Lecture Notes on Principles of Complier Design

By D.R.Nayak,Aasst.prof Govt.College of Engg.Kalahandi,Bhawanipatna

1. Introduction to Compilers

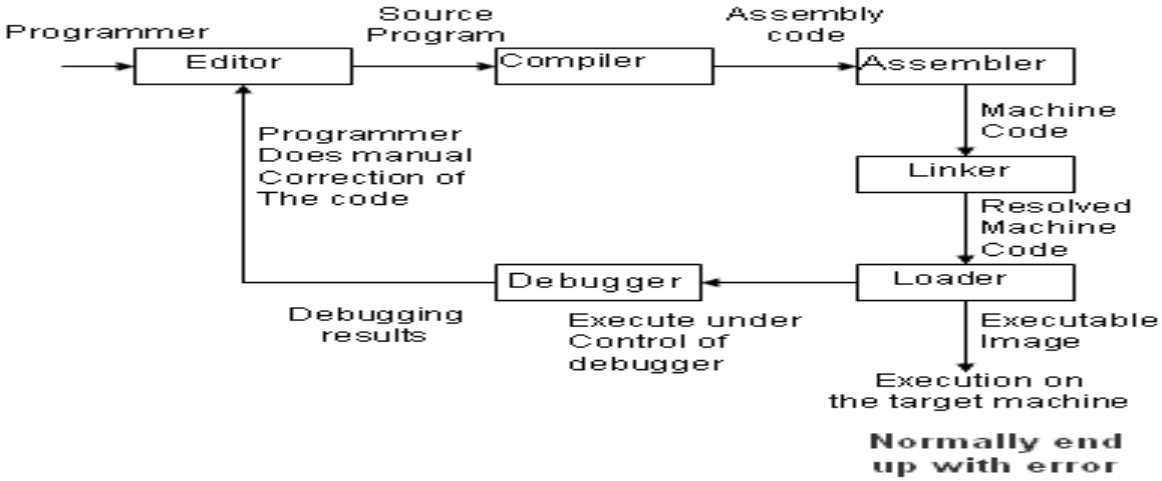
What is Compiler?

Compiler is a program which translates source program written in one language to an equivalent program in other language (the target language). Usually the source language is a high level language like Java, C, C++ etc. whereas the target language is machine code or "code" that a computer's processor understands.

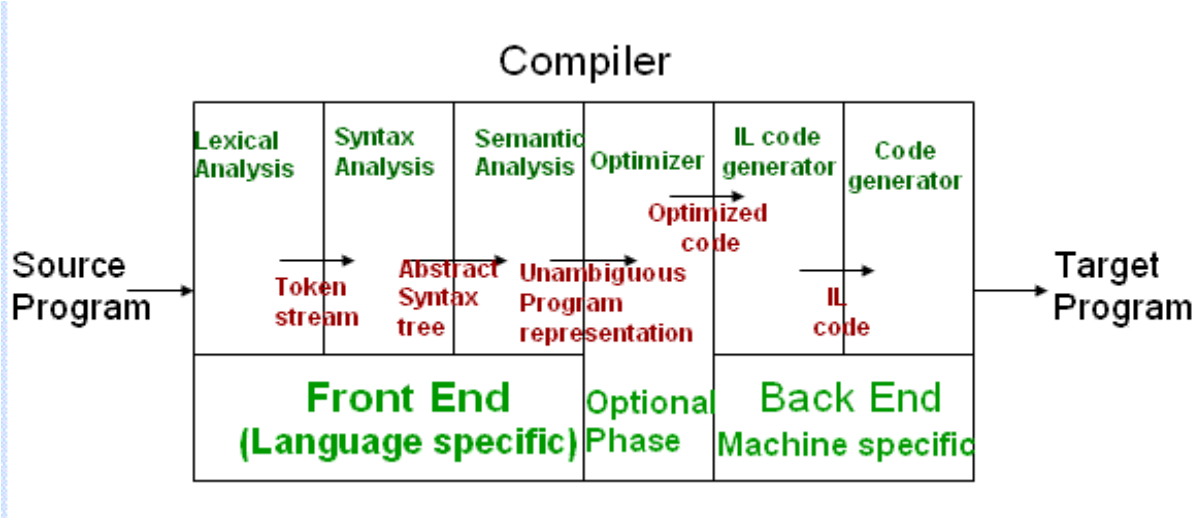
Simple Design of Complier

Many modern compilers have a common 'two stage' design. The "front end" translates the source language or the high level program into an intermediate representation. The second stage is the "back end", which works with the internal representation to produce low level code .

The Enhanced Design



Phases of complier



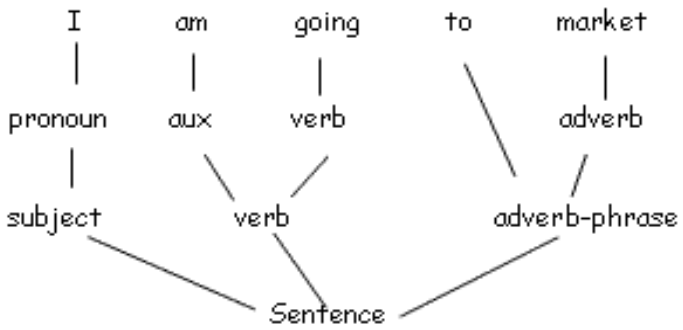
Lexical Analysis

- Recognizing words is not completely trivial. For example: ist his ase nte nce?
- Therefore, we must know what the word separators are
- The language must define rules for breaking a sentence into a sequence of words.
- Normally white spaces and punctuations are word separators in languages.
- In programming languages a character from a different class may also be treated as word separator.
- The lexical analyzer breaks a sentence into a sequence of words or tokens: - If a == b then a = 1 ; else a = 2 ; - Sequence of words (total 14 words) if a == b then a = 1 ; else a = 2 ;

In simple words, lexical analysis is the process of identifying the *words* from an input string of characters, which may be handled more easily by a parser. These words must be separated by some predefined delimiter or there may be some rules imposed by the language for breaking the sentence into tokens or words which are then passed on to the next phase of syntax analysis.

The Second step

Syntax checking or parsing



Syntax analysis is a process of imposing a hierarchical structure on the token stream. It is basically like generating sentences for the language using language specific grammatical rules.

Semantic Analysis

Since it is too hard for a compiler to do semantic analysis, the programming languages define strict rules to avoid ambiguities and make the analysis easier. This has been done by putting one outside the scope of other so that the compiler knows that these two aditya are different by the virtue of their different scopes.

```
{
        int Aditya = 4;
        {
                int Aditya = 6;
                cout << Aditya;
        }
}
```

Code Optimization

.It is the optional phase and Run faster
- Use less resource (memory, registers, space, fewer fetches etc.)
- Common sub-expression elimination
- Copy propagation
- Dead code elimination
- Code motion
- Strength reduction
- Constant folding

Example1:

```
int x = 3;
int y = 4;
int *array[5];
for (i=0; i<5;i++)
*array[i] = x + y;
```

Because **x** and **y** are invariant and do not change inside of the loop, their addition doesn't need to be performed for each loop iteration. Almost any good compiler optimizes the code. An optimizer moves the addition of **x** and **y** outside the loop, thus creating a more efficient loop. Thus, the optimized code in this case could look like the following:

```
int x = 3;
int y = 4;
int z = x + y;
int *array[5];
for (i=0; i<5;i++)
*array[i] = z;
```

Some of the different optimization methods are:

1) Constant Folding - replacing y= 5+7 with y=12 or y=x*0 with y=0
2) Dead Code Elimination - e.g.,

```
        If (false)
        a = 1;
        else
```

a = 2;
with a = 2;

3) Peephole Optimization - a machine-dependent optimization that makes a pass through low-level assembly-like instruction sequences of the program(called a peephole), and replacing them with a faster (usually shorter) sequences by removing redundant register loads and stores if possible.

4) Flow of Control Optimizations

5) Strength Reduction - replacing more expensive expressions with cheaper ones - like pow(x,2) with x*x

6) Common Sub expression elimination - like a = b*c, f= b*c*d with temp = b*c, a= temp, f= temp*d;

Code Generation

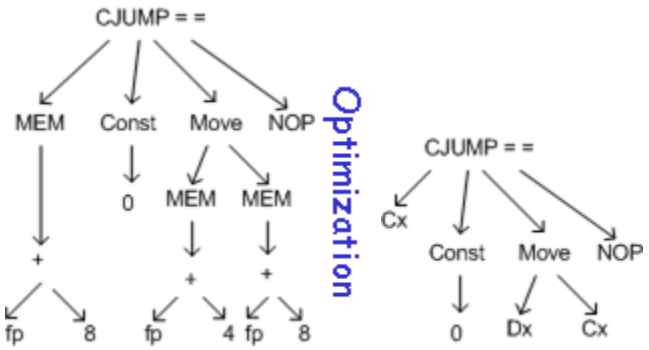
Usually a two step process

- Generate intermediate code from the semantic representation of the program
- Generate machine code from the intermediate code

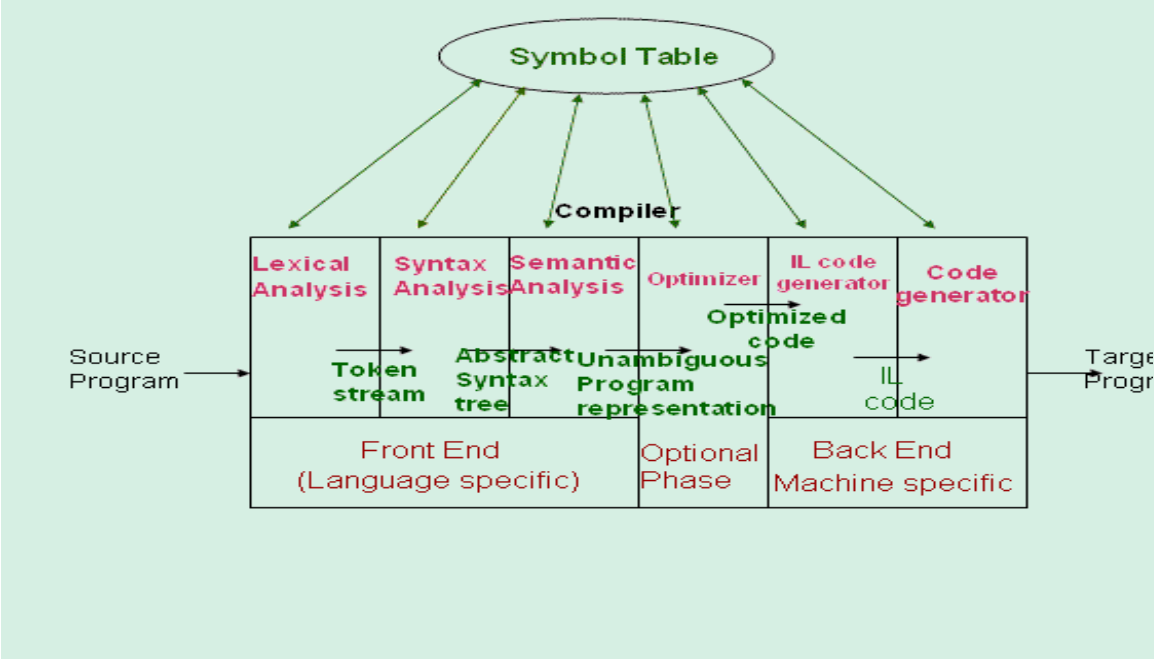
Intermediate Code Generation

1. Abstraction at the source level identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
2. Abstraction at the target level memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems
3. Code generation is mapping from source level abstractions to target machine abstractions
4. Map identifiers to locations (memory/storage allocation)
5. Explicate variable accesses (change identifier reference to relocatable/absolute address

