Lecture 19

LL(1) Table Construction

Here now is the algorithm to construct a predictive parsing table.

1. For each production $A \rightarrow \alpha$

    1. for each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A,a].

    2. If ε is in FIRST(α), add $A \rightarrow \alpha$ to M[A,b] for each terminal b in FOLLOW(A). If ε is in FIRST(α), and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A,\$].

2. Make each undefined entry of M be error.

Let us apply the algorithm to the expression grammar. Since FIRST(TE') = FIRST(T) = { (, id }, the production $E \rightarrow TE'$ cause M[E,(] and M[E,id] to get $E \rightarrow TE'$. The production $E' \rightarrow +TE'$ causes M[E',+] to get $E' \rightarrow +TE'$. The production $E' \rightarrow \varepsilon$ causes M[E',)] and M[E',\$] to get $E' \rightarrow \varepsilon$ since FOLLOW (E') = {), \$ }. And so on. The final parsing table produced is:

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

Left Factoring

Consider the grammar

E → T + E | T
T → int | int T | (E)

It is impossible to predict because for T, two productions start with int. For E, it

is not clear how to predict; the two productions start with the non-terminal T. A grammar must be *left factored* before use for predictive parsing. The procedure to left-factor a grammar is as follows:

If $\alpha \neq \varepsilon$, replace all productions

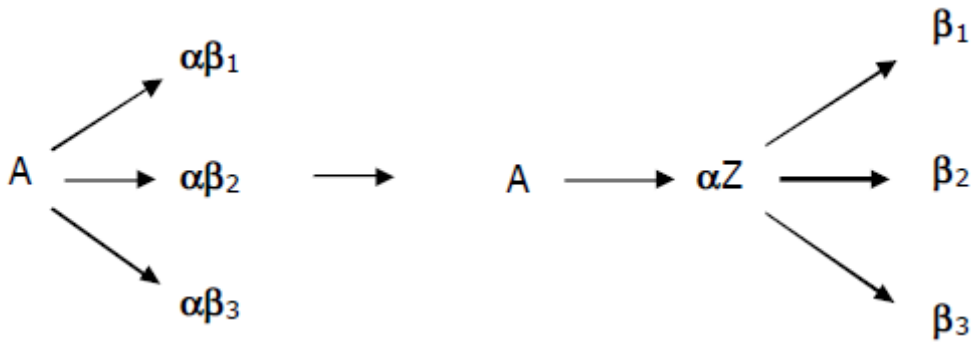
$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

with

$$A \rightarrow \alpha Z \mid \gamma$$
$$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where Z is a new non-terminal

A graphical explanation:



Example: consider following fragment of expression grammar

Factor → id
| id [ExprList]
| id (ExprList)

After left factoring, the grammar becomes

Factor → id Args
Args → [ExprList]
| (ExprList)
| ε

Given a CFG that does not meet the LL(1) condition, it is *un-decidable* whether or not an equivalent LL(1) grammar exists.

Lecture 20

Bottom-up Parsing

Bottom-up parsing is more general than top-down parsing. Bottom-up parsers handle a large class of grammars. It is the preferred method in practice. It is also called *LR* parsing; *L* means that tokens are read left to right and *R* means that the parser constructs a rightmost derivation. LR parsers do not need left-factored grammars. LR parsers can handle left-recursive grammars.

LR parsing *reduces* a string to the start symbol by inverting productions. A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$$

Each γ_i is a *sentential form.* If γ contains only terminals, γ is a *sentence* in L(G). If γ contains ≥ 1 nonterminals, γ is a *sentential form*. A bottom-up parser builds a derivation by working from input sentence *back* towards the start symbol S.

Consider the grammar

S → aABe
A → Abc | b
B → d

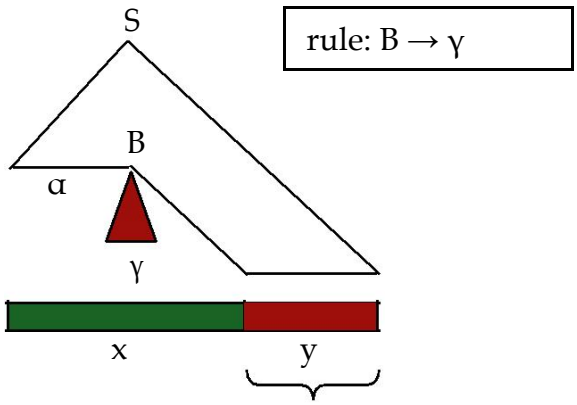
The sentence abbcde can be reduced to S:

abbcde
aAbcde
aAde
aABe
S

These reductions, in fact, trace out the following right-most derivation in reverse:

S → aABe
→aAde
→aAbcde
→abbcde

S →aBy →aγy →xy



Consider the grammar

1. E → E + (E)
2. | int

The bottom-up parse of the string int + (int) + (int) would be

int + (int) + (int)
E + (int) + (int) E
+ (E) + (int)
E + (int)
E + (E)
E

The consequence of an LR parser tracing a rightmost derivation in reverse is that given αβγ be a step of a bottom-up parse, assuming that next reduction is A → β τhen γ is a string of terminals. The reason is that αAγ → αβγ is a step in a rightmost derivation. This observation provides a strategy for building bottom up parsers: split the input string into two substrings. Right substring (a string of terminals) is as yet unexamined by parser and left substring has terminals and non-terminals. The dividing point is marked by a ► (the ► is not part of the string). Initially, all input is unexamined: ►x_1 x_1 . . . x_n.

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

1. Shift
2. Reduce

Shift moves ► one place to the right which shifts a terminal to the left string

E + (► int) ⇒ E + (int ►)

In the *reduce* action, the parser applies an inverse production at the right end of the left string. If E → E + (E) is a production, then

E + (E+(E)►) ⇒ E + (E ►)

Shift-Reduce Example

►int + (int) + (int) $	shift
int ► + (int) + (int) $	reduce E → int
E ► + (int) + (int) $	shift 3 times
E + (int ►) + (int) $	reduce E → int
E + (E ►) + (int) $	shift
E + (E) ► + (int) $	reduce E → E+(E)
E ► + (int) $	shift 3 times
E + (int ►) $	reduce E → int
E + (E ►) $	shift
E + (E) ► $	red E → E+(E)
E ► $	*accept*

Lecture 21

Shift-Reduce: The Stack

A stack can be used to hold the content of the left string. The Top of the stack is marked by the ► symbol. The shift action pushes a terminal on the stack. Reduce pops zero or more symbols from the stack (production *rhs*) and pushes a non-terminal on the stack (production *lhs*)

Discovering Handles

A bottom-up parser builds the parse tree starting with its leaves and working toward its root. The upper edge of this partially constructed parse tree is called its *upper frontier*. At each step, the parser looks for a section of the upper frontier that matches right-hand side of some production. When it finds a match, the parser builds a new tree node with the production's left-hand non-terminal thus extending the frontier upwards towards the root. The critical step is developing an efficient mechanism that finds matches along the tree's current frontier.

Formally, the parser must find some substring β, of the upper frontier where

1. β is the right-hand side of some production A → β, and
2. A → β is one step in right-most derivation of input stream

We can represent each potential match as a pair $\langle A \rightarrow \beta, k \rangle$, where k is the position on the tree's current frontier of the right-end of β. The pair $\langle A \rightarrow \beta, k \rangle$ is called the *handle* of the bottom-up parse.

Handle Pruning

A bottom-up parser operates by repeatedly locating handles on the frontier of the partial parse tree and performing reductions that they specify. The bottom-up parser uses a stack to hold the frontier. The stack simplifies the parsing algorithm in two ways.

First, the stack trivializes the problem of managing space for the frontier. To extend the frontier, the parser simply pushes the current input token onto the top of the stack. Second, the stack ensures that all handles occur with their right end at the top of the stack. This eliminates the need to represent handle's position.

Shift-Reduce Parsing Algorithm

```
push $ onto stack
sym ←
nextToken()
repeat until (sym == $ and the stack contains exactly Goal on top
        of $) if a handle for A → β on top of stack
                pop |β| symbols off the
                stack push A onto the
                stack
        else if (sym ≠ $ )
         push sym onto stack sym
         ← nextToken() else /* no
                handle, no input */
                report error and halt
```

Lecture 22

Example: here is the bottom-up parser's action when it parses the expression grammar sentence

x – 2 × y

(tokenized as id – num * id)

	word	Stack	Handle	Action
1	id	▶	- none -	*shift*
2	–	id ▶	<Factor → id,**1**>	*reduce*
3	–	Factor ▶	<Term → Factor,**1**>	*reduce*
4	–	Term ▶	<Expr → Term,**1**>	*reduce*
5	–	Expr ▶	- none -	*shift*
6	num	Expr – ▶	- none -	*shift*
7	×	Expr – num ▶	<Factor → num,**3**>	*reduce*
8	×	Expr – Factor ▶	<Term → Factor,**3**>	*shift*
9	×	Expr – Term ▶	- none -	*shift*
10	id	Expr – Term × ▶	- none -	*shift*
11	$	Expr – Term × id▶	<Factor → id,**5**>	*reduce*
12	$	Expr–Term × Factor▶	<Term→Term×Factor,**5**>	*reduce*
13	$	Expr – Term ▶	<Expr → Expr – Term,**3**>	*reduce*
14	$	Expr ▶	<Goal → Expr,**1**>	*reduce*
15	$	Goal	- none -	***accept***

Handles

The handle-finding mechanism is the key to efficient bottom-up parsing. As it process an input string, the parser must find and track all potential handles. For example, every legal input eventually reduces the entire frontier to grammar's goal symbol. Thus,
<Goal → Expr,1> is a potential handle at the start of every parse. As the parser builds a derivation, it discovers other handles. At each step, the set of potential handles represent different suffixes that lead to a reduction. Each potential handle represent a string of grammar symbols that, if seen, would complete the right-hand side of some production.

For the bottom-up parse of the expression grammar string, we can represent the potential handles that the shift-reduce parser should track. Using the placeholder • to represent top of the stack, there are nine handles:

	Handles
1	<Factor → id •>
2	<Term → Factor•>
3	<Expr → Term •>
4	<Factor → num•>
5	<Term → Factor•>
6	<Factor → id •>
7	<Term → Term × Factor •>
8	<Expr → Expr – Term •>
9	<Goal → Expr •>

This notation shows that the second and fifth handles are identical, as are first and sixth. It also create a way to represent the potential of discovering a handle in future. Consider the parser's state in step 6: The parser has recognized Expr –. Using the stack-relative notation, we can represent the parser's state as Expr → Expr – • Term. The parser has already recognized an Expr and a –. If the parser reaches a state where it shifts a Term on top of Expr and –, it will complete the handle Expr → Expr – Term •. How many potential handles must the parser recognize? The right-hand side of each production can have a placeholder at its start, at its end and between any two consecutive symbols.

Expr → • Expr –Term
Expr → Expr • – Term
Expr → Expr – •Term
Expr → Expr – Term •

If the right-hand side of a production has *k* symbols, it has *k +1* placeholder positions.

Lecture 23

Handles

The number of potential handles for the grammar is simply the sum of the lengths of the right-hand side of all the productions. The number of complete handles is simply the number of productions. These two facts lead to the critical insight behind LR parsers:

> *A given grammar generates a finite set of handles (and potential handles) that the parser must recognize*

However, it is not a simple matter of putting the placeholder in right-hand side to generate handles. The parser needs to recognize the correct handle by different right contexts. Consider the parser's action at step 9.

	word	**Stack**	**Handle**	**Action**
9	×	Expr – Term ▶	- none -	*shift*
10	id	Expr – Term × ▶	- none -	*shift*
11	$	Expr – Term × id▶	<Factor → id,**5**>	*reduce*
12	$	Expr–Term × Factor▶	<Term→Term×Factor,**5**>	*reduce*
13	$	Expr – Term ▶	<Expr → Expr – Term,**3**>	*reduce*
14	$	Expr ▶	<Goal → Expr,**1**>	*reduce*
15	$	Goal	- none -	***accept***

The frontier is Expr – Term, suggesting a handle <Expr → Expr – Term>•. However, the parser decides to extend the frontier by shifting × on to the stack rather than reducing frontier to Expr. Clearly, this the correct move for the parser. No potential handle contains Expr followed by ×. At step 9, the set of potential handles is

<Expr → Expr – Term•>
<Term →Term• × Factor>
<Term →Term• / Factor >

The next input symbol clearly matches the second choice. The parser needs a basis for deciding between first (reduce) and second (shift) choices:

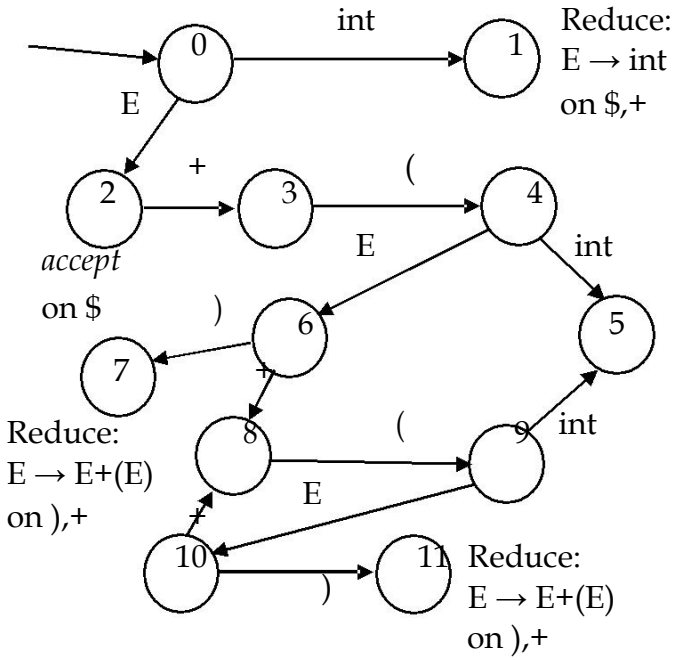
<Expr → Expr – Term•>
<Term →Term• × Factor>

This requires more context than the parser has in the frontier (stack). To choose between reducing and shifting, the parser must recognize which symbols can occur to the right of Expr and Term in valid phrases.

