low2+1)+i2-low2)*width

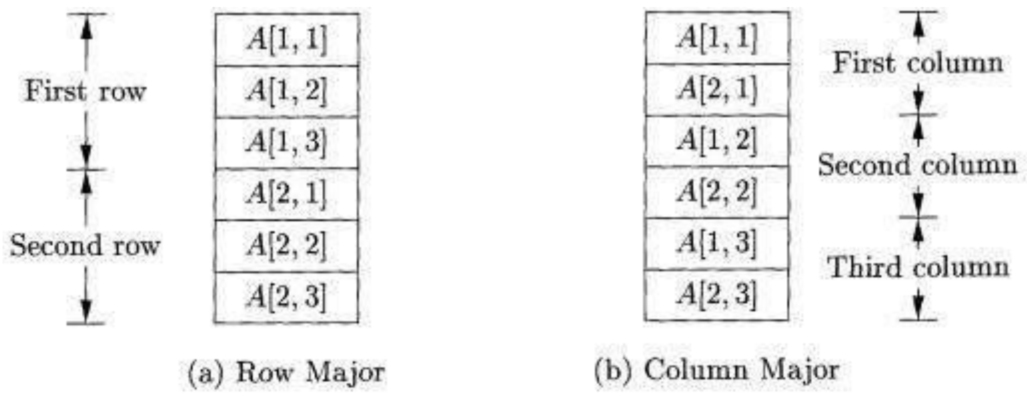


Figure 6.21: Layouts for a two-dimensional array.

4TranslationofArrayReferences

*L*generate anarraynamefollowedbyasequenceofindexexpressions:

$$L \rightarrow L\,[\,E\,]\ \mid\ \mathbf{id}\,[\,E\,]$$

Calculate addresses based on widths, using the formularather than on numbers of elements. Thetranslation scheme generates three-address code for expressions with array references. It consists of theproductionsand semanticactions together with productionsinvolving nonterminal

```
S → id = E ;     { gen( top.get(id.lexeme) '=' E.addr); }

  | L = E ;      { gen(L.addr.base '[' L.addr ']' '=' E.addr); }

E → E1 + E2      { E.addr = new Temp ();
                   gen(E.addr '=' E1.addr '+' E2.addr); }

  | id           { E.addr = top.get(id.lexeme); }

  | L            { E.addr = new Temp ();
                   gen(E.addr '=' L.array.base '[' L.addr ']'); }

L → id [ E ]     { L.array = top.get(id.lexeme);
                   L.type = L.array.type.elem;
                   L.addr = new Temp ();
                   gen(L.addr '=' E.addr '*' L.type.width); }

  | L1 [ E ]     { L.array = L1.array;
                   L.type = L1.type.elem;
                   t = new Temp ();
                   L.addr = new Temp ();
                   gen(t '=' E.addr '*' L.type.width); }
                   gen(L.addr '=' L1.addr '+' t); }
```

Figure 6.22: Semantic actions for array references

.

TypeChecking

1RulesforTypeChecking2

TypeConversions

3 Overloadingof FunctionsandOperators

4 TypeInferenceandPolymorphicFunctions5

An Algorithm forUnification

Typecheckingacompiler needstoassignatype expressiontoeachcomponentofthesourceprogram. The compiler must then determine that these type expressions conform to a collection oflogicalrules that is calledthetypesystemforthe sourcelanguage.

1 Rules forTypeChecking

Type checking can take on two forms: **synthesis and inference**. *Type synthesis* builds up the typeof an expression from the types of its subexpressions. It requires names to be declared before they areused. The type of *E1* +*E*₂ is defined in termsof the types of *E1* and *E*₂ • A typical rule for typesynthesishas the form

$$\begin{aligned}&\textbf{if } f \text{ has type } s \to t \textbf{ and } x \text{ has type } s,\\&\textbf{then } \text{expression } f(x) \text{ has type } t\end{aligned} \qquad (6.8)$$

Typeinferencedeterminesthetypeofalanguageconstructfromthewayitisused.Rulefortypeinferencehas theform

$$\begin{aligned}&\textbf{if } f(x) \text{ is an expression,}\\&\textbf{then } \text{for some } \alpha \text{ and } \beta,\ f \text{ has type } \alpha \to \beta \textbf{ and } x \text{ has type } \alpha\end{aligned} \qquad (6.9)$$

2 TypeConversions

integersareconvertedtofloatswhennecessary,usingaunaryoperator(float).Forexample,theinteger**2** is converted to a floatin thecodefortheexpression**2*3.14:**

```
t1 = (float) 2
t2 = t1 * 3.14
```