Intermediate code generation



The final structure of complier



Lexical Analysis

Lexical Analysis

. Recognize tokens and ignore white spaces, comments

i	f		(	x	1		*	x	2	<	1	.	0	)	{

Generates token stream

if	(	x1	*	x2	<	1.0	)	{

- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata

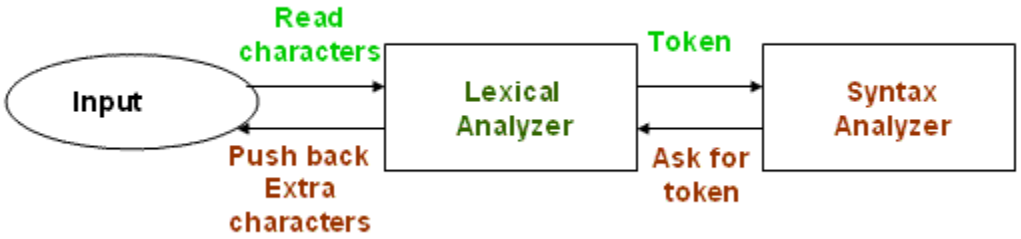
The first phase of the compiler is lexical analysis. The lexical analyzer breaks a sentence into a sequence of words or tokens and ignores white spaces and comments. It generates a stream of tokens from the input.

Token: A token is craving source program into logical entity. Sentences consist of a string of tokens. For example number, identifier, keyword, string,constants etc are tokens.

Lexeme: Sequence of characters in a token is a lexeme.

Pattern: Rule of description is a pattern. For example letter (letter | digit)* is a pattern to symbolize a set of strings which consist of a letter followed by a letter or digit.

Interface to other phases



Regular expressions in specifications

Regular expressions describe many useful languages. A regular expression is built out of simpler regular expressions using a set of defining rules. Each regular expression **R** denotes a regular language **L(R).**

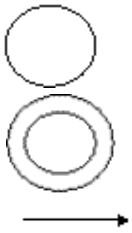
Finite Automata

A finite automata consists of - An input alphabet belonging to S
- A set of states S
- A set of transitions $state_i \longrightarrow state_j$
- A set of final states F
- A start state n

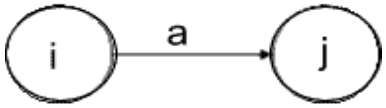
Transition $s_1 \longrightarrow s_2$ is read:
In state s_1 on input a go to state s_2
. If end of input is reached in a final state then accept

Pictorial notation

. A state



. A final state
. Transition

. Transition from state i to state j on an input a 

A state is represented by a circle, a final state by two concentric circles and a transition by an arrow.

How to recognize tokens

. Consider

relop ⟶ < | <= | = | <> | >= | >

id ⟶ letter(letter|digit)*

num ⟶ digit + ('.' digit +)? (E('+'|'-')? digit +)?

delim ⟶ blank | tab | newline

ws ⟶ delim +

. Construct an analyzer that will return <token, attribute> pairs

We now consider the following grammar and try to construct an analyzer that will return <token, attribute> pairs.

relop ⟶ < | = | = | <> | = | >

id ⟶ letter (letter | digit)*

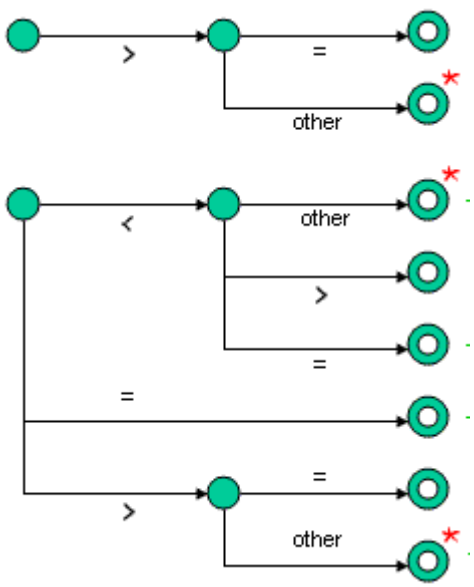
num ⟶ digit+ ('.' digit+)? (E ('+' | '-')? digit+)?

delim ⟶ blank | tab | newline

ws ⟶ delim+

Using set of rules as given in the example above we would be able to recognize the tokens. Given a regular expression R and input string x , One approach is build MINIMIZE DFA by combining all NFAs.

Transition diagram for relops



token is relop , lexeme is >=

token is relop, lexeme is >

token is relop, lexeme is <

token is relop, lexeme is <>

token is relop, lexeme is <=

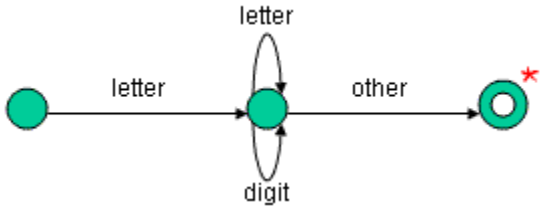
token is relop, lexeme is =

token is relop , lexeme is >=

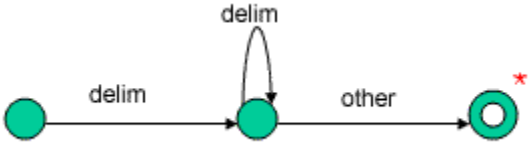
token is relop, lexeme is >

In case of < or >, we need a lookahead to see if it is a <, = , or <> or = or >. We also need a global data structure which stores all the characters. In lex, yylex is used for storing the lexeme. We can recognize the lexeme by using the transition diagram shown in the slide. Depending upon the number of checks a relational operator uses, we land up in a different kind of state like >= and > are different. From the transition diagram in the slide it's clear that we can land up into six kinds of relops.

Transition diagram for identifier



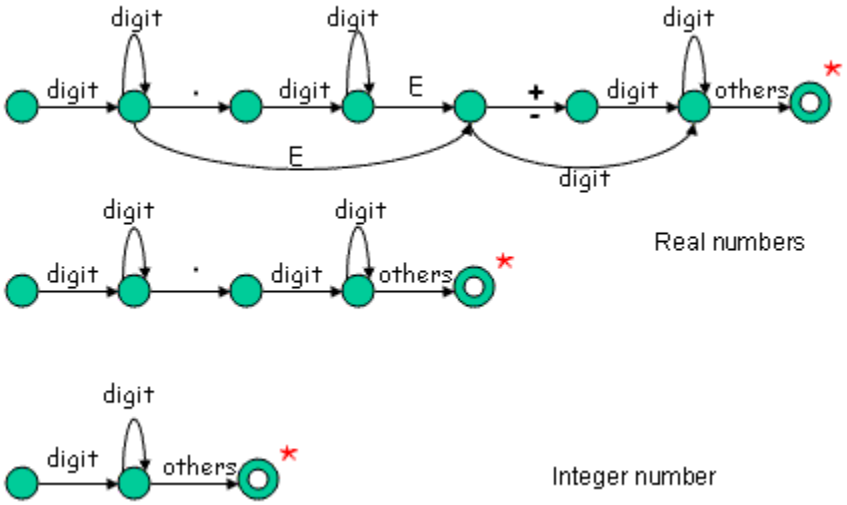
Transition diagram for white spaces



Transition diagram for identifier : In order to reach the final state, it must encounter a letter followed by one or more letters or digits and then some other symbol. Transition diagram for

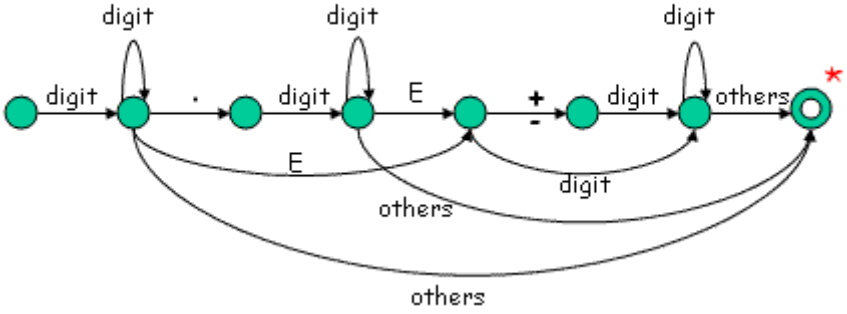
white spaces : In order to reach the final state, it must encounter a delimiter (tab, white space) followed by one or more delimiters and then some other symbol.

Transition diagram for unsigned numbers



Transition diagram for Unsigned Numbers : We can have *three* kinds of unsigned numbers and hence need three transition diagrams which distinguish each of them. The first one recognizes *exponential* numbers. The second one recognizes *real* numbers. The third one recognizes *integers.*

Another transition diagram for unsigned numbers



Lexical analyzer generator

. Input to the generator

- List of regular expressions in priority order

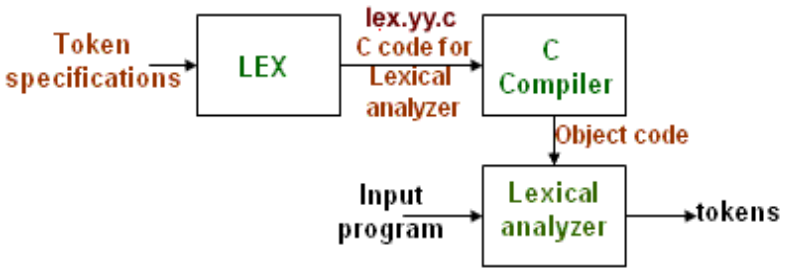
- Associated actions for each of regular expression (generates kind of token and other book keeping information)

. Output of the generator

- Program that reads input character stream and breaks that into tokens

- Reports lexical errors (unexpected characters), if any

LEX: A lexical analyzer generator



How does LEX work?

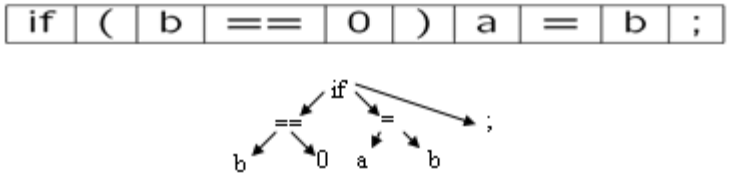
. Regular expressions describe the languages that can be recognized by finite automata
. Translate each token regular expression into a non deterministic finite automaton (NFA)
. Convert the NFA into an equivalent DFA
. Minimize the DFA to reduce number of states
. Emit code driven by the DFA tables

.

Syntax Analysis

Syntax Analysis

Check syntax and construct abstract syntax tree



. Error reporting and recovery

. Model using context free grammars

. Recognize using Push down automata/Table Driven Parsers

This is the second phase of the compiler. In this phase, we check the syntax and construct the abstract syntax tree. This phase is modeled through context free grammars and the structure is recognized through push down automata or table-driven parsers.