LR(1) Parsers

The LR(1) parsers can recognize precisely those languages in which one-symbol lookahead suffices to determine whether to shift or reduce. The LR(1) construction algorithm builds a handle-recognizing DFA. The parsing algorithm uses this DFA to recognize handles and potential handles on the parse stack

Parsing DFA



In order to remember the state the DFA goes into on a symbol, the parser stores the DFA state in the stack along with the symbol. Initial entry in the stack will be '<dummy,0>'.

Parsers represent DFA as a 2D table. The rows correspond to DFA states and columns correspond to terminals and non-terminals. The columns with

terminals and the rows form the *action table* while the columns with non-terminals and rows are called the *goto table*. It is customary to show these tables together.

Building LR(1) Tables

To construct *Action* and *Goto* tables, the LR(1) parser generator builds a model of handle-recognizing DFA. The model is used to fill in the tables. The LR(1)-table construction needs a concrete representation for the handles and their associated lookahead symbols. We call this representation an *LR(1) item*.

Lecture 24

An LR(1) item is a pair [X → α•β, a] where X → αβ is a production and a∈**T** (terminals) is look-ahead symbol. The model uses a set of LR(1) items to represent each parser state. The model is called the *canonical collection (CC) of set of LR(1) items.*

Canonical Collection

Each set in *CC* represents a state in the eventual parser DFA. The construction of CC begins by building a model of parser's initial state. The initial state consists of the set of LR(1) items that represent the parser's initial state, along with any items that must also hold in the initial state. To simplify the task of building this initial state, the construction requires that the grammar have a unique goal symbol. The convention is to add a new start symbol S to grammar and a production

S → E top most symbol

This leads to the augmented grammar

S → E
E → E + (E) | int

The Closure Procedure

The item [S → •E, $] describes the parser's initial state. It represents a configuration in which recognizing S followed by $ would be a valid parse. This item, i.e., [S → •E, $] becomes the core of the first state in CC, labeled I_0. If the grammar has several distinct productions for the start symbol, each of them generates an item in this initial core of I_0. The procedure *closure* does this.

closure(*s*)
= *repeat*
 for each [X → α•Yβ, a] ∈ *s*
  for each *production* Y → α
  for each b ∈ FIRST(βa)
  s ← *s* ∪ [Y → • γ,
 b] *until s is unchanged*

Let's apply this procedure to the augmented grammar.

The first set is I_0 = *closure*({[S → •E, $] }) . Equating the terms in the

procedure, *s* = {[S → •E, $]}, [X → α •Yβ, **a**] ⇔ [S → •E, $], X = S, α = ε, Y = E,
β = ε, a = **$, Y** → γ ∪E → E + (E) and E → intFIRST(βa) = FIRST($) = $.

This leads to expansion of *s*.

s = { [S → •E, $] } ∪ { [E → •E + (E), $] } ∪ { [E → •int, $] } =
{ [S → •E, $] , [E → •E + (E), $] , [E → •int, $] }

The set *s* changed so we repeat. The item [S → •E, $] is already processed. The for loop considers [X → α•Yβ, a] ⇔ [E→•E+(E),$], which leads to the match up X = E,
α= ε, Y = E, β= +(E), a = $, **Y** → γ ⇔ E → E + (E), ⇔ E → int
FIRST(βa) = FIRST(+(E)$) = +. The set *s* is extended

s = *s* ∪ { [E → •E+(E), +] } ∪ { [E → •int, +] }

Lecture 25

The set *s* changed so the repeat loop is executed again. This time, however, the item [E → •int, $/+] does not yield any more items because the dot is followed by the terminal int. The first set of items is

I_0 = {
[S → •E, $],
[E → •E+(E), $/+],
[E → •int, $/+]
}

Let's consider the rationale behind the *Closure* procedure. If [A → β•Cδ,a] ∈ *s*, then one potential completion for the left context is to find a string that reduces to C, followed by δa. This completion should cause a reduction to A, since it fills out the production's right-hand side (Cδ), and follows it with a valid look-ahead symbol. For a production
C → γ, *closure* must insert '•' before γ and add appropriate look-ahead symbols – all terminals that can appear as the initial symbol in δa. This includes every terminal in FIRST(δ). If ε ∈ FIRST(δ), it also includes a, thus FIRST(δa) in the algorithm.

The *goto* Procedure

The second critical step in the construction is to derive other parser states from I_0. To accomplish this, we compute, for each state I_i and each grammar symbol y, the state that would arise if the parser recognized a y while in state I_i. A state *s* that contains
[X → α • yβ, b] has a transition (*goto*) labeled y to the state that contains the items *goto*(*s*, y) where y can be terminal or a non-terminal.

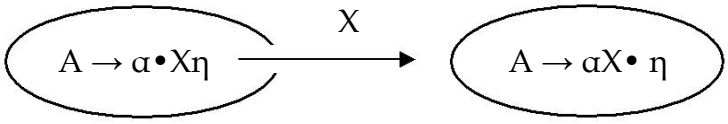
goto(*s*, y)
m ← { }
for each item [X → α •yβ, b] ∈ *s*
m ← *m* ∪ {[X → α y•β, b]}
return *closure*(*m*)

Finite Automaton of Items

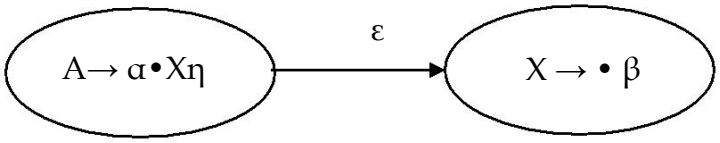
The LR(1) items are used as the states of a finite automaton (FA) that maintains information about the parsing stack and progress of a shift-reduce parser. The FA will start out as a nondeterministic finite automaton (NFA). A DFA can be constructed from this NFA using the subset construction, similar to one we used for lexical analysis.

Consider the NFA of LR(0) items, i.e., no look-ahead. What are the transitions of the NFA of LR(0) items? Consider the item A→ α•γ. Suppose γ begins with symbol X which may be a terminal (token) or non-terminal. The item can be written as A→ α•Xη.

Then there is a transition on symbol X for state represented by item A→ α•Xη to state represented by item A→ αX•η. If X is a terminal, then this transition corresponds to a shift of X from input to top of parse stack.



If X is a non-terminal, then the interpretation of this transition is more complex because non-terminals do not appear in input. In fact, such a transition will correspond to pushing of X onto the stack during the parse. But this can only occur during a reduction by the production X → β. Such a reduction must be preceded by recognition of a β. The state given by X → •β represents the beginning of this process (dot indicates we are about to recognize β). Then for every item A→ α•Xη we must add an ε-transition for every production X → β.



Lecture 26

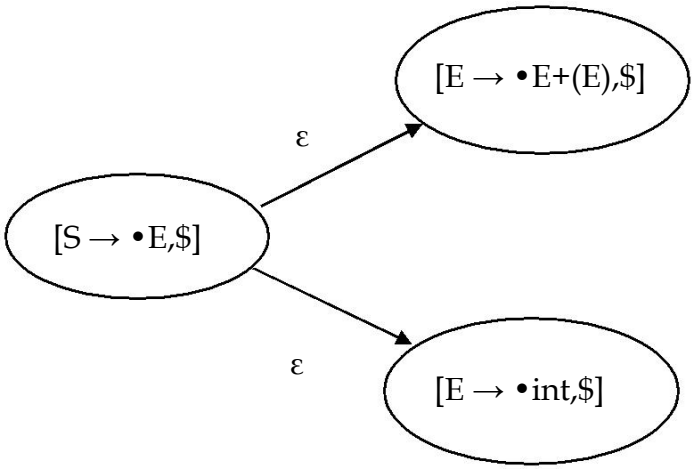
The initial DFA state I_0 we computed is the ε-closure of the set consisting of item

$S \rightarrow \bullet E$

Recall the stage in the closure

$s = \{ [S \rightarrow \bullet E, \$] , [E \rightarrow \bullet E + (E), \$] , [E \rightarrow \bullet int, \$] \}$

The NFA states and transitions required are



Algorithm:
Construction of collection of canonical sets of LR(1) items.
Input:
An augmented grammar G'
Output:
Collection of canonical (CC) sets of LR(1)
CC(G')
$I_0 \leftarrow$ {closure([S' → •S, $])}
CC ← { I_0 }
repeat
for each unmarked set $I_j \in$ CC
mark I_j *as processed*
for each X *following* • *in an item in* I_j
$I_k \leftarrow$
goto(I_j,X) *if* I_k
$\notin$ CC *then*
CC ← CC $\cup$ I_k
record transition from I_j *to* I_k *on*

X *until* CC *is not changing*

We use the algorithm to compute the sets of LR(1) items for the augmented grammar G'

$S \rightarrow E$
$E \rightarrow E + (E) \mid \underline{int}$

We computed I_0; we now compute the sets goto(I_0,X) for various values of X. X can be E, int, +, (and) .

I_1 = goto(I_0,int): invokes *closure*({[E → int•,$/+]}). No additional closure is possible since the dot is at the right end of the production. Thus I_1 = {[E → int•, $/+]} and we have the transition from I_0 to I_1 on int

I_2 = goto(I_0,E)

m ← {}
for each [X → •E , b] ∈ I_0
□[S → •E, $]
□[E → •E+(E), $/+]

m = *m* ∪ {[S → E•, $]}∪{[E → E•+(E), $/+]}
return closure(*m*)

No further closure for the first item because • is at the end
In the second item, a terminal + appears after • so no further closure is possible. Thus I_2 = {[S → E•, $], [E → E• +(E), $/+]}.

We repeat the process in similar fashion.

I_3 = goto(I_2,+) = {[E → E + • (E), $/+]}

I_4 = goto(I_3,( ) = { [E → E + (• E), $/+], [E → • E + (E),)/+], [E → • int,)/+]}

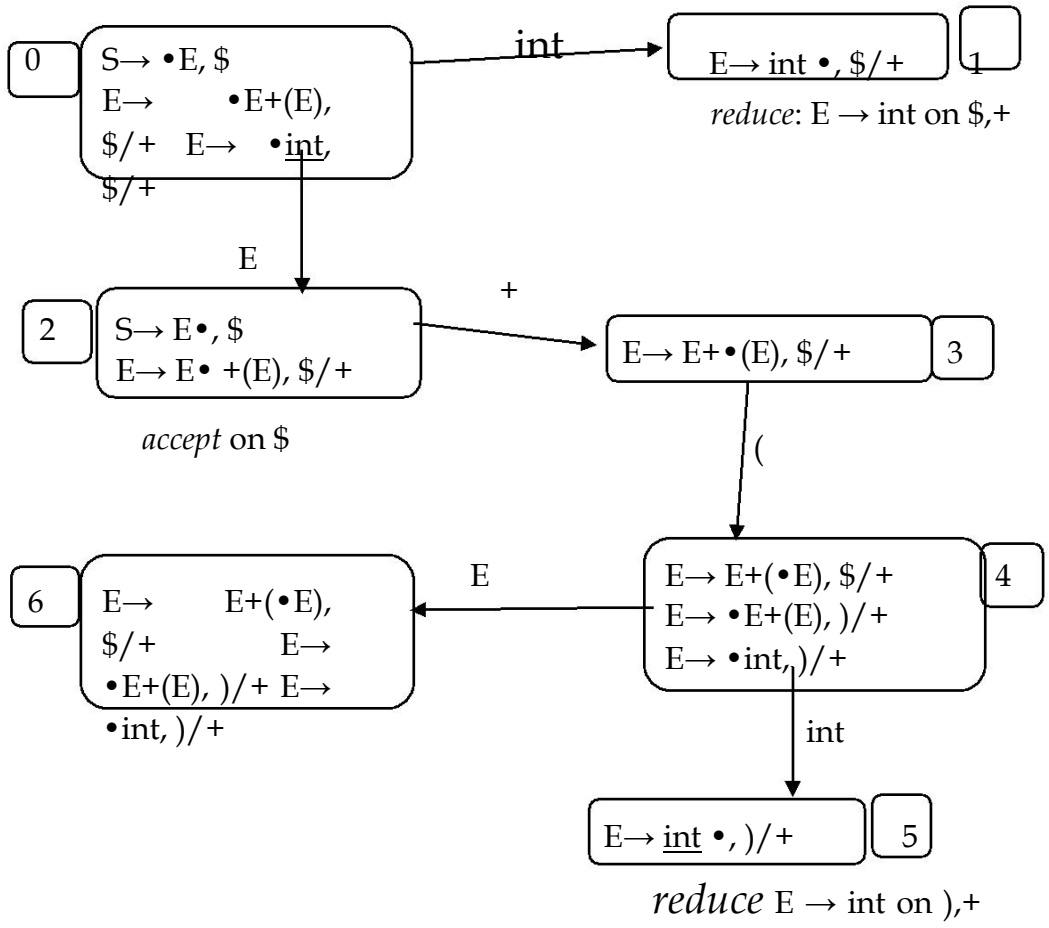
I_3 = goto(I_2,+) = {[E → E + • (E), $/+]}