Type conversion rules vary from language to language. The rules for Java in Fig. 6.25 distinguishbetween *widening* conversions, which are intended to preserve information, and *narrowing* conversions,whichcan lose information.
Conversionfromonetypetoanotherissaidtobe*implicit* ifitisdoneautomaticallybythecompiler.Implicittype conversions, also called*coercions*,

$$\begin{tikzcd}
 & double \arrow[d, dash] & \\
 & float \arrow[d, dash] & \\
 & long \arrow[d, dash] & \\
 & int \arrow[dl, dash] \arrow[dr, dash] & \\
short \arrow[d, dash] & & char \\
byte & &
\end{tikzcd}$$

(a) Widening conversions

$$\begin{tikzcd}
 & double \arrow[d] & \\
 & float \arrow[d] & \\
 & long \arrow[d] & \\
 & int \arrow[dl] \arrow[d] \arrow[dr] & \\
char \arrow[r, leftrightarrow] \arrow[rr, leftrightarrow, bend right] & short \arrow[r, leftrightarrow] & byte
\end{tikzcd}$$

(b) Narrowing conversions

Figure 6.25: Conversions between primitive types in Java

3 OverloadingofFunctionsandOperators

An *overloaded* symbol has different meanings depending on its context. The + operator in Javadenoteseither string concatenation oraddition

Type-synthesisruleforoverloaded functions:

$$\begin{aligned}
&\textbf{if } f \text{ can have type } s_i \rightarrow t_i, \text{ for } 1 \leq i \leq n, \text{ where } s_i \neq s_j \text{ for } i \neq j \\
&\textbf{and } x \text{ has type } s_k, \text{ for some } 1 \leq k \leq n \\
&\textbf{then } \text{expression } f(x) \text{ has type } t_k
\end{aligned} \qquad (6.10)$$

4 TypeInferenceandPolymorphicFunctions

TypeinferenceisusefulforalanguagelikeML,whichisstronglytyped,butdoesnotrequirenamesto bedeclared beforetheyareused.Typeinferenceensuresthat namesareusedconsistently.

Theterm"polymorphic"referstoanycodefragmentthatcanbeexecutedwithargumentsofdifferenttypes.

Thetypeoflengthcanbedescribedas,"foranytypea,lengthmapsalistofelementsoftypeatoaninteger."

```
fun length(x) =
        if null(x) then 0 else length(tl(x)) + 1;
```

Figure 6.28: ML program for the length of a list

The program fragment defines function *length* with one parameter *x*. The body of the function consistsofaconditionalexpression.Thepredefinedfunction *null* testswhetheralistisempty,andthepredefinedfunction*tl* (shortfor "tail")returns the remainder ofalistafter thefirst elementis removed.

5 AnAlgorithm forUnification

Unification is the problem of determining whether two expressions s and t can be made identicalby substituting expressions for the variables in s and t. Testing equality of expressions is a special caseof unification; if s and t have constants but no variables, then s and t unify if and only if they areidentical.so it can beusedto test structural equivalenceofcircular types .[7]

Graph-theoreticformulationofunification,wheretypesarerepresentedbygraphs.Typevariables are represented by leaves and type constructors are represented by interior nodes. Nodes aregroupedintoequiv-alenceclasses; iftwonodes areinthesameequivalenceclass,thenthetypeexpressions they represent must unify. Thus, all interior nodes in the same class must be for the sametypeconstructor, and their correspondingchildrenmust beequivalent.

Example6.18:Consider thetwotypeexpressions

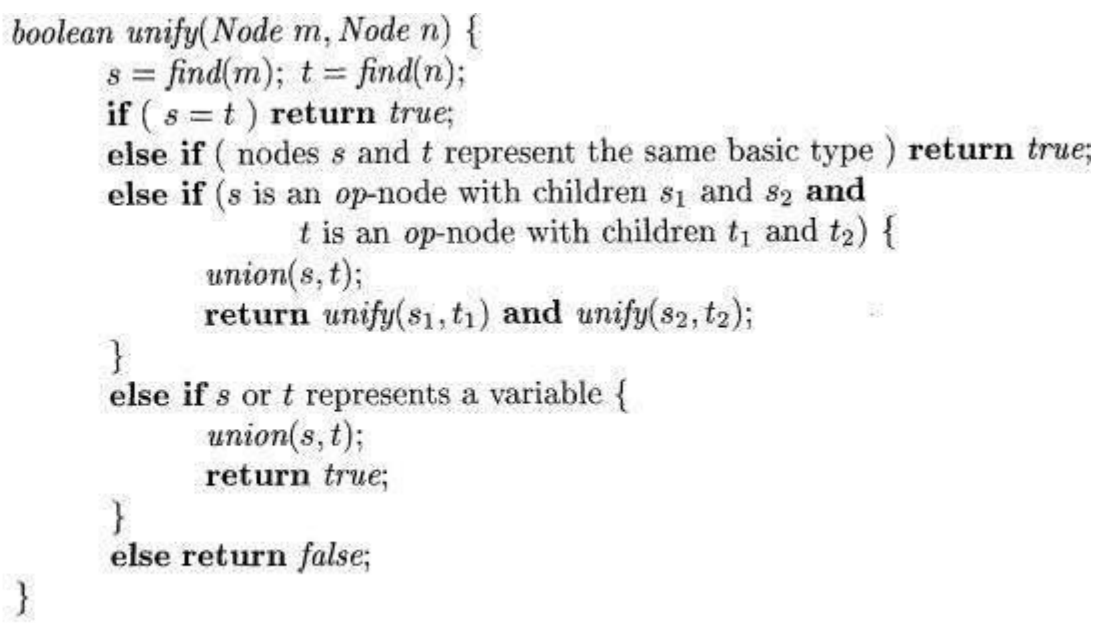
$$((\alpha_1 \to \alpha_2) \times list(\alpha_3)) \to list(\alpha_2)$$
$$((\alpha_3 \to \alpha_4) \times list(\alpha_3)) \to \alpha_5$$