Syntax definition

. Context free grammars

- a set of tokens (terminal symbols)

- a set of non terminal symbols

- a set of productions of the form nonterminal $\longrightarrow$ String of terminals & non terminals

- a start symbol <T, N, P, S>

Syntax analyzers

. Testing for membership whether w belongs to L(G) is just a "yes" or "no" answer

. - Must generate the parse tree

- Handle errors gracefully if string is not in the language

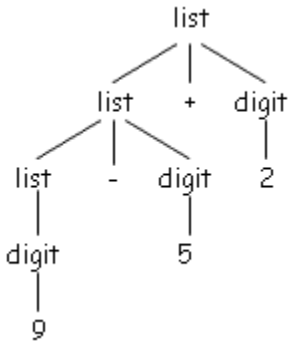
. Form of the grammar is important

Parse tree

- It shows how the start symbol of a grammar derives a string in the language
- root is labeled by the start symbol
- leaf nodes are labeled by tokens
- Each internal node is labeled by a non terminal
- if A is a non-terminal labeling an internal node and x_1 , x_2 , .x_n are labels of the children of that node then A $\longrightarrow$ x_1 x_2 . x_n is a production

Example

Parse tree for 9-5+2



The parse tree for 9-5+2 implied by the derivation in one of the previous slides is shown.
. 9 is a *list* by production (3), since 9 is a digit.
. 9-5 is a *list* by production (2), since 9 is a list and 5 is a digit.
. 9-5+2 is a *list* by production (1), since 9-5 is a list and 2 is a digit.

Ambiguity

A Grammar can have more than one parse tree for a string
Consider grammar

string $\longrightarrow$ string + string
| string - string
| 0 | 1 | . | 9

String 9-5+2 has two parse trees

A grammar is said to be an ambiguous grammar if there is some string that it can generate in more than one way (i.e., the string has more than one parse tree or more than one leftmost derivation). A language is inherently ambiguous if it can only be generated by ambiguous grammars.

.

Parsing

. Process of determination whether a string can be generated by a grammar

. Parsing falls in two categories:

Top-down parsing - A parser can start with the start symbol and try to transform it to the input. Intuitively, the parser starts from the largest elements and breaks them down into incrementally smaller parts. LL parsers are examples of top-down parsers.

Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers.

Left recursion

. A top down parser with production A $\longrightarrow$ A α may loop forever

. From the grammar A $\longrightarrow$ A α | b left recursion may be eliminated by transforming the grammar to

A $\longrightarrow$ b R

R $\longrightarrow$ α R | ε

Example . Consider grammar for arithmetic expressions

E $\longrightarrow$ E + T | T
T $\longrightarrow$ T * F | F
F $\longrightarrow$ (E) | id
. After removal of left recursion the grammar becomes
E $\longrightarrow$ T E'
E' $\longrightarrow$ + T E' | ε
T $\longrightarrow$ F T'
T' $\longrightarrow$ * F T' | ε
F $\longrightarrow$ (E) | id

As another example, a grammar having left recursion and its modified version with left recursion removed has been shown.

In general

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$

transforms to

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots\ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots\ldots \mid \alpha_m A' \mid \in$$

The general algorithm to remove the left recursion follows. Several improvements to this method have been made. For each rule of the form
A $\longrightarrow$ A $\mathbf{a}_1$ | A $\mathbf{a}_2$ | ... | A $\mathbf{a}_m$ | β_1 |β_2 | .. |β_n

Where:
. **A** is a left-recursive non-terminal.
. **a** is a sequence of non-terminals and terminals that is not null ($a \neq \varepsilon$).
. **T** is a sequence of non-terminals and terminals that does not start with **A** .
Replace the A-production by the production:
$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$
And create a new non-terminal
$A' \longrightarrow \mathbf{a}_1 A' \mid \mathbf{a}_2 A' \mid ... \mid \mathbf{a}_m A' \mid \varepsilon$

Left factoring

. In top-down parsing when it is not clear which production to choose for expansion of a symbol

defer the decision till we have seen enough input.

In general if $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$

defer decision by expanding A to a A'

we can then expand A' to β_1 or β_2

. Therefore $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$

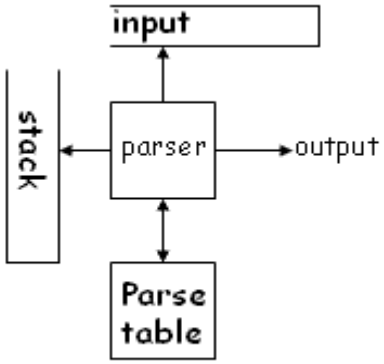
transforms to

$A \longrightarrow \alpha A'$

$A' \longrightarrow \beta_1 \mid \beta_2$

Predictive parsers
. A non recursive top down parsing method
. **Parser "predicts" which production to use**
. **It removes backtracking by fixing one production for every non-terminal and input token(s)**
. **Predictive parsers accept LL(k) languages**
- First L stands for left to right scan of input
- Second L stands for leftmost derivation
- k stands for number of lookahead token

Predictive parsing
. Predictive parser can be implemented by maintaining an external stack



Parse table is a two dimensional array M[X,a] where "X" is a non terminal and "a" is a terminal of the grammar

It is possible to build a non recursive predictive parser maintaining a stack explicitly, rather than implicitly via recursive calls. 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 end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with a $ 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 [X,a] , where X is a non-terminal, and a is a terminal or the symbol $. The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive parser looks up the production to be applied in the parsing table.

Parsing algorithm

The parser is controlled by a program that behaves as follows. The program considers X , the symbol on top of the stack, and a , the current input symbol. These two symbols determine the action of the parser. Let us assume that a special symbol ' $ ' is at the bottom of the stack and terminates the input string. 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 $\longrightarrow$ UVW}, the parser replaces X on top of the stack by UVW (with U on the top). If M[X,a] = error, the parser calls an error recovery routine.

Example: Consider the grammar

E $\longrightarrow$ T E'

E' $\longrightarrow$ +T E' | ε

T $\longrightarrow$ F T'

T' $\longrightarrow$ * F T' | ε

F $\longrightarrow$ (E) | id

As an example, we shall consider the grammar shown. A predictive parsing table for this grammar is shown in the below

Parse table for the grammar

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

Blank entries are error states.

Parsing action with input id + id * id using parsing algorithm

Example

Stack	input	action
$E	id + id * id $	expand by E ⟶ TE'
$E'T	id + id * id $	expand by T ⟶ FT'
$E'T'F	id + id * id $	expand by F ⟶ id
$E'T'id	id + id * id $	pop id and ip++
$E'T'	+ id * id $	expand by T' ⟶ ε
$E'	+ id * id $	expand by E' ⟶ +TE'
$E'T+	+ id * id $	pop + and ip++
$E'T	id * id $	expand by T ⟶ FT'

Let us work out an example assuming that we have a parse table. We follow the predictive parsing algorithm which was stated a few slides ago. With input **id ⟶ id * id** , the predictive parser makes the sequence of moves as shown. The input pointer points to the leftmost symbol of the string in the INPUT column. If we observe the actions of this parser carefully, we see that it is tracing out a leftmost derivation for the input, that is, the productions output are those of a leftmost derivation. The input symbols that have already been scanned, followed by the grammar symbols on the stack (from the top to bottom), make up the left-sentential forms in the derivation.

Constructing parse table

Input:grammer G
Output:parsing table M
Method
1.for each production A→α of the grammer,do step 2 and 3
2.for each terminal a in first(α),add A→α to M[A,a]
3.if Ɛ is in first(α) ,add A→α to M[A,b] for each terminal b in follow(A).
4.make each undefined entry of M as error.
Compute first sets

1. If X is terminal, then First (X) is {X}.
2. If X ⟶ ε is a production then add e to FIRST(X).
3. If X is a non terminal and X ⟶ $Y_1 Y_k$ .....Y_k is a production, then place ***a*** in First (X) if for some i, ***a*** is in FIRST(Y_i) and e is in all of FIRST(Y_1), FIRST(Y_2),.., FIRST(Y_{i-1});that is, Y_1 ..Y_{i-1} ⇒* **ε** . If **ε** is in FIRST(Y_j) for all i = 1,2,..,k, then add **ε** to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive **ε** , then we add nothing more to FIRST(X), but if Y_1 ⇒* **ε** , then we add FIRST(Y_2) and so on.

Example
For the expression grammar
E ⟶ T E'
E' ⟶ +T E' | ε
T ⟶ F T'
T' ⟶ * F T' | ε
F ⟶ (E) | id

First(E) = First(T) = First(F) = { (, id }
First(E') = {+, ε }
First(T') = { *, ε }

Consider the grammar shown above. For example, id and left parenthesis are added to FIRST(F) by rule (3) in the 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 are in FIRST(T) as well. As another example, e is in FIRST(E') by rule (2).

Compute follow sets

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

2. If there is a production A ⟶ a Bß, then everything in FIRST(ß) except for e is placed in FOLLOW(B).

3. If there is a production A ⟶ a ß, or a production A ⟶ a Bß where FIRST(ß) contains e (i.e., ß ⇒ * e), then everything in FOLLOW(A) is in FOLLOW(B).

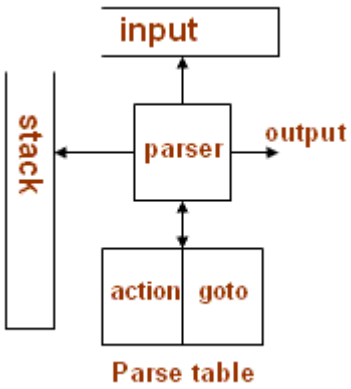
Example
For the expression grammar
E → T E'
E' → +T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

FOLLOW(E)=FOLLOW(E')=[ ),$]
FOLLOW(T)=FOLLOW(T')=[+,),$]
FOLLOW(F)=[+,*,),$]

Bottom up parsing

Bottom-up parsing is a more powerful parsing technique. It is capable of handling almost all the programming languages. . It can fastly handle left recursion in the grammar. . It allows better error recovery by detecting errors as soon as possible.