I_4 = goto(I_3,( ) = {[E → E + (• E), $/+], [E → • E + (E),)/+], [E → • int,)/+]}

I_5 = goto(I_4, int) = { [E → int •,)/+] }

I_6 = goto(I_4, E) = { [E → E + (E •), $/+], [E → E • + (E),)/+] }

and so on. The sets and transitions so far yield the DFA



Lecture 27

LR Table Construction

Construct CC = {I_0,I_1,I_2, ... , I_n}, for G'. State i of the parser is constructed from the set I_i The parsing actions for state i are determined as follows:

for each item $I_i \in$ CC
  if [A → α•a β, b] ∈ I_i *and* goto(I_i, a) == I_j
        then Action[i,a] ← "shift j"
  else if [A → •, a] ∈ I_i *and* A ≠ S' *then*
        Action[i,a] ← "reduce A → α
  " *else if* [S' → **S**•, $] ∈ I_i *then*
         Action[i,a] ← "accept"
end for

// the goto table
for each non-terminal A ∈ G
  if goto(I_i, A) = I_j *then*
        Goto[i, A] ← j

The initial state is the one that contains the item [S' → •**S**, $]. All remaining entries are marked "error". Let us go through an example and construct the LR table for the augmented grammar

1. S' → E
2. E → T – E
3. E → T
4. T → F × T
5. T → F
6. F → id

The FIRST sets we would need are

Symbol	FIRST
S'	{ id }
E	{ id }
T	{ id }
F	{ id }
id	{ id }
×	{ × }
–	{ – }

We construct the canonical collection of set of LR(1)

I_0 = {closure([S' → •E, $])}
{
[S' → •E, $],
[E → •T – E, $], [E → •T, $],
[T → •F×T, $], [T → •F, $]
[T → •F×T, –], [T → •F, –],
[F → •id, $], [F → •id, –],
[F → •id, ×]
}

I_1 = {goto(I_0, E)} = {[S' → E•, $] }

I_2 = {goto(I_0, T)} = {[E → T• – E,$], [E →T•,$]}

I_3 = {goto(I_0, F)} = {
[T → F• ×T, $],
[T → F•, $],
[T → F• ×T, –],
[T → F•, –] }

I_4 = {goto(I_0, id)} = {
[F → id •, $],
[F → id •, –],
[F → id •, ×] }

I_5 = {goto(I_2, –)} = {
[E →T– • E, $], [E → •T–E, $], [E → •T, $],
[T → •F×T, $], [T → •F×T, –], [T → •F, $], [T → •F, –],
[F → •id, $], [F → •id, –], [F → •id, ×] }

I_6 = {goto(I_3, ×)} = {
[T → F× •T, $], [T → F× •T, –],
[T → •F×T, $], [T → •F×T, –], [T → •F, $], [T → •F, –],
[F → •id, $], [F → •id, –], [F → •id, ×] }

I_7 = {goto(I_5, E)} = {[E → T – E •, $] }

I_2 = {goto(I_5, T)}, i.e., goto(I_5, T) yields the same set as I_2.

I_3 = {goto(I_5, F)}

I_4 = {goto(I_5, id)}

I_8 = {goto(I_6, T)} = { [T → F×T •, $], [T → F×T •, –] }

I_3 = {goto(I_6, F)}

I_4 = {goto(I_6, id)}

We now filling the LR(1) table by applying the rules.

Apply
1. if [A → α•aβ, b] ∈ I_i and goto(I_i, a) = I_j
then set Action[i,a] ← "shift j". // here, a is a terminal.

I_0={[S' → •E,$],[E →•T– E, $],
[E → •T, $], [T → •F×T, $],
[T → •F×T, –], [T → •F, $],
[T → •F, –], [F → •id, $], [F
→ •id, –], [F → •id, ×] }
goto(I_0, id) = I_4
→ Action[0, id] ← shift 4

I_2= {[E → T• – E, $], [E →T•,$]}, goto(I_2, –) = I_5
→ Action[2, –] ← shift 5

I_3={[T → F• ×T, $], [T → F•, $],[T → F• ×T, –], [T → F•, –]}, goto(I_3, ×) = I_6
→Action[3, ×] ← shift 6

goto(I_5, id) = I_4
→ Action[5, id] ← shift 4

goto(I_6, id) = I_4
→ Action[6, id] ← shift 4

Apply
2. if [A → α•, a] ∈ I_i and A ≠ S' then set action[i,a] to "reduce A → α"

I_2 = { [E → T• – E, $], [E →T•,$] }
→Action[2, $] ← reduce 3

I_3 = {[T → F• ×T, $], [T → F•, $], [T → F• ×T, –], [T → F•, –] }
→Action[3, $] ← reduce 5
→Action[3, –] ← reduce 5

I_4 = {[F → id•, $], [F → id•, –], [F → id•, ×] }

→ Action[4, $] ← reduce 6

→Action[4, –] ← reduce 6

→Action[4, ×] ← reduce 6

I_7 = {[E → T – E•, $] }

→ Action[7, $] ← reduce 2

I_8 = { [T → F×T•, $], [T → F×T•, –] }

→Action[8, $] ← reduce 4

→Action[8, –] ← reduce 4

Apply

3. if [S' → **S**•, $] ∈ I_i

then set action[i,$] to "accept"

I_1 = {[S' → E•, $] }

→ Action[1, $] ← accept

Apply

for each non-terminal A ∈ G

if goto(I_i, A) = I_j then goto[i, A] ← j.

goto(I_0, E) = I_1 → goto[0,E] ← 1
goto(I_0, T) = I_2 → goto[0,T] ← 2
goto(I_0, F) = I_3 → goto[0,F] ← 3
goto(I_5, E) = I_7 → goto[5,E] ← 7
goto(I_5, T) = I_2 → goto[5,T] ← 2
goto(I_5, F) = I_3 → goto[5,F] ← 3
goto(I_6, T) = I_8 → goto[6,T] ← 8
goto(I_6, F) = I_3 → goto[6,F] ← 3

The final table we get is

	Action				Goto		
	id	–	×	$	E	T	F
0	s4				1	2	3
1				*acc*			
2		s5		r3			
3		r5	s6	r5			
4		r6	r6	r6			
5	s4				7	2	3
6	s4					8	3
7				r2			
8		r4		r4			

Let us parse the expression x – y × z using the LR(1) table. The scanner will encode the input string as id – id × id $ where $ is the EOF marker

Stack	Input	
¤0	id – id × id $	s4
¤0id4	– id × id $	r6 F→ id
¤0F3	– id × id $	r5 T→ F
¤0T2	– id × id $	s5
¤0T2–5	id × id $	s4
¤0T2–5id4	× id $	r6 F→ id
¤0T2–5F3	× id $	s6
¤0T2–5F3×6	id $	s4
¤0T2–5F3×6id4	$	r6 F→ id
¤0T2–5F3×6F3	$	r5 T→ F
¤0T2–5F3×6T8	$	r4 T→ F×T
¤0T2–5T2	$	r3 E→T
¤0T2 –5E7	$	r2 E→T–E
¤0E1	$	***accept***

Lecture 28

LR(1) Skeleton Parser

```
stack.push(dummy);
stack.push(0); done = false;
token = scanner.next(); while
(!done) {
        s = stack.top();
        if( Action[s,token] == "reduce
                A→β") { stack.pop(2× | β | );
                s = stack.top();
                stack.push(A);
                stack.push(Goto[s,A]
                );
        }
        else if( Action[s,token] == "shift i"){
                stack.push(token);
                stack.push(i);
        token = scanner.next();
        }
        else if(Action[s,token] ==
                "accept" && token ==
                "$" )
                done = true;
        else
                report error and recover;
}
report success;
```

Shift/Reduce Conflicts

If a DFA states contains both [X → α•aβ, b] and [Y → γ•, a]
Then on input "a" we could either shift into state [X → αa•β,b], or reduce with Y → γ. This is called a *shift-reduce conflict.* Typically, this is due to ambiguities in the grammar. The classic example of a shift-reduce conflict is the dangling else. Consider the grammar

```
stmt    →  if E then stmt
        |     if E then stmt else stmt
```

We will have DFA state containing

[stmt → if E then stmt•, else]
[stmt → if E then stmt •else stmt, x]

If else follows, we can shift

[stmt → if E then stmt else • stmt, x]

or reduce

[stmt → if E then stmt•, else]

Typical action is shift so that else matches with most recent if.

Lecture 29

Shift/Reduce Conflicts

Consider the ambiguous grammar

$$E \rightarrow E + E \mid E \times E \mid int$$

We will DFA state containg

$$[E \rightarrow E \times E\bullet, +]$$
$$[E \rightarrow E \bullet + E, +]$$

Again we have a shift/reduce conflict. We need to reduce because × has precedence over +

Reduce/Reduce Conflicts

If a DFA states contains both $[X \rightarrow \alpha \bullet, a]$ and $[Y \rightarrow \beta \bullet, a]$, then on input "a" we don't know which production to reduce with. This is called a *reduce-reduce conflict*. Usually due to gross ambiguity in the grammar.

LR(1) Table Size

LR(1) parsing table for even a simple language can be extremely large with thousands of entries. It is possible to reduce the size of the table. Many states in the DFA are similar.

The *core* of set of LR items is the set of first components without the lookahead terminals. For example the core of the item $\{ [X \rightarrow \alpha\bullet\beta, b], [Y \rightarrow \gamma\bullet\delta, d] \}$ is
$\{ X \rightarrow \alpha\bullet\beta, Y \rightarrow \gamma\bullet\delta \}$. Consider the LR(1) states

$$\{ [X \rightarrow \alpha \bullet, a], [Y \rightarrow \beta \bullet, c] \}$$
$$\{ [X \rightarrow \alpha \bullet, b], [Y \rightarrow \beta \bullet, d] \}$$

They have the same core and can be merged. The merged state contains

$$\{ [X \rightarrow \alpha \bullet, a/b], [Y \rightarrow \beta \bullet, c/d] \}$$

These are called the *LALR(1) states*. LALR(1) stands for **L**ook**A**head **LR**(1). This leads to tables that have 10 times fewer states than LR(1).

Here is the algorithm to generate LALR(1) DFA.

1. Repeat until all states have distinct code
2. choose two distinct states with same core
3. merge states by creating a new one with the union of all the items point edges from predecessors to new state
4. new state points to all the previous successors

LALR languages are not natural. They are an efficiency hack on LR languages. Any reasonable programming language has a LALR(1) grammar. LALR(1) has become a standard for programming languages and for parser generators.

Lecture 30

Parser Generators

Parser generators exist for LL(1) and LALR(1) grammars. For example,

- LALR(1) - YACC, Bison, CUP
- LL(1) – ANTLR
- Recursive Descent - JavaCC

YACC Parser Generator

YACC – Yet Another Compiler Compiler, appeared in 1975 as a Unix application. The other companion application Lex appeared at the same time. These two greatly aided the construction of compilers and interpreters. The input to YACC consists of a specification text file. The structure of the file is

definitions
%%
rules
%%
C/C++ functions

Here, for example, is the YACC file for a calculator

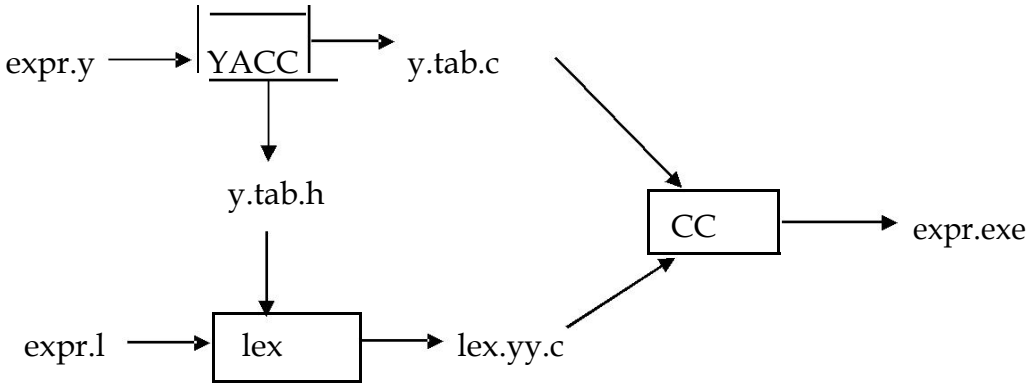
```
%token NUMBER LPAREN RPAREN
%token PLUS MINUS TIMES DIVIDE
%%
        expr : expr PLUS expr
                        | expr MINUS expr
                        | expr TIMES expr
                        | expr DIVIDE expr
                        | LPAREN expr RPAREN
                        | MINUS expr
                       | NUMBER
                      ;
%%
```

The Flex input file for a calculator is

```
%{
#include "y.tab.h"
%}
digit                   [0-9]
ws                      [ \t\n]+
%%
{ws}                    ;
{digit}+        {return NUMBER;}
"+"                     {return PLUS;}
"*"                     {return TIMES;}
"/"                     {return DIVIDE;}
"–"                     {return MINUS;}

%%
```

The following diagram outlines the process of building a parser with YACC and Lex.



