**CONTEXT FREE GRAMMAR**

A context free grammar consists of terminals, non-terminals, a start symbol and productions.

1. 1.    Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages.
2. 2.    Non-terminals are syntactic variables that denote sets of strings that help define language generated by the grammar. They impose a hierarchical structure on the language.
3. 3.    In a grammar one non-terminal is distinguished as the start symbol, and the sets of string it denotes is the language denoted by the grammar.
4. 4.    The productions of a grammar specify the manner in which terminals and non-terminals can be combined to form strings . Each production consists of a non-terminal followed by an arrow(==>) followed by a string of non-terminals and terminals.

Eg:

E ==>EAE | (E) | -E | id

A==> + | - | \* | / |

**Where E,A are the non-terminals while id, +, \*, -, /,(, ) are the terminals.**

**Ambiguity**

A grammar that produces more than one parse tree for some sentence is said to be ambiguous, Put another way, an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence. Carefully writing the grammar can eliminate ambiguity.

**Elimination of Left Recursion**

A grammar is left recursion if it has a nonterminal A such that there is a derivation A Afor some string . Top-Down parsing method cannot handle left-recursion grammars, so a transformation that eliminates left recursion is needed. A left-recursive pair of productions

A A  could be replaced by the non-left-recursive productions

AA'

A'A'| 

without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

No matter how many A-productions there are, we can eliminate immediate left recursion from them by the follwing technique. First, we group the A-productions as

AA1| A 2|..........A m |1|2|........ |n

where noi begins with an A. Then, we replace the A-production by

A1A'|2A'|........ |nA'

A'1A'|  2A'|.......... mA' |

The nonterminal A generates the same strings as before but is no longer left recursive.This procedure eliminates all immediate left recursion from the A and A' productions , but it does not eliminate left recursion involving derivation of two or more steps. For example, consider the grammar

S==>Aa|b

A==>Ac|Sd|

The nonterminal S is left-recursive because S gives Aa gives Sda, but it is not immediately left recursive.

Algorithm below will systematically eliminates left recursion from a grammar. It is guaranteed to work if the grammar has no cycles or productions

Cycles can be systematically eliminated from a grammar as can - productions.

**Algorithm** Eliminating left recursion.

*Input.* Grammar with no cycles or -productions.

*Output.* An equivalent grammar has no left recursion.

*Method.* Apply the algorithm to G . Note that the resulting non-left-recursive grammar may have  -productions.

1. Arrange the nonterminals in some order A1, A2, ,...........,An.

2. **for** i := 1 to *n* **do begin**

**for** j: = 1 to i - 1 **do begin**

replace each production of the form Ai==> Ajα by the productions

Ai==> α1|α2...............|αk

where Aj==>α1|α2|.........|αk are all current Aj productions.

**end**

eliminate the immediate left recursion among Ai productions

**end**

**Left Factoring**

Left factoring is a grammar transformation that is useful producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a non terminal A, we may rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice. If A==>α are two A-productions and the input begins with a non empty string derived from , we do not know whether to expand A to or to. We may defer decision by expanding A to αA' . Then after seeing the input derived from  we expand A' to 1or to 2

A==>αA'

A'==>α1|α2

**Algorithm: left factoring a**  .**grammar**

*Input.* Grammar G

*Output*. An equivalent left factored grammar.

*method.* For each non terminal A find the longest prefix  common to two or more of its alternatives. If != E, i.e., there is a non trivial common prefix, replace all the A productions

A==>1|2|..............|n|where  represents all alternatives that do not begin with  by

A==>A'|

A'==>1|2|.............|n

Here A' is new nonterminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

**THE ROLE OF A PARSER**

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion.

The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)

2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– [top-down parsing](http://www.mec.ac.in/resources/notes/notes/compiler/module2/module2/tdp.htm) and [bottom-up parsing](http://www.mec.ac.in/resources/notes/notes/compiler/module2/module2/bup.htm).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Nonrecursive Predictive Parsing

It is possible to build a nonrecursive predictive parser by maitaining a stack explicitly, rather than implictly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a nonterminal . The nonrecursive parser in figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.

  A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1 If X= a=$, the parser halts and announces successful completion of parsing.

2 If X=a!=$, the parser pops X off the stack and advances the input pointer to the next input symbol.

3 If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a]={X->UVW}, the parser replaces X on top of the stack by WVU( with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine.

The behavior of the parser can be described in terms of its configurations, which give the stack contents and the remaining input.

Algorithm for Nonrecursive predictive parsing.

Input. A string w and a parsing table M for grammar G.

Output. If w is in L(G), a leftmost derivation of w; otherwise, an error indication.

Method. Initially, the parser is in a configuration in which it has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown in Fig.

set ip to point to the first symbol of w$.

repeat

let X be the top stack symbol and a the symbol pointed to by ip.

if X is a terminal of $ then

if X=a then

pop X from the stack and advance ip

else error()

else

if M[X,a]=X->Y1Y2...Yk then begin

pop X from the stack;

push Yk,Yk-1...Y1 onto the stack, with Y1 on top;

output the production X-> Y1Y2...Yk

end

else error()

until X=$

FIRST and FOLLOW

 The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during panic-mode error recovery.

 If x is any string of grammar symbols, let FIRST(x) be the set of terminals that begin the strings derived from x. If x=\*>e, then e is also in FIRST(x).

Define FOLLOW(A), for nonterminals A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form S=\*>xAab for some x and b. Note that there may, at some time during the derivation, have been symbold beteween A and a, but if so, they derived e and disappeared. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is {X}.

2. If X->e is a production, then add e to FIRST(X).

3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1); that is, Y1...Yi-1=\*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=\*>e, then we add FIRST(Y2) and so on.

 Now, we can compute FIRST for any sting X1X2X3...Xn as follows. Add to FIRST(X1X2X3...Xn) all the non-e symbols of FIRST(X1). Also add the non-e symbols of FIRST(X2) if e is in FIRST(X1), the non-e symbols of FIRST(X3) if e is in both FIRST(X1) and FIRST(X2), and so on. Finally , add e to FIRST(X1X2...Xn) if, for all i, FIRST(Xi) contains e.

 To compute the FIRST(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. PLace $ in FOLLOW(S), where S is the start symbol and $ in the input right endmarker.

2. If there is a production A=>aBß where FIRST(ß) except e is placed in FOLLOW(B).

3. If there is aproduction A->aB or a production A->aBß where FIRST(ß) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Follow.

Find the first and follow of all nonterminals in the Grammar-

E -> TE'

E'-> +TE'|e

T -> FT'

T'-> \*FT'|e

F -> (E)|id

Then:

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

FIRST(E')={+,e}

FIRST(T')={\*,e}

FOLLOW(E)=FOLLOW(E')={),$}

FOLLOW(T)=FOLLOW(T')={+,),$}

FOLLOW(F)={+,\*,),$}

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with i=1 in each case, since FIRST(id)=(id) and FIRST('(')= {(} by rule 1. Then by rule 3 with i=1, the production T -> FT' implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW,we put $ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied to production F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E -> TE', $ and right parenthesis are in FOLLOW(E').

Construction Of Predictive Parsing Tables

For any grammar G, the following algorithm can be used to construct the predictive parsing table.

The algrithm is -->

Input : Grammar G

Output : Parsing table M

Method

1. For each production A-> a of the grammar, do steps 2 and 3.

2. For each terminal a in FIRST(a), add A->a, to M[A,a].

3. If e is in First(a), add A->a to M[A,b] for each terminal b in FOLLOW(A). If e is in FIRST(a) and $ is in FOLLOW(A), add A->a to M[A,$].

4. Make each undefined entry of M be error.

LL(1) Grammar

 The above algorithm can be applied to any grammar G to produce a prasing table M. FOr some Grammars, for example if G is left recursive or ambiguous, then M will have atleast one multiply-defiend entry.

 A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no muliply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rule to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

 The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar.although leftrecursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes.to alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control constructs and to use operator precedence for expressions.however, if an lr parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

 Error Recovery in Predictive Parsing

 The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty.

 Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows:

 1. As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

 2. It is not enough to use FOLLOW(A) as the synchronizingset for A. Fo example , if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generaitn expressions.

3. If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

4. If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missedThis approach reduces the number of nonterminals that have to be considered during error recovery.

5. If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Top Down parsing Bottom Up parsing

Bottom-up (shift-reduce) Parsers

Consider the grammar:

*S* -> **a***AB***e**

*A* -> *A***bc** | **b**

*B* -> **d**

The sentence **abcde** can be **reduced** to *S*, thus:

**abbcde**

**a***A***bcde**

**a***A***de**

**a***AB***e**

*S*

The technique used here is to scan the string looking for a **substring** which matches the right side of a production, and which is replaced by the left side of that production.

The preceding reduction steps trace the following **rightmost derivation**, in reverse:

*S* -> **a***AB***e** -> **a***A***de** -> **a***A***bcde** -> **abbcde**

Note in the derivation, the choice of productions:

**abbcde**

**a***A***bcde**

**a***A***de** ...etc

After the first **b** was replaced by *A*, there were 3 substrings which matched the right sides of productions (*A***bc**, **b** and **d**). The substring *A***bc** is a **handle**, defined as the **leftmost simple phrase** of a sentential form and is always chosen in this technique.

Terminology and Concepts

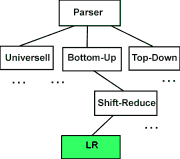
A **shift-reduce** parser operates by scanning the input symbols and either **shifting** them onto a stack or, if the top of stack holds the right end of a handle of the grammar, **reducing** the handle to a simpler sentential form which becomes the new top of stack item.

A **conflict** occurs when the parser cannot decide whether to either: shift or reduce the top of stack (a shift/reduce conflict), or reduce the top of stack using one of two possible productions (a reduce/reduce conflict) .The grammars which can be parsed in this way are called **LR**, and are more general than LL grammars.

**LR parsing introduction**

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse

**LR-Parser**



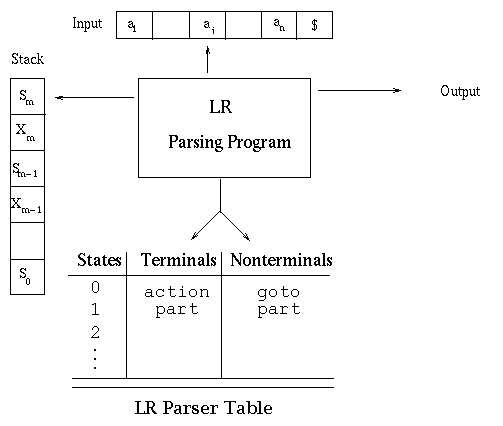
Advantages of LR parsing:

LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods. The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

**The LR parsing algorithm**

The schematic form of an LR parser is shown below.



The program uses a stack to store a string of the form s0X1s1X2...Xmsm where sm is on top. Each Xi is a grammar symbol and each si is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision. The parsing table consists of two parts: a parsing action function *action* and a goto function *goto*. The program driving the LR parser behaves as follows: It determines sm the state currently on top of the stack and ai the current input symbol. It then consults action[sm, ai], which can have one of four values:

shift s, where s is a state

reduce by a grammar production A -> b

accept

error

The function goto takes a state and grammar symbol as arguments and produces a state. For a parsing table constructed for a grammar *G*, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of *G*. Recall that the viable prefixes of *G* are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser because they do not extend past the rightmost handle.

A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

(s0 X1 s1 X2 s2... Xm sm, ai ai+1... an$)

This configuration represents the right-sentential form

X1 X1 ... Xm ai ai+1 ...an

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading ai and sm, and consulting the parsing action table entry action[sm, ai]. Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later.

The configurations resulting after each of the four types of move are as follows:

If action[sm, ai] = shift s, the parser executes a shift move entering the configuration

(s0 X1 s1 X2 s2... Xm sm ai s, ai+1... an$)

Here the parser has shifted both the current input symbol ai and the next symbol.

If action[sm, ai] = reduce A -> b, then the parser executes a reduce move, entering the configuration,

(s0 X1 s1 X2 s2... Xm-r sm-r A s, ai ai+1... an$)

where s = goto[sm-r, A] and r is the length of b, the right side of the production. The parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state sm-r. The parser then pushed both A, the left side of the production, and s, the entry for goto[sm-r, A], onto the stack. The current input symbol is not changed in a reduce move.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production reduced.

If action[sm, ai] = accept, parsing is completed.

If action[sm, ai] = error, the parser has discovered an error and calls an error recovery routine.

**LR parsing algorithm**

**Input**: Input string *w* and an LR parsing table with functions action and goto for a grammar *G*.  
**Output**: If *w* is in L(*G*), a bottom-up parse for *w*. Otherwise, an error indication.  
**Method**: Initially the parser has s0, the initial state, on its stack, and *w*$ in the input buffer.

repeat forever begin

let s be the state on top of the stack

and a the symbol pointed to by ip;

if action[s, a] = shift s' then begin

push a, then push s' on top of the stack; // <symbol, state> pair

advance ip to the next input symbol;

else if action[s, a] = reduce A -> b then begin

pop 2\* |b| symbols off the stack;

let s' be the state now on top of the stack;

push A, then push goto[s', A] on top of the stack;

output the production A -> b; // for example

else if action[s, a] = accept then

return

else error();

end

Let's work an example to get a feel for what is going on,

An Example

(1) E -> E \* B

(2) E -> E + B

(3) E -> B

(4) B -> 0

(5) B -> 1

The Action and Goto Table

The two LR(0) parsing tables for this grammar look as follows:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | action | | | | |  | goto | |
| state | **\*** | **+** | **0** | **1** | **$** |  | **E** | **B** |
| **0** |  |  | s1 | s2 |  |  | 3 | 4 |
| **1** | r4 | r4 | r4 | r4 | r4 |  |  |  |
| **2** | r5 | r5 | r5 | r5 | r5 |  |  |  |
| **3** | s5 | s6 |  |  | acc |  |  |  |
| **4** | r3 | r3 | r3 | r3 | r3 |  |  |  |
| **5** |  |  | s1 | s2 |  |  |  | 7 |
| **6** |  |  | s1 | s2 |  |  |  | 8 |
| **7** | r1 | r1 | r1 | r1 | r1 |  |  |  |
| **8** | r2 | r2 | r2 | r2 | r2 |  |  |  |

**The action table** is indexed by a state of the parser and a terminal (including a special nonterminal $ that indicates the end of the input stream) and contains three types of actions: a shift that is written as 'sn ' and indicates that the next state is n, a reduce that is written as 'rm ' and indicates that a reduction with grammar rule m should be performed and an accept that is written as 'acc' and inidcates that the parser accepts the string in the input stream

Simple LR parsing

An *LR(0) item* (or just *item*) of a grammar *G* is a production of *G* with a dot at some position of the right side indicating how much of a production we have seen up to a given point. For example, for the production E -> E + T we would have the following items:

[E -> .E + T]

[E -> E. + T]

[E -> E +. T]

[E -> E + T.]