LR parsing



Actions in an LR (shift reduce) parser
. Assume S_i is top of stack and a_i is current input symbol
. Action [S_i ,a_i] can have four values
1. shift a_i to the stack and goto state S_j
2. reduce by a rule
3. Accept
4. error
Example
Consider the grammer and parsing table for this grammer and find parsing action shown below
E → E + T | T
T → T * F | F
F → (E) | id

State	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parsing action for id + id * id

Stack	Input	Action
0	id+id*id$	shift 5
0 id 5	+id*id$	reduce by F→id
0 F 3	+id*id$	reduce by T→F
0 T 2	+id*id$	reduce by E→T
0 E 1	+id*id$	shift 6
0 E 1 + 6	id*id$	shift 5
0 E 1 + 6 id 5	*id$	reduce by F→id
0 E 1 + 6 F 3	*id$	reduce by T→F
0 E 1 + 6 T 9	*id$	shift 7
0 E 1 + 6 T 9 * 7	id$	shift 5
0 E 1 + 6 T 9 * 7 id 5	$	reduce by F→id
0 E 1 + 6 T 9 * 7 F 10	$	reduce by T→T*F
0 E 1 + 6 T 9	$	reduce by E→E+T
0 E 1	$	ACCEPT

Constructing parse table

Augment the grammar

. G is a grammar with start symbol S

. The augmented grammar G' for G has a new start symbol S' and an additional production S' ⟶ S

LR(0) items

. An LR(0) item of a grammar G is a production of G with a special symbol "." at some position of the right side

. Thus production A ⟶ XYZ gives four LR(0) items

A ⟶ .XYZ

A → X.YZ

A → XY.Z

Start state

. Start state of DFA is an empty stack corresponding to S' → .S item

- This means no input has been seen

- The parser expects to see a string derived from S

Closure operation

1. Initially, every item in I is added to closure (I).

2. If A → α .B ϒis in closure(I) and B → **γ** is a production then add the item B → . **γ** to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Example

Consider the grammar
E' → E
E → E + T | T
T → T * F | F
F → (E) | id
If I is { E' → .E } then closure(I) is
E' → .E
E → .E + T
E → .T
T → .T * F
T → .F
F → .id
F → .(E)

Consider the example described here. Here I contains the LR(0) item E' → .E . We seek further input which can be reduced to E. Now, we will add all the productions with E on the LHS. Here, such productions are E → .E+T and E → .T. Considering these two productions, we will need to add more productions which can reduce the input to E and T respectively. Since we have already added the productions for E, we will need those for T. Here these will be T → .T+F and T → .F. Now we will have to add productions for F, viz. F → .id and F → .(E).

Goto operation

Goto(I,X) , where I is a set of items and X is a grammar symbol,

- is closure of set of item A → α X. ϒ

- such that A → α .X ϒ is in I

. Intuitively if I is a set of items for some valid prefix a then goto(I,X) is set of valid items for prefix α X

. If I is { E' ⟶ E. , E ⟶ E. + T } then goto(I,+) is

E ⟶ E + .T

T ⟶ .T * F

T ⟶ .F

F ⟶ .(E)

F ⟶ .id

The second useful function is Goto(I,X) where I is a set of items and X is a grammar symbol. Goto(I,X) is defined to be the closure of the set of all items [A ⟶ a X. T] such that [A ⟶ a .X T] is in I. Intuitively, if I is set of items that are valid for some viable prefix a , then goto(I,X) is set of items that are valid for the viable prefix a X. Consider the following example: If I is the set of two items { E ' ⟶ E. , E ⟶ E. + T }, then goto(I,+) consists of

E ⟶ E + .T

T ⟶ .T * F

T ⟶ .F

F ⟶ .(E)

F ⟶ .id

We computed goto(I,+) by examining I for items with + immediately to the right of the dot. E' ⟶ E. is not such an item, but E ⟶ E. + T is. We moved the dot over the + to get {E ⟶ E + .T} and the took the closure of this set.

Sets of items

C : Collection of sets of LR(0) items for grammar G'
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 ADD goto(I,X) to Cuntil no more additions
We are now ready to give an algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G'; the algorithm is as shown below:
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

Example

Grammar:

E ' → E

E → E+T | T

T → T*F | F

F → (E) | id

I_0 : closure(E ' → .E)

E ' → .E

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

I_1 : goto(I_0 ,E)

E ' → E.

E → E. + T

I_2 : goto(I_0 ,T)

E → T.

T → T. *F

I_3 : goto(I_0 ,F)

T → F.

I_4 : goto(I_0 ,( )

F → (.E)

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

I_5 : goto(I_0 ,id)

F → id.

I_6 : goto(I_1 ,+)

E → E + .T

T → .T * F

T → .F

F → .(E)

F → .id

I_7 : goto(I_2 ,*)

T → T * .F

F → .(E)

F → .id

I_8 : goto(I_4 ,E)

F → (E.)

E → E. + T

goto(I_4 ,T) is I_2

goto(I_4 ,F) is I_3

goto(I_4 ,( ) is I_4

goto(I_4 ,id) is I_5

I_9 : goto(I_6 ,T)

E → E + T.

T → T. * F

goto(I_6 ,F) is I_3

goto(I_6 ,( ) is I_4

goto(I_6 ,id) is I_5

I_{10} : goto(I_7 ,F)

T → T * F.

goto(I_7 ,( ) is I_4

goto(I_7 ,id) is I_5

I_{11} : goto(I_8 ,))

F → (E).

goto(I_8 ,+) is I_6

goto(I_9 ,*) is I_7

Let's take an example here. We have earlier calculated the closure I_0 . Here, notice that we need to calculate goto (I_0 ,E), goto(I_0 ,T), goto(I_0 , F) , goto (I_0 , ( ) and goto(I_0 , id). For calculating goto(I_0 , E), we take all the LR(0) items in I 0 , which expect E as input (i.e. are of the form A → α .E β), and advance ".". Closure is then taken of this set. Here, goto(I 0 , E) will be closure { E ' → E. , E → E.+T }. The closure adds no item to this set, and hence goto(I 0 , E)={ E ' → E. , E → E.+T }.

Construct SLR parse table

. Construct C={I_0 , . , I_n } the collection of sets of LR(0) items
. If A → α .a is in I_i and goto(I_i , a) = I_j then action[i,a] = shift j
. If A → a . is in I_i
then action[i,a] = reduce A → a for all a in follow(A)
. If S' → S. is in I_i then action[i,$] = accept
. If goto(I i ,A) = I_j then goto[i,A]=j for all non terminals A

. All entries not defined are errors

Notes

. This method of parsing is called SLR (Simple LR)

. LR parsers accept LR(k) languages

- L stands for left to right scan of input
- R stands for rightmost derivation
- k stands for number of lookahead token

. SLR is the simplest of the LR parsing methods. SLR is too weak to handle most languages!

. If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous

. All SLR grammars are unambiguous

. Are all unambiguous grammars in SLR?

Example

. Consider following grammar and its SLR parse table:

S ' ⟶ S

S ⟶ L = R

S ⟶ R

L ⟶ *R

L ⟶ id

R ⟶ L

I_0 : S' ⟶ .S

S ⟶ .L=R

S ⟶ .R

L ⟶ .*R

L ⟶ .id

R ⟶ .L

I_1 : goto(I_0 , S)

S' ⟶ S.

I_2 : goto(I_0 , L)

S ⟶ L.=R

R ⟶ L.

Construct rest of the items and the parse table.

Given grammar:

S ⟶ L = R

S ⟶ R

L ⟶ *R

L ⟶ id

R ⟶ L

Augmented grammar:

S ' ⟶ S

S ⟶ L = R

S ⟶ R

L ⟶ *R

L ⟶ id

R ⟶ L

Constructing the set of LR(0) items:

Using the rules for forming the LR(0) items, we can find parser states as follows:

I_0 : closure(S ' ⟶ .S)

S ' ⟶ .S

S ⟶ .L = R

S ⟶ .R

L ⟶ .*R

L ⟶ .id

R ⟶ .L

I_1: goto(I_0 , S)

S ' → S.

I_2 : goto(I_0 , L)

S → L.=R

R → L.

I_3 : goto(I_0 , R)

S → R.

I_4 : goto(I_0 , *)

L → *.R

R → .L

L → .*R

L → .id

I_5 : goto(I_0 , id)

L → id.

I_6 : goto(I_2 , =)

S → L=.R

R → .L

L → .*R

L → .id

I_7 : goto(I_4 , R)

L → *R.

I_8 : goto(I_4 , L)

R → L.

goto(I_4 , *) = I_4

goto(I_4 , id) = I_5

I_9 : goto(I_6 , R) S → L=R.

I_{10} : goto(I_6 , L) R → L.

goto(I_6, *) = I_4

goto(I_6 , id) = I_5

So, the set is C = { I0, I1, I2, ... I10 } SLR parse table for the grammar

	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6,r6			r6			
3				r3			
4		s4	s5			8	7
5	r5			r5			
6		s4	s5			8	9
7	r4			r4			
8	r6			r6			
9				r2			

The table has multiple entries in action[2,=]
Using the rules given for parse table construction and above set of LR(0) items, parse table as shown can be constructed. For example:
Consider [6,*] : I_6 contains L $\longrightarrow$.*R and goto (I_6 , *) = I_4. So [6,*] = shift 4 or s4.
Consider [4,L] : goto(I_4 , L) = I_8. So [4,L] = 8.
Consider [7,=] : I_7 contains L $\longrightarrow$ *R. and ' = ' is in follow(L).
So [7,=] = reduce L $\longrightarrow$ *R or r4 i.e. reduce by 4th rule.
Similarly the other entries can be filled .
Consider the entry for [2,=]
I_2 contains S $\longrightarrow$ L.=R and goto(I_2 , =) = I_6. So [2,=] contains 'shift 6'.
I_2 contains R $\longrightarrow$ L. and '=' is in follow(R).
So [2,=] contains 'reduce 6'. So [2,=] has multiple entries viz. r6 and s6.
There is both a shift and a reduce entry in action[2,=]. Therefore state 2 has a shift-reduce conflict on symbol "=", However, the grammar is not ambiguous.
. Parse id=id assuming reduce action is taken in [2,=]

Stack	**input**	**action**
0	id=id	shift 5
0	=id	reduce by L $\longrightarrow$ id
0 L 2	=id	reduce by R $\longrightarrow$ L
0 R 3	=id	error

. if shift action is taken in [2,=]

Stack	**input**	**action**
0	id=id$	shift 5

0 id 5	=id$	reduce by L → id
0 L 2	=id$	shift 6
0 L 2 = 6	id$	shift 5
0 L 2 = 6 id 5	$	reduce by L → id
0 L 2 = 6 L 8	$	reduce by R → L
0 L 2 = 6 R 9	$	reduce by S → L=R
0 S 1	$	ACCEPT

We can see that [2,=] has multiple entries, one shift and other reduce, which makes the given grammar ambiguous but it is not so. 'id = id' is a valid string in S' as
S ' → S → L=R → L=L → L=id → id=id
but of the given two possible derivations, one of them accepts it if we use shift operation while if we use reduce at the same place, it gives error as in the other derivation.

Canonical LR Parsing

. Carry extra information in the state so that wrong reductions by A → α will be ruled out
. Redefine LR items to include a terminal symbol as a second component (look ahead symbol)
. The general form of the item becomes [A → α . β , a] which is called LR(1) item.
. Item [A → α ., a] calls for reduction only if next input is a. The set of symbols
Canonical LR parsers solve this problem by storing extra information in the state itself. The problem we have with SLR parsers is because it does reduction even for those symbols of follow(A) for which it is invalid. So LR items are redefined to store 1 terminal (look ahead symbol) along with state and thus, the items now are LR(1) items. An LR(1) item has the form : [A→ a . β , a] and reduction is done using this rule only if input is 'a'. Clearly the symbols a's form a subset of follow(A).