Lecture 31

Beyond Syntax

These questions are part of context-sensitive analysis. Answers depend on values, not parts of speech. Answers may involve computation.

These questions can be answered by using formal methods such as context-sensitive grammars and attribute grammars or by using ad-hoc techniques.

One of the most popular is the use of attribute grammars.

Attribute Grammars

A CFG is augmented with a *set of rules*. Each symbol in the derivation has a set of values or *attributes*. Rules specify how to *compute* a value for each attribute

Consider the grammar for *signed binary numbers (SBN)*

Number	→	Sign List
Sign	→	+ –
List	→	List Bit | Bit
Bit	→	0 1

The string "–1" can be derived as follows:

Number	→	Sign List
	→	– List
	→	– Bit
	→	– 1

Similarly, the derivation for "-101" is

Number	→	Sign List
	→	Sign List Bit
	→	Sign List 1
	→	Sign List Bit 1
	→	Sign List 0 1
	→	Sign Bit 0 1
	→	Sign 1 0 1
	→	– 1 0 1

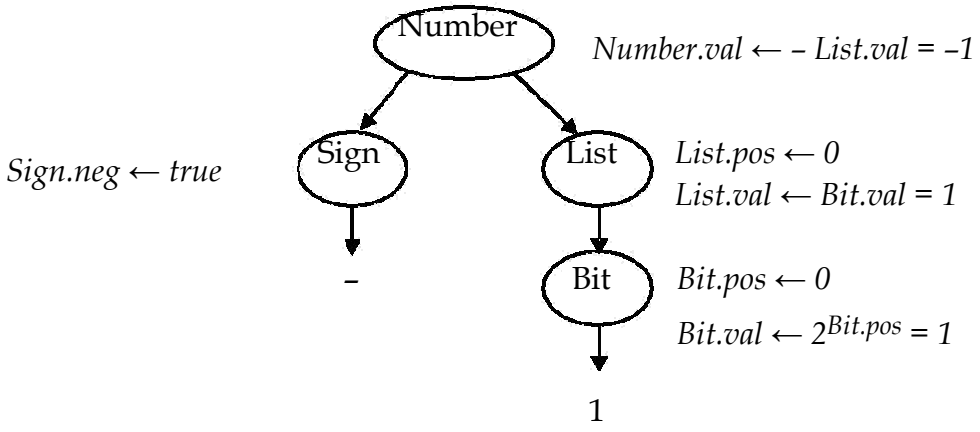
For an *attributed version* of *SBN*, the following attributes are needed

Symbol	*Attributes*
Number	val
Sign	neg
List	pos, val
Bit	pos, val

We will add rules to compute decimal value of a signed binary number

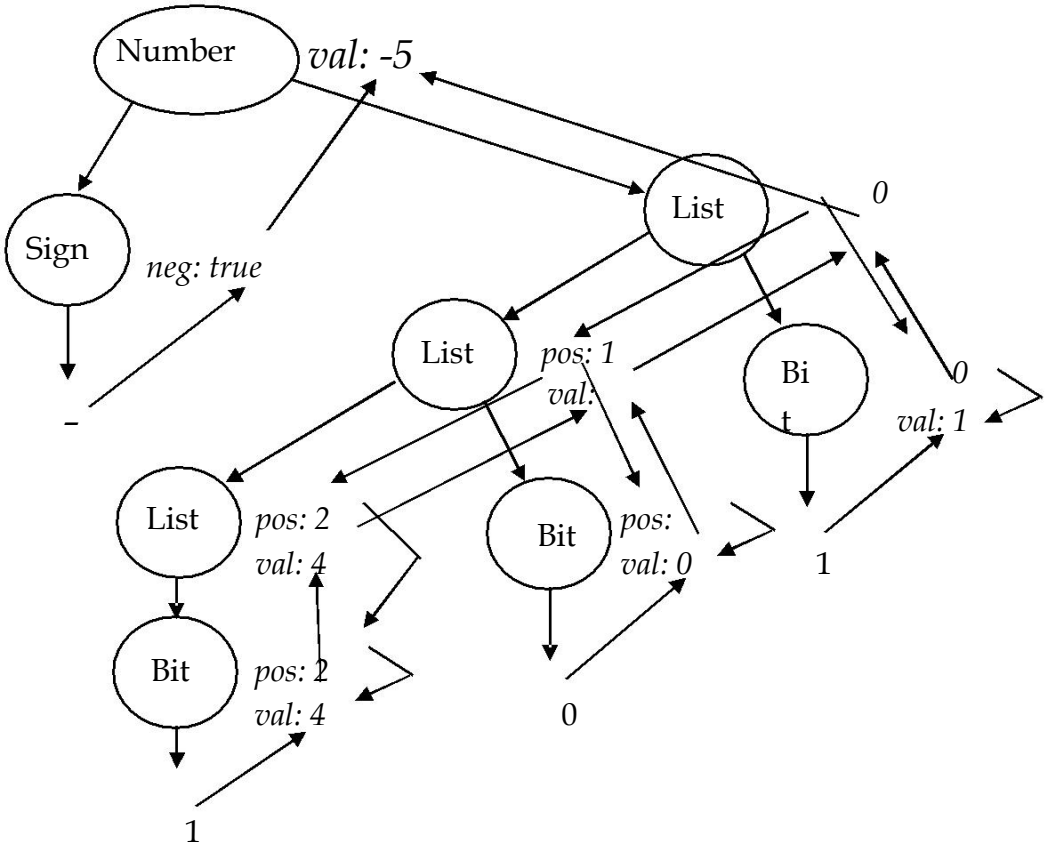
Productions	***Attribution Rules***
Number → Sign List	List.pos ← 0 if Sign.neg then Number.val ← **–** List.val else Number.val ← List.val
Sign → +	Sign.neg ← false
Sign → –	Sign.neg ← true
List0 → List1 Bit	List1.pos ← List0.pos + 1 Bit.pos ← List0.pos List0.val ← List1.val + Bit.val
List → Bit	Bit.pos ← List.pos List.val ← Bit.val
Bit → 0	Bit.val ← 0
Bit → 1	Bit.val ← $2^{\text{Bit.pos}}$

Attributes are associated with nodes in parse tree. Rules are value assignments associated with productions. Rules and parse tree define an attribute dependence graph which must be acyclic.

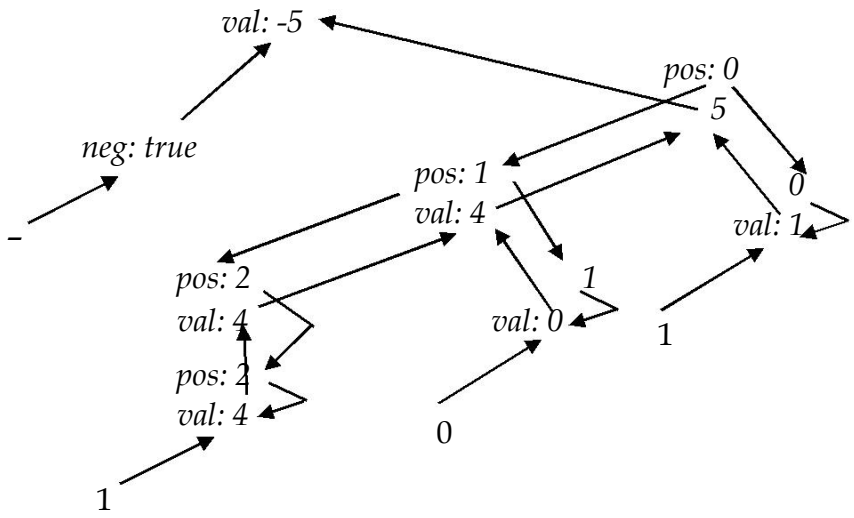


Attributes are distinguished based on the direction of value flow. Attributes of a node whose values are defined wholly in terms of attributes of node's children and from constants are called *synthesized attributes*. Values used to compute synthesized attributes flow bottom-up in the parse tree.

Attributes whose values are defined in terms of a node's own attributes, node's siblings and node's parent are called *inherited attributes*. Values flow top-down and laterally in the parse tree. The following attributed tree shows the inherited and synthesized attributes for the input signed binary number -101.



When the parse tree is peeled away, we get the attribute dependence graph



Lecture 32

Evaluation Methods

A number of ways can be used to evaluate the attributes. When using *Dynamic method*, the compiler application builds the parse tree and then builds the dependence graph. A topological sort of the graph is carried out and attributes are evaluated or defined in topological order. In *rule-based (or treewalk)* methodology, the attribute rules are analyzed at compiler-generation time. A fixed (static) ordering is determined and the nodes in the dependency graph are evaluated this order. In *oblivious (passes, dataflow)* methodology, the attribute rules and parse tree are ignored. A convenient order is picked at compiler design time and used.

Attribute grammars have not achieved widespread use due to a number of problems. For example: non-local computation, traversing parse tree, storage management for short-lived attributes and lack of high-quality inexpensive tools. However, a variation of attribute grammars and evaluation schemes is used in many compilers. This variation is called *ad-hoc analysis*.

In rule-based evaluators, a sequence of actions is associated with grammar productions. Organizing actions required for context-sensitive analysis around structure of the grammar leads to powerful, albeit ad-hoc, approach which is used on most parsers. A snippet of code (*action*) is associated with each production that executes at parse time In top-down parsers, the snippet is added to the appropriate parsing routine. In a bottom-up shift-reduce parsers, the actions are performed each time the parser performs a reduction. Here the LR(1) skeleton parser indicating the place where the snippet is executed.

```
stack.push(dummy);
stack.push(0); done = false; token
= scanner.next(); while (!done) {
        s = stack.top();
        if( Action[s,token] == "reduce A→β")
                { invoke the code snippet
                stack.pop(2×|β|);
                s = stack.top();
                stack.push(A);
                stack.push(Goto[s,A]);
        }
        else if( Action[s,token] == "shift i"){
```

```
                stack.push(token);
                stack.push(i); token =
                scanner.next();
        }
}
```

The following table shows the code snippets for the SBN example.

Productions	*Code snippet*
Number → Sign List	Number.val ← – Sign.val × List.val
Sign → +	Sign.val ← 1
Sign → –	Sign.val ← –1
List → Bit	List.val ← Bit.val
$List_0$ → $List_1$ Bit	$List_0$.val ← 2×$List_1$.val + Bit.val
Bit → 0	Bit.val ← 0
Bit → 1	Bit.val ← 1

Lecture 33

Implementing Ad-Hoc Scheme

The parser needs a mechanism to pass values of attributes from definitions in one snippet to uses in another. We will adopt notation used by YACC for snippets and passing values. Recall that the skeleton LR(1) parser stored two values on the stack <*symbol, state*>. We can replace this with triples <*value, symbol, state*>. On a reduction
by $A \rightarrow \beta$, the parser pops $3\times|\beta|$ items from the stack rather than $2\times|\beta|$. It pushes value along with the symbol.

Lecture 34

Let's go through an example of using YACC to implement the ad-hoc scheme for an arithmetic calculator.