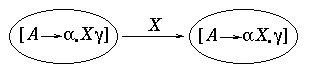
We call them LR(0) items because they contain no explicit reference to lookahead. More on this later when we look at canonical LR parsing.

The central idea of the SLR method is first to construct from the grammar a deterministic finite automaton to recognize viable prefixes. With this in mind, we can easily see the following:

the symbols to the left of the dot in an item are on the stack and the symbols to the right are still to come (where are they?). up until the time when we have the dot to the right of the last symbol of the production, we have a viable prefix (why is this true?). when the dot reaches the right side of the last symbol of the production, we have a handle for the production and can do a reduction (the text calls this a completed item; similarly it calls [E -> .E + T] an initial item). an item is a summary of the recent history of a parse (how so?) items correspond to the states of a NFA (why an NFA and not a DFA?). Now, if items correspond to states, then there must be transitions between items (paralleling transitions between the states of a NFA). Some of these are fairly obvious. For example, consider the transition from [E -> .(E)] to [E -> (.E)] which occurs when a "(" is shifted onto the stack. In a NFA this would correspond to following the arc labelled "(" from the state corresponding to [E -> .(E)] to the state corresponding to [E -> (.E)]. Similarly, we have [T -> .F] and [T -> F.] which occurs when F is produced as the result of a reduction and pushed onto the stack. Other transitions can occur on e-transitions.

The insight that items correspond to states leads us to the explanation for why we need e-transitions.

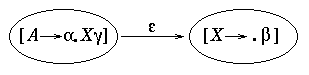
Consider a transition on symbol *X* from [*A* -> a.*X*g] to [*A* -> a*X*.g]. In a transition diagram this looks like:



If *X* is a terminal symbol, this transition corresponds to shifting *X* from the input buffer to the top of the stack. Things are more complicated if *X* is a nonterminal because nonterminals cannot appear in the input and be shifted onto the stack as we do with terminals. Rather, nonterminals only appear on the stack as the result of a reduction by some production *X* -> b.

How does such a reduction take place? We would have to have the handle b on top of the stack, which means we must have already recognized b.

\*\*\* The process of recognizing b begins with the initial item [*X* -> .b]. So, for every item [*A* -> a.*X*g] we add an e-transition for each production choice *X* -> b of *X*, which says that *X* can be produced by recognizing any of the right-hand sides of its production choices. In the transition diagram it looks like this:



To complete our understanding of the creation of a NFA from the items, we need to decide on the choices for start state and final states.

We'll consider final states first. Recall that the purpose of the NFA is not to recognize strings, but to keep track of the current state of the parse, thus it is the parser that must decide when to do an accept and the NFA need not contain that information.

For the start state, consider the initial configuration of the parser: the stack is empty and we want to recognize *S*, the start symbol of the grammar. But there may be many initial items [*S* -> .a] from which to choose.

To solve the problem, we augment our grammar with a new production *S*' -> *S*, where *S*' is the new start symbol and [*S*' -> .*S*] becomes the start state for the NFA. What will happen is that when doing the reduction for this production, the parser will know to do an accept.

The following example makes the need for e-transitions and an augmented grammar more concrete. Consider the following augmented grammar:

E' -> E

E -> E + T

E -> T

T -> T \* F

T -> F

F -> (E)

F -> id

A quick examination of the grammar reveals that any legal string must begin with either **(** or **id**, resulting in one or the other being pushed onto the stack.

So we would have either the state transition [F -> .(E)] to [F -> (.E)] or the transition from [F -> .id] to [F -> id.].

But clearly to make either of these transitions we must already be in the corresponding state ([F -> .(E)] or [F -> .id]).

Recall, though, that we always begin with our start state [E' -> E] and note that there is no transition from the start state to either [F -> .(E)] or [F -> .id].

To get from the start state to one of these two states without consuming anything from the input we must have e-transitions.

The example from the book makes this a little clearer. We want to parse "(id)".

|  |  |  |
| --- | --- | --- |
| items and e-transitions | | |
| **Stack** | **State** | **Comments** |
| Empty | [E'-> .E] | can't go anywhere from here |
|  | e-transition | so we follow an e-transition |
| Empty | [F -> .(E)] | now we can shift the ( |
| **(** | [F -> (.E)] | building the handle (E); This state says: "I have ( on the stack and expect the input to give me tokens that can eventually be reduced to give me the rest of the handle, E)." |

Now let's try to bring all this together with a further example. Let's build the NFA for the following (very simple) grammars:

S -> **(** S **)** S | e

S -> **(** S **)** | **a**

Here's how:

Construct the list of LR(0) items.

Build a first version of the NFA without worrying about e-transitions.

Add the e-transitions using the rule: every item with a dot before *S* has an e-transition to every initial item of *S*. (Can you generalize this rule?)

We make a distinction between kernel items and non-kernel items (also known as closure items).

Which items are kernel items?  
kernel item includes *S*' -> .*S* and all items whose dots are not at the left end (these will be the items that originate a state as targets of non-e-transitions).

Which items are closure items?

 closure item

those items which have their dots at the left end (these are the items added to a state during the e-closure step).

constructing the LR parsing table

To construct the parser table we must convert our NFA into a DFA.

**\*\*\*** The states in the LR table will be the e-closures of the states corresponding to the items!! SO...the process of creating the LR state table parallels the process of constructing an equivalent DFA from a machine with e-transitions. Been there, done that - this is essentially the subset construction algorithm so we are in familiar territory here! We need two operations: closure() and goto().

closure()

If I is a set of items for a grammar *G*, then closure(I) is the set of items constructed from I by the two rules:

Initially every item in I is added to closure(I)

If *A* -> a.*B*b is in closure(I), and *B* -> g is a production, then add the initial item [*B* -> .g] to I, if it is not already there. Apply this rule until no more new items can be added to closure(I).

From our grammar above, if I is the set of one item {[E'-> .E]}, then closure(I) contains:

I0: E' -> .E

E -> .E + T

E -> .T

T -> .T \* F

T -> .F

F -> .(E)

F -> .id

goto()

goto(I, *X*), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items [*A* -> a*X*.b] such that [*A* -> a.*X*b] is in I.

The idea here is fairly intuitive: if I is the set of items that are valid for some viable prefix g, then goto(I, *X*) is the set of items that are valid for the viable prefix g*X*.

Building a DFA from the LR(0) items

Now we have the tools we need to construct the canonical collection of sets of LR(0) items for an augmented grammar *G*'.

Sets-of-Items-Construction: to construct the canonical collection of sets of LR(0) items for augmented grammar G'.

procedure items(G')

begin

C := {closure({[S' -> .S]})};

repeat

for each set of items in C and each grammar symbol X

such that goto(I, X) is not empty and not in C do

add goto(I, X) to C;

until no more sets of items can be added to C

end;

algorithm for constructing an SLR parsing table

**Input**: augmented grammar G'  
**Output**: SLR parsing table functions action and goto for G'  
**Method**:

Construct C = {I0, I1 , ..., In} the collection of sets of LR(0) items for G'.

State i is constructed from Ii:

if [*A* -> a.ab] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.

if [*A* -> a.] is in Ii, then set action[i, a] to "reduce *A* -> a" for all a in FOLLOW(*A*). Here *A* may not be *S*'.

if [*S*' -> *S*.] is in Ii, then set action[i, $] to "accept"

If any conflicting actions are generated by these rules, the grammar is not SLR(1) and the algorithm fails to produce a parser.

The goto transitions for state i are constructed for all nonterminals *A* using the rule: If goto(Ii, A) = Ij, then goto[i, *A*] = j.

All entries not defined by rules 2 and 3 are made "error".

The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S].

**Example**: Build the canonical LR(0) collections and DFAs for the following grammars:

Ex 1:

S -> **(** S **)** S | e

Ex 2:

S -> **(** S **)** | **a**

Ex 3:

E' -> E

E -> E + T

E -> T

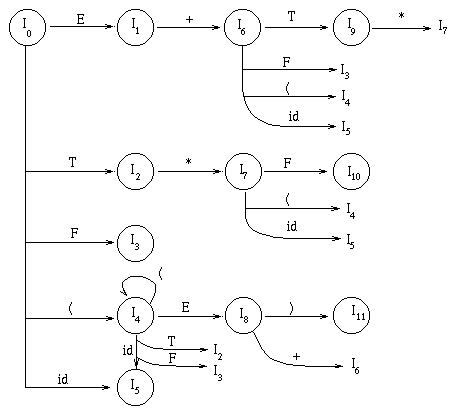
T -> T \* F

T -> F

F -> ( E )

F -> id

Here is what the corresponding DFA looks like:



Dealing with conflicts

Recall that the actions of a parser are one of: 1) shift, 2) reduce, 3) accept, and 4) error.

A grammar is said to be a **LR(0) grammar** if rules 1 and 2 are unambiguous. That is, if a state contains a completed item [*A* -> a.], then it can contain no other items. If, on the other hand, it also contains a "shift" item, then it isn't clear if we should do the reduce or the shift and we have a **shift-reduce conflict**. Similarly, if a state also contains another completed item, say, [*B* -> b.], then it isn't clear *which* reduction to do and we have a **reduce-reduce conflict**.

Constructing the action and goto table as is done for LR(0) parsers would give the following item sets and tables:

**Item set 0**

S → · E

+ E → · 1 E

+ E → · 1

**Item set 1**

E → 1 · E

E → 1 ·

+ E → · 1 E

+ E → · 1

**Item set 2**

S → E ·

**Item set 3**

E → 1 E ·

The action and goto tables:

|  |  |  |  |
| --- | --- | --- | --- |
|  | action | | goto |
| state | **1** | **$** | **E** |
| **0** | s1 |  | 2 |
| **1** | s2/r2 | r2 | 3 |
| **2** |  | acc |  |
| **3** | r1 | r1 |  |

As can be observed there is a shift-reduce conflict for state 1 and terminal '1'.

For shift-reduce conflicts there is a simple solution used in practice: always prefer the shift operation over the reduce operation. This automatically handles, for example, the dangling else ambiguity in if-statements. See the book's discussion on this.

Reduce-reduce problems are not so easily handled. The problem can be characterized generally as follows: in the SLR method, state i calls for reducing by *A* -> a if the set of items Ii contains item [*A* -> a.] (a completed item) and a is in FOLLOW(*A*). But sometimes there is an alternative ([*B* -> a.]) that could also be taken and the reduction is made.

Canonical LR parsing

We want to avoid the conflicts that can occur with some grammars using the SLR method.

By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle a for which there is a possible reduction to *A*. As the text points out, sometimes the FOLLOW sets give too much information and doesn't (can't) discriminate between different reductions.

The general form of an LR(k) item becomes [*A* -> a.b, *s*] where *A* -> ab is a production and s is a string of terminals. The first part (*A* -> a.b) is called the core and the second part is the lookahead. In LR(1) |*s*| is 1, so *s* is a single terminal.

*A* -> ab is the usual righthand side with a marker; any *a* in *s* is an incoming token in which we are interested. Completed items used to be reduced for every incoming token in FOLLOW(*A*), but now we will reduce only if the next input token is in the lookahead set *s*. SO...if we get two productions *A* -> a and *B* -> a, we can tell them apart when a is a handle on the stack if the corresponding completed items have different lookahead parts.

Furthermore, note that the lookahead has no effect for an item of the form [*A* -> a.b, a] if b is not e. Recall that our problem occurs for completed items, so what we have done now is to say that an item of the form [*A* -> a., a] calls for a reduction by *A* -> a only if the next input symbol is a. More formally, an LR(1) item [*A* -> a.b, a] is valid for a viable prefix g if there is a derivation *S* =>\* s ab*w*, where

g = sa, and

either a is the first symbol of *w*, or *w* is e and a is $.