The following substitution S is the most general unifier for these expressions

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

This substitution maps the two type expressions to the following expression

$$((\alpha_1 \to \alpha_2) \times list(\alpha_1)) \to list(\alpha_2)$$



```
boolean unify(Node m, Node n) {
        s = find(m);  t = find(n);
        if ( s = t ) return true;
        else if ( nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s_1 and s_2 and
                        t is an op-node with children t_1 and t_2) {
                union(s, t);
                return unify(s_1, t_1) and unify(s_2, t_2);
        }
        else if s or t represents a variable {
                union(s, t);
                return true;
        }
        else return false;
}
```

Figure 6.32: Unification algorithm.

Theunificationalgorithm,usesthefollowingtwooperationson nodes:

find{n)returns therepresentative nodeofthe equivalence class currently containingnode *n.*

union(m, n) merges the equivalence classes containing nodes *m* and n. If one of the representativesfor the equivalence classes of m and n is a non-variable node, *union* makes that nonvariable node bethe representative for the merged equivalence class; otherwise, *union* makes one or the other of theoriginal representatives be the new representative. This asymme-try in the specification of *union* isimportant because a variable cannot be used as the representative for an equivalence class for anexpression containing a type constructor or basic type. Otherwise, two inequivalent expressions maybeunified through that variable.

ControlFlow

1 BooleanExpressions

2 Short-CircuitCode

3 Flow-of-ControlStatements

4 Control-Flow Translation of Boolean

ExpressionsInprogramminglanguages,booleanexpressionsareoften

usedto

1. **Alterthe flowofcontrol**.Booleanexpressionsareusedasconditionalexpressionsinstatementsthat alter the flow of control. The value of such boolean expressions is implicit in a position reached in aprogram.For example, in if*(E)*5, the expression*E*must betrueif statement*S* isreached.

2. **Computelogicalvalues.**Aboolean expressioncanrepresent*true*Or*false*asvalues.Suchboolean

expressions can be evaluated in analogy to arithmetic expressions using three-address instructionswithlogical operators.

1 BooleanExpressions

Boolean expressions are composed of the boolean operators (which we denote &&, I I, and !, using theC convention for the operators AND, OR, and NOT, respectively) applied to elements that are booleanvariablesorrelationalexpressions. Booleanexpressionsgeneratedby the following grammar:

$$B \rightarrow B \mid\mid B \mid B \,\&\&\, B \mid\ !B \mid (B) \mid E\ \textbf{rel}\ E \mid \textbf{true} \mid \textbf{false}$$

Given the expression Bi I I B2, if we determine that B1 is true, then we can conclude that the entireexpression is true without having to evaluate B2.Similarly, given B1 && B2, if Bi is false, then theentireexpression is false.

2 Short-CircuitCode

In *short-circuit* (or *jumping)* code,thebooleanoperators&&,II,and!translateintojumps.Theoperators themselves do not appear in the code; instead, the value of a boolean expression is representedbyaposition in thecodesequence.

Example The statement

i f (x <100||x >200 && x!=y)x =0;

might be translated into the code of Fig. **6.34.** In this translation, the boolean expression is true if controlreacheslabel *L2*. Iftheexpressionisfalse,controlgoesimmediatelyto *Lu* skipping *L2* andtheassignment x=**0.**