The YACC file for a calculator grammar is as follows:

```
%token NUMBER LPAREN RPAREN
%token PLUS MINUS TIMES DIVIDE
%%
        expr : expr PLUS expr
                        | expr MINUS expr
                        | expr TIMES expr
                        | expr DIVIDE expr
                        | LPAREN expr RPAREN
                        | MINUS expr
                       | NUMBER
                      ;
%%
```

We will add the code snippets

```
%{
#include <iostream>
%}
// type of value entries in the parse stack
%union          {int val;}

%token NUMBER LPAREN RPAREN EQUAL
%token PLUS MINUS TIMES DIVIDE
/* associativity and precedence:in order of increasing
        precedence */
%nonassoc       EQUAL
%left PLUS MINUS
%left TIMES     DIVIDE
%left UMINUS /* dummy token to use as
                        precedence marker */
%type <val>     NUMBER expr

%%
prog : expr     { cout << $1 << endl;}
```

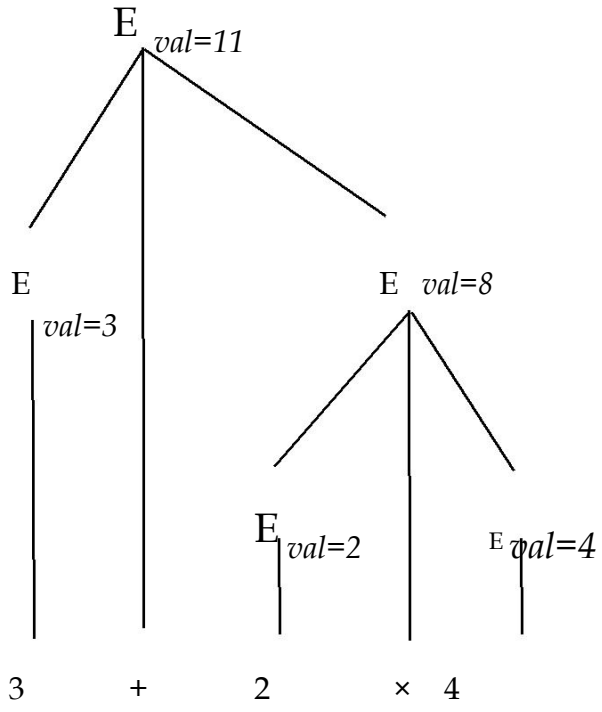
```
        ;
expr : expr       PLUS expr                {$$ = $1 + $3;}

| expr            MINUS expr {$$ = $1 - $3;}
| expr TIMES expr {$$ =            $1      * $3;}
| expr DIVIDE expr {if($3)         $$ = $1 / $3;}
| LPAREN expr RPAREN {$$ =   $2;}
| MINUS expr                        {$$ = -$2;}
| NUMBER                            {$$ = $1;}

;
```

The '$' notation is used by YACC to refer to values of symbols on the right hand side of the grammar production. For example, for **expr : expr PLUS expr**, **$1** refers to first **expr** on the right, **$2** refers to **PLUS** and **$3** refers to the second non-terminal **expr**. The notation $$ refers to the symbol on the left hand side of the production. Internally, the $1 refers to the attribute value associated with the first grammar symbol, $2 with the second, $3 with the third and so on. These values are stored in the parse stack. The notation $$ instructs YACC to push a computed attribute value on the stack and associate it with the symbol on the left when the reduction takes place.

The following attributed tree shows the values as they are computed in a bottom-up parse



(note: please see the file "**lex_yacc.pdf**" for additional information on using YACC.)

Intermediate Representations

Compilers are organized as a series of passes. This creates the need for an *intermediate representation* (IR) for the code being compiled. Compilers use some internal form– an IR –to represent the code being analyzed and translated. Many compilers use more than one IR during the course of compilation.

The IR must be expressive enough to record all of the useful facts that might be passed between passes of the compiler. During translation, the compiler derives facts that have no representation in the source code. For example, the addresses of variables and procedures are not specified in the code being compiled. Typically, the compiler augments the IR with a set of tables that record additional information. Foremost among these is the *symbol table*. These tables are considered part of the IR

Selecting an appropriate IR for a compiler project requires an understanding of both the source language and the target machines and the properties of the programs to be compiled. Thus, a source-to-source translator e.g., C++ to Java, might keep its internal information in a form quite to close to the source. In contrast, a compiler that produces assembly code might use a form close to the target machine's instruction set.

Lecture 35

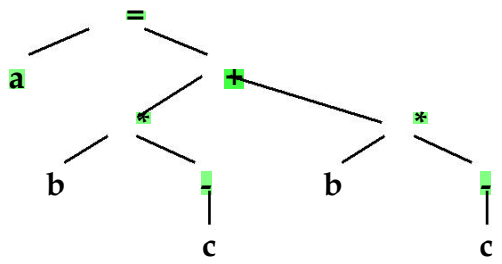
IR Taxonomy

IRs fall into three organizational categories:

1. *Graphical* IRs encode the compiler's knowledge in a graph.
2. *Linear* IRs resemble pseudo-code for some abstract machine
3. *Hybrid* IRs combine elements of both graphical (structural) and linear IRs

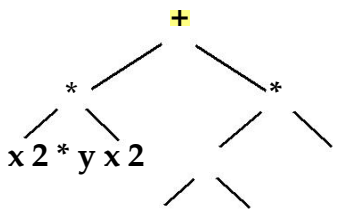
Graphical IRs

Parse trees are graphs that represent source-code form of the program. The structure of the tree corresponds to the syntax of the source code. Parse trees are used primarily in discussion of parsing and in attribute grammar systems where they are the primary IR In most other applications, compilers use one of the more concise alternatives. An *abstract syntax tree* (AST) retains the essential structure of the parse tree but eliminates extraneous nodes. Here, for example, is the AST for the expression
a = b*-c + b*-c. Notice how all derivation related information has been removed.

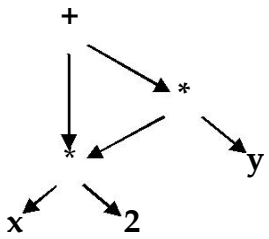


ASTs have been used in many practical compiler systems such as source-to-source systems, automatic parallelization tools, pretty-printing etc.

AST is more concise than a parse tree. It faithfully retains the structure of the original source code. Consider the AST for **x*2+x*2*y**



The AST contains two distinct copies of **x*2.**A *directed acyclic graph* (DAG) is a contraction of the AST that avoids duplication.



If the value of **x** does not change between uses of **x*2**, the compiler can generate code that evaluates the subtree once and uses the result twice.

The task of building AST fits neatly into an ad hoc-syntax-directed translation scheme. Assume that the compiler has routines mknode and mkleaf for creating tree nodes. The following rules can be attached to the expression grammar to create AST.

Production	***Semantic Rule***
E → E1 + E2	E.nptr = mknode('+', E1.nptr, E2.nptr)
E → E1 ∗ E2	E.nptr = mknode('∗', E1.nptr, E2.nptr)
E → – E1	E.nptr = mknode('–', E1.nptr)
E → (E1)	E.nptr = E1.nptr
E → num	E.nptr = mkleaf('num', num.val)

The following table shows the same rules using YACC syntax.

Production	***Semantic Rule (yacc)***
E → E1 + E2	$$.nptr = mknode('+', $1.nptr, $3.nptr)
E → E1 ∗ E2	$$.nptr = mknode('∗', $1.nptr, $3.nptr)
E → – E1	$$.nptr = mknode('–', $1.nptr)
E → (E1)	$$.nptr = $1.nptr
E → num	$$.nptr = mkleaf('num', $1.val)

We will use another IR, called *three-address code*, for actual code generation The semantic rules for generating three-address code for common programming languages constructs are similar to those for AST.

Linear IRs

The alternative to graphical IR is a linear IR. An assembly-language program is a form of linear code. It consists of a sequence of instructions that execute in order of appearance Two linear IRs used in modern compilers are *stack-machine code* and *three-address code*.

Stack-machine code is sometimes called one-address code. It assumes the presence of an operand stack. Most operations take their operands from the stack and push results back onto the stack. Here, for example, is the linear IR for x – 2 × y

stack-machine	*three-address*
push 2	t1 ← 2
push y	t2 ← y
multiply	t3 ← t1 × t2
push x	t4 ← x
subtract	t5 ← t4 – t1

Stack-machine code is compact; it eliminates many names from IR. This shrinks the program in IR form. All results and arguments are transitory unless explicitly moved to memory. Stack -machine code is simple to generate and execute. Smalltalk-80 and Java use byte-codes which are abstract stack-machine code. The byte-code is either interpreted or translated into target machine code (JIT).

In *three-address code* most operations have the form

x ← y *op* z

with an operator (**op**), two operands (**y and z**) and one result (**x**). Some operators, such as an immediate load and a jump, will need fewer arguments.

Lecture 36

Three -address code is attractive for several reasons. Absence of destructive operators gives the compiler freedom to reuse names and values. Three-address code is reasonably compact: operations are 1 to 2 bytes; addresses are 4 bytes. Many modern processors implement three-address operations; a three-address code models their properties well

We now consider *syntax-directed translation* schemes using three-address code for various programming constructs. We start with the *assignment statement*.

Assignment Statement

Production	*translation scheme*
S → id = E	{ p = lookup(id.name); emit(p, '=', E.place); }
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, '=', '–' ,E1.place); }
E → (E1)	{ E.place = E1.place; }
E → id	{ p = lookup(id.name); emit(E.place, '=', p); }

The translation scheme uses a *symbol table* for identifiers and temporaries. Every time the parser encounters an identifier, it installs it in the symbol table. The symbol table can be implemented as a hash table or using some other efficient data structure for table. The routine **lookup(name)** checks if there an entry for the name in the symbol table. If the **name** is found, the routine returns a pointer to entry. The routine **newtemp()** returns a new temporary in response to successive calls. Temporaries can be placed in the symbol table. The routine **emit()** generates a three-address statement which can either be held in memory or written to a file. The attribute E.place records the symbol table location.

Lecture 37

Here is the bottom-up parse of the assignment statement **a = b*-c + b*-c** and the syntax-directed translation into three-address code.

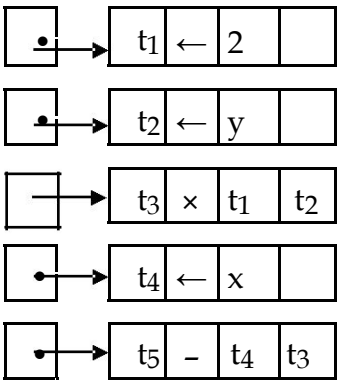
Parser action	*attribute*	*Three- address code*
id=id ∗ –id + id ∗ – id		
id=E1 ∗ –id + id ∗ – id	E1.place = b	
id=E1 ∗ –E2 + id ∗ – id	E2.place = c	
id=E1 ∗ E2 + id ∗ – id	E2.place = t1	t1 = – c
id=E1 + id ∗ –id	E1.place = t2	t2 = b∗t1
id=E1 + E2 ∗ –id	E2.place = b	
id=E1 + E2 ∗ –E3	E3.place = c	
id=E1 + E2 ∗ E3	E3.place = t3	t3 = – c
id=E1 + E2	E2.place = t4	t4 = b∗t3
id=E1	E1.place = t5	t5 = t2+t4
S		a = t5

Representing Linear Codes

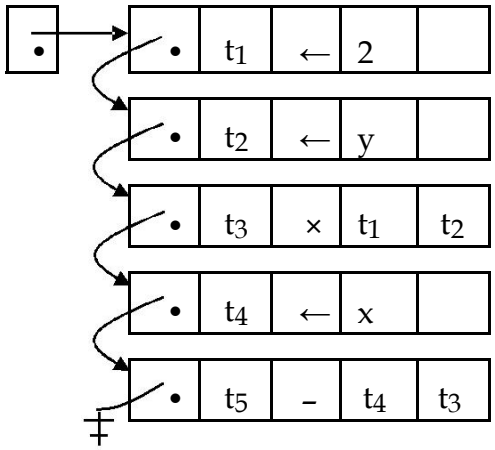
Three-address codes are often implemented as a *set of quadruples*. Each quadruple has four fields: an operator, two operands (or sources) and a destination. In C++, for example, one can design a quadruple class and then declare a simple array of quadruples. This leads to the following arrangement; the index of the array element acts as the number of quadruple generated.

Target	*Op*	*Arg1*	*Arg2*
t1	←	2	
t2	←	y	
t3	×	t1	t2
t4	←	x	
t5	–	t4	t3

An array of pointers to quads can be employed which leads to the following structure:



Both simple array and array of pointers have maximum size limitation. This limitation can be overcome by using a linked list of quads:



Flow-of-Control Statements

We now use the syntax-directed translation scheme for the flow-of-control statements found in most procedural programming languages.

S → if E then S_1
| if E then S_1 else S_2
| while E do S_1

where E is a boolean expression. Consider the statement

if c < d then x = y
+ z
else
x = y – z

One possible 3-address code could be

if c < d goto L1 goto

```
        L2
L1:     x = y + z
        goto L3
L2: x = y – z L3:
nop
```

We will assume that a three-address statement can be symbolically labeled; the function newlabel() returns a new symbolic label each time it is called

Lecture 38

Three-Address Statement Types

Prior to proceeding with flow-of-control construct, here are the types of three-Address statements that we will use

- *Assignment statement*
  x = y *op* z
  op is a binary arithmetic or logical operation

- *Copy statement*
  x = y
  value of ***y*** is assigned to **x**

- *Unconditional jump*
  goto L
  The three-address statement with label **L** is executed next

- *Conditional jump*
  if x *relop* y goto L
  relop is <, =, >=, etc. If **x** stands in relation ***relop*** to **y**, execute statement with label **L**, next otherwise

- *Indexed assignment*
  a) **x = y[i]**
  b) **x[i] = y**
  In a), set **x** to value in location **i** memory units beyond location **y**.
  In b), set contents of location **i** memory units beyond **x** to **y**.