**algorithm for construction of the sets of LR(1) items**

**Input**: grammar G'  
**Output**: sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'  
**Method**:

closure(I)

begin

repeat

for each item [*A* -> a.Bb, a] in I,

each production *B* -> g in G',

and each terminal b in FIRST(ba)

such that [*B* -> .g, b] is not in I do

add [*B* -> .g, b] to I;

until no more items can be added to I;

end;

goto(I, X)

begin

let J be the set of items [*A* -> a*X*.b, a] such that

[*A* -> a.*X*b, a] is in I

return closure(J);

end;

procedure items(G')

begin

C := {closure({*S'* -> .*S*, $})};

repeat

for each set of items I in C and each grammar symbol *X* such

that goto(I, *X*) is not empty and not in C do

add goto(I, *X*) to C

until no more sets of items can be added to C;

end;

An example,

Consider the following grammer,

S’->S

S->CC

C->cC

C->d

Sets of LR(1) items

I0: S’->.S,$

S->.CC,$

C->.Cc,c/d

C->.d,c/d

I1:S’->S.,$

I2:S->C.C,$

C->.Cc,$

C->.d,$

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

C->.Cc,c/d

C->.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

I6: C->c.C,$

C->.cC,$

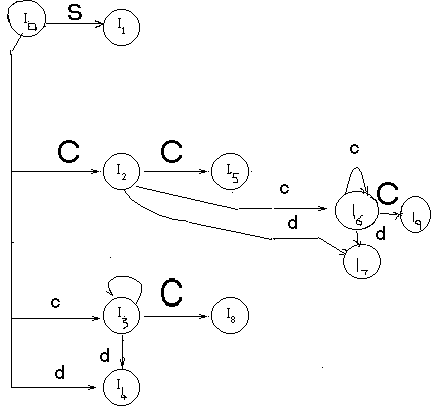
C->.d,$

I7:C->d.,$

I8:C->cC.,c/d

I9:C->cC.,$

Here is what the corresponding DFA looks 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 |  |  |  |

**algorithm for construction of the canonical LR parsing table**  
**Input**: grammar G'  
**Output**: canonical LR parsing table functions action and goto

Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.

State i is constructed from Ii:

if [*A* -> a.ab, b>] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.

if [*A* -> a., a] is in Ii, then set action[i, a] to "reduce *A* -> a" for all a in FOLLOW(*A*). Here *A* may not be *S*'.

if [*S*' -> *S*.] is in Ii, then set action[i, $] to "accept"

If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.

The goto transitions for state i are constructed for all nonterminals *A* using the rule: If goto(Ii, A) = Ij, then goto[i, *A*] = j.

All entries not defined by rules 2 and 3 are made "error".

The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S, $].

Example: Let's rework the following grammar:

A -> **(** A **)** | **a**

Every SLR(1) grammar is an LR(1) grammar. The problem with canonical LR parsing is that it generates a lot of states. This happens because the closure operation has to take the lookahead sets into account as well as the core items.

The next parser combines the simplicity of SLR with the power of LR(1).

LALR parsing

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of *core* (or first) components and differ only in their second component, the lookahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing.

This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same *core*. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the lookahead. But it is possible for the merger to produce a reduce/reduce conflict.

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

We'll look at the simple method. :-)

**Algorithm for easy construction of an LALR table**  
**Input**: G'  
**Output**: LALR parsing table functions with action and goto for G'.

**Method**:

Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.

For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.

Let C' = {J0, J1 , ..., Jm} be the resulting sets of LR(1) items. The parsing actions for state i are constructed from Ji in the same manner as in the construction of the canonical LR parsing table. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.

The goto table is constructed as follows: If J is the union of one or more sets of LR(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 K be the union of all sets of items having the same core as goto(I1, X). Then goto(J, X) = K.

Consider the above example,

I3 & I6 can be replaced by their union

I36:C->c.C,c/d/$

C->.Cc,C/D/$

C->.d,c/d/$

I47:C->d.,c/d/$

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

Parsing Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| state | **c** | d | **$** | S | **C** |
| **0** | S36 | S47 |  | 1 | 2 |
| **1** |  |  | Acc |  |  |
| **2** | S36 | S47 |  |  | 5 |
| **36** | S36 | S47 |  |  | 89 |
| 47 | R3 | R3 |  |  |  |
| **5** |  |  | R1 |  |  |
| **89** | R2 | R2 | R2 |  |  |

handling errors

The LALR parser may continue to do reductions after the LR parser would have spotted an error, but the LALR parser will never do a shift after the point the LR parser would have discovered the error and will eventually find the error.

The YACC Parser Generator - Introduction

There are several varieties of LR parsers (LR(0), LR(1), SLR and LALR), with differences depending on amount of lookahead and on construction of the parse table. Their characteristics and use are outside the scope of this unit.

It is possible, however, to automatically generate LALR parsers (which are considered very powerful) using the YACC parser generator provided on Unix (and other systems).

YACC operates in an analogous manner to lex, in that a yacc source file (of similar format) is compiled into C code which implements the parser.

YACC Input Format

The yacc source program format is very similar to that of lex:

declarations

%%

translation rules

%%

supporting C functions

The production (translation) rules:

*S* -> *A* | *B* | *C*

would be written in yacc source as:

S : A { action 1 }

| B { action 2 }

| C { action 3 }

;

YACC Example

A yacc source file for a simple grammar (a very simple desk calculator) looks like:

%token NAME NUMBER

%%

statement: NAME '=' expression

| expression

{printf("= %d\n", $1;}

;

expression: expression '+' NUMBER

{ $$ = $1 + $3; }

| expression '-' NUMBER

{ $$ = $1 - $3; }

| NUMBER

{ $$ = $1; }

;

%%

Notes on the Example

The rules section (before the first %%) defines **tokens** (see later). Single quoted characters such as '+' are automatically allowable as tokens without speciying them here.

Every symbol has a **value**, which is of a type appropriate to the type of the symbol (eg: for a number, the value would be the number itself). Yacc obtains this from yylval. In this parser, the type of the value of a NUMBER will be an int, and the value of a NAME will be a pointer to a symbol table entry.

Whenever a yacc parser reduces a production, it can execute an **action rule** associated with it. The action rule can refer to the values of the right side as $1, $2, etc, and the value of the left side as $$. Note that the last action code specified is redundant, since $$ = $1 is the default action if none is specified.

Theoretical note -- observe that yacc handles left recursion in the grammar: in fact, it works best if left recursion is present! Left factoring is also unnecessary. This is rather more general than recursive descent ( LL(1)) parsers can handle.

A Simple lexer For The Example yacc Program

This simple lexer doesn't (yet) define the NAME token:

%{

#include "y.tab.h"

%}

%%

[0-9]+ {yylval = atoi(yytext); return NUMBER; }

[ \t] ; /\* ignore whitespace \*/

\n return 0; /logical EOF \*/

. return yytext[0];

%%

The commands to compile the combined yacc/lex program are:

% yacc -d calc.y

% lex calc.l

% cc -o calc y.tab.c lex.yy.c -ly -ll

% calc

some interesting topics:

*LEX-the Lexical Analyzer*

The Lex utility generates a 'C' code which is nothing but a yylex() function which can be used as an interface to YACC. A good amount of details on Lex can be obtained from the Man Pages itself. A Practical approach to certain fundamentals are given here.  
        The General Format of a Lex File consists of three sections:  
            1. Definitions  
            2. Rules  
            3. User Subroutines  
Definitions consists of any  external 'C' definitions used in the lex actions or subroutines . e.g all preprocessor directives like #include, #define macros etc. These are simply copied to the lex.yy.c file.  The other type of definitions are Lex definitions which are essentially the lex substitution strings,lex start states and lex table size declarations.The Rules is the basic part which specifies the regular expressions and their corresponding actions. The User Subroutines are the function definitions of the functions that are used in the Lex actions.

Things to remember:  
1. If there is no R.E for the input string , it will be copied to the standard output.  
2. The Lex resolves the ambiguity in case of matching by choosing the longest match first or by choosing the rule given first.  
3. All the matched expressions are contained in yytext whose length is yyleng.

When shall we use the Lex Substitution Strings ?  
Lex Substitution Strings are of importance when the regular expressions become very large and unreadable. In that case it is better to break them into smaller substitution strings and then use them in the Rules Sections. Another use is when a particular Regular Expression appears quite often in number of Rules.

When shall one use the Start states ?  
Start states helps in resolving the reduce -reduce error in the Parser. It is particularly important when one wants to return different tokens for the same Regular Expression depending upon what is the previous scanned token.  
e.g suppose we have two tokens /keywords CustName & ItemName followed by a string. If the BNF is such that the string reduces to two non terminals then there is reduce/reduce conflict. To avoid this it will be better to return two different tokens for the string depending upon whether CustName was scanned previously or ItemName.

How can one determine correct Table Size Declarations?  
There is indeed no hard and fast rule for the above. The best method is not to give any sizes initially. The default table sizes are sufficient for small applications. However if the limits are crossed then Lex itself will generate errors messages giving info on the size violations. Then one can experiment by incrementally increasing the corresponding sizes.

When does it become important to give 'C' code to identify a token instead of a R.E?  
Sometimes one might notice that after adding one more R.E to the Lex rules , the number of Lex states increases tremendously. This happens when the particular R.E has a subexpression which clashes with other R.E. Huge number of states implies more case statements and hence reduces the execution speed of yylex functions. Thus it is more economical to identify an initial yet deterministic part of the token and then use  input() & unput() functions to identify the remaining part of the token.  
One nice example is that of a 'C' comment ( /\* comment \*/). After identifying the '/\*' one can write a simple 'C' action to identify the remaining token.

How do we redirect the Stdin to a particular FILE \* ?  
Normally the Lex takes the input from the Standard input through a macro definition  
  
#define  lex\_input()    (((yytchar=yysptr>yysbuf?U(\*--yysptr):getc(yyin))==10?                                              (yylineno++,yytchar):yytchar)==EOF?0:yytchar)

To redirect the yyyin from stdin just do the following  
        FILE \* fp=fopen("Filename","r");  
        yyin=fp;  
Another  leagal but bad workaround is to redefine this Macro in the definitions section by replacing yyin with fp. However this will always give a Warning during compilation.

What is the significance of the yywrap() function ?  
The yywrap() function is executed when the lexical analyzer reaches the end of the input file. It is generally used to print a summary of lexical analysis or to open another file for reading. The yywrap() function should return 0 if it has arranged for additional input, 1 if the end of the input has been reached.

***YACC-Yet Another Compiler-Compiler***

Yacc is the Utility which generates the function 'yyparse' which is indeed the Parser. Yacc describes a context free , LALR(1) grammar and supports both bottom-up and top-down parsing.The general format for the YACC file is very similar to that of the Lex file.

          1. Declarations  
          2. Grammar Rules  
          3. Subroutines  
In Declarations apart from the legal 'C' declarations there are few Yacc specific declarations which begins with a %sign.  
  
          1.  %union    It defines the Stack type for the Parser.  
                        It is a union of various datas/structures/                         objects.   
  
          2.  %token    These are the terminals returned by  the yylex  
                        function to the yacc. A token can also have type  
                        associated with it for good type checking and   
                        syntax directed translation. A type of a token   
                        can be specified as %token <stack member>  
                        tokenName.  
  
          3. %type      The type of a non-terminal symbol in  
                        the Grammar rule can be specified with this.  
                        The format is %type <stack member>  
                        non-terminal.  
  
          4. %noassoc   Specifies that there is no associativity   
                        of a terminal symbol.  
  
          5. %left      Specifies the left associativity of  
                        a Terminal Symbol  
  
          6. %right     Specifies the right assocoativity of  
                        a Terminal Symbol.  
  
          7. %start     Specifies the L.H.S non-terminal symbol of a  
                        production rule which should be taken as the  
                        starting point of the grammar rules.  
  
          8. %prec     Changes the precedence level associated with  
                       a particular rule to that of the following  
                       token name or literal.  
                       The grammar rules are specified as follows:  
                       Context-free grammar production-  
                         p->AbC  
                       Yacc Rule-  
                           p : A b C   { /\*   'C' actions   \*/}  
The general style for coding the rules is to have all Terminals in upper-case and all non-terminals in lower-case.  
To facilitates a proper syntax directed translation the Yacc has something called pseudo-variables which forms a bridge between the values of terminal/non-terminals and the actions. These pseudo variables are $$,$1,$2,$3......   The $$ is the L.H.S value of the rule whereas $1 is the first R.H.S value of the rule and so is $2 etc. The default type for pseudo variables is integer unless they are specified by %type ,   
%token <type> etc.

How are Recursions handled in the grammar rule ?  
Recursions are of two types left or right recursions. The left recursion is of form  
           list :  item    { /\* first item \*/ }  
                 |  list  item  { /\* rest of the items \*/ }  
The right recursion is of form  
          list  : item  { /\* first item \*/ }  
                 | item list  { /\* rest of items \*/ }  
In right Recursion the Parser is a bit bigger then that of left recursion and the items are matched from right to left.

How are symbol table or data structures built in the actions ?  
For a proper syntax directed translation it is important to make full use of the pseudo variables. One must have structures/classes defined of all the productions which can form a proper abstraction. The yystack should then be a union of pointers of all such structures/classes. The reason why pointers should be used instead of structures is to save space and also to avoid copying structures when the rule is reduced. Since the stack is always updated i.e on reduction the stack elements are popped and repaced by L.H.S so any data that was refered gets lost.

# 

 SYMBOL TABLES

A compiler uses a symbol table to keep track of scope and binding information about names. The symbol table is searched every time a name is encountered in the source text. Changes to the symbol table occur if a new name or new information about an existing name is discovered.

A symbol table mechanism must allow us to add new entries and find existing entries. The two symbol table mechanisms are linear lists and hash tables. Each scheme is evaluated on the basis of time required to add n entries and make e inquiries. A linear list is the simplest to implement, but its performance is poor when n and e are large. Hashing schemes provide better performance for greater programming effort and space overhead.

It is useful for a compiler to be able to grow the symbol table dynamically at compile time. If the symbol table is fixed when the compiler is written, the size must be chosen large enough to handle any source program that might be presented.

### SYMBOL-TABLE ENTRIES

Each entry in the symbol table is for the declaration of a name. The format of entries does have to be uniform, because the information saved about a same depends on the usage of time. Each entry can be implemented as a record consisting of a sequence of consecutive words of memory. To keep symbol table entries uniform, it may be convenient for some of the information about a name to be kept outside the table entry, with only a pointer to this information stored in the record.