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Figure 6.34: Jumping code

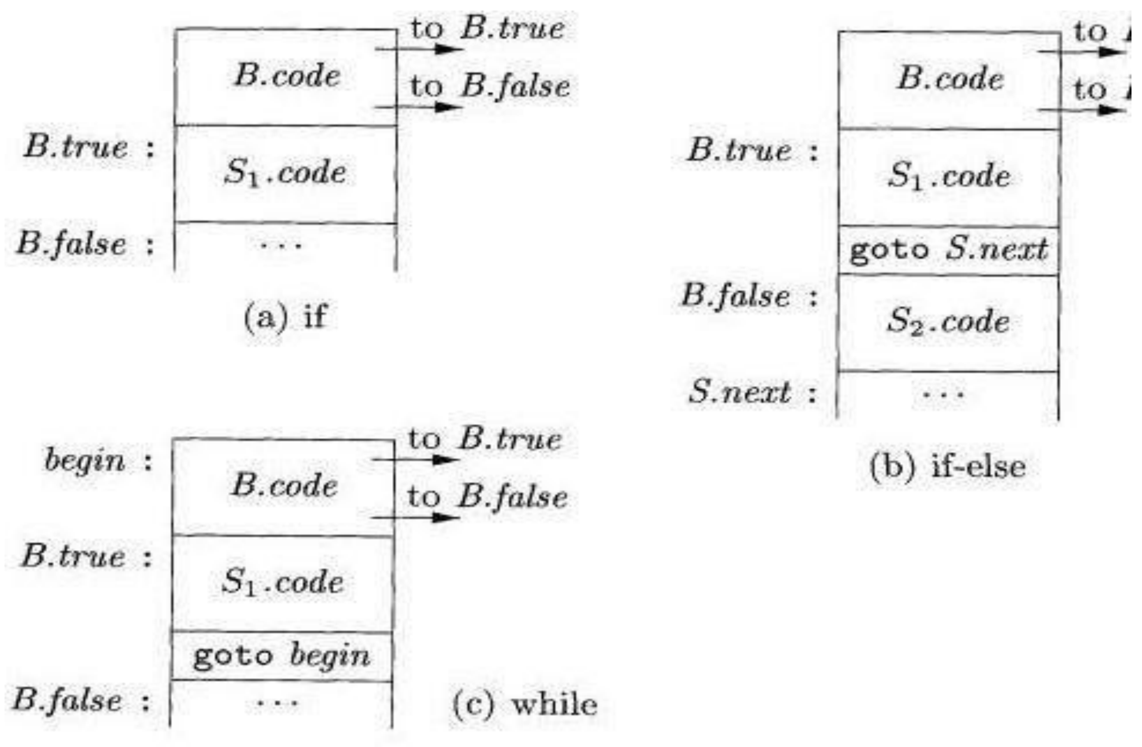
3 Flow-of-ControlStatements

$$\begin{array}{lll} S & \rightarrow & \textbf{if}\ (\ B\ )\ S_1 \\ S & \rightarrow & \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2 \\ S & \rightarrow & \textbf{while}\ (\ B\ )\ S_1 \end{array}$$

Intheseproductions,nonterminal*B*representsabooleanexpressionandnon-terminalSrepresentsastatement.

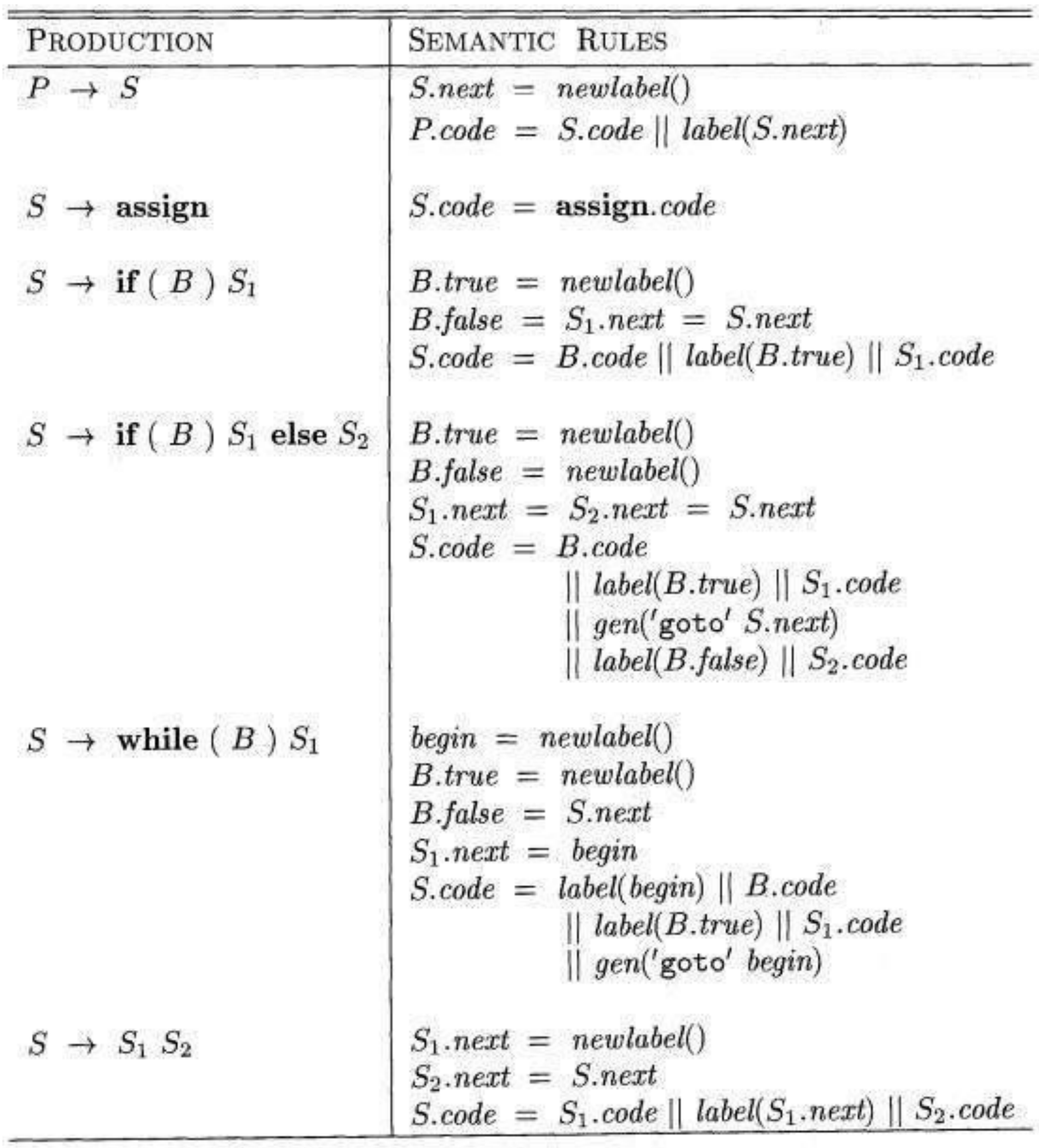
B and S have a synthesized attribute *code*, which gives the translation into three-address instructions. webuildup the translations*B.code*and *S.code*as strings, usingsyntax directed definitions.