We associate with a boolean expression E two labels (attributes); E.true and E.false. The control flows to E.true if the expression evaluates to true, to E.false otherwise. Following is syntax-directed translation for

$S \rightarrow$ if E then S_1

E.true = newlabel()
E.false = S.next
S_1.next = S.next
S.code = E.code || gen(E.true ':') || S_1.code

The attribute "**next**" records the label of the next statement to execute. "**code**" is string-valued attribute that holds the actual code generated in the form of a

character string. The code can be eventually written out to a file. The || is the string concatenation operator, that is "hello" || " world" will yield the combined string "hello world".
Suppose E is "a < b". E.code would be

```
if a < b goto E.true
goto E.false
```

We will discuss semantic rules for boolean expressions shortly

The syntax-directed translation for

$S \rightarrow$ if E then S_1 else S_2

is

```
E.true = newlabel()
E.false = newlabel()
S1.next = S.next
S2.next = S.next
S.code = E.code || gen(E.true ':') ||
                   S1.code ||
                   gen('goto' S.next)
                   || gen(E.false ':')
                   || S2.code
```

Similarly, the syntax-directed translation for the while loop is

$\underline{S} \rightarrow \underline{\text{while E do } S_1}$

```
S.begin = newlabel()
E.true = newlabel()
E.false = S.next
S1.next = S.begin
S.code = gen(S.begin ':') ||
                   E.code ||
                   gen(E.true ':')
                   || S1.code ||
                   gen('goto' S.begin)
```

Lecture 39

Boolean Expressions

In programming languages, boolean expressions have two primary purposes:

- compute logical values such as **x = a < b && d > e**
- conditional expressions in flow-of-control statements

Consider the grammar for Boolean expressions

E	→	E or E
	|	E and E
	|	not E
	|	(E)
		id relop
	|	id
	|	true
	|	false

We will implement the translation for boolean expressions by flow of control method, i.e., representing the value of a boolean expression by a position reached in the program. Here are the syntax directed translation for the grammar rules

E → id_1 relop id_2

E.code = gen('if' id_1 relop id_2 'goto' E.true) || gen('goto' E.false)

E → true

E.code = gen('goto' E.true)

E → false

E.code = gen('goto' E.false)

E → E_1 or E_2

E_1.true = E.true
E_1.false = newlabel()
E_2.true = E.true
E_2.false = E.false
E.code = E_1.code || gen(E_1.false ':') || E_2.code

E → E_1 and E_2

E_1.true = newlabel()
E_1.false = E.false
E_2.true = E.true
E_2.false = E.false

E.code = E_1.code || gen(E_1.true ':') || E_2.code

$E \rightarrow not\ E_1$

E_1.true = E.false
E_1.false = E.true
E.code = E_1.code

$E \rightarrow (E_1)$

E_1.true = E.true
E_1.false = E.false
E.code = E_1.code

Example: Consider the expression

a < b or c < d and e < f

Suppose the true and false exits for the entire expression are **Ltrue** and **Lfalse**.

The syntax directed translation scheme will generate the code

```
        if a < b goto Ltrue goto
        L1
L1:     if c < d goto L2 goto
        Lfalse
L2:     if e < f goto Ltrue goto
        Lfalse
```

Example: Consider the while statement

```
while a < b
   if c < d then x =
         y + z
   else
         x = y – z
```

The translation scheme will generate the following code:

```
L1:     if a < b goto L2 goto
        Lnext
L2:     if c < d goto L3 goto
        L4
L3:     t1 = y + z x =
        t1 goto L1
L4:     t2 = y – z x =
        t2 goto L1
Lnext:          nop
```

Implementation of Syntax-directed Translation

The easiest way to implement syntax-directed definitions is to use two passes: construct a syntax tree for the input in the first pass and then walk the tree in depth-first order evaluating attributes and emitting code. We would like to use only one pass if possible. The problem in generating three-address code in one pass is that we may not know the labels that the control must go to when we generate jump statements. However, by using a technique called *back-patching*, we can generate code in one pass.

As we generate code, we will generate the jumps (conditional or unconditional) with targets temporarily left unspecified. Each such statement will be put on a list of goto statements that have targets missing. We will fill the labels when the proper label can be determined; this is the backpatching step. Backpatching is especially suited for bottom-up parsers.

Assume that the quadruples are put into a simple array. Labels will be indices into this array.

To manipulate list of goto labels, we will use three functions:

1. makelist(i)
   creates and returns a new list containing only i, the index of quadruple

2. merge(p_1, p_2)
   concatenates lists pointed to by p_1 and p_2 and returns the concatenated list.

3. backpatch(p, i)
   inserts i as the target label for each of the goto statements on list pointed to by p

We now construct a translation scheme suitable for producing quads (IR) for boolean expressions during bottom-up parsing. The grammar we use is

$$
\begin{array}{lcl}
E & \rightarrow & E_1 \text{ or } M\ E_2 \\
 & | & E_1 \text{ and } M\ E_2 \\
 & | & \text{not } E_1 \\
 & | & (\ E_1\ ) \\
 & | & id_1 \text{ relop } id_2 \\
 & | & \text{true} \\
 & | & \text{false}
\end{array}
$$

$$M \rightarrow \varepsilon$$

We will associate synthesized attributes truelist and falselist with the nonterminal E. Incomplete jumps will be placed on these list.

We associate the semantic action

{ M.quad = nextquad() }

with the production M → ε. The function nextquad() returns the index of the next quadruple to follow. The attribute **quad** will record this index.

1. $E \rightarrow E_1$ and M E_2
   {
   backpatch(E_1.truelist, M.quad); E.truelist = E_2.truelist;
   E.falselist = merge(E_1.falselist, E_2.falselist);
   }
   Let's look at the mechanics. If E_1 is false, E is false because of the and clause. If E_1 is true, we need to evaluate E_2. The start of E_1, i.e., the index of the first quad for E_1 is recorded by M.quad; in a bottom up parse, the reduction M → ε will occur before reduction to E_2. The backpatch sets the targets of goto's in E_1.truelist to the start of E_2.

2. $E \rightarrow E_1$ or M E_2
   {
   backpatch(E_1.falselist, M.quad);
   E.truelist = merge(E_1.truelist, E_2.truelist); E.falselist = E_2.falselist;
   }

3. $E \rightarrow$ not E_1
   {
   E.truelist = E_1.falselist;
   E.falselist = E_1.truelist;
   }

4. $E \rightarrow (E_1)$
   {
   E.truelist = E_1.truelist;
   E.falselist = E_1.falselist;
   }
5. $E \rightarrow id_1$ relop id_2
   {

```
            E.truelist = makelist(nextquad());
            E.falselist =
            makelist(nextquad()+1);
            emit('if' id1 relop id2 'goto _')       ;
            emit('goto _' );
    }
```

6. E → true
   {
   E.truelist = makelist(nextquad());
   emit('goto _');
   }

7. E → false
   {
   E.falselist = makelist(nextquad());
   emit('goto _');
   }

Lecture 40

Example: consider, the boolean expression

a < b or c < d and e < f

Recall the syntax directed translation for the production

$E \rightarrow id_1$ relop id_2
{
E.truelist = makelist(nextquad());
E.falselist = makelist(nextquad()+1);
emit('if' id_1 relop id_2 'goto _') ;
emit('goto _');
}

We carry out a bottom-up parse. In response to reduction of a < b to E, the list E.truelist gets {100} and E.falselist gets {101} and the two quadruples

100: if a < b goto _

101: goto _

are generated. Notice that the goto's are generated with targets. These are precisely the goto's whose quad indices 100 and 101 are recorded in the truelist and falselist attributes of E. These will be patched later in the parse via the backpatching mechanism.

The next reduction to happen is $M \rightarrow \varepsilon$ which is in the production

$E \rightarrow E_1$ or M E_2

This reduction will eventually take place when reduction to E_2 happens. This marker non-terminal M records the value of nextquad which at this time is 102. Next, the reduction of c < d to E leads to the list E.truelist getting {102}, E.falselist getting {103} and the two quadruples

102: if c < d goto _

103: goto _

are generated.

Next reduction is $M \rightarrow \varepsilon$. The marker non-terminal M in the production

$E \rightarrow E_1$ and M E_2

records the value of nextquad which at this time is 104, the quad index of first quad of E_2 . Reducing e < f to E causes E.truelist to get {104}, E.falselist to get {105} and the generation of quads

104: if e < f goto _

105: goto _

We now reduce by the production

$E \rightarrow E_1$ and M E_2

Recall the semantic actions associated with this rule:

$E \rightarrow E_1$ and M E_2
{
backpatch(E_1.truelist, M.quad);
E.truelist = E_2.truelist;
E.falselist = merge(E_1.falselist, E_2.falselist);
}

The six quadruples generated so far are

100: if a < b goto _

101: goto _
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _

The semantic action calls

backpatch({102},104)

The backpatch fills in 104 as the target of the goto in quad 102.

100: if a < b goto _

101: goto _
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _

The next two semantic actions define E.truelist and E.falselist. This way, the synthesized attributes propagate the attributes up the parse tree.

We now reduce by the production

$E \rightarrow E_1$ or M E_2

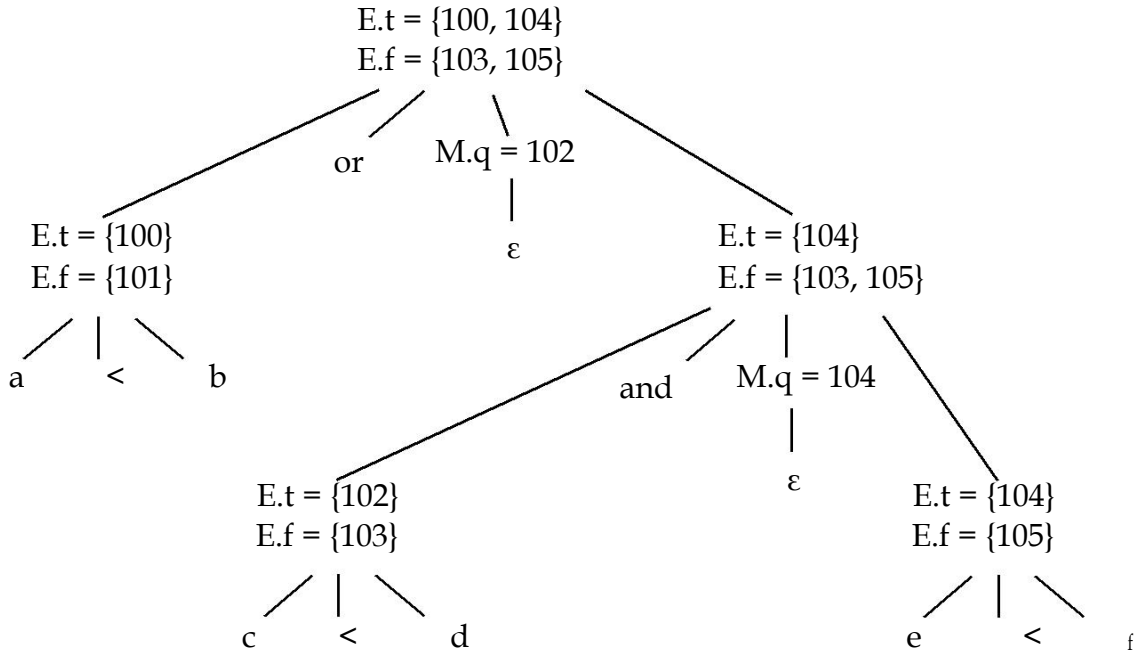
The semantic action calls

backpatch({101},102)

which fills in 102 in statement 101:

100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _

The remaining goto's will have their targets backpatched later in the parse. The attributed parse tree at this stage is



Lecture 41

Flow-of-Control Statements

We now use backpatching to translate flow-of-control statements in one pass. We will use the same list-handling procedures as before.

S → if E then S
| if E then S else S
| while E do S
| begin L end
| A

L → L ; S
| S

The semantic actions associated with each production are

1. S → if E then M_1 S_1 N else M_2 S_2
{
backpatch(E.truelist, M_1.quad);
backpatch(E.falselist, M_2.quad);
S.nextlist = merge(S_1.nextlist, merge(N.nextlist,S_2.nextlist));
}

2. N → ε
{
N.nextlist = makelist(nextQuad());
emit('goto_');
}

3. M → ε
{
M.quad = nextQuad();
}

4. S → if E then M S_1
{
backpatch(E.truelist, M.quad);
S.nextlist = merge(E.falselist,S_1.nextlist);
}

5. S → while M_1 E do M_2 S_1
{

backpatch(S_1.nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist = E.falselist; emit('goto' M_1.quad);
}

6. S → begin L end
{
S.nextlist = L.nextlist;
}

7. S → A
{
S.nextlist = nil;
}

8. S → L_1 ; M S
{
backpatch(L_1.nextlist, M.quad);
L.nextlist = S.nextlist;
}

9. L → S
{
L.nextlist = S.nextlist;
}

Example: Let go through the example with the following input statement

if a < b or c < d and e < f then x = y+z else x = y-z

The bottom-up parse will reduce the compound boolean expression
a < b or c < d and e < f to E_1 or M E_2 which we have already covered in the previous example. We thus assume that the quads for the boolean expression have been generated. The sentential form at this stage is

if E_1 or M E_2 then x = y+z else x = y-z

The reduction E → E_1 or M E_2 yields

if E then x = y+z else x = y-z

The semantic actions define the synthesized attributes E.truelist=[100,104] and E.falselist=[103,105].