Information is entered into the symbol table at various times. Keywords are entered initially. The lexical analyzer looks up sequences of letters and digits in the symbol table to determine if a reserved keyword or a name has been collected. With this approach, keywords must be in the symbol table before lexical analysis begins. Alternatively, if lexical analyzer intercepts reserved keywords, they should be entered into the symbol table with a warning of their possible use as a keyword.

The symbol table entry itself can be set up when the role of a name becomes clear, with the attribute values being filled in as the information become available. In some cases, the entry can be initiated from the lexical analyzer as soon as a name is seen in the input. More often, one name may denote several different objects, even in the same block or procedure.

For example, the C declarations

int x;

struct x {float y, z;};

use x as both an integer and as the tag of a structure with two fields. In such cases, the lexical analyzer can only return to the parser the name itself rather than a pointer to the symbol table entry. The record in the symbol table is created when the syntactic role played by the name is discovered. Attributes of a name are entered in response to declarations. Labels are identifiers followed by a colon, so one action associated with recognizing such an identifier may be to enter this fact into symbol table.

**CHARACTERS IN A NAME**

There is a distinction between the token id for an identifier or name, the lexeme consisting of the character string forming the name, and the attributes of the name. The lexeme is needed when a symbol table entry is set up for the first time and when we look up a lexeme found in the input to determine whether it is a name that has already appeared. A common representation of a name is a pointer to a symbol table entry for it.

If there is a modest upper bound on the length of a name, then the characters in the name can be stored in the symbol table entry as shown in figure. If there is no limit on the length of a name the indirect scheme can be used. Rather than allocating in each symbol table entry the maximum possible amount of space to hold a lexeme, utilize space more efficiently if there is only space for a pointer in the symbol table entry. In the record for a name, a pointer is placed to a separate array of characters giving the position of the first character of the lexeme. The indirect scheme permits the size of the name field of the symbol table entry itself to remain a constant.

The complete lexeme constituting a name must be stored to ensure that all uses of the same name can be associated with the symbol table record.

Name Attributes

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | |  |
| s | o | r | t |  |  |  |  |  |  |  |
| a |  |  |  |  |  |  |  |  |  |  |
| r | e | a | d | A | r | r | a | Y |  |  |
| i |  |  |  |  |  |  |  |  |  |  |
|  | | | | | | | | | |  |

In fixed size space within a record.

#### STORAGE ALLOCATION INFORMATION

Information about the storage locations that will be bound to names at run time is kept in the

Symbol table If machine code is to be generated by the compiler, then the position of each data object relative to a fixed origin, such as the beginning of an activation record must be ascertained. The same remark applies to a block of data loaded as a module separate from the program. For example, COMMON blocks in Fortran are loaded separately, and the positions of names relative to the beginning of the COMMON block in which they lie must be determined.

**THE LIST DATA STRUCTURE FOR SYMBOL TABLES**

The simplest and easiest to implement data structure for a symbol table is a linear list of records. Arrays are used to store their names and their associated information. New names are added to the list in the order in which they are encountered. The position of the end of the array is marked by the pointer available, pointing to where the next symbol table entry will go. The search for a name proceeds backwards from the end of the array to the beginning. When the name is located, associated information can be found in the words following next. If we reach the beginning of the array without finding the name, a fault occurs- an expected name is not in the table.

Making for an entry for a name and looking up the name in the symbol table are independent operations. In a block-structured language, an occurrence of a name is in the scope of the most closely nested declaration of the name. This scope can be implemented by making a fresh entry for a name every time it is declared. A new entry is made in the words immediately following the pointer available; that pointer is increased by the size of the symbol table record. Since entries are inserted in order, starting from the beginning of the array, they appear in the order they are created in.

|  |
| --- |
| id1 |
| Info1 |
| Id2 |
| Info2 |
| ……….. |
| Infon |
|  |

available\_\_\_\_\_\_\_\_

A linear list of records

If the symbol table contains n names the work necessary to insert a new name is constant if we do the insertion without checking to see the name is already in the table. If multiple entries for names are not allowed look the entire table before discovering that a name is not in the table. Insertions and inquiries take time proportional to n, names and m inquiries is at most c n (n + e),c is a constant.

Hash tables

Variation of the searching technique is known as hashing. Open hashing refers to the property that there need be no limit on the number of entries that can be in the table. This scheme gives the capability of performing e enquiries on n names in time proportional to n (n + e) / m. Since m can be made large up to n this is more efficient than linear list. In the basic hashing scheme, there are two parts to the data structure:

1.         A hash table consisting of a fixed array of m pointers to table entries.

2.         Table entries organized into m separate linked lists, called buckets. Each record in the symbol table appears on exactly one of these lists. Storage for the records may be drawn from an array of records. The dynamic storage allocations facilities of the implementation language can be used to obtain space for the records.

To determine whether there is an entry for string s in the symbol table, apply a hash function h to s, such that h(s) returns an integer between 0 and m-1. If s is in the symbol table, then it is on the list numbered h(s). If s is not in the symbol table, it is entered by creating a record for s that is linked at the front of the list numbered h (s). The average list is n/m record long if there are n names in a table of size m. By choosing m so that n/m is bounded by a small constant the time to access a table entry is constant. This space taken by the symbol table consist m words for the hash table and cn words for the table entries, where c is the no. of words per table entry. Thus the space for the hash table depends only on m and the space for table entries depends only on the number of entries.

The choice of m depends on the intended application for a symbol table. One suitable approach for computing hash functions is to proceed as follows:

1. Determine a positive integer h from the characters c1,c2, ……….ck in string s. The conversion of single characters to integers is usually supported by the implementation language.
2. Convert the integer h determined above into the no. of the list. Divide by m and take the reminder.

A technique for computing h is to add up the integer values of the characters in a string. Multiply the old value of h by a constant @ before adding in the next character.

That is h0=0, hi=@ hi-1+ci

(1)               #define PRIME 211

(2)               #define EOS ‘\0’

(3)               int hashpjw(s)

(4)               char \*s;

(5)               {

(6)               char \*p;

(7)               unsigned h=0, g;

(8)               for( p=0;\*p !=EOS; p=p+1){

(9)               h=(h << 4)+(\*p)’

(10)           if(g= h@0xf0000000){

(11)           h=h^(g >>24);

(12)           h=h ^ g ;

(13)           }

(14)           }

(15)           return h % PRIME;

(16)           }

In the hash function hashpjw, the sizes included the first primes larger than 100,200,……,1500.A close second was the function that computed the old value by 6559,ignoring overflows, adding in the next character. Function hashpjw is computed by starting with h=0.For each character c, shift bits of h left 4 positions and add in c. If any of four high-order bits of h is 1, shift four bits right 24 positions , exclusively-or them into h, and reset to 0 any of the high order bits that was 1.

Representing scope information

The entries in the symbol table are for declarations of names.When an occurrence of a name in the source text is looked up in the symbol table,the entry for the appropriate declration of that name must be returned.

A simple approach is to maintain a separate symbol table for each scope.Information for the nonlocals of a procedure is found by scanning the symbol tables for the existing program.With this approach the symbol table is integrated into the intermediate representation of the input.Most closely nested scope rules can be implemented by adapting the data structures.We keep track of the local names of the procedure by giving each procedure a unique number.The number of each procedure can be computed in a syntax directed manner from semantic rules that recognize the beginning and ending of each procedure.The procedure number is made a part of all locals declared in the procedure.

When we look up a newly scanned name,a match occurs only if the characters of the name match an entry character for character,and the associated number in the symbol table entry is the number of the procedure which is processed.Most closely nested scope rules can be implemented in terms of the following operations on a name:

Lookup : find the most recently created entry

Insert : make a new entry

Delete : remove the most recently created entry

Deleted entries must be preserved,they are just removed from the active symbol table.In a one-pass compiler ,information in the symbol table about a scope consisting of a procedure body ,is not needed at compile time after the procedure body is processed.However ,it may be needed at run time.In this case ,the information in the symbol table must be added to the generated code for user by the linker .

When a linear list consisting of an array of records was described ,it was said that lookup can be implemented by inserting entries at one end .A scan starting from the end and proceeding to the beginning of an array,finds the most recently created entry for the name.A pointer front points to the most recently created entry in the list.The implementation of the insert takes constant time because a new entry is created at the front of the list.The implementation of the lookup is done by scanning the list starting from entry pointed by front and following links until the desired one is found.

A hash table consists of m lists accessed through an array.For implementing the delete operation ,we would rather not have to scan the entire hash table.Suppose each entry has two links

1. 1.      A hash link that chains the entry to other entries whose names hash to the same value
2. 2.      A scope link that chains all entries in the same scope

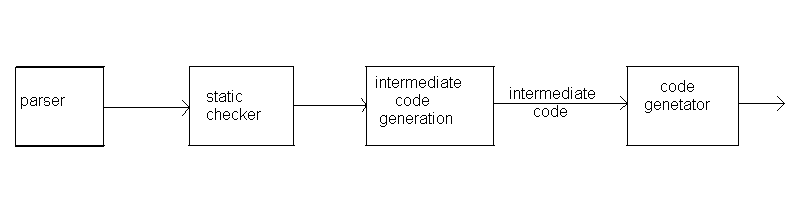
Deletion of entries from the hash table must be done with the care, because deletion of an affects the previous one on its list.When we delete the i-1 st entry points to i+1 st entry.

The i-1 st entry can be found if the hash links from a circular link list.We can use a stack to keep track of the lists containing entries to be deleted.A marker is placed in the stack when a new procedure is scanned.When we finish processing the procedure,the list numbers can be popped from the stack until the marker for the procedure is reached.

# **INTERMEDIATE CODE GENERATION**

Introduction

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



# 

# **INTERMEDIATE LANGUAGES**

There are three types of intermediate representation:-

1. Syntax Trees

2. Postfix notation

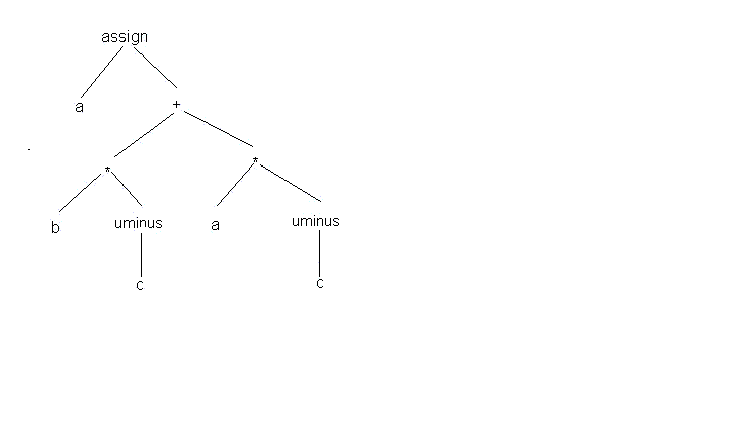
3. Three Address Code

Semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees of for generating postfix notation.

Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified. A syntax tree for

the assignment statement a:=b\*-c+b\*-c appear in the figure.

.fig8.2

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

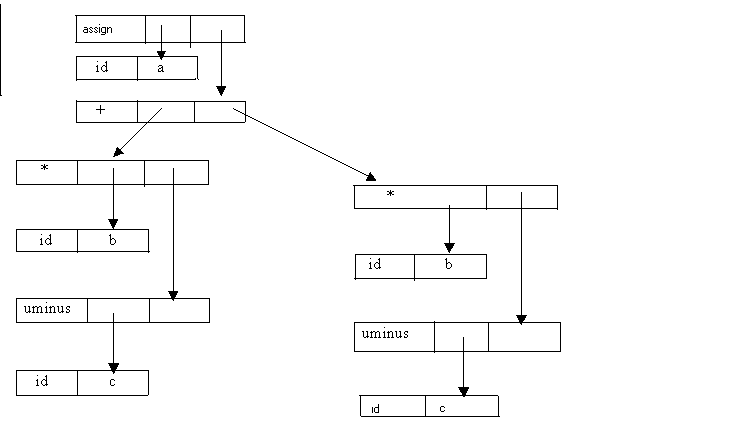
a b c uminus + b c uminus \* + assign

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

Syntax tree for assignment statements are produced by the syntax directed definition in fig.

|  |  |
| --- | --- |
| Production | Semantic Rule |
| S → **id** **:**= E | S**.**nptr **:**= mknode( ‘assign’, mkleaf(**id**, **id.**place), E**.**nptr) |
| E → E1 + E2 | E**.**nptr **:**= mknode(‘+’, E1**.**nptr ,E2**.**nptr) |
| E → E1 \* E2 | E**.**nptr **:**= mknode(‘\* ’, E1**.**nptr ,E2**.**nptr) |
| E → - E1 | E**.**nptr **:**= mkunode(‘uminus’, E1**.**nptr) |
| E → ( E1 ) | E**.**nptr **:**= E1**.**nptr |
| E → **id** | E**.**nptr **:**= mkleaf(**id**, **id.**place) |

This same syntax-directed definition will produce the dag if the functions mkunode(op, child) and mknode(op, left, right) return a pointer to an existing node whenever possible, instead of constructing new nodes. The token id has an attribute place that points to the symbol-table entry for the identifier id.name, representing the lexeme associated with that occurrence of id. lf the lexical analyzer holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme. Two representations of the syntax tree in Fig8.2 appear in Fig.8.4. Each node is represented as a record with a field for its operator and additional fields for pointers to its children. In Fig 8.4(b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position IO.

fig8.4(a)

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | id | b |  |
| 1 | id | c |  |
| 2 | uminus | 1 |  |
| 3 | \* | 0 | 2 |
| 4 | id | b |  |
| 5 | id | c |  |
| 6 | uminus | 5 |  |
| 7 | \* | 4 | 6 |
| 8 | + | 3 | 7 |
| 9 | id | a |  |
| 10 | assign | 9 | 8 |
| 11 | …… |  |  |

### fig8.4(b)

### 

### 

### Three-Address Code

Three-address code is a sequence of statements of the general form

X:= Op Z

where x, y, and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement. Thus a source language expression like x+y\*z might be translated into a sequence

Z

X +

where t1 and t2 are compiler-generated temporary names. This unraveling cf complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization. The use of names for the intermediate values computed by a program allow- three-address code to be easily rearranged – unlike postfix notation. three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag in Fig. 8.2 are represented by the three-address code sequences in Fig. 8.5. Variable names can appear directly in three-address statements, so Fig. 8.5(a) has no statements corresponding to the leaves in Fig. 8.4.