Thetranslationofif(B)S1consistsofB.codefollowedbySi.code,asillustratedin Fig.6.35(a). WithinB.codearejumpsbasedon thevalueof B.IfB istrue, control flowsto thefirst instructionofSi.code,and ifB is false, control flowsto theinstructionimmediately following S1. code.



Codeforif,if else,whilestatements

Thesyntax-directeddefinitioninFig.6.36-6.37producesthree-addresscodeforbooleanexpressionsinthecontext ofif-, if-else-, and while-statements.

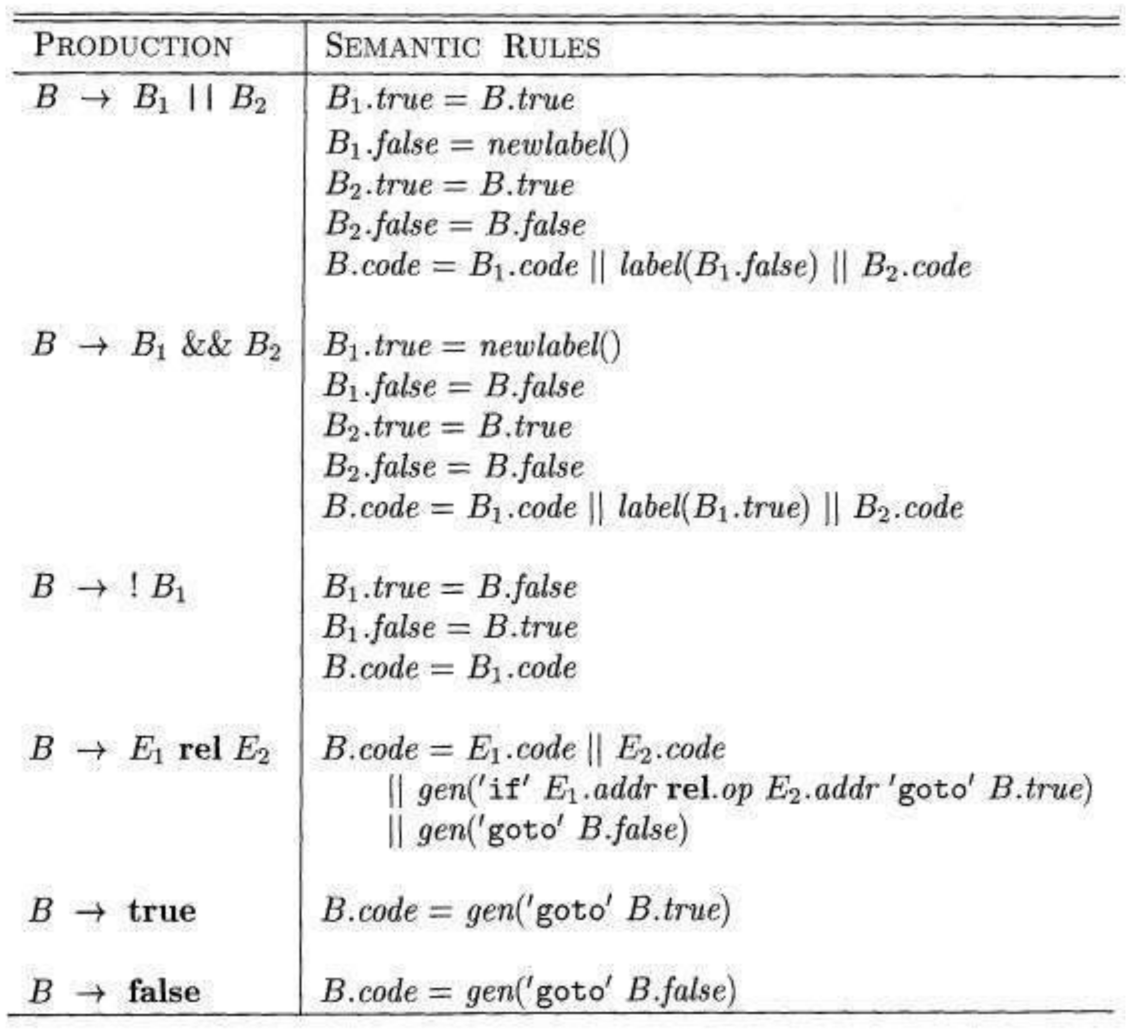


PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Figure 6.36: Syntax-directed definition for flow-of-control statements.

4 Control-FlowTranslationofBooleanExpressions

Boolean expression *B* is translated into three-address instructions that evaluate *B* using createslabelsonlywhentheyareneeded.Alternatively,unnecessarylabelscanbeeliminatedduringasubsequen toptimization phase.



PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \ ! \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen(\texttt{'if'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \texttt{'goto'} \ B.true)$ $\mid\mid gen(\texttt{'goto'} \ B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen(\texttt{'goto'} \ B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen(\texttt{'goto'} \ B.false)$

Figure 6.37: Generating three-address code for booleans

Backpatching

1One-

PassCodeGenerationUsingBackpatching2Backpa

tching for Boolean Expressions

3Flow-of-ControlStatements

1 One-PassCodeGenerationUsingBackpatching

The problem in generating three address codes in a single pass is that we may not know thelabelsthat controlmust goto atthe timejump statementsaregenerated.Sotoget aroundthis problem a series of branching statements with the targets of the jumps temporarily left unspecified isgenerated.BackPatchingisputtingtheaddressinsteadoflabelswhentheproperlabelisdetermined.

Tomanipulatelists ofjumps,Backpatching Algorithmsperform threetypes ofoperations

1.*makelist(i)* creates a new list containing only *i*, an index into the array of instructions; *makelist* returnsapointer to thenewly created list.