We now trace the remaining bottom up parse and execute the semantic actions:

if E then M_1 x=y+z else x=y-z	$M_1 \rightarrow \varepsilon$ { M_1.quad = 106 }
$\Rightarrow$ if E then M_1 A else x=y-z	A → x=y+z { emit('x=y+z') }
$\Rightarrow$ if E then M_1 S_1 else A	$S_1 \rightarrow$ A { S_1.nextlist = nil}
$\Rightarrow$ if E then M_1 S_1 N else x=y-z	N → ε { N.nextlist = [107] emit('goto _' }
$\Rightarrow$ if E then M_1 S_1 N else M_2 x=y-z	$M_2 \rightarrow \varepsilon$ { M_2.quad = 108 }
$\Rightarrow$ if E then M_1 S_1 N else M_2 A	A → x=y-z { emit('x=y-z') }
$\Rightarrow$ if E then M_1 S_1 N else M_2 S_2	$S_2 \rightarrow$ A { S_2.nextlist = nil }
$\Rightarrow$ S	{ backpatch([100,104],106) backpatch([103,105],108) S.nextlist=[107]}

The array of quadruples at this stage will contain

100	if a < b goto 106
101	goto 102
102	if c < d goto 104
103	goto 108
104	if e < f goto 106
105	goto 108
106	x=y+z
107	goto _
108	x=y-z
109	

Semantic Actions in YACC

The syntax-directed translation statements can be conveniently specified in YACC

The %**union** will require more fields because the attributes vary. The actual mechanics will be covered in the handout for the syntax-directed translation phase of the course project

Lecture 42

Code Generation

The code generation problem is the task of mapping intermediate code to machine code. The generated code must be correct for obvious reasons. It should be efficient both in terms of memory space and execution time.

The input to the code generation module of compiler is intermediate code (optimized or not) and its task is typically to produce either machine code or assembly language code for a target machine.

The code generation module has to tackle a number of issues.

- Memory management: mapping names to data objects in the run-time system.
- Instruction selection: the assembly language instructions to choose to encode intermediate code statements
- Instruction scheduling: instruction chosen must utilize the CPU resources effectively. Hardware stalls must be avoided.
- Register allocation: operands are placed in registers before executing machine operation such as ADD, MULTIPLY etc. Most processors have a limited set of registers available. The code generator has to make efficient use of this limited resource

For our discussion, we will target a machine that has the following general characteristics. Most actual processors are similar to such architecture.

The machine is byte-addressable with 4-byte words. It has N general -purpose registers. It uses two-address instructions of the form *op source, destination*. The target assembly language operations are:

- MOV source, destination
- ADD source, destination
- SUB source, destination (dest = dest – source)
- GOTO address
- CJ conditional jump

More instruction will be added to the instruction set as needed.

The following table presents the addressing modes for source or destination operands.

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + contents(R)	1
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents(c+contents(R))	1
literal	#c	c	1
stack	SP	SP	0
indexed stack	c(SP)	c + contents(SP)	1

We associate a cost with each instruction. This will allow us to compute the cost of generated code. The cost corresponds to length of instruction. For example the instruction

```
MOV R0,R1  ; R0 = c(R1)
```

has cost 1 while

```
MOV R5,M   ; M = c(R5)
```

has cost 2: 1 for instruction, 1 additional for memory address. The column title "ADDED COST" indicates this additional cost.

Simple Code Generation

We start with a simple code generation strategy: define a target code sequence for each intermediate code (such as 3-address code) statement type. Thus,

Intermediate code	becomes…
a = b	MOV b,a

a = b[c]	MOV addr(b),R0
	ADD c, R0
	MOV *R0,a

a = b + c	MOV b,a ADD c,a

a[b] = c	MOV addr(a),R0
	ADD b,R0
	MOV c,*R0

Consider the C statement: a[i] = b[c[j]]; the simple code generator will emit

```
t1 := c[j]          MOV addr(c), R0
                    ADD j, R0
                    MOV *R0, t1
t2 := b[t1]         MOV addr(b), R0
                    ADD t1, R0
                    MOV *R0, t2
a[i] := t2          MOV address(a), R0
                    ADD i, R0
                    MOV t2, *R0
```

The cost of this code is 18 and we are forced to allocate space for two temporaries. While the simple approach works, it does not produce good code. There a number of reasons for this. The generator considers each IR (3-address in this case) alone and makes local decision. It does not take temporary variables into account. One optimization possible is to get rid of the temporaries:

```
MOV addr(c), R0
ADD j, R0
MOV addr(b), R1
ADD *R0, R1
MOV addr(a), R2
ADD i, R2
MOV *R1, *R2
```

The cost of this code is 12. We can optimize further:

```
MOV addr(c), R0
ADD j, R0
MOV addr(a), R2
ADD i, R2
MOV *addr(b)(R0), *R2
```

The cost of this code is 10. What is needed is a way to generate machine code based on past and future use of the data.

Lecture 43

Control Flow Graph - CFG

A *control flow graph* is the triplet CFG = < V, E, Entry >, where V = vertices or nodes, representing an instruction or *basic block* (group of statements), E = (V x V) edges, potential flow of control. Entry is an element of V, the unique program entry.

Basic Blocks

A *basic block* is a sequence of consecutive statements with single entry/single exit. Flow of control only enters at the beginning and only leaves at the end. The can be variants of basic blocks with single entry/multiple exit, multiple entry/single exit.

Generating CFGs

In order to generate a CFG, we partition the intermediate code (3-address code, for example) into basic blocks. Edges are added corresponding to control flow between blocks. An unconditional goto in the IR will lead to a single edge to another or the same block. A conditional goto will lead to multiple edges. If there is no goto at the end of a block, the control passes to first statement of next block.

Here is the algorithm for partitioning intermediate code into basic blocks. The input to the algorithm is a sequence of three-address statements. The algorithm will output a list of basic blocks with each three-address statements in exactly one block.

Algorithm: partition 3-address statements into basic blocks:

1. Determine the set of leaders – the first statements of basic blocks. The rules are:
    - The first statement is a *leader*
    - Any statement that is the target of a conditional or unconditional goto is a leader
    - Any statement that immediately follows a goto or conditional goto 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: consider the C fragment for computing dot product $a^T b$ of two vectors a and b of length 20

$a^T b = a_1b_1 + a_2b_2 + \ldots\ldots\ldots + a_{20}b_{20}$

```
prod = 0; i =
1; do {
        prod = prod + a[i]*b[i]; i = i + 1;
} while ( i <= 20 );
```

The 3-address code for the dot product with the two leaders highlighted

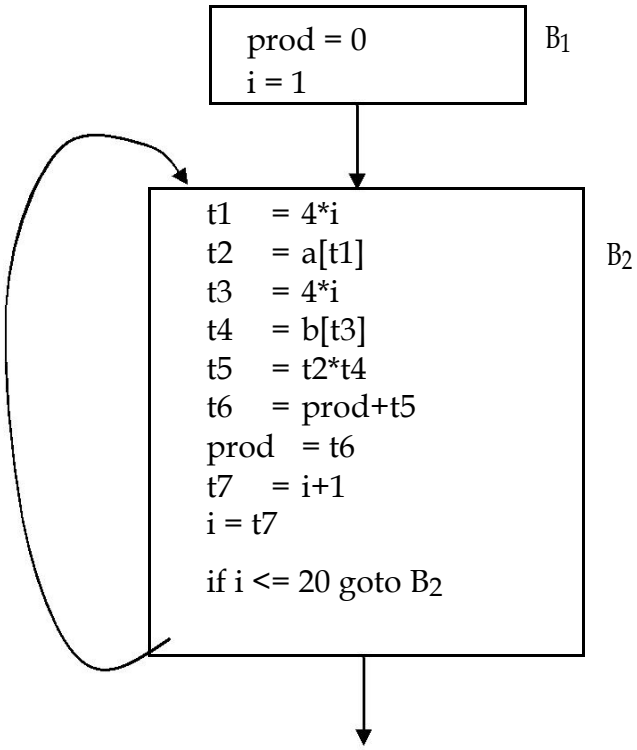
next line of goto will also be the leader

```
1       prod = 0
2       i = 1
 3      t1   = 4*i        /* offset */
 4      t2   = a[t1]      /*   a[i] */
 5      t3   =  4*i
 6      t4   =  b[t3]     /*   b[i] */
7       t5 = t2*t4
8       t6 = prod+t5
9       prod = t6
10      t7 = i+1
11      i = t7
12      if i <= 20 goto (3)
```

The two basic blocks are

```
1       prod = 0                                   B1
2       i = 1
 3      t1   = 4*i        /* offset */
 4      t2   = a[t1]      /*   a[i] */             B2
 5      t3   =  4*i
 6      t4   =  b[t3]     /*   b[i] */
7       t5 = t2*t4
8       t6 = prod+t5
9       prod = t6
10      t7 = i+1
11      i = t7
12      if i <= 20 goto (3)
```

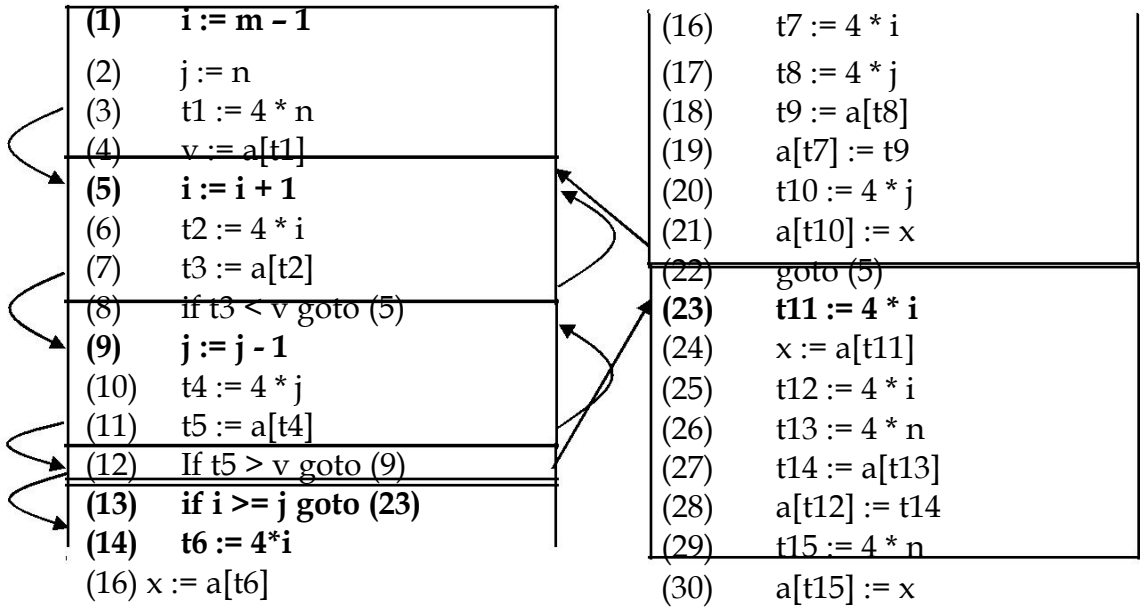
This yields the following CFG; note that the target of the condition goto at the end of the second block has been replaced by reference to block 2.



Let us consider a more complex example. Here is the quicksort algorithm encoded as a recursive function in C++

```
i = 1 Quick
Sort
void quicksort(int m, int n)
{
        int i,j,v,x;
        if( n <= m ) return; i=m-1;
        j=n; v=a[n]; while(true) {
                do i=i+1; while( a[i] < v); do j=j-1;
                while( a[j] > v); i( i >= j ) break;
                x=a[i]; a[i]=a[j]; a[j]=x;
        }
        x=a[i]; a[i]=a[n]; a[n]=x; quicksort(m,j);
        quicksort(i+1,n);
}
```

The 3-address for the highlighted portion of the routine (the recursive calls have been left out) with the leaders highlighted and the resulting CFG is



Basic Block Code Generation

The code generation can carried out at the basic block level. A number of strategies are available to generate code from a basic block. The three we will discuss are

1. Basic - using liveness information
2. Using DAGS - node numbering
3. Register Allocation

In case of the basic code generation strategy, the generator deals with each basic block individually to emit machine code for the block using liveness information. At the end of the block, the generator emits code to save any live values left in registers.