Closure(I)

repeat

for each item [A → α .B β, a] in I

for each production B → γ in G'

and for each terminal b in First(βa)

add item [B → . γ , b] to I

until no more additions to I

To find closure for Canonical LR parsers:

Example
Consider the following grammar
S' → S
S → CC
C → cC | d
Compute closure(I) where I={[S' → .S, $]}

S' → .S, $

S → .CC, $

C → .cC, c

C → .cC, d

C → .d, c

C → .d, d

For the given grammar:
S ' → S
S → CC
C → cC | d
I : closure([S ' → S, $])

S '→ .S	$	as first(e $) = {$}
S → .CC	$	as first(C$) = first(C) = {c, d}
C → .cC	c	as first(Cc) = first(C) = {c, d}
C → .cC	d	as first(Cd) = first(C) = {c, d}
C → .d	c	as first(e c) = {c}
C → .d	d	as first(e d) = {d}

Example

Construct sets of LR(1) items for the grammar on previous slide

I_0 :	S ' ⟶ .S,	$
	S ⟶ .CC,	$
	C ⟶ .cC,	c /d
	C ⟶ .d,	c /d
I_1 :	goto(I_0 ,S)	
	S ' ⟶ S.,	$
I_2 :	goto(I 0 ,C)	
	S ⟶ C.C,	$
	C ⟶ .cC,	$
	C ⟶ .d,	$
I_3 :	goto(I 0 ,c)	
	C ⟶ c.C,	c/d
	C ⟶ .cC,	c/d
	C ⟶ .d,	c/d
I_4 :	goto(I 0 ,d)	
	C ⟶ d.,	c/d
I_5 :	goto(I 2 ,C)	
	S ⟶ CC.,	$
I_6 :	goto(I 2 ,c)	
	C ⟶ c.C,	$
	C ⟶ .cC,	$
	C ⟶ .d,	$
I_7 :	goto(I 2 ,d)	

C ⟶ d., $

I_8 : goto(I 3 ,C)

C ⟶ cC., c/d

I_9 : goto(I 6 ,C)

C ⟶ cC., $

To construct sets of LR(1) items for the grammar given in previous slide we will begin by computing closure of {[S ´ ⟶ .S, $]}.

To compute closure we use the function given previously.

In this case α = ε , B = S, ß =ε and a=$. So add item [S ⟶ .CC, $].

Now first(C$) contains c and d so we add following items

we have A=S, α = ε , B = C, ß=C and a=$

Now first(C$) = first(C) contains c and d

so we add the items [C ⟶ .cC, c], [C ⟶ .cC, d], [C ⟶ .dC, c], [C ⟶ .dC, d].

Similarly we use this function and construct all sets of LR(1) items.

Construction of Canonical LR parse table

. Construct C={I_0 , . , I_n } the sets of LR(1) items.
. If [A ⟶ α .a β , b] is in I_i and goto(I_i , a)=I_j then action[i,a]=shift j
. If [A ⟶ α ., a] is in I_i then action[i,a] reduce A ⟶ α
. If [S ' ⟶ S., $] is in I_i then action[i,$] = accept
. If goto(I i , A) = I_j then goto[i,A] = j for all non
We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ ⟶ S., $] ε I_1 .

Parse 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		

We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ ⟶ S., $] ε I_1

Notes on Canonical LR Parser

. Consider the grammar discussed in the previous two slides. The language specified by the grammar is c*dc*d.

. When reading input cc.dcc.d the parser shifts cs into stack and then goes into state 4 after reading d. It then calls for reduction by C ⟶ d if following symbol is c or d.

. IF $ follows the first d then input string is c*d which is not in the language; parser declares an error

. On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error

. **Problem** : Canonical LR parse table has a large number of states

An LR parser will not make any wrong shift/reduce unless there is an error. But the number of states in LR parse table is too large. To reduce number of states we will combine all states which have same core and different look ahead symbol.

LALR Parse table

. Look Ahead LR parsers

. Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items

I_4 : C ⟶ d. , c/d I_7 : C ⟶ d., $

. Replace I_4 and I_7 by a new state I_{47} consisting of (C ⟶ d., c/d/$)

. Similarly I_3 & I_6 and I_8 & I_9 form pairs

. Merge LR(1) items having the same core

We will combine Ii and Ij to construct new Iij if Ii and Ij have the same core and the difference is only in look ahead symbols. After merging the sets of LR(1) items for previous example will be as follows:

I_0 : S' ⟶ S $

S ⟶ .CC $

C ⟶ .cC c/d

C ⟶ .d c/d

I_1 : goto(I 0 ,S)
S' → S. $
I_2 : goto(I 0 ,C)
S → C.C $
C → .cC $
C → .d $
I_{36} : goto(I 2 ,c)
C → c.C c/d/$
C → .cC c/d/$
C → .d c/d/$
I_4 : goto(I 0 ,d)
C → d. c/d
I_5 : goto(I 2 ,C)
S → CC. $
I_7 : goto(I 2 ,d)
C → d. $
I_{89} : goto(I 36 ,C)
C → cC. c/d/$

Construct LALR parse table

. Construct C={I_0 , .. ,I_n } set of LR(1) items
. For each core present in LR(1) items find all sets having the same core and replace these sets by their union
. Let C' = {J_0 , .. .,J_m } be the resulting set of items
. Construct action table as was done earlier
. Let J = I_1 U I_2 .. .U I_k
since I_1 , I_2 .. ., I_k have same core, goto(J,X) will have he same core
Let K=goto(I_1 ,X) U goto(I_2 ,X) .. goto(I_k ,X) the goto(J,X)=K
The construction rules for LALR parse table are similar to construction of LR(1) parse table.

LALR parse table .

State	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

The construction rules for LALR parse table are similar to construction of LR(1) parse table.

Notes on LALR parse table

. Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.
. SLR and LALR parse tables have same number of states.

Semantic analysis
In semantic analysis the information is stored as attributes of the nodes of the abstract syntax tr ee. The values of those attributes are calculated by semantic rule.

There are two ways for writing attributes:

1) **Syntax Directed Definition** : It is a high level specification in which implementation details are hidden, e.g., $$ = $1 + $2; /* does not give any implementation details. It just tells us.
2) **Translation scheme** : Sometimes we want to control the way the attributes are evaluated, the order and place where they are evaluated. This is of a slightly lower level.
Conceptually both:
- parse input token stream
- build parse tree
- traverse the parse tree to evaluate the semantic rules at the parse tree nodes

. Evaluation may:
- generate code
- save information in the symbol table
- issue error messages
- perform any other activity

To avoid repeated traversal of the parse tree, actions are taken simultaneously when a token is found. So calculation of attributes goes along with the construction of the parse tree.

Attributes
. attributes fall into two classes: *synthesized* and *inherited*
. value of a synthesized attribute is computed from the values of its children nodes
. value of an inherited attribute is computed from the sibling and parent nodes

Synthesized Attributes
A syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

S-attributed grammars are a class of attribute grammars, comparable with L-attributed grammars but characterized by having no inherited attributes at all. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis, pose a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created ***after*** creation of all of their children. Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing .

Syntax Directed Definitions for a desk calculator program

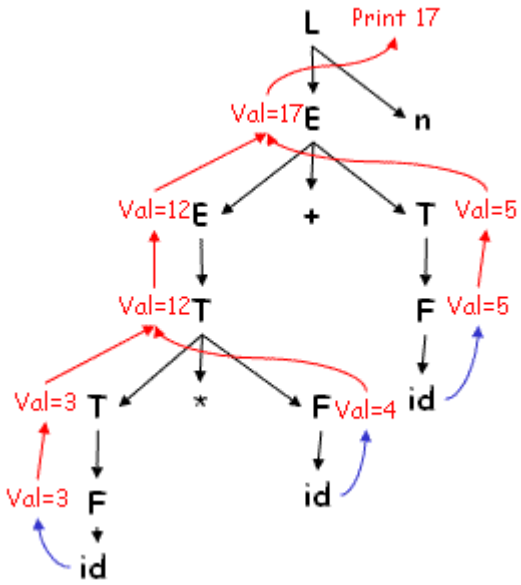
L ⟶ E n	Print (E.val)
E ⟶ E + T	E.val = E.val + T.val
E ⟶ T	E.val = T.val
T ⟶ T * F	T.val = T.val * F.val
T ⟶ F	T.val = F.val
F ⟶ (E)	F.val = E.val
F ⟶ digit	F.val = digit.lexval

. terminals are assumed to have only synthesized attribute values of which are supplied by lexical analyzer

. start symbol does not have any inherited attribute

This is a grammar which uses only synthesized attributes. Start symbol has no parents, hence no inherited attributes.

Parse tree for 3 * 4 + 5 n



Using the previous attribute grammar calculations have been worked out here for 3 * 4 + 5 n. Bottom up parsing has been done.

Inherited Attributes

. an inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings

. Used for finding out the context in which it appears

. possible to use only S-attributes but more natural to use inherited attributes

D ⟶ T L	L.in = T.type
T ⟶ real	T.type = real
T ⟶ int	T.type = int
L ⟶ L_1 , id	L_1 .in = L.in; addtype(id.entry, L.in)
L ⟶ id	addtype (id.entry,L.in)

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

Dependency Graph : Directed graph indicating interdependencies among the synthesized and inherited attributes of various nodes in a parse tree.

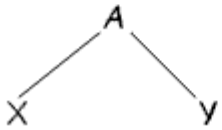
Algorithm to construct dependency graph
for each node **n** in the parse tree do
for each attribute **a** of the grammar symbol do
construct a node in the dependency graph
for **a**
for each node n in the parse tree do
for each semantic rule b = f (c_1 , c_2 , ..., c_k) do
{ associated with production at n }
for i = 1 to k do
construct an edge from c_i to b

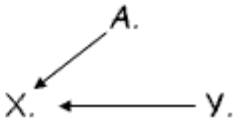
An algorithm to construct the dependency graph. After making one node for every attribute of all the nodes of the parse tree, make one edge from each of the other attributes on which it depends.

Example

- **Suppose A.a = f(X.x , Y.y) is a semantic rule for A → X Y**

- **If production A → X Y has the semantic rule X.x = g(A.a, Y.y)**





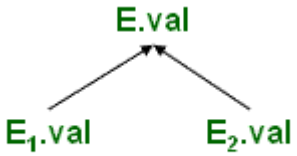
The semantic rule **A.a = f(X.x , Y.y)** for the production A -> XY defines the synthesized attribute **a** of **A** to be dependent on the attribute **x** of **X** and the attribute **y** of **Y** . Thus the dependency graph will contain an edge from X.x to A.a and Y.y to A.a accounting for the two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

Example

. Whenever following production is used in a parse tree

E ⟶E 1 + E 2 E.val = E 1 .val + E 2 .val

we create a dependency graph



The synthesized attribute E.val depends on E1.val and E2.val hence the two edges one each from E 1 .val & E 2 .val

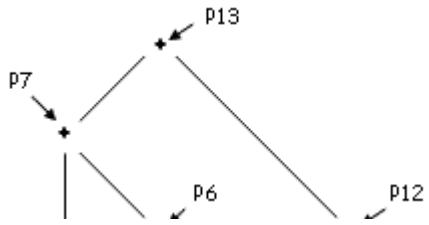
DAG for Expressions

Expression a + a * (b - c) + (b - c) * d make a leaf or node if not present, otherwise return pointer to the existing node

P 1 = makeleaf(id,a)

P 2 = makeleaf(id,a)

P 3 = makeleaf(id,b)



P 4 = makeleaf(id,c)

P 5 = makenode(-,P 3 ,P 4)

P 6 = makenode(*,P 2 ,P 5)

P 7 = makenode(+,P 1 ,P 6)

P 8 = makeleaf(id,b)

P 9 = makeleaf(id,c)

P 10 = makenode(-,P 8 ,P 9)

P 11 = makeleaf(id,d)

P 12 = makenode(*,P 10 ,P 11)

P 13 = makenode(+,P 7 ,P 12)

A directed acyclic graph (DAG) for the expression : a + a * (b V c) + (b V c) * d All the function calls are made as in the order shown. Whenever the required node is already present, a pointer to it is returned so that a pointer to the old node itself is obtained. A new node is made if it did not exist before. The function calls can be explained as:

P1 = makeleaf(id,a)

A new node for identifier Qa R made and pointer P1 pointing to it is returned.