2. merge(pi,p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenatedlist.
3. backpatch(p,i)insertsiasthe targetlabel for each oftheinstructionson thelist pointedto byp.

2 BackpatchingforBooleanExpressions

Construct a translation scheme suitable for generating code for boolean expressions during bottom-upparsing. A marker nonterminal *M* in the grammar causes a semantic action to pick up, at appropriatetimes,theindex ofthe next instruction to begenerated. Thegrammar is as follows:

$$B \rightarrow B_1 \mid\mid M\, B_2 \mid B_1 \;\&\&\; M\, B_2 \mid !\, B_1 \mid (\, B_1\,) \mid E_1 \;\mathbf{rel}\; E_2 \mid \mathbf{true} \mid \mathbf{false}$$
$$M \rightarrow \epsilon$$

The translation scheme is in Fig. 6.43.

1)	$B \rightarrow B_1 \mid\mid M\ B_2$	$\{$ *backpatch*$(B_1.\mathit{falselist}, M.\mathit{instr})$; $B.\mathit{truelist} = \mathit{merge}(B_1.\mathit{truelist}, B_2.\mathit{truelist})$; $B.\mathit{falselist} = B_2.\mathit{falselist}$; $\}$
2)	$B \rightarrow B_1\ \&\&\ M\ B_2$	$\{$ *backpatch*$(B_1.\mathit{truelist}, M.\mathit{instr})$; $B.\mathit{truelist} = B_2.\mathit{truelist}$; $B.\mathit{falselist} = \mathit{merge}(B_1.\mathit{falselist}, B_2.\mathit{falselist})$; $\}$
3)	$B \rightarrow\ !\ B_1$	$\{$ $B.\mathit{truelist} = B_1.\mathit{falselist}$; $B.\mathit{falselist} = B_1.\mathit{truelist}$; $\}$
4)	$B \rightarrow (\ B_1\ )$	$\{$ $B.\mathit{truelist} = B_1.\mathit{truelist}$; $B.\mathit{falselist} = B_1.\mathit{falselist}$; $\}$
5)	$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$\{$ $B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr})$; $B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr}+1)$; *emit*('`if`' $E_1.\mathit{addr}$ **rel**.*op* $E_2.\mathit{addr}$ '`goto _`'); *emit*('`goto _`'); $\}$
6)	$B \rightarrow \mathbf{true}$	$\{$ $B.\mathit{truelist} = \mathit{makelist}(\mathit{nextinstr})$; *emit*('`goto _`'); $\}$
7)	$B \rightarrow \mathbf{false}$	$\{$ $B.\mathit{falselist} = \mathit{makelist}(\mathit{nextinstr})$; *emit*('`goto _`'); $\}$
8)	$M \rightarrow \epsilon$	$\{$ $M.\mathit{instr} = \mathit{nextinstr}$; $\}$