Computing Live/Next Use Information

For the statement:

x = y + z

x has a *next use* if there is a statement s that references x and there is some way for control to flow from the original statement to s.

```
        x = y + z
              .....
              .....
s       t1 = x – t3
```

A variable is *live* at a given point in time if it has a next use. Liveness tells us whether we care about the value held by a variable. Here is the algorithm for computing live status of variables in a basic block"

Algorithm: Computing live status
Input:
A basic block.
Output:
For each statement, set of live variables

Method:

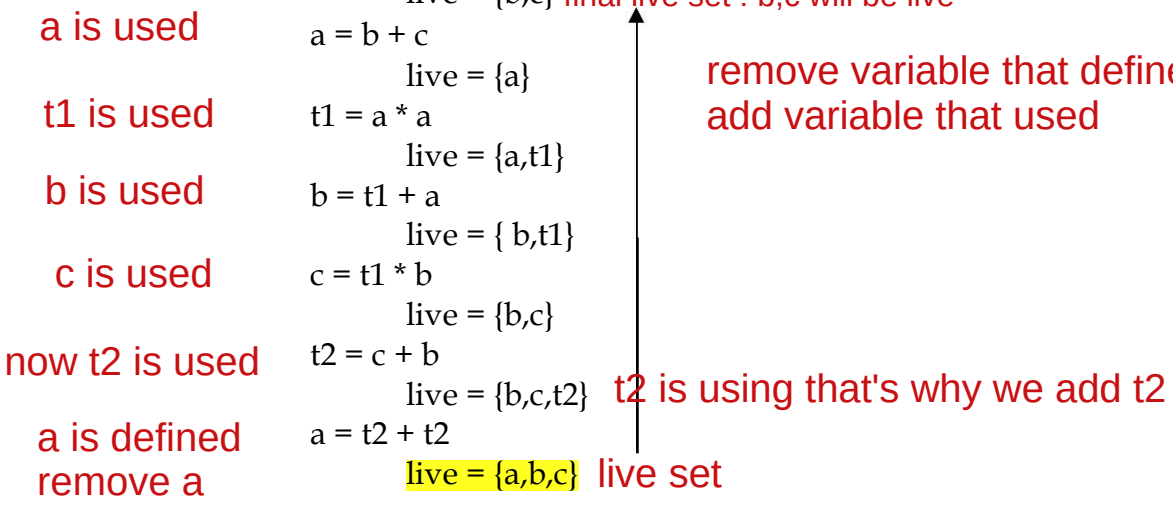
1. Initially all non-temporary variables go into live set.
2. for i = last statement to first statement:
   for statement i: x = y op z
   attach to statement i, current live
   set. remove x from set.
   add y and z to set.

Lecture 44

Example: let us apply the algorithm to the following segment of 3-address code:

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

start in reverse order



Basic Code Generation

With live/next use information computed, the basic code generation algorithm proceeds as follows. Process the 3 -address instructions from beginning to end of a block. For each instruction, use machine registers to hold operands whenever possible. A non-live value in a register can be discarded, freeing that register. The code generator uses two data structures for keeping track of register usage:

1. Register descriptor - register status (empty, inuse) and contents (one or more "values")
2. Address descriptor - the location (or locations) where the current value for a variable can be found (register, stack, memory)

Instruction type: x = y op z

1. If y is non-live and in register R (alone) then generate

OP z′, R

where z′ = best location for z. i.e., lookup address descriptor for z. Prefer register location if z is present in a register.

2. If operation is commutative, z is non-live and is in register R (alone), generate

OP y′, R

(y′ = best location for y)

3. If there is a free register R, generate

MOV y′, R
OP z′, R

4. Use a memory location. Generate

MOV y′,x
OP z′,x

After generating machine instructions, update information about the current best location of x. If x is in a register, update that register's information (descriptor). If y and/or z are not live after this instruction, update register and address descriptors according.

Let us return to the 3-address code example and apply the basic code generation algorithm. Recall the basic block with liveness information:

```
        live = {b,c}
a = b + c
        live = {a}
t1 = a * a
        live = {a,t1}
b = t1 + a
        live = { b,t1}
c = t1 * b
        live = {b,c}
t2 = c + b
        live = {b,c,t2}
a = t2 + t2
        live = {a,b,c}
```

Initially

three registers:
(-, -, -) all
empty current

values:
(a,b,c,t1,t2) = (m,m,m, -, -)

1: a = b + c,
Live =
a
getreg(): L = R1
MOV b,R1
ADD c,R1 ; R1 := R1 + c
Registers: (a, -, -)
current values: (R1,m,m, -, -)

2: t1 = a * a, Live
= a,t1
L = R2 (since a is
live) MOV
R1,R2
MUL R2,R2 ; R2 = R2* R2
Registers: (a,t1, -)
current values: (R1,m,m,R2, -)

3: b = t1 + a,
Live = b,t1
Since a is not live L = R1

ADD R2,R1 ; R1 = R1+R2

Registers: (b,t1, -)
current values: (m,R1,m,R2, -)

4: c = t1 * b, Live
= b,c
Since t1 is not live L = R2
MUL R1,R2 ; R2 = R1*R2
Registers: (b,c, -)
current values: (m,R1,R2, -, -)

5: t2 = c + b,
Live =
b,c,t2 L =
R3
MOV R2,R3
ADD R1,R3 ; R3 = R1+R2
Registers: (b,c,t2)
current values: (m,R1,R2, -,R3)

6: a = t2 + t2,

Live =
a,b,c
ADD R3,R3
Registers: (b,c,a)
current values: (R3,R1,R2,-,R3)

End of block

move all live variables to
memory: MOV R3,a
MOV R1,b
MOV R2,c
all registers
available

Thus the machine code (assembly language) generated is

```
; a := b + c
        LOAD b,R1
        ADD c,R1            ; R1 := R1 + c
; t1 := a * a
        MOV R1,R2
        MUL R2,R2           ; R2 = R2* R2
; b := t1 + a
        ADD R2,R1           ; R1 = R1+R2
; c := t1 * b
        MUL R1,R2           ; R2 = R1*R2
; t2 := c + b
        MOV R2,R3
        ADD R1,R3           ; R3 = R1+R2
; a := t2 + t2
        ADD R3,R3
        MOV R3,a            ; mov live
                            ; var to
        MOV R1,b            memory
        MOV R2,c
```

Lecture 45

Liveness information allows us to keep values in registers if they will be used later. An obvious concern is why do we assume all variables are live at the end of blocks? Why do we need to save live variables at the end? It seems reasonable to perceive that we might have to reload them in the next block. To do this, we need to determine live/next use information across blocks and not just within a block. This requires global data-flow analysis.

Global Data-Flow Analysis

A Directed Acyclic Graph (DAG) for a basic block has the following labels for the nodes:

- Leaves are labeled by unique identifiers.
- Interior nodes are labeled by operator symbols.
- Nodes can have multiple labels since they represent computed values.

Algorithm: Generate DAG from 3-address code

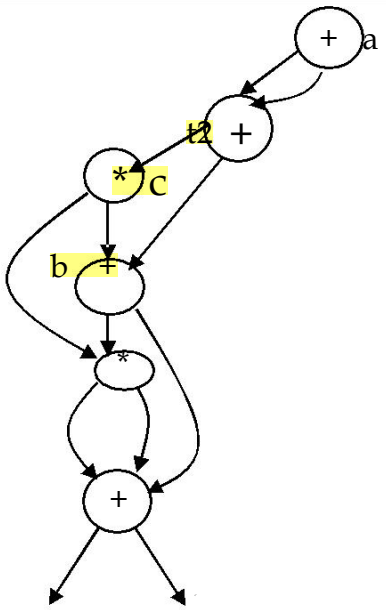
For statement i: x = y op z

- if y op z node exists,
  add x to the label for that node.
  else
  add node for y op z.
- if y or z exist in the dag,
  point to existing locations
  else
  add leaves for y and/or z and have the op node point to them.

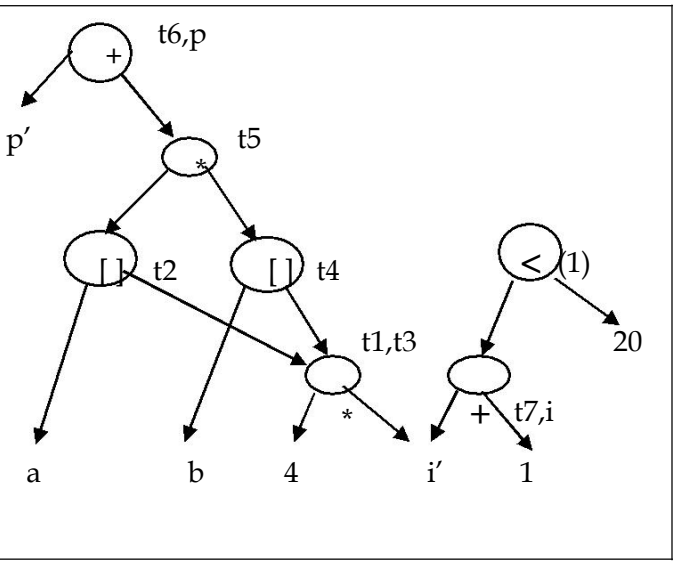
- label the op node with x.

- if x existed previously as a leaf,
  subscript that previous entry.
- if x is associated with other interior nodes,
  remove them from that list.

Here and an example of the DAG generated for the 3-address code

a = b + c t1 = a * a b = t1 + a c = t1 * b t2 = c + b a = t2 + t2	

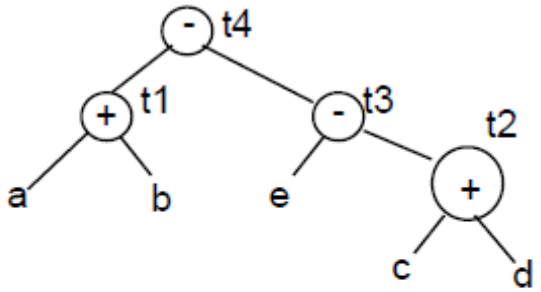
Here is another example

t1 = 4 * i t2 = a[t1] t3 = 4 * i t4 = b[t3] t5 = t2 * t4 t6 = p + t5 p = t6 t7 = i + 1 i = t7 if i <20 goto (1)	

DAGs and optimization

DAGs play an important role in code optimization. It is possible to detect *common sub-expressions* and eliminate them; a node in the DAG with more than one parent is common sub-expression.

The order in which the DAG is traversed can lead to better code. For example, the following DAG can be traversed in two ways:



Order 1	***Order 2***
t1 = a + b	t2 = c + d
t2 = c + d	t3 = e – t2
t3 = e – t2	t1 = a + b
t4 = t1 – t3	t4 = t1 – t3

The code generated for order one with 2 registers is

```
MOV a, R0
ADD b,R0
MOV c, R1
ADD D, R1
MOV R0,t1
MOV e, R0
SUB R1,R0
MOV t1,R1
SUB R0, R1
MOV R1,t4
```

Ten machine instructions are generated.

Whereas, order #2 with 2 registers leads to

```
MOV c, R0
ADD D, R0
```

```
MOV e, R1
SUB R0,R1
MOV a,R0
ADD b, R0
MOV R0,t4
```

Only seven instructions are required, a saving of three machine instructions. Reordering improved code because computation of t4 immediately followed computation of t1, its left operand. t1 must be in a register and it is.

Register Allocation

Registers in a machine are a scarce resource. The issue faced by the code generator is this how to best use the bounded number of registers. The matter is complicated by the fact that a few registers are reserved for special purposes. For example, the program counter is kept in a registers. A register is used for the keeping track of the top of the function call stack. Certain operators require multiple registers, often in pairs. Division is an example of such an operator.

The general register allocation problem is NP- complete. Heuristic algorithms exist to solve the problem. One such strategy is the by using the *graph coloring* algorithm: given a graph, color the nodes with different colors such that no two nodes that have an edge between them have the same color. Here is how the graph coloring algorithm can be used to compute register allocation for K registers in a machine.

Algorithm: K registers allocation with graph coloring

1. Compute liveness information.
2. Create interference graph G
3. one node for each variable, an edge connects two variables if one is live at a point where the other is defined
4. Simplify: for any node m with less than K neighbors, remove it from the graph and push it onto a stack. If (G - m) can be colored with K colors, so can G. If we reduce the entire graph, goto step 5.
5. Spill: if we get to the point where we are left with only nodes with degree >= K, mark some node for potential spilling (to memory). Remove and push onto stack. Back to step 3.
6. Assign colors: starting with empty graph, rebuild graph by popping elements off the stack and assigning a color different from neighbors.

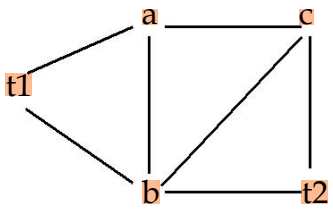
Potential spill nodes may or may not be colorable.

Process may require iterations and rewriting of some of the code to create more temporaries.

Let us apply the algorithm to the following 3-address code

		live
a	= b + c	{a}
t1	= a * a	{t1,a}
b	= t1 + a	{b,t1}
c	= t1 * b	{b,c}
t2	= c + b	{b,c,t2}
a	= t2 + t2	{a,b,c}

The interference graph generated is



Upon coloring the nodes, the final register allocation assuming 3 registers is