Code for syntax tree

t1 := -c

t2 := b \* t1

t3 := -c

t4 := b \* t3

t5 := t2 + t4

a := t5

Code for DAG

t1 := -c

t2 := b \* t1

t5 := t2 + t2

a := t5

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

### Types Of Three-Address Statements

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding inter- mediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using ”back patching,” discussed in Section 8.6. Here are the common three-address statements used in the remainder of this book:

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

2. Assignment instructions of the form x:= op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3. Copy statements of the form x: = y where the value of y is assigned to x.

4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

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

6. param x and call p, n for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of three-address statements

param x1

param x2

param xn

call p, n

generated as part of a call of the procedure p(x,, x~,..., x”). The integer n indicating the number of actual parameters in ”call p, n” is not redundant because calls can be nested. The implementation of procedure calls is outline d in Section 8.7.

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

8. Address and pointer assignments of the form x:= &y, x:= \*y and \*x: = y. The first of these sets the value of x to be the location of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an I-value such as A[i, j], and x is a pointer name or temporary. That is, the r-value of x is the l-value (location) of some object!. In the statement x: = ~y, presumably y is a pointer or a temporary whose r- value is a location. The r-value of x is made equal to the contents of that location. Finally, +x: = y sets the r-value of the object pointed to by x to the r-value of y.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must clearly be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restricted instruction set may force the front end to generate long sequences of statements for some source, language operations. The optimizer and code generator may then have to work harder if good code is to be generated.

### Syntax-Directed Translation into Three-Address Code

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of non-terminal E on the left side of E 🡪 E1 + E will be computed into a new temporary t. In general, the three- address code for id: = E consists of code to evaluate E into some temporary t, followed by the assignment id.place: = t. If an expression is a single identifier, say y, then y itself holds the value of the expression. For the moment, we create a new name every time a temporary is needed; techniques for reusing temporaries are given in Section S.3. The S-attributed definition in Fig. 8.6 generates three-address code for assignment statements. Given input a: = b+ – c + b+ – c, it produces the code in Fig. 8.5(a). The synthesized attribute S.code represents the three- address code for the assignment S. The non-terminal E has two attributes:

1. E.place, the name that will hold the value of E, and

2. E.code, the sequence of three-address statements evaluating E.

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





These attributes represent labels created by a function new label that returns a new label every time it is called. Note that S.after becomes the label of the statement that comes after the code for the while statement. We assume that a non-zero expression represents true; that is, when the value of F becomes zero, control leaves the while statement. f:expressions that govern the flow of control may in general be Boolean expressions containing relational and logical operators. The semantic rules for while statements in Section 8.6 differ from those in Fig. 8.7 to allow for flow of contro1 within Boolean expressions. Postfix notation -an be obtained by adapting the semantic rules in Fig. 8.6 (or see Fig. 2.5). 1he postfix notation for an identifier is the identifier itself. The rules for the other productions concatenate only the operator after the code for the operands. For example, associated with the production E – E, is the semantic rule

E.code:= E1.code || ’uminus’

1n general, the intermediate form produced by the syntax-directed translations in this chapter can he changed by making similar modifications to the semantic rules.

### Implementations of three-Address Statements

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

###### **Quadruples**

A quadruple is a record structure with four fields, which we call op, arg l, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unary operators like x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result. The quadruples in Fig. H.S(a) are for the assignment a: = b+ – c + b i – c. They are obtained from the three-address code in Fig. 8.5(a). The contents of fields arg 1, arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

###### **Triples**

To avoid entering temporary names into the symbol table. we might refer to a temporary value bi the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: op, arg 1 and arg2, as in Fig. 8.8(b). The fields arg 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). Since three fields are used, this intermediate code format is known as triples.’ Except for the treatment of programmer-defined names, triples correspond to the representation of a syntax tree or dag by an array of nodes, as in Fig. 8.4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | op | Arg1 | Arg2 |
| (0) | uminus | c |  |
| (1) | \* | b | (0) |
| (2) | uminus | c |  |
| (3) | \* | b | (2) |
| (4) | + | (1) | (3) |
| (5) | **:**= | a | (4) |

fig8.8(a) Qudraples fig8.8(b) Triples

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves. In practice, the information needed to interpret the different kinds of entries in the arg 1 and arg2 fields can be encoded into the op field or some additional fields. The triples in Fig. 8.8(b) correspond to the quadruples in Fig. 8.8(a). Note that the copy statement a:= t5 is encoded in the triple representation by placing a in the arg 1 field and using the operator assign. A ternary operation like x[ i ]: = y requires two entries in the triple structure, as shown in Fig. 8.9(a), while x: = y[i] is naturally represented as two operations in Fig. 8.9(b).



###### **Indirect Triples**

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples. For example, let us use an array statement to list pointers to triples in the desired order. Then the triples in Fig. 8.8(b) might be represented as in Fig. 8.10.



### Comparison of Representations: The Use of Indirection

The difference between triples and quadruples may be regarded as a matter of how much indirection is present in the representation. When we ultimately produce target code, each name, temporary or programmer-defined, will be assigned some run-time memory location. This location will be placed in the symbol-table entry for the datum. Using the quadruple notation, a three- address statement defining or using a temporary can immediately access the location for that temporary via the symbol table.

A more important benefit of quadruples appears in an optimizing compiler, where statements are often moved around. Using the quadruple notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use. If we move a statement computing x, the statements using x require no change. However, in the triples declaration. moving a statement that defines a temporary value requires us to change all references to that statement in the arg 1 and arg2 arrays. This problem makes triples difficult to use in an optimizing compiler.

Indirect triples present no such problem. A statement can be moved by reordering the statement list. Since pointers to temporary values refer to the r>p-arg 1-arg2 array(s), which are not changed, none of those pointers need be changed. Thus, indirect triples look very much like quadruples as far as their utility is concerned. The two notations require about the same amount of space and they are equally efficient for reordering of code. ~s with ordinary triples. Allocation of storage to those temporaries needing it must be deferred in the code generation phase. However, indirect triples can save some space compared with quadruples if the same temporary value is used more than once. The reason is that two or more entries in the statement array can point to the same line of the op-arg1-arg2 structure. For example, lines (l4) and (l6) of Fig. 8.10 could be combined and we could then combine (l5) and (17).

# **DECLARATIONS**

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record. When the front end generates addresses, it may have a target machine in mind. Suppose that addresses of consecutive integers differ by 4 on a byte- addressable machine. The address calculations generated by the front end may therefore include multiplications by 4. The instruction set of the target machine may also favor certain layouts of data objects, and hence their addresses. We ignore alignment of data objects here, Example 7.3 shows how data objects are aligned by two compilers.

### Declarations in a Procedure

The syntax of languages such as C, Pascal, and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say offset, can keep track of the next avai1able relative address. In the translation scheme of Fig. S.I1 non-terminal P generates a sequence of declarations of the form id: T. Before ’.he first declaration is considered, offset is set to 0. As each new name is seen, that name is entered in the symbol table with offset equal to the current value of offset, and offset is incremented by the width of the data object denoted by that name. The procedure enter(name, type, offset) creates a symbol-table entry for name, gives it type and relative address offset in its data area. We use synthesized attributes type and width for non-terminal T to indicate the type and width, or number of memory units taken by objects of that type. Attribute type represents a type expression constructed from the basic types integer and real by applying the type constructors pointer and array, as in Section 6.l. If type expressions are represented by graphs, then attribute type might be a pointer to the node representing a type expression. In Fig. 8. I , integers have width 4 and real have width 8. The width of an array is obtained by multiplying the width of each element by the number of elements in the array.- The width of each pointer is assumed to be 4.

P 🡪 D

D 🡪 D ; D

D 🡪 id : T {enter (id.name, T.type, offset);

Offset:= offset + T.width }

T 🡪 integer {T.type :=integer;

T.width :=4}

T 🡪 real {T.type := real;

T.width := 8}

T 🡪 array [num ] of T1 {T.type :=array(num.val, T1.type);

T.width :=num.val X T1.width}

T 🡪 ^T1 {T.type :=pointer (T.type);

T.width:=4}

In Pascal and C, a pointer may be seen before we learn the type of the object pointed to Storage allocation for such types is simpler if all pointers have the same width. The initialization of offset in the translation scheme of Fig. 8.1 is more evident if the first production appears on one line as:

P 🡪{offset:= 0 } D

Non-terminals generating a. called marker non-terminals in Section 5.6, can be used to rewrite productions so that all actions appear at the ends of right sides. Using a marker non-terminal M, (8.2) can be restated as:

P 🡪 M D

M 🡪ε (offset:= 0}

### Keeping Track of Scope Information

In a language with nested procedures, names local to each procedure can be assigned relative addresses using the approach of Fig. 8.11 . When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will he illustrated by adding semantic rules to the following language.

P 🡪 D

D 🡪 D;D | id: T proc id; D;S

The production for non-terminals S for statements and T for types are not shown because we focus on declarations. The non-terminal T has synthesized attributes type and width, as in the translation scheme of Fig. For simplicity, suppose that there is a separate symbol table for each procedure in the language (8.3). One possible implementation of a symbol table is a linked list of entries for names. Clever implementations can be substituted if desired. A new symbol table is created when a procedure declaration D proc id D~; 5 is seen, and entries for the declarations in D~ are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations in Fig. 8.11 is that the procedure enter is told which symbol table to make an entry in. For example, symbol tables for five procedures are shown in Fig. 8. l2. The nesting structure of the procedures can be deduced from the links between the symbol tables; the program is in Fig. 7.22. The symbol tables for procedures readarray, exchange, and quicksort point back to that for the containing procedure sort, consisting of the entire program. Since partition is declared within quicksort, its table points to that of quick sort.



### The semantic rules are defined in terms of the following operations:

I. mktable(previous) creates a new symbol table and returns a pointer to the new table. The argument previous points to a previously created symbol table, presumably that for the enclosing procedure. The pointer previous is placed in a header for the new symbol table, along with additional information such as the nesting depth of a procedure. We can also number the procedures in the order they are declared and keep this number in the header.

2. enter(table, name, type, offset) creates a new entry for name *name* in the symbol table pointed to by table. Again, enter places type and relative address offset in fields within the entry.

3. addwidth(table, width) records the cumulative width of all the entries table in the header associated with this symbol table.

4. enterproc (table, name, newtable) creates a new entry for procedure name in the symbol table pointed to by table. The argument newtable points to the symbol table for this procedure name.

The translation scheme in Fig. S. l3 shows how data can be laid out in one pass, using a stack tblptr to hold pointers to symbol tables of the enclosing procedures. With the symbol tables ot’ Fig. 8.12, tblptr will contain pointers to the tables for -ort, quicksort, and partition when the declarations in partition are considered. The pointer to the current symbol table is on top. The other stack offset is the natural generalization to nested procedures of attribute offset in Fig. 8. l I. The top element of offset is the next available relative address for a local of the current procedure. All semantic actions in the sub-trees for B and C in

A B C { actionA}

are done before actionA the end of the production occurs. Hence, the action associated with the marker M in Fig. 8.l3 is the first to be done. The action for non-terminal M initializes stack tblptr with a symbol table for the outermost scope, created by operation mktable(nil). The action also pushes relative address 0 onto stack offset. The non-terminal V plays a similar role when a procedure declaration appears. Its action uses the operation mktable(top(tblptr)) to create a new symbol table. Here the argument top(tblptr) gives the enclosing scope of the new table. A pointer to the new table is pushed above that for the enclosing scope. Again, 0 is pushed onto offset.

For each variable declaration id: T. an entry is created for id in the current symbol table. This declaration leaves the stack pointer unchanged; the top of stack offset is incremented by T.width. when the action on the right side of D proc id: N D,; S occurs. the width of all

declarations generated by D1 is on top of stack offset.’, it is recorded using addwidth. and offset are then popped, and we revert to examining the declarations in the closing procedure. At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

P 🡪 M D {addwidth(top(tblptr), top(offset));

Pop(tblptr); pop(offset)}

M 🡪 ε { t := mktable(nil);

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

D 🡪 D1 ;D2

D 🡪 proc id ; N D1 ;S { t := top(tblptr);

addwidth(t.top(offset));

pop(tblptr); pop(offset);

enterproc(top(tblptr), id.name, t)}

D 🡪 id : T {enter(top(tblptr),id.name,T.type,top(offset));

top(offset) := top(offset) +T.width }

N 🡪 ε { t := mktable(top(tblptr));

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

### Field Names in Records

The following production allows non-terminal T to generate records in addition to basic types, pointers, and arrays:

T 🡪 record D end

The actions in the translation scheme of Fig. S.I4 emphasize the similarity between the layout of records as a language construct and activation records. Since procedure definitions do not affect the width computations in Fig. 8.13, we overlook the fact that the above production also allows procedure definitions to appear within records.

T 🡪 record L D end {T.type := record(top(tblptr));

T.width := top(offset);

Pop(tblptr); pop(offset) }

L🡪 ε { t:= mktable(nil);

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

After the keyword record is seen, the acting associated with the marker

creates a new symbol table for the field names. A pointer to this symbol table is pushed onto stack tblptr and relative address 0 is pushed onto stack . The action for D 🡪 id: T in Fig. 8.13 therefore enters information about the field name id into the symbol table for the record. Furthermore, the top of stack will hold the width of all the data objects within the record after the fields have been examined. The action following end in Fig. 8. 14 returns the width as synthesized attribute T.width. The type T.type is obtained by applying the constructor *record* to the pointer to the symbol table for this record.