P2 = makeleaf(id,a)

Node for Qa R already exists so a pointer to that node i.e. P1 returned.

P3 = makeleaf(id,b)

A new node for identifier Qb R made and pointer P3 pointing to it is returned.

P4 = makeleaf(id,c)

A new node for identifier Qc R made and pointer P4 pointing to it is returned.

P5 = makenode(-,P3,P4)

A new node for operator Q- R made and pointer P5 pointing to it is returned. This node becomes the parent of P3,P4.

P6 = makenode(*,P2,P5)

A new node for operator Q- R made and pointer P6 pointing to it is returned. This node becomes the parent of P2,P5.

P7 = makenode(+,P1,P6)

A new node for operator Q+ R made and pointer P7 pointing to it is returned. This node becomes the parent of P1,P6.

P8 = makeleaf(id,b)

Node for Qb R already exists so a pointer to that node i.e. P3 returned.

P9 = makeleaf(id,c)

Node for Qc R already exists so a pointer to that node i.e. P4 returned.

P10 = makenode(-,P8,P9)

A new node for operator Q- R made and pointer P10 pointing to it is returned. This node becomes the parent of P8,P9.

P11 = makeleaf(id,d)

A new node for identifier Qd R made and pointer P11 pointing to it is returned.

P12 = makenode(*,P10,P11)

A new node for operator Q* R made and pointer P12 pointing to it is returned. This node becomes the parent of P10,P11.

P13 = makenode(+,P7,P12)

A new node for operator Q+ R made and pointer P13 pointing to it is returned. This node becomes the parent of P7, P12.

L-attributed definitions

. When translation takes place during parsing, order of evaluation is linked to the order in which nodes are created

. A natural order in both top-down and bottom-up parsing is depth first-order

. L-attributed definition: where attributes can be evaluated in depth-first order

L-attributed definitions are a class of syntax-directed definitions where attributes can always be evaluated in depth first order. (L is for left as attribute information appears to flow from left to right). Even if the parse tree is not actually constructed, it is useful to study translation during parsing by considering depth-first evaluation of attributes at the nodes of a parse tree.

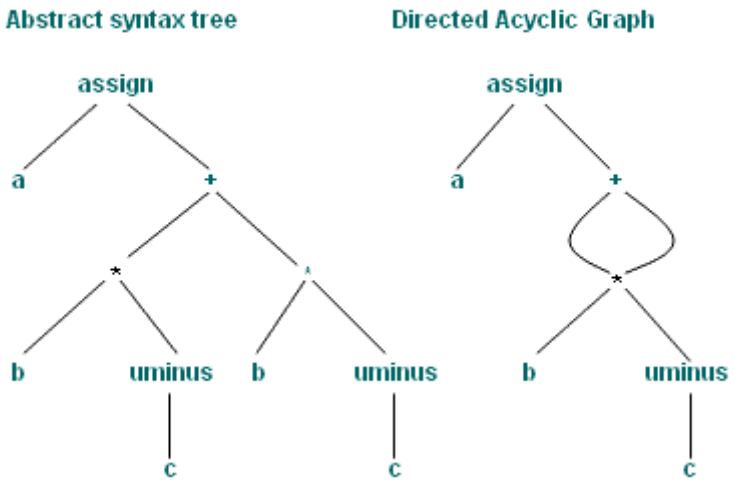
Abstract Syntax Tree/DAG

. Condensed form of a parse tree
. useful for representing language constructs
. Depicts the natural hierarchical structure of the source program
- Each internal node represents an operator
- Children of the nodes represent operands
- Leaf nodes represent operands
. DAG is more compact than abstract syntax tree because common sub expressions are eliminated

A syntax tree depicts the natural hierarchical structure of a source program. Its structure has already been discussed in earlier lectures.

DAGs are generated as a combination of trees: operands that are being reused are linked together, and nodes may be annotated with variable names (to denote assignments). This way, DAGs are highly compact, since they eliminate local common sub-expressions. On the other hand, they are not so easy to optimize, since they are more specific tree forms. However, it can be seen that proper building of DAG for a given sequence of instructions can compactly represent the outcome of the calculation.An example of a syntax tree and DAG has been given in the next slide .

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



You can see that the node " * " comes only once in the DAG as well as the leaf " b ", but the meaning conveyed by both the representations (AST as well as the DAG) remains the same.

Inter mediate codes

4 types intermediate codes are there
1.postfix notation
2.syntax trees
3.quadruples
4,triples

Postfix notation

. No parenthesis are needed in postfix notation because
- the position and parity of the operators permit only one decoding of a postfix expression
. Postfix notation for
a = b * -c + b * - c
is
a b c - * b c - * + =

Three address code

. It is a sequence of statements of the general form X := Y op Z where
- X, Y or 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

Three address code is a sequence of statements of the general form: x := y 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 or boolean - valued data. Compilers use this form in their IR.

Three address code .

. Only one operator on the right hand side is allowed
. Source expression like x + y * z might be translated into t 1 := y * z t 2 := x + t 1 where t 1 and t 2 are compiler generated temporary names
. Unraveling of complicated arithmetic expressions and of control flow makes 3-address code desirable for code generation and optimization
. The use of names for intermediate values allows 3-address code to be easily rearranged
. Three address code is a linearized representation of a syntax tree where explicit names correspond to the interior nodes of the graph

Three address instructions

. Assignment

- x = y op z

- x = op y

- x = y

. Jump

- goto L

- if x relop y goto L

. Function

- param x

- call p,n

- return y

. Pointer

- x = &y

.Indexed assignment - x = *y

- x = y[i] - *x = y

- x[i] = y

The various types of the three-address codes. Statements can have symbolic label and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding intermediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using back patching.

Intermediate Code Generation

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. We can think of this IR as a program for an abstract machine. This IR should have two important properties: It should be easy to produce and it should be easy to translate into target program. IR should have the abstraction in between of the abstraction at the source level (identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)) and of the abstraction at the target level (memory locations, registers, stack, opcodes, addressing modes, system libraries and interface to the operating systems). Therefore IR is an intermediate stage of the mapping from source level abstractions to target machine abstractions.

Intermediate Code Generation ...

. Front end translates a source program into an intermediate representation

. Back end generates target code from intermediate representation

. Benefits

- Retargeting is possible
- Machine independent code optimization is possible

In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code. Details of the target language are confined to the back end, as far as possible. Although a source program can be translated directly into the target language, some benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated: a compiler for a different machine can be created by attaching a back-end for the new machine to an existing front-end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Syntax directed translation of expression into 3-address code

$S \rightarrow id := E$	S.code = E.code ||
	gen(id.place:= E.place)
$E \rightarrow E_1 + E_2$	E.place:= newtmp
	$E.code := E_1.code \|\| E_2.code \|\|$
	$gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	E.place:= newtmp
	$E.code := E_1.code \|\| E_2.code \|\|$
	$gen(E.place := E_1.place * E_2.place)$

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

X := y *op* z

Where x, y and z are names, constants, or compiler generated temporaries. op stands for any operator, such as fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. Note that no built up arithmetic expression 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

t1 := y * z

t2 := x + t1

where t1 and t2 are compiler-generated temporary names. This unraveling of complicated arithmetic expression 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 allows three-address code to be easily rearranged unlike postfix notation. We can easily generate code for the three-address code given above. The S-attributed definition above generates three-address code for assigning statements. The synthesized attribute *S.code* represents the three-address code for the assignment S. The nonterminal E has two attributes:

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

. *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.

Syntax directed translation of expression .

$E \rightarrow -E_1$	E.place := newtmp
	E.code := E_1 .code ||
	gen(E.place := - E_1 .place)
$E \rightarrow (E_1)$	E.place := E_1 .place
	E.code := E_1 .code
$E \rightarrow id$	E.place := id.place
	E.code := ' '

.

Example for Numerical representation

. a or b and not c
t_1 = not c
t_2 = b and t_1
t_3 = a or t_2
. relational expression a < b is equivalent to if a < b then 1 else 0
1. if a < b goto 4.
2. t = 0
3. goto 5
4. t = 1
5.

Consider the implementation of Boolean expressions using 1 to denote true and 0 to denote false. Expressions are evaluated in a manner similar to arithmetic expressions.

For example, the three address code for a or b and not c is:

t1 = not c

t2 = b and t1

t3 = a or t2

Syntax directed translation of boolean expressions

$E \rightarrow E_1$ or E_2
E.place := newtmp
emit(E.place ':=' E_1 .place 'or' E_2 .place)
$E \rightarrow E_1$ and E_2
E.place:= newtmp
emit(E.place ':=' E_1 .place 'and' E_2 .place)
$E \rightarrow$ not E_1
E.place := newtmp

emit(E.place ':=' 'not' E_1 .place)
E $\longrightarrow$ (E_1) E.place = E_1 .place

Example of 3-address code
Code for a < b or c < d and e < f
if a < b goto Ltrue
goto L1
L1: if c < d goto L2
goto Lfalse
L2: if e < f goto Ltrue
goto Lfalse
Ltrue:
Lfalse:
Code for a < b or c < d and e < f
It is equivalent to a<b or (c<d and e<f) by precedence of operators.
Code:
if a < b goto L.true
goto L1
L1 : if c < d goto L2
goto L.false
L2 : if e < f goto L.true
goto L.false
where L.true and L.false are the true and false exits for the entire expression.
(The code generated is not optimal as the second statement can be eliminated without changing the value of the code).

Example .

Code for

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

```
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:    t1 = Y - Z
      X= t1
      goto LI
Lnext:
```

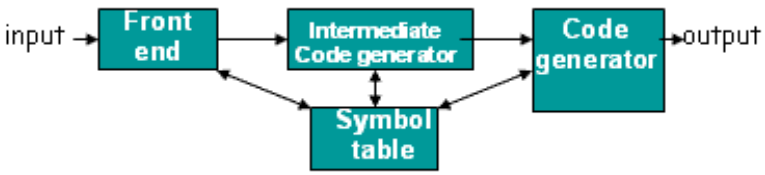
Code for while a < b do
if c < d then
x = y + z
else

x = y - z
L1 : if a < b goto L2 //no jump to L2 if a>=b. next instruction causes jump outside the loop
goto L.next
L2 : if c < d goto L3
goto L4
L3 : t1 = Y + Z
X= t1
goto L1 //return to the expression code for the while loop
L4 : t1 = Y - Z
X= t1
goto L1 //return to the expression code for the while loop
L.next:
Here too the first two goto statements can be eliminated by changing the direction of the tests (by translating a relational expression of the form id1 < id2 into the statement if id1 id2 goto E.false).