Figure 6.43: Translation scheme for boolean expressions

Consider semantic action (1) for the production B -> B1|| M B2. If Bx is true, then B is alsotrue,sothejumpsonBi.truelistbecomepartofB.truelist.IfBiisfalse,however,wemustnexttestB2,so the target forthe jumps B>i .falselist must be the beginning of the code generated for B2 • ThistargetisobtainedusingthemarkernonterminalM.Thatnonterminalproduces,asasynthesizedattributeM.instr, the index ofthenext instruction, just beforeB2 codestarts beinggenerated.

Example

Considerexpression

$$x < 100 \mid\mid x > 200 \ \&\& \ x \ != \ y$$

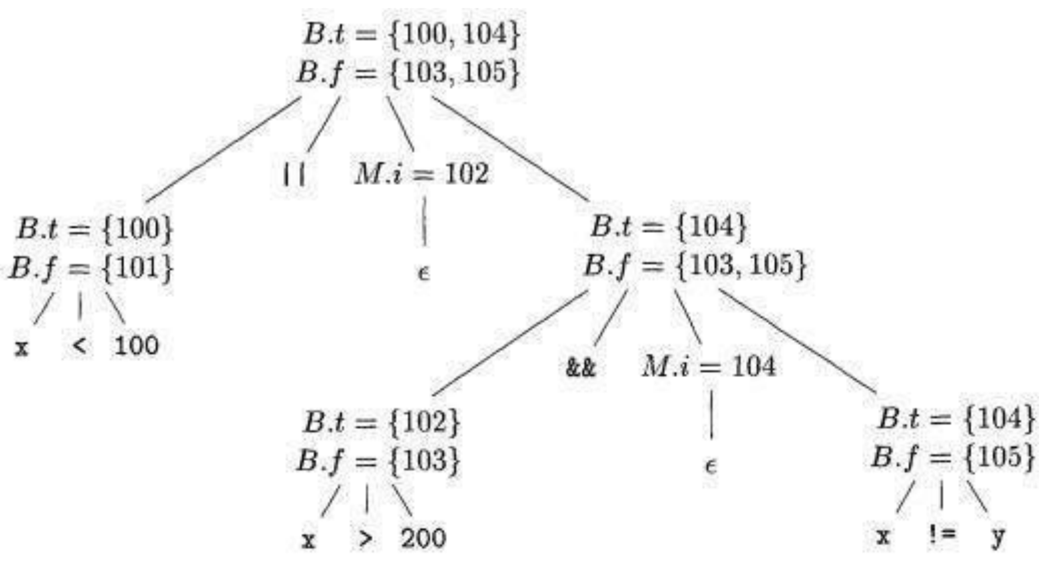


Figure 6.44: Annotated parse tree for $x < 100 \mid\mid x > 200 \ \&\& \ x \ != \ y$

AnannotatedparsetreeisshowninFig.6.44;attributes *truelist,falselist,* and *instr* arerepresented by their initial letters. The actions are performed during a depth-first traversal of the tree.Since all actions appear at the ends of right sides, they can be performed in conjunction with reductionsduring a bottom-up parse. In response to the reduction of *x* <100 to *B* by production (5), the twoinstructions

```
100:  if x < 100 goto _
101:  goto _
```

aregenerated.(startinstructionnumbersat100.) ThemarkernonterminalMin theproduction

$$B \rightarrow B_1 \mid\mid M \, B_2$$

recordsthevalueofnextinstr,whichatthistimeis102.

Thereductionofx>200toBbyproduction(5) generates the instructions

```
102:  if x > 200 goto _
103:  goto _
```

The subexpression $x > 200$ corresponds to B_1 in the production

$$B \rightarrow B_1 \ \&\& \ M \ B_2$$

The marker nonterminal *M* records the current value of *nextinstr*, which is

nowReducing*x!=y*into *B*by production (5)generates

```
104:  if x != y goto _
105:  goto _
```

WenowreducebyB—>B1&&MB2.Thecorrespondingsemanticactioncallsbackpatch(B1.truelist,M.instr) to bind the true exit of B1 to the first instruction of B2. Since B1.truelistis {102} and M.instr is 104, this call to backpatch fills in 104 in instruction 102. The six instructionsgeneratedso far arethus as shown in Fig. 6.45(a).