# **ASSIGNMENT STATEMENTS**

Expressions can be of type integer, real, array and record in this section As part of the translation of assignments into three-address code, we show how names can be looked up in the symbol table and how elements of arrays and records can be accessed.

### 

### Names in the Symbol Table

In the previous Section we formed three-address statements using names themselves, with the understanding that the names stood for pointers to their symbol-table entries. The translation scheme in Fig. 8.15 shows how such symbol-table entries can be found. The lexeme for the name represented by id is given by attribute *id.name.* Operation *lookup (id.name)* checks if there is an entry for this occurrence of the name in the symbol table. If so, a pointer to the entry is returned; otherwise, *lookup* returns *nil* to indicate that no entry was found.

The semantic actions in Fig. 8.15 use procedure *emit* to emit three-address statements to an output file, rather than building up *code* attributes for non-terminals, as in Fig. 8.6. From Section 2.3, translation can be done by emitting to an output file if the *code* attributes of the non-terminals on the left side of productions are formed by concatenating the *code* attributes of the non-terminals on the right in the same order that the non-terminals appear on the right side. Perhaps with some additional strings in between. By reinterpreting the *lookup* operation in Fig.8.15, the translation scheme can be used even if the most closely nested scope rule applies to nonlocal names, as in Pascal. For concreteness, suppose that the context in which an assignment appears is given by the following grammar

P 🡪 MD

M 🡪 ε

D 🡪 d ; d | id :T | proc id ; N D ; S

N 🡪 ε

Non-terminal P becomes the new start symbol when these productions are added to those in Fig. 8.15.

S 🡪 id := D {p=lookup(id.name);

if p!=nil then

emit(P’:=’ E.place)

else error }

E 🡪 E1 + E2 {E.place := newtemp;

emit(E.place ‘:=’E1.place ‘+’ E2.place)}

E 🡪 E1 \* E2 {E.place ;= newtemp;

Emit(E.place ‘:=’ E1.place ‘\*’ E2.place)}

E 🡪 -E1 {E.place := newtemp;

Emit(E.place ‘:=’ ‘uminus’ E1.place)}

E 🡪 ( E1 ) {E.place := E1.place }

E 🡪 id { p:= lookup(id.name);

If p != nil then

E.place := p

else error}

Translation scheme to produce three-address code for assignments.

For each procedure generated by this grammar, the translation scheme in Fig. 8. l3 sets up a separate symbol table. Each such symbol table has a header containing a pointer to the table for the enclosing procedure. When the statement forming a procedure body is examined. A pointer to the symbol table for the procedure appears on top of the stack *tblptr.* This pointer is pushed onto the stack by actions associated with the marker non-terminal *N* on the right side of *D* 🡪proc id; *N D1 ; S.* Let the productions for non-terminal *S* be those in Fig. 8.l5. Names in an assignment generated by *S* must have been declared in either the procedure that *S* appears in. or in some enclosing procedure. When applied to *name,* the modified *lookup* operation first checks if *name* appears in the current symbol table, accessible through *top(tblptr).* If not, *lookup* uses the pointer in the header of a table to find the symbol table for the enclosing procedure and looks for the name there. If the name cannot be found in any of these scopes, then *lookup* returns *nil.*

For example, suppose that the symbol tables are as in Fig. 8.l2 and that an assignment in the body of procedure partition is being examined. Operation *lookup(i)* will find an entry in the symbol table for partition. Since v is not in this symbol table, lookup(v) will use the pointer in the header in this symbol table to continue the search in the symbol table for the enclosing procedure quicksort.

**Reusing Temporary Names**

 We have been going along assuming that *newtemp* generates a new temporary name each time a temporary is needed. It is useful, especially in optimizing compilers, to actually create a distinct name each time *newtemp* is called. However, the temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values. Temporaries can be reused by changing *newtemp.* An alternative approach of packing distinct temporaries into the same location during code generation is explored in the next chapter. The, bulk of temporaries denoting data are generated during the syntax-directed translation of expressions, by rules such as those in Fig. 8.15. The code generated by the rules for *E🡪 E1* + *E2* has the general form:

evaluate *E1* into t1

evaluate *E2 i*nto t2

t:= t1 + t2

From the rules for the synthesized attribute *E.place* it follows that t1 and t2 are not used elsewhere in the program. The lifetimes of these temporaries are nested like matching pairs of balanced parentheses. In fact, the lifetimes of all temporaries used in the evaluation of *E2* are contained in the lifetime of t1. It is therefore possible to modify *newtemp* so that it uses, as if it were a stack, a small array in a procedure’s data area to hold temporaries. Let us assume for simplicity that we are dealing only with integers. Keep a count *c,* initialized to zero. Whenever a temporary name is used as an operand, decrement *c* by 1. Whenever a new temporary name is generated, use $c and increase *c* by 1. Note that the ”stack” of temporaries is not pushed or popped at run time, although it happens that stores and loads of temporary values are made by the compiler to occur at the ”top.”

  Temporaries that may be assigned and/or used more than once, for example, in a conditional assignment, cannot be assigned names in the last-in first-out manner described above. Since they tend to be rare, all such temporary values can he assigned names of their own. The same problem of temporaries defined or used more than once occurs when we perform code optimization such as combining common sub-expressions or moving a computation out of a loop . A reasonable strategy is to create a new name whenever we create an additional definition or use for a temporary or move its computation.

|  |  |
| --- | --- |
| Statement | Value of c |
|  | 0 |
| $0**:**= a \* b | 1 |
| $1**:**= c \* d | 2 |
| $0**:**= $0 + $1 | 1 |
| $1**:**= e \* f | 2 |
| $0**:**= $0 - $1 | 1 |
| x **:**= $0 | 0 |

Fig8.16 Three-address code with stacked temporaries.

**Addressing Array Elements**

 Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w, then the *i th* element of array A begins in location

*base + (i – low) x w* (8.4)

where *low* is the lower bound on the subscript and *base is* the relative address of the storage allocated for the array. That is, *base* is the relative address of A[low]. The expression (8.4) can be partially evaluated at compile time if it is rewritten as

*i X w + (base – low x w)*

The sub-expression c = *base – low* x w can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A, so the relative address of A[i] is obtained by simply adding *i* x w to *c.* Compile-time pre-calculation can also be applied to address calculations of elements of multi-dimensional arrays. A two-dimensional array is normally stored in one of two forms, either *row-major* (row-by-row) or *column-major* (column-by-column). Figure 8.*l7* shows the layout of a 2x3 array A in (a) row-major form and (b) column-major form. Fortran uses column-major form; Pascal uses row-major form, because A[i,j] is equivalent to A[i] [ j], and the elements of each array A[i] are stored consecutively. In the case of a two-dimensional array stored in row-major form, the relative address of A[i1,*i2*] can be calculated by the formula

*base + ((i1 – low1) x n2 + i2 – low2) x w*

where *low1* and *low2* are the lower bounds on the values of *i1* and *i2* and *n2* is the number of values that *i2* can take. That is, if *high 2* is the upper bound on the value of *i2,* then *n2 = high2 – low2* + 1. Assuming that *i1* and *i2* are the only values that are not known at compile time, we can rewrite the above expression as

((i1 \* n2) + i2) X w + *(base –* ((low1 X n2) + low2) X w) (8. 5)



The last term in this expression can be determined at compile time. We can generalize row- or column-major form to many dimensions. The generalization of row-major form is to store the elements in such a way that, as we scan down a block of storage, the rightmost subscripts appear to vary fastest, like the numbers on an odometer. The expression (8.5) generalizes to the following expression for the relative address of A[i1, *i2,..., ik]*

( ( ……((i1n2+i2)n3+i3)…… )nk + ik) X w (8. 6)

+ *base –* (( ……((low1 n2 + low2) n3 +low3)…… )nk + lowk) x w

Since for all *j, nj = highj – low* + 1 is assumed fixed, the term on the second line of (8.6) can be computed by the compiler and saved with the symbol-table entry for A. Column-major form generalizes to the opposite arrangement, with the leftmost subscripts varying fastest. Some languages permit the sizes of arrays to be specified dynamically, when a procedure is called at run-time. The formulas for accessing the elements of such arrays are the same as for fixed-size arrays, but the upper and lower limits are not known at compile time.

The chief problem in generating code for array references is to relate the computation of (8.6) to a grammar for array references. Array references can be permitted in assignments if non-terminal *L* with the following productions is allowed where id appears in Fig. 8.15:

*L 🡪 id [Elist ] | id*

*Elist 🡪 Elist, E | E*

In order that the various dimensional limits *nj* of the array be available as we group index expressions into an *Elist,* it is useful to rewrite the productions as

*L 🡪 Elist] | id*

*Elist 🡪 Elist, E | id [ E*

That is, the array name is attached to the leftmost index expression rather than being joined to *Elist* when an *L* is formed. These productions allow a pointer to the symbol-table entry for the array name to be passed as a synthesized attribute *array* of *Elist.*

We also use *Elist.ndim* to record the number of dimensions (index expressions) in the *Elist.* The function *limit(array, j)* returns *nj,* the number of elements along the j th dimension of the array whose symbol -table entry is pointed to by *array.* Finally, *Elist.place* denotes the temporary holding a value computed from index expressions in *Elist.* An *Elist* that produces the first *m* indices of a k-dimensional array reference A[i1,*i2,..., ik* ] will generate three-address code to compute

( ……((i*1 n2* + i2)n3 + i3)…… )nm + im (8.7)

using the recurrence

e1 = i1

em = e(m-1) X nm + im (8.8)

Thus, when *m = k,* a multiplication by the width w is all that will be needed to compute the term on the firs( line of (8.6). Note that the *ij’s* here may really he values of expressions and code to evaluate those expressions will be interspersed with code to compute (8.7).

An l-value *L* will have two attributes. *L.place* and *L.offset.* In the case that *L* is a simple name. *L.place* will he a pointer to the, symbol-table entry for that name, and *L.offset* will be null, indicating that the l-value is a simple name rather than an array reference. The non-terminal *E* has the same translation *E.place,* with the same meaning as in Fig 8.15.

**The Translation Scheme for Addressing Array Elements**

Semantic actions will be added to the grammar:

1. 1.      S 🡪 L := E
2. 2.      E 🡪 E + E
3. 3.      E 🡪 ( E )
4. 4.      E 🡪 L
5. 5.      L 🡪 Elist ]
6. 6.      L 🡪 id
7. 7.      Elist 🡪 Elist , E
8. 8.      Elist 🡪 id [ E

As in the case of expressions without array references, the three-address code itself is produced by the *emit* procedure invoked in the semantic actions. We generate a normal assignment if *L* is a simple name, and an indexed assignment into the location denoted by *L* otherwise:

1. (1)   S 🡪 *L:=E* { if *L.offset =* null then /\* *L* is a simple id \*/

*emit (L.place ’:=’ E.place* );

else *emit (L.place’*[*’L.offset’*]’ ’:=’ *E.place)*

The code for arithmetic expressions is exactly the same as in Fig. 8.15:

(2) *E 🡪 E1* + E2 {*E.place:= newtemp;*

*emit (E.place ’:=’ E1.place ’+’ E2.place) }*

(3) *E* 🡪 ( *E1* ) { *E.place:= E1.place }*

When an array reference *L* is reduced to *E,* we want the r-value of *L.* Therefore we use indexing to obtain the contents of the location *L.place [L.offset*]:

(4) *E 🡪 L*  {if *L.offset =* null then /\* *L* is a simple id \*/ *E.place:= L.place*

else begin

*E.place: = newtemp;*

*emit (E.place ’:=’ L.place ’*[‘  *L.offset’*]’ )

end }

Below, *L.offset* is a new temporary representing the first term of (8.6); function *width(Elist.array)* returns w in (8.6). *L.place* represents the second term of (8.6), returned by the function *c(Elist.array).*

(5) *L 🡪 Elist* ] {L.*place:= newtemp;*

*L.offset:= newtemp;*

*emit(L.place ’:=’ c(Elist.array));*

*emit(L.offset ’:=’ Elist.place’\*width(Elist.array)))*

A null offset indicates a simple name.

(6) *L* 🡪 id *{ L.place:= id.place;*

*L.offset: = null*

When the next index expression is seen, we apply the recurrence (8.8). In the following action, *Elist1.place* corresponds to *e(m-1)* in (8.8) and *Elist.place* to *em .* Note that if *Elist1*  has *m–*l components, then *Elist* on the left side of the production has *m* components.

(7) *Elist 🡪 Elist1, E*  *{ t:= newtemp;*

*m:= Elist1.ndim + l;*

*emit(t ’:=’ Elist1.place ’\*’limit(Elist1.array, m));*

*emit(t ’:=’ t ’+’ E.place);*

*Elist.array: = Elist1.array;*

*Elist.place:= t;*

*Elist.ndim:= m }*

*E.place holds both the value of the expression E and the value of (8.7) for m=1.*

(8) *Elist* 🡪 id [ *E { Elist.array:= id.place;*

*Elist.place:= E.place;*

*Elist.ndim: = l )*



### Type Conversions Within Assignments

In practice, there would be many different types of variables and constants, so the compiler must either reject certain mixed-type operations or generate appropriate coercion (type conversion) instructions. Consider the grammar for assignment statements as above, but suppose there are two types – real and integer, with integers converted to reals when necessary. We introduce another attribute *E.type,* whose value is either *real* or *integer.* The semantic rule for *E.type* associated with the production *E 🡪 E* + *E* is:

*E 🡪 E+E {E.type :=*

*if E1.type = integer and*

*E2.type = integer then integer*

*else real }*

The entire semantic rule for *E 🡪 E* + *E.* and most of the other productions must be modified to generate, when necessary, three-address statements of the form x: = inttoreal y, whose effect is to convert integer y to a real of equal value, called x. We must also include with the operator code an indication of whether fixed or floating-point arithmetic is intended. The complete semantic action for a production of the form *E 🡪 E,* + *E.* is listed in Fig. 8. 19.