Case Statement
. switch expression
begin
case value: statement
case value: statement
..
case value: statement
default: statement
end
.evaluate the expression
. find which value in the list of cases is the same as the value of the expression
. - Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression
. execute the statement associated with the value found
.

Code Generation

Code generation and Instruction Selection



.
. output code must be correct
. output code must be of high quality
. code generator should run efficiently

As we see that the final phase in any compiler is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program, as shown in the figure. Optimization phase is optional as far as compiler's correct working is considered. In order to have a good compiler following conditions should hold:

1. Output code must be correct: The meaning of the source and the target program must remain the same i.e., given an input, we should get same output both from the target and from the source program. We have no definite way to ensure this condition. What all we can do is to maintain a test suite and check.

2. Output code must be of high quality: The target code should make effective use of the resources of the target machine.

3. Code generator must run efficiently: It is also of no use if code generator itself takes hours or minutes to convert a small piece of code.

Issues in the design of code generator

. Input: Intermediate representation with symbol table assume that input has been validated by the front end

. target programs :

- absolute machine language fast for small programs
- relocatable machine code requires linker and loader
- assembly code requires assembler, linker, and loader

Let us examine the generic issues in the design of code generators.

1. 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 the information in the symbol table that is used to determine the runtime addresses of the data objects denoted by the names in the intermediate representation. We assume that prior to code generation the input has been validated by the front end i.e., type checking, syntax, semantics etc. have been taken care of. The code generation phase can therefore proceed on the assumption that the input is free of errors.

2. Target programs: The output of the code generator is the target program. This output may take a variety of forms; absolute machine language, relocatable machine language, or assembly language.

. Producing an absolute machine language as output has the advantage that it can be placed in a fixed location in memory and immediately executed. A small program can be thus compiled and executed quickly.

. Producing a relocatable machine code as output allows subprograms to be compiled separately. 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.

. Producing an assembly code as output makes the process of code generation easier as we can generate symbolic instructions. The price paid is the assembling, linking and loading steps after code generation.

Instruction Selection

. Instruction selection
. uniformity
. completeness
. Instruction speed
. Register allocation Instructions with register operands are faster
- store long life time and counters in registers
- temporary locations
- Even odd register pairs
. Evaluation order

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. So, the instruction selection depends upon:

1. Instructions used i.e. which instructions should be used in case there are multiple instructions that do the same job.
2. Uniformity i.e. support for different object/data types, what op-codes are applicable on what data types etc.
3. Completeness: Not all source programs can be converted/translated in to machine code for all architectures/machines. E.g., 80x86 doesn't support multiplication.
4. Instruction Speed: This is needed for better performance.
5. Register Allocation:
. Instructions involving registers are usually faster than those involving operands memory.
. Store long life time values that are often used in registers.
6. Evaluation Order: The order in which the instructions will be executed. This increases performance of the code.

Instruction Selection

. straight forward code if efficiency is not an issue

a=b+c	Mov b, R_0	
d=a+e	Add c, R_0	
	Mov R_0 , a	
	Mov a, R_0	can be eliminated
	Add e, R_0	
	Mov R_0 , d	
a=a+1	Mov a, R_0	Inc a
	Add #1, R_0	
	Mov R_0, a	

Here is an example of use of instruction selection: Straight forward code if efficiency is not an issue

a=b+c	**Mov b, R_0**	
d=a+e	**Add c, R_0**	
	Mov R_0 , a	
	Mov a, R_0	**can be eliminated**
	Add e, R_0	
	Mov R_0 , d	
a=a+1	**Mov a, R_0**	**Inc a**
	Add #1, R_0	
	Mov R_0, a	

Here, "Inc a" takes lesser time as compared to the other set of instructions as others take almost 3 cycles for each instruction but "Inc a" takes only one cycle. Therefore, we should use "Inc a" instruction in place of the other set of instructions.

Target Machine

. Byte addressable with 4 bytes per word

. It has n registers R_0 , R_1 , ..., R_{n-1}

. Two address instructions of the form opcode source, destination

. Usual opcodes like move, add, sub etc.

. Addressing modes

MODE	FORM	ADDRESS
Absolute	M	M
register	R	R
index	c(R)	c+cont(R)
indirect register	*R	cont(R)

indirect index	*c(R)	cont(c+cont(R))
literal	#c	c

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Our target computer is a byte addressable machine with four bytes to a word and n general purpose registers, R_0 , R_1 ,..R_{n-1} . It has two address instructions of the form

op source, destination

In which op is an op-code, and source and destination are data fields. It has the following op-codes among others:

MOV (move source to destination)

ADD (add source to destination)

SUB (subtract source from destination)

The source and destination fields are not long enough to hold memory addresses, so certain bit patterns in these fields specify that words following an instruction contain operands and/or addresses. The address modes together with their assembly-language forms are shown above.

Basic blocks

. sequence of statements in which flow of control enters at the beginning and leaves at the end
. Algorithm to identify basic blocks
. determine leader
- first statement is a leader
- any target of a goto statement is a leader
- any statement that follows a goto statement is a leader
. for each leader its basic block consists of the leader and all statements up to next leader
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 algorithm can be used to partition a sequence of three-address statements into basic blocks:
1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are the following:
. 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 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.

Flow graphs

. add control flow information to basic blocks
. nodes are the basic blocks
. there is a directed edge from B_1 to B_2 if B_2 can follow B_1 in some execution sequence

- there is a jump from the last statement of B_1 to the first statement of B_2
- B_2 follows B_1 in natural order of execution
. initial node: block with first statement as leader

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

. There is conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 , or

. B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump. We say that B_1 is the predecessor of B_2 , and B_2 is a successor of B_1 .

Next use information

. for register and temporary allocation
. remove variables from registers if not used
. statement X = Y op Z defines X and uses Y and Z
. scan each basic blocks backwards
. assume all temporaries are dead on exit and all user variables are live on exit

The use of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j uses the value of x computed at i. We wish to determine for each three-address statement x := y op z what the next uses of x, y and z are. We collect next-use information about names in basic blocks. If the name in a register is no longer needed, then the register can be assigned to some other name. This idea of keeping a name in storage only if it will be used subsequently can be applied in a number of contexts. It is used to assign space for attribute values. The simple code generator applies it to register assignment. Our algorithm is to determine next uses makes a backward pass over each basic block, recording (in the symbol table) for each name x whether x has a next use in the block and if not, whether it is live on exit from that block. We can assume that all non-temporary variables are live on exit and all temporary variables are dead on exit.

Algorithm to compute next use information

. Suppose we are scanning i : X := Y op Z in backward scan
- attach to i, information in symbol table about X, Y, Z
- set X to not live and no next use in symbol table
- set Y and Z to be live and next use in i in symbol table

As an application, we consider the assignment of storage for temporary names. Suppose we reach three-address statement i: x := y op z in our backward scan. We then do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and live ness of x, y and z.

2. In the symbol table, set x to "not live" and "no next use".

3. In the symbol table, set y and z to "live" and the next uses of y and z to i. Note that the order of steps (2) and (3) may not be interchanged because x may be y or z.

If three-address statement i is of the form x := y or x := op y, the steps are the same as above, ignoring z.

Example

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

For example, consider the basic block shown above

Example

STATEMENT

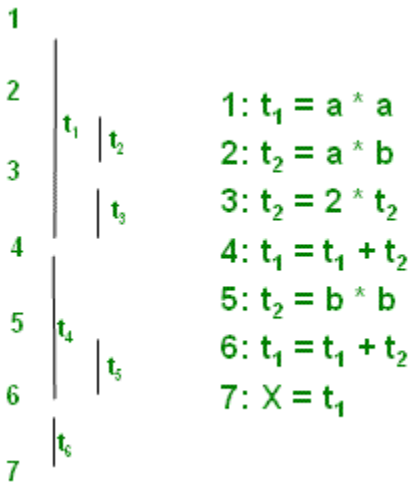
7: no temporary is live
6: t_6:use(7), t_4 t_5 not live
5: t_5:use(6)
4: t_4:use(6), t_1 t_3 not live
3: t_3:use(4), t_2 not live
2: t_2:use(3)
1: t_1:use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

We can allocate storage locations for temporaries by examining each in turn and assigning a temporary to the first location in the field for temporaries that does not contain a live temporary. If a temporary cannot be assigned to any previously created location, add a new location to the data area for the current procedure. In many cases, temporaries can be packed into registers rather than memory locations, as in the next section.

Example .



The six temporaries in the basic block can be packed into two locations. These locations correspond to t 1 and t 2 in:

1: $t_1 = a * a$

2: $t_2 = a * b$
3: $t_2 = 2 * t_2$
4: $t_1 = t_1 + t_2$
5: $t_2 = b * b$
6: $t_1 = t_1 + t_2$
7: $X = t_1$

Code Generator

. consider each statement
. remember if operand is in a register
. **Register descriptor**
- Keep track of what is currently in each register.
- Initially all the registers are empty

. **Address descriptor**
- Keep track of location where current value of the name can be found at runtime
- The location might be a register, stack, memory address or a set of those

The code generator generates target code for a sequence of three-address statement. It considers each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact, if possible. The code-generation uses descriptors to keep track of register contents and addresses for names.

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

2. An address descriptor keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these, since when copied, a value also stays where it was. This information can be stored in the symbol table and is used to determine the accessing method for a name.

Code Generation Algorithm

for each X = Y op Z do

. invoke a function getreg to determine location L where X must be stored. Usually L is a register.
. Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate

Mov Y', L

. Generate

op Z', L

Again prefer a register for Z. Update address descriptor of X to indicate X is in L. If L is a register update its descriptor to indicate that it contains X and remove X from all other register descriptors.
. If current value of Y and/or Z have no next use and are dead on exit from block and are in registers, change register descriptor to indicate that they no longer contain Y and/or Z.

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

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L will usually be a register, but it could also be a memory location. We shall describe getreg shortly.
2. Consult the address descriptor for u to determine y', (one of) the current location(s) of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of u is not already in L, generate the instruction MOV y', L to place a copy of y in L.
3. Generate the instruction OP z', L where z' is a current location of z. Again, prefer a register to a memory location if z is in both. Update the address descriptor to indicate that x is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.
4. If the current values of y and/or y have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x := y op z, those registers no longer will contain y and/or z, respectively.

Function getreg

1. If Y is in register (that holds no other values) and Y is not live and has no next use after

X = Y op Z

then return register of Y for L.

2. Failing (1) return an empty register
3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.
4. else select memory location X as L

The function getreg returns the location L to hold the value of x for the assignment x := y op z.

1. If the name y is in a register that holds the value of no other names (recall that copy instructions such as x := y could cause a register to hold the value of two or more variables simultaneously), and y is not live and has no next use after execution of x := y op z, then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there is one.

3. Failing (2), if x has a next use in the block, or op is an operator such as indexing, that requires a register, find an occupied register R. Store the value of R into memory location (by MOV R, M) if it is not already in the proper memory location M, update the address descriptor M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory.

4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

Example

Stmt	code	reg desc	addr desc
t_1 =a-b	mov a,R_0	R_0 contains t_1	t_1 in R_0
	sub b,R_0		
t_2 =a-c	mov a,R_1	R_0 contains t_1	t_1 in R_0
	sub c,R_1	R_1 contains t_2	t_2 in R_1
t_3 =t_1 +t_2	add R_1 ,R_0	R_0contains t_3	t_3 in R_0
		R_1 contains t_2	t_2 in R_1
d=t_3 +t_2	add R_1 ,R_0	R_0contains d	d in R_0
	mov R_0 ,d		d in R_0 and
			memory

For example, the assignment d := (a - b) + (a - c) + (a - c) might be translated into the following three- address code sequence:

t_1 = a - b

t_2 = a - c

t_3 = t_1 + t_2

d = t_3 + t_2

The code generation algorithm that we discussed would produce the code sequence as shown. Shown alongside are the values of the register and address descriptors as code generation progresses.

Conditional Statements

. branch if value of R meets one of six conditions negative, zero, positive, non-negative, non-zero, non-positive

if X < Y goto Z Mov X, R0

Sub Y, R0

Jmp negative Z

. Condition codes: indicate whether last quantity computed or loaded into a location is negative, zero, or positive

Machines implement conditional jumps in one of two ways. One way is to branch if the value of a designated register meets one of the six conditions: negative, zero, positive, non-negative, non-zero, and non-positive. On such a machine a three-address statement such as if x < y goto z can be implemented by subtracting y from x in register R, and then jumping to z if the value in register is negative. A second approach, common to many machines, uses a set of condition codes to indicate whether the last quantity computed or loaded into a register is negative, zero or positive.

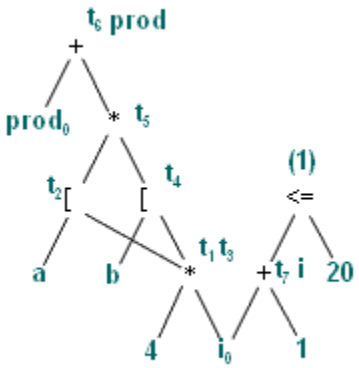
DAG representation of basic blocks

. useful data structures for implementing transformations on basic blocks
. gives a picture of how value computed by a statement is used in subsequent statements
. good way of determining common sub-expressions
. A dag for a basic block has following labels on the nodes
- leaves are labeled by unique identifiers, either variable names or constants
- interior nodes are labeled by an operator symbol
- nodes are also optionally given a sequence of identifiers for labels

DAG (Directed Acyclic Graphs) are useful data structures for implementing transformations on basic blocks. A DAG gives a picture of how the value computed by a statement in a basic block is used in subsequent statements of the block. Constructing a DAG from three-address statements is a good way of determining common sub-expressions (expressions computed more than once) within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block. A DAG for a basic block is a directed cyclic graph with the following labels on nodes: 1. Leaves are labeled by unique identifiers, either variable names or constants. From the operator applied to a name we determine whether the l-value or r-value of a name is needed; most leaves represent r- values. The leaves represent initial values of names, and we subscript them with 0 to avoid confusion with labels denoting "current" values of names as in (3) below. 2. Interior nodes are labeled by an operator symbol. 3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labeling a node are deemed to have that value.

DAG representation: example

1. $t_1 := 4 * i$
2. $t_2 := a[t_1]$
3. $t_3 := 4 * i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2 * t_4$
6. $t_6 := prod + t_5$
7. $prod := t_6$
8. $t_7 := i + 1$
9. $i := t_7$
10. if i <= 20 goto (1)



For example, the slide shows a three-address code. The corresponding DAG is shown. We observe that each node of the DAG represents a formula in terms of the leaves, that is, the values possessed by variables and constants upon entering the block. For example, the node labeled t 4 represents the formula
b[4 * i]
that is, the value of the word whose address is 4*i bytes offset from address b, which is the intended value of t 4 .

Code Generation from DAG

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

We see how to generate code for a basic block from its DAG representation. The advantage of doing so is that from a DAG we can more easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address

statements or quadruples. If the DAG is a tree, we can generate code that we can prove is optimal under such criteria as program length or the fewest number of temporaries used. The algorithm for optimal code generation from a tree is also useful when the intermediate code is a parse tree.

Rearranging order of the code

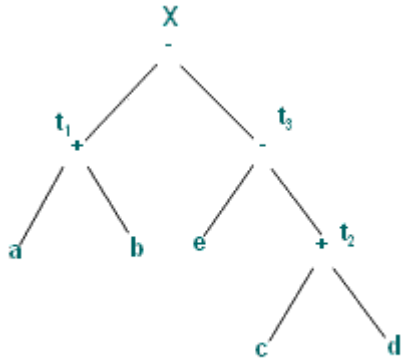
. Consider following basic block

t 1 = a + b

t 2 = c + d

t 3 = e -t 2

X = t 1 -t 3



and its DAG

Here, we briefly consider how the order in which computations are done can affect the cost of resulting object code. Consider the basic block and its corresponding DAG representation as shown in the slide.

Rearranging order .

Three adress code for the DAG (assuming only two registers are available)

MOV a, R_0

ADD b, R_0

MOV c, R_1

ADD d, R_1

MOV R_0 , t_1

Register spilling

Rearranging the code as

t_2 = c + d

t_3 = e -t 2

t_1 = a + b

X = t_1 -t_3

gives

MOV c, R_0

ADD d, R_0

MOV e, R_1

MOV e, R_0		SUB R_0 , R_1
SUB R_1 , R_0		MOV a, R_0
MOV t_1 , R_1	Register reloading	ADD b, R_0
SUB R_0 , R_1		SUB R_1 , R_0
MOV R_1 , X		MOV R_1 , X

If we generate code for the three-address statements using the code generation algorithm described before, we get the code sequence as shown (assuming two registers R0 and R1 are available, and only X is live on exit). On the other hand suppose we rearranged the order of the statements so that the computation of t 1 occurs immediately before that of X as:

t_2 = c + d

t_3 = e -t 2

t_1 = a + b

X = t_1 -t_3

Then, using the code generation algorithm, we get the new code sequence as shown (again only R0 and R1 are available). By performing the computation in this order, we have been able to save two instructions; MOV R0, t 1 (which stores the value of R0 in memory location t 1) and MOV t 1 , R1 (which reloads the value of t 1 in the register R1).

Peephole Optimization

. target code often contains redundant instructions and suboptimal constructs
. examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
. peephole is a small moving window on the target systems

A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. A simple but effective technique for locally improving the target code is peephole optimization, a method for trying to improve 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 be contiguous, although some implementations do require this.

Peephole optimization examples.

Redundant loads and stores

. Consider the code sequence

Move R_0 , a Move a, R_0

. Instruction 2 can always be removed if it does not have a label.

Now, we will give some examples of program transformations that are characteristic of peephole optimization: Redundant loads and stores: If we see the instruction sequence

Move R_0 , a

Move a, R_0

We can delete instruction (2) because whenever (2) is executed, (1) will ensure that the value of a is already in register R0. Note that is (2) has a label, we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Peephole optimization examples.

Unreachable code

- Consider following code sequence

```
#define debug 0
if (debug) {
      print debugging info
}

this may be translated as
      if debug = 1 goto L1
      goto L2
L1: print debugging info
L2:

Eliminate jump over jumps
      if debug <> 1 goto L2
            print debugging information
L2:
```

Another opportunity for peephole optimization is the removal of unreachable instructions.

Unreachable code example .

constant propagation
if 0 <> 1 goto L2
print debugging information
L2:
Evaluate boolean expression. Since if condition is always true the code becomes
goto L2
print debugging information
L2:
The print statement is now unreachable. Therefore, the code becomes
L2:

Peephole optimization examples.

- flow of control: replace jump sequences

```
goto L1                          goto L2
...                                ...
...          by                    ...
L1 : goto L2                     L1: goto L2
```

- Simplify algebraic expressions

remove x := x+0 or x:=x*1

Peephole optimization examples.

. Strength reduction
- Replace X^2 by X*X
- Replace multiplication by left shift
- Replace division by right shift

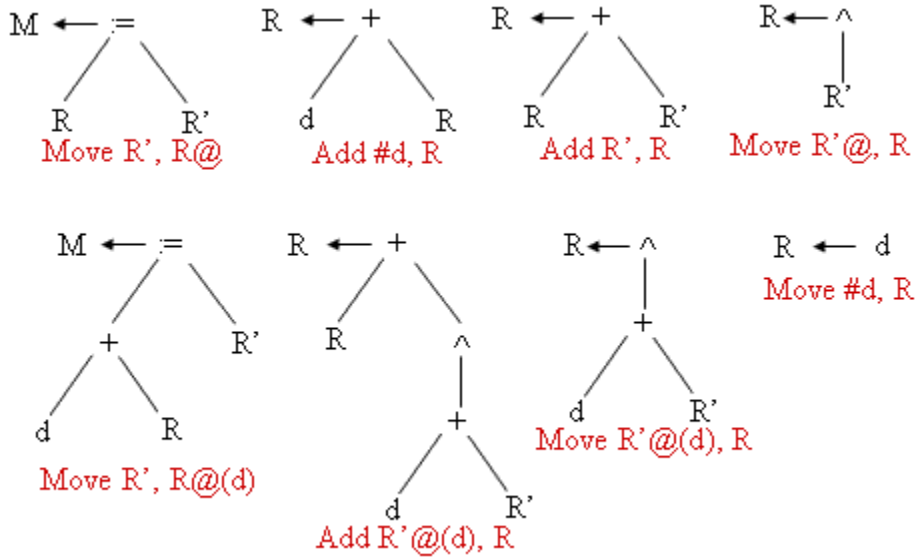
. Use faster machine instructions

replace Add #1,R
by Inc R

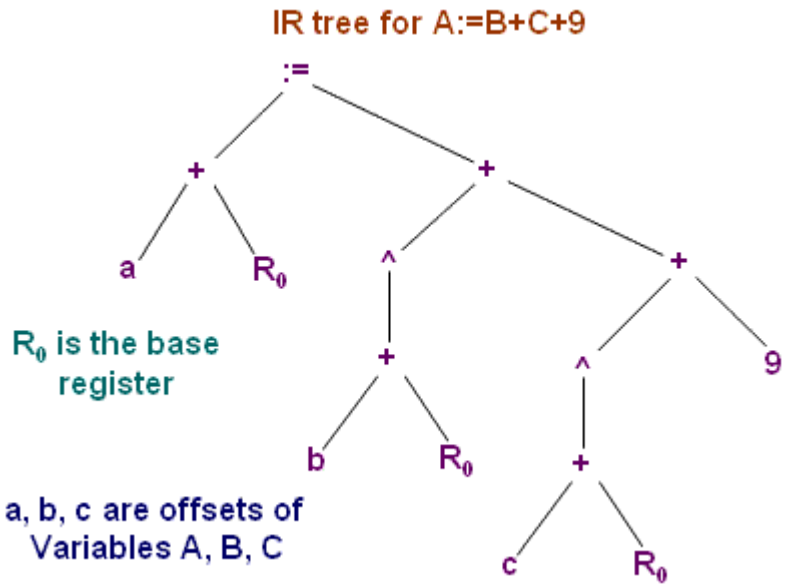
Code Generator Generator

. Code generation by tree rewriting
. target code is generated during a process in which input tree is reduced to a single node
. each rewriting rule is of the form replacement →template { action} where
- replacement is a single node
- template is a tree
- action is a code fragment