Thesemanticactionassociatedwiththefinalreductionby B—>B1||MB2callsbackpatch({101},102)which leaves theinstructions as in Fig

Theentireexpressionistrueifandonlyifthegotosofinstructions100or104arereached,andisfalseif and only if the gotos of instructions 103 or 105 are reached. These instructions will have their targetsfilledin laterin thecompilation, whenit is seenwhat mustbe donedepending on thetruth orfalsehood

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(a) After backpatching 104 into instruction 102.

```
100:  if x < 100 goto _
101:  goto 102
102:  if y > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(b) After backpatching 102 into instruction 101.

Figure 6.45: Steps in the backpatch process

oftheexpression.as

3 Flow-of-ControlStatements

usebackpatchingto translateflow-of-control statementsin onepass.

$$
\begin{aligned}
S &\rightarrow \textbf{if}\,(\,B\,)\,S \;\mid\; \textbf{if}\,(\,B\,)\,S\;\textbf{else}\;S \;\mid\; \textbf{while}\,(\,B\,)\,S \;\mid\; \{\,L\,\} \mid A\;; \\
L &\rightarrow L\,S \;\mid\; S
\end{aligned}
$$

ThetranslationschemeinFig.6.46maintainslistsofjumpsthatarefilledinwhentheirtargets are found.

$$
\begin{array}{lll}
1) & S \rightarrow \textbf{if}\,(\,B\,)\,M\,S_1 & \{\, backpatch(B.truelist,\ M.instr); \\
 & & \;\; S.nextlist = merge(B.falselist,\ S_1.nextlist);\ \} \\[1ex]
2) & S \rightarrow \ \textbf{if}\,(\,B\,)\,M_1\,S_1\,N\ \textbf{else}\ M_2\,S_2 & \\
 & & \{\, backpatch(B.truelist,\ M_1.instr); \\
 & & \;\; backpatch(B.falselist,\ M_2.instr); \\
 & & \;\; temp = merge(S_1.nextlist,\ N.nextlist); \\
 & & \;\; S.nextlist = merge(temp,\ S_2.nextlist);\ \} \\[1ex]
3) & S \rightarrow \ \textbf{while}\ M_1\,(\,B\,)\,M_2\,S_1 & \\
 & & \{\, backpatch(S_1.nextlist,\ M_1.instr); \\
 & & \;\; backpatch(B.truelist,\ M_2.instr); \\
 & & \;\; S.nextlist = B.falselist; \\
 & & \;\; emit(\text{'}\texttt{goto}\text{'}\ M_1.instr);\ \} \\[1ex]
4) & S \rightarrow \{\,L\,\} & \{\, S.nextlist = L.nextlist;\ \} \\[1ex]
5) & S \rightarrow A\ ; & \{\, S.nextlist = \textbf{null};\ \} \\[1ex]
6) & M \rightarrow \epsilon & \{\, M.instr = nextinstr;\ \} \\[1ex]
7) & N \rightarrow \epsilon & \{\, N.nextlist = makelist(nextinstr); \\
 & & \;\; emit(\text{'}\texttt{goto \_}\text{'});\ \} \\[1ex]
8) & L \rightarrow L_1\ M\ S & \{\, backpatch(L_1.nextlist,\ M.instr); \\
 & & \;\; L.nextlist = S.nextlist;\ \} \\[1ex]
9) & L \rightarrow S & \{\, L.nextlist = S.nextlist;\ \}
\end{array}
$$

Figure 6.46: Translation of statements

Backpatch the jumps when B is true to the instruction Mi.instr; the latter is the beginning of thecode for Si. Similarly, we backpatch jumps when B is false to go to the beginning of the code for S 2 .The list S.nextlist includes all jumps out of Si and S 2 , as well as the jump generated by N. (Variabletempis atemporary thatis used only for merginglists.

Semanticactions(8)and(9)handlesequencesofstatements.In

$$L \to L_1 \, M \, S$$

theinstructionfollowingthecodeforL±inorderofexecutionisthebeginningofS.ThustheL1.nextlist list is backpatched to the beginning of the code for S, which is given by M.instr. In L —> S,L.nextlistis thesame as S.nextlist.

Notethatnonewinstructionsaregeneratedanywhereinthesesemanticrules,exceptforrules(3)and (7). All other code is generated by the semantic actions associated with assignment-statements andexpressions. The flow of control causes the proper backpatching so that the assignments and booleanexpressionevaluations will connect properly.