*E.place := newtemp;*

*if E1.type = integer and E2.type = integer then begin*

*emit(E.place’:=’ E1.place ’int+’ E2.place);*

*E.type: = integer*

end

else if *E1.type = real* and *E2.type = real* then begin

*emit (E.place ’:=’ E1.place* ’real +’ *E2.place);*

*E.type := real*

end

else if *E1.type = integer* and *E2.type = real* then begin

*u := newtemp;*

*emit(u ’:=’* ’inttoreal’ *E1.place);*

*emit(E.place ’:=’ u* ’real+’ *E2.place);*

*E.type:= real*

end

else if *E1.type = real* and *E2.type = integer* then begin

*u := newtemp;*

*emit(u ’:=’* ’inttoreal’ *E2.place);*

*emit(E.place ’:=’ E1.place* ’real+’ u);

*E.type: = real*

end

else

*E.type:= type error;*

*Fig. 8.19. Semantic action for E 🡪 E1 + E2.*

The semantic action of Fig. 8. l9 uses two attributes *E.place* and *E.type* for the non-terminal *E.* As the number of types subject to conversion increases. The number of cases that arise increases quadratically (or worse, if there are operators with more than two arguments). Therefore with large numbers of types, careful organization of the semantic actions becomes more important.

**Accessing Fields in Records**

 The compiler must keep track of both the types and relative addresses of the fields of a record. An advantage of keeping this information in symbol-table entries for the field names is that the routine for looking up names in the symbol table can also be used for field names. With this in mind, a separate symbol table was created for each record type by the semantic actions in Fig. 8.14. lf *r* is a pointer to the symbol table for a record type, then the type *record(t)* formed by applying the constructor *record* to the pointer was returned as *T.type* We use the expression

p↑.info + 1

to illustrate how a pointer to the symbol table can be extracted from an attribute *E.type.* From the operations in this expression it follows that p must be a pointer to a record with a field name info whose type is arithmetic. If types are constructed as in Fig. 8.13 and 8.14, the type of p must be given by a type expression

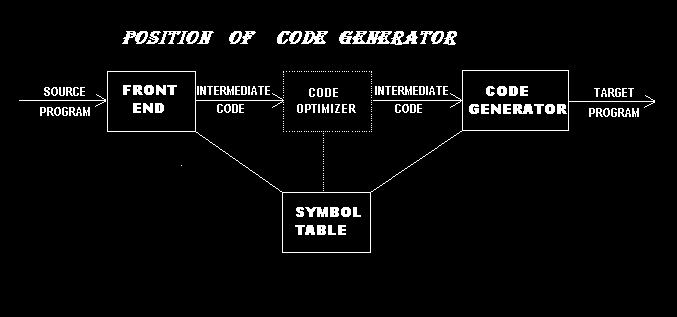
*pointer (record(t))*

The type of pt is then *record(t),* from which *t* can be extracted. The field name info is looked up in the symbol table pointed to by *t.*

***CODE GENERATION***

The final phase in our compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.



**fig. 1**

## 

## ISSUES IN THE DESIGN OF A CODE GENERATOR

  While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation, and evaluation order are inherent in almost all code generation problems.

**INPUT TO THE CODE GENERATOR**

  The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation.

There are several choices for the intermediate language, including: linear representations such as postfix notation, three address representations such as quadruples, virtual machine representations such as syntax trees and dags.

We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.). We also assume that the necessary type checking has take place, so type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.

TARGET PROGRAMS

  The output of the code generator is the target program. The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed quickly. A number of “student-job” compilers, such as WATFIV and PL/C, produce absolute code.

Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.

Producing an assembly language program as output makes the process of code generation somewhat easier .We can generate symbolic instructions and use the macro facilities of the assembler to help generate code .The price paid is the assembly step after code generation.

Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must uses several passes.

  MEMORY MANAGEMENT

  Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the “back patching”. Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as j: *goto i* is encountered, and i is less than j, the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple i. If, however, the jump is forward, so i exceeds j, we must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. Then we process quadruple i, we fill in the proper machine location for all instructions that are forward jumps to i.

 INSTRUCTION SELECTION

  The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three- address statement we can design a code skeleton that outlines the target code to be generated for that construct.

For example, every three address statement of the form x := y + z, where x, y, and z are statically allocated, can be translated into the code sequence

MOV y, R0 /\* load y into register R0 \*/

ADD z, R0 /\* add z to R0 \*/

MOV R0, x /\* store R0 into x \*/

Unfortunately, this kind of statement – by - statement code generation often produces poor code. For example, the sequence of statements

a := b + c

d := a + e

would be translated into

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

Here the fourth statement is redundant, and so is the third if ‘a’ is not subsequently used.

The quality of the generated code is determined by its speed and size.

A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may lead to correct, but unacceptably inefficient target code. For example if the target machine has an “increment” instruction (INC), then the three address statement a := a+1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, add one to the register, and then stores the result back into a.

MOV a, R0

ADD #1,R0

MOV R0, a

Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain. Deciding which machine code sequence is best for a given three address construct may also require knowledge about the context in which that construct appears.

 REGISTER ALLOCATION

  Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two subproblems:

1.           During **register allocation,** we select the set of variables that will reside in registers at a point in the program.

2.           During a subsequent **register assignment** phase, we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require **register pairs** (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

M x, y

where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form

D x, y

where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

Now consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c).

Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which ‘a’ is to be loaded depends on what will ultimately happen to e.

  t := a + b t := a + b

t := t \* c t := t + c

t := t / d t := t / d

1. (a)                                                                                                                      (b)

fig. 2 Two three address code sequences

L R1, a L R0, a

A R1, b A R0, b

M R0, c A R0, c

D R0, d SRDA R0, 32

ST R1, t D R0, d

ST R1, t

**(a) (b)**

**fig.3** Optimal machine code sequence

#### CHOICE OF EVALUATION ORDER

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

APPROCHES TO CODE GENERATION

  The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

##### RUN-TIME STORAGE MANAGEMENT

The semantics of procedures in a language determines how names are bound to storage during allocation. Information needed during an execution of a procedure is kept in a block of storage called an activation record; storage for names local to the procedure also appears in the activation record.

  An **activation record** for a procedure has fields to hold parameters, results, machine-status information, local data, temporaries and the like. Since run-time allocation and de-allocation of activation records occurs as part of the procedure call and return sequences, we focus on the following three-address statements:

1. 1.      call
2. 2.      return
3. 3.      halt
4. 4.      action, a placeholder for other statements

For example, the three-address code for procedures c and p in fig. 4 contains just these kinds of statements. The size and layout of activation records are communicated to the code generator via the information about names that is in the symbol table. For clarity, we show the layout in Fig. 4 rather than the form of the symbol-table entries.

We assume that run-time memory is divided into areas for code, static data, and a stack.

three-address code activation record for c activation record for p

(64 bytes) (64 bytes)

return address

|  |
| --- |
|  |

return address

|  |
| --- |
|  |

/\* code for c\*/

action1

call p

action2

halt

|  |
| --- |
|  |

0: 0:

## 8: 0:

buf

|  |
| --- |
|  |

arr

|  |
| --- |
|  |

## 8: 4:

/\* code for p\*/

action3

return

|  |
| --- |
|  |

## 

i

i

|  |
| --- |
|  |

## 

n

|  |
| --- |
|  |

## 56:

j

|  |
| --- |
|  |

## 84:

## 60:

## 

## 

## fig 4 . Input to a code generator

## 

## STATIC ALLOCATION

Consider the code needed to implement static allocation. A call statement in the intermediate code is implemented by a sequence of two target-machine instructions. A MOV instruction saves the return address, and a GOTO transfers control to the target code for the called procedure:

MOV #*here* +20, *callee.static\_area*

GOTO *callee.code\_area*

The attributes *callee.statatic\_area* and *callee.code\_area* are constants referring to the address of the activation record and the first instruction for the called procedure, respectively. The source #*here*+20 in the MOV instruction is the literal return address; it is the address of instruction following the GOTO instruction.

The code for a procedure ends with a return to the calling procedure ends with a return to the calling procedure, except the first procedure has no caller, so its final instruction is HALT, which presumably returns control to the operating system. A return from procedure callee is implemented by

GOTO \**callee.static\_area*

which transfers control to the address saved at the beginning of the activation record.

## Example 1: The code in Fig. 5 is constructed from the procedures c and p in Fig. 4. We use the pseudo-instruction ACTION to implement the statement action, which represents three-address code that is not relevant for this discussion. We arbitrarily start the code for these procedures at addresses 100 and 200, respectively, and assume that each ACTION instruction takes 20 bytes. The activation records for the procedures are statically allocated starting at location 300 and 364, respectively.

/\*code for c\*/

100: ACTION1

120: MOV #140,364 /\*save return address 140 \*/

132: GOTO 200 /\* call p \*/

140: ACTION2

160: HALT

……

/\*code for p\*/

200: ACTION3

220: GOTO \*364 /\*return to address saved in location 364\*/

……

/\*300-363 hold activation record for c\*/

300: /\*return address\*/

304: /\*local data for c\*/

…… /\*364-451 hold activation record for p\*/

364: /\*return address\*/

368: /\*local data for p\*/

**fig 5.** Target code for input in fig 4.

The instructions starting at address 100 implement the statements

action1 ; call p; action2; halt

of the first procedure c. Execution therefore starts with the instruction ACTION1 at address 100. The MOV instruction at address 120 saves the return address 140 in the machine-status field, which is the first word in the activation record of p. The GOTO instruction at address 132 transfers control to the first instruction is the target code of the called procedure.

Since 140 was saved at address 364 by the call sequence above, \*364 represents 140 when the GOTO statement at address 220 is executed. Control therefore returns to address 140 and execution of procedure c resumes.

## STACK ALLOCATION

## Static allocation can become stack allocation by using relative addresses for storage in activation records. The position of the record for an activation of a procedure is not known until run time. In stack allocation, this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register. The indexed address mode of our target machine is convenient for this purpose.

Relative addresses in an activation record can be taken as offsets from any known position in the activation record. For convenience, we shall use positive offsets by maintaining in a register SP a pointer to the beginning of the activation record on top of the stack. When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure. After control returns to the caller, it decrements SP, thereby de-allocating the activation record of the called procedure.

The code for the 1st procedure initializes the stack by setting SP to the start of the stack area in memory.

MOV #*stackstart*, SP /\*initialize the stack\*/

code for the first procedure

HALT /\*terminate execution\*/

A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

ADD #*caller.recordsize*, SP

MOV #*here+*16, SP /\* save return address\*/

GOTO *callee.code\_area*

The attribute *caller.recordsize* represents the size of an activation record, so the ADD instruction leaves SP pointing to the beginning of the next activation record. The source #*here*+16 in the MOV instruction is the address of the instruction following the GOTO; it is saved in the address pointed to by SP.

The return sequence consists of two parts. The called procedure transfers control to the return address using

GOTO \*0(SP) /\*return to caller\*/

The reason for using \*0(SP) in the GOTO instruction is that we need two levels of indirection: 0(SP) is the address of the first word in the activation record and \*0(SP) is the return address saved there.

The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. That is, after the subtraction SP points to the beginning of the activation record of the caller:

SUB #*caller.recordsize*, SP

## BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

# BASIC BLOCKS

A **basic block** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

t1 := a\*a

t2 := a\*b

t3 := 2\*t2

t4 := t1+t3

t5 := b\*b

t6 := t4+t5

A three-address statement x := y+z is said to *define* x and to *use* y or z. A name in a basic block is said to *live* at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 1: Partition into basic blocks.

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the set of **leaders**, the first statements of basic blocks.

The rules we use are the following:

I) The first statement is a leader.

II) Any statement that is the target of a conditional or unconditional goto is a leader.

III) Any statement that immediately follows a goto or conditional goto statement is a leader.

2.      For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example 3: Consider the fragment of source code shown in fig. 7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

begin

prod := 0;

i := 1;

do begin

prod := prod + a[i] \* b[i];

i := i+1;

end

while i<= 20

end

**fig 7**: program to compute dot product

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic blocks. statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block.

(1)   prod := 0

(2)   i := 1

(3)   t1 := 4\*i

(4)   t2 := a [ t1 ]

(5)   t3 := 4\*i

(6)   t4 :=b [ t3 ]

(7)   t5 := t2\*t4

(8)   t6 := prod +t5

(9)   prod := t6

(10)  t7 := i+1

(11) i := t7

(12)   if i<=20 goto (3)

**fig 8.** Three-address code computing dot product

TRANSFORMATIONS ON BASIC BLOCKS

A basic block computes a set of expressions. These expressions are the values of the names live on exit from block. Two basic blocks are said to be *equivalent* if they compute the same set of expressions.

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraic transformations.

STRUCTURE-PRESERVING TRANSFORMATIONS

The primary structure-preserving transformations on basic blocks are:

1. 1.      common sub-expression elimination
2. 2.      dead-code elimination
3. 3.      renaming of temporary variables
4. 4.      interchange of two independent adjacent statements

We assume basic blocks have no arrays, pointers, or procedure calls.

1. 1.      Common sub-expression elimination

Consider the basic block

a:= b+c

b:= a-d

c:= b+c

d:= a-d

The second and fourth statements compute the same expression,

namely b+c-d, and hence this basic block may be transformed into the equivalent block

a:= b+c

b:= a-d

c:= b+c

d:= b

Although the 1st and 3rd statements in both cases appear to have the same expression on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

1. 2.      Dead-code elimination

Suppose x is dead, that is, never subsequently used, at the point where the statement x:= y+z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

1. 3.      Renaming temporary variables

Suppose we have a statement t:= b+c, where t is a temporary. If we change this statement to u:= b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed. In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary. We call such a basic block a *normal-form* block.

4. Interchange of statements

Suppose we have a block with the two adjacent statements

t1:= b+c

t2:= x+y

Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2. A normal-form basic block permits all statement interchanges that are possible.

# CODE OPTIMIZATION

**Criteria for Code-Improving Transformations**

Simply stated, the best program transformations are those that yield the most benefit for the least effort. The transformations provided by an optimizing compiler should have several properties.

First, a transformation must preserve the meaning of programs. That is, an "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source program. The influence of this criterion pervades this chapter; at all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

Second, a transformation must, on the average, speed up programs by a measurable amount. Sometimes we are interested in reducing the space taken by the compiled code, although the size of code has less importance than it once had. Of course, not every transformation succeeds in improving every program, and occasionally an "optimization" may slow down a program slightly, as long as on the average it improves things.

Third, a transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. Certain local or "peephole" transformations of the kind are simple enough and beneficial enough to be included in any compiler.

Some transformations can only be applied after detailed, often time-consuming, analysis of the source program, so there is little point in applying them to programs that will be run only a few times. For example, a fast, nonoptimizing, compiler is likely to be more helpful during debugging or for "student jobs” that will be run successfully a few times and thrown away. Only when the program in question takes up a significant fraction of the machine's cycles does improved code quality justify the time spent running an optimizing compiler on the program.

Before we get into optimization as such we need to familiarize ourselves with a few things

# ALGEBRAIC TRANSFORMATION

# 

# Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set. The useful ones are those that simplify expressions or replace expensive operations by cheaper ones. For example, statements

such as

x := x +0

Or

x := x\*1

 can be eliminated from a basic block without changing the set of expressions it computes. The exponentiation operator in the statements

  x := y \*\* 2

 usually requires a function call to implement. Using an algebraic transformation, this statement can be replaced by cheaper, but equivalent statement

x := y\*y

# FLOW GRAPHS

# 

# We can add the flow-of –control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes of the flow graph are the basic blocks. One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B1 to block B2can be immediately follow B1in some execution sequence; that is, if

1. there is a conditional or unconditional jump from the last statement of B2, or
2. B2 immediately follow B1in the order of the program, and B1 does not end in the unconditional jump

B1 is a predecessor of B2, and B2is a successor of B1.

# ***Example 4:The flow graph of the program of fig. 7 is shown in fig. 9, B1 is the initial node.***

|  |
| --- |
| **Prod := 0**  **I:=1** |

**B1**

|  |
| --- |
| **t1 := 4 \* i**  **t2 := a [ t1 ]**  **t3 := 4 \* i**  **t4 := b [ t3 ]**  **t5 := t2 \* t4**  **t6:= prod + t5**  **t7:=i+1**  **i := t7**  **if I <= 20 goto B2** |

**B2**

**Fig .9** flow graph for program

REPRESENTATION OF BASIC BLOCKS

Basic Blocks are represented by variety of data structures. For example, after partitioning the three address statements by Algorithm 1, each basic block can be represented by a record consisting of a count of number of quadruples in the block, followed by a pointer to the leader of the block, and by the list of predecessors and successors of the block. For example the block B2 running from the statement (3) through (12) in the intermediate code of figure 2 were moved elsewhere in the quadruples array or were shrunk, the (3) in if i<=20 goto(3) would have to be changed.

# ***LOOPS***

**Loop** is a collection of nodes in a flow graph such that

1. All nodes in the collection are *strongly connected*; from any node in the loop to any other, there is path of length one or more, wholly within the loop, and

2. The collection of nodes has a unique *entry*, a node in the loop such that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

A loop that contains no other loops is called an *inner loop*.

## PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying “optimizing” transformations to the target program.

A simple but effective technique for improving the target code is *peephole optimization*, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this. We shall give the following examples of program transformations that are characteristic of peephole optimizations:

• Redundant-instructions elimination

• Flow-of-control optimizations

• Algebraic simplifications

• Use of machine idioms

REDUNTANT LOADS AND STORES

 If we see the instructions sequence

  (1)   MOV R0,a

  (2)   MOV a,R0

-we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

# ***UNREACHABLE CODE***

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1.In C, the source code might look like:

**#define debug 0**

**….**

**If ( debug ) {**

**Print debugging information**

**}**

In the intermediate representations the if-statement may be translated as:

**If debug =1 goto L2**

**Goto L2**

**L1: print debugging information**

**L2: …………………………(a)**

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**, (a) can be replaced by:

**If debug ≠1 goto L2**

**Print debugging information**

**L2: ……………………………(b)**

As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

**If debug ≠0 goto L2**

**Print debugging information**

**L2: ……………………………(c)**

As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto [L2. Then](mailto:L@.Then) all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

### FLOW-OF-CONTROL OPTIMIZATIONS

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

**goto L2**

**….**

**L1 : gotoL2**

by the sequence

**goto L2**

**….**

**L1 : goto L2**

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

**if a < b goto L1**

**….**

**L1 : goto L2**

can be replaced by

**if a < b goto L2**

**….**

**L1 : goto L2**

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

**goto L1**

**……..**

**L1:if a<b goto L2**

**L3: …………………………………..(1)**

may be replaced by

**if a<b goto L2**

**goto L3**

**…….**

**L3: ………………………………….(2)**

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

# ***ALGEBRAIC SIMPLIFICATION***

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

x := x+0

Or

x := x \* 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

ELIMINATION OF COMMON SUBEXPRESSIONS

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where its referenced when encountered again – of course providing the variable values in the expression still remain constant.

# ***ELIMINATION OF DEAD CODE***

Its possible that a large amount of dead(useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code

REDUCTION IN STRENGTH

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x² is invariably cheaper to implement as x\*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

# ***USE OF MACHINE IDIOMS***

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and

auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

**Getting Better Performance**

Dramatic improvements in the running time of a program-such as cutting the running time form a few hours to a few seconds-are usually obtained by improving the program at all levels, from the source level to the target level, as suggested by fig. At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations are performed.

Algorithmic transformations occasionally produce spectacular improvements in running time. For example, Bentley relates that the running time of a program for sorting N elements dropped from 2.02N^2 microseconds to 12Nlog2N microseconds then a carefully coded "insertion sort" was replaced by "quicksort".

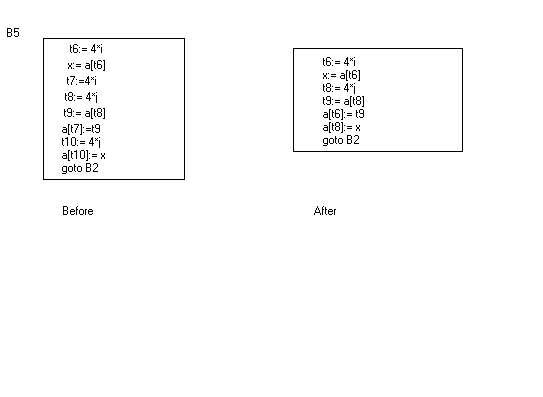
**THE PRINCIPAL SOURCES OF OPTIMIZATION**

Here we introduce some of the most useful code-improving transformations. Techniques for implementing these transformations are presented in subsequent sections. A transformation of a program is called local if it can be performed by looking only at the statements in a bas9ic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

**Function-Preserving Transformations**

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B5 shown in fig recalculates 4\*i and 4\*j.

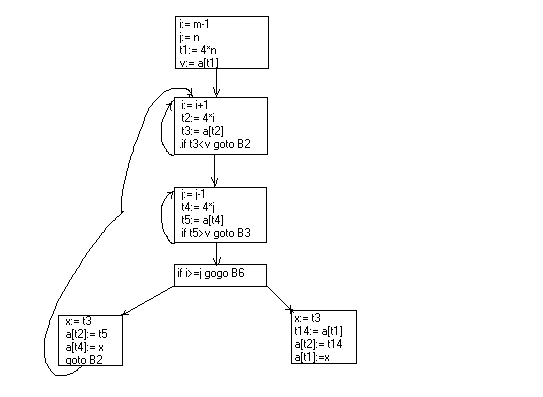


Local common subexpression elimination

**Common Subexpressions**

An occurrence of an expression E is called a common subexpression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value. For example, the assignments to t7 and t10 have the common subexpressions 4\*I and 4\*j, respectively, on the right side in Fig. They have been eliminated in Fig by using t6 instead of t7 and t8 instead of t10. This change is what would result if we reconstructed the intermediate code from the dag for the basic block.

Example: Fig shows the result of eliminating both global and local common subexpressions from blocks B5 and B6 in the flow graph of Fig. We first discuss the transformation of B5 and then mention some subtleties involving arrays.



B5 and B6 after common subexpression elimination

After local common subexpressions are eliminated B5 still evaluates 4\*i and 4\*j, as shown in the earlier fig. Both are common subexpressions; in particular, the three statements

t8:= 4\*j; t9:= a[t[8]; a[t8]:=x

in B5 can be replaced by

t9:= a[t4]; a[t4:= x using t4 computed in block B3. In Fig. observe that as control passes from the evaluation of 4\*j in B3 to B5, there is no change in j, so t4 can be used if 4\*j is needed.

Another common subexpression comes to light in B5 after t4 replaces t8. The new expression a[t4] corresponds to the value of a[j] at the source level. Not only does j retain its value as control leaves b3 and then enters B5, but a[j], a vlue computed into a temporary t5, does too becaoude there are no assignments to elements of the array a in the interim. The statement

t9:= a[t4]; a[t6]:= t9

in B5 can therefore be replaced by

a[t6]:= t5

The expression in blocks B1 and B6 is not considered a common subexpression although t1 can be used in both places.After control leaves B1 and before it reaches B6,it can go through B5,where there are assignments to a.Hence, a[t1] may not have the same value on reaching B6 as it did in leaving B1, and it is not safe to treat a[t1] as a common subexpression.

**Copy Propagation**

Block B5 in Fig. can be further improved by eliminating x using two new transformations. One concerns assignments of the form f:=g called copy statements, or copies for short. Had we gone into more detail in Example 10.2, copies would have arisen much sooner, because the algorithm for eliminating common subexpressions introduces them, as do several other algorithms. For example, when the common subexpression in c:=d+e is eliminated in Fig., the algorithm uses a new variable t to hold the value of d+e. Since control may reach c:=d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c:=d+e by either c:=a or by c:=b.

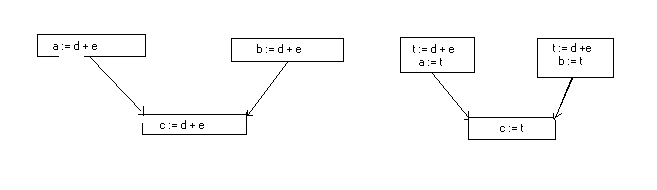
The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement f:=g. For example, the assignment x:=t3 in block B5 of Fig. is a copy. Copy propagation applied to B5 yields:

x:=t3

a[t2]:=t5

a[t4]:=t3

goto B2



Copies introduced during common subexpression elimination.

This may not appear to be an improvement, but as we shall see, it gives us the opportunity to eliminate the assignment to x.

**Dead-Code Eliminations**

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. For example, we discussed the use of debug that is set to true or false at various points in the program, and used in statements like

If (debug) print …

By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is false. Usually, it is because there is one particular statement

Debug :=false

That we can deduce to be the last assignment to debug prior to the test no matter what sequence of branches the program actually takes. If copy propagation replaces debug by false, then the print statement is dead because it cannot be reached. We can eliminate both the test and printing from the o9bject code. More generally, deducing at compile time that the value of an expression is a co9nstant and using the constant instead is known as constant folding.

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms 1.1 into

a [t2 ] := t5

a [t4] := t3

goto B2

**Loop Optimizations**

We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization: code motion, which moves code outside a loop; induction-variable elimination, which we apply to eliminate I and j from the inner loops B2 and B3 and, reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

**Code Motion**

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed ( a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

While (i<= limit-2 )

Code motion will result in the equivalent of

t= limit-2;

while (i<=t)

**Induction Variables and Reduction in Strength**

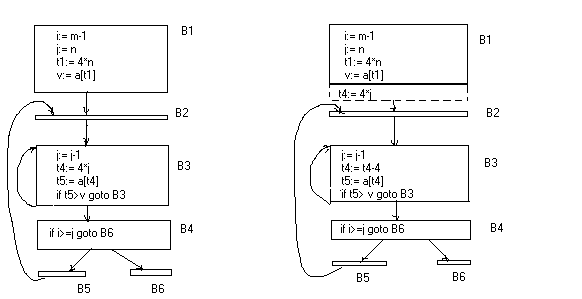
While code motion is not applicable to the quicksort example we have been considering the other two transformations are.Loops are usually processed inside out.For example consider the loop around B3.

Note that the values of j and t4 remain in lock-step;every time the value of j decreases by 1 ,that of t4 decreases by 4 because 4\*j is assigned to t4.Such identifiers are called induction variables.

When there are two or more induction variables in a loop, iit may be possible to get rid of all but one, by the process of induction-variable elimination.For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely.; t4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

 Example: As the relationship t4:=4\*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4\*j-4 must hold. We may therefore replace the assignment t4:= 4\*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4\*j on entry to the block B3, we place an intializations\ of t4 at the end of the blcok where j itself is initialized, shown by the dashed addt\ition to block B1 in second Fig.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.



Before After

strength reduction applied to 4\*j in block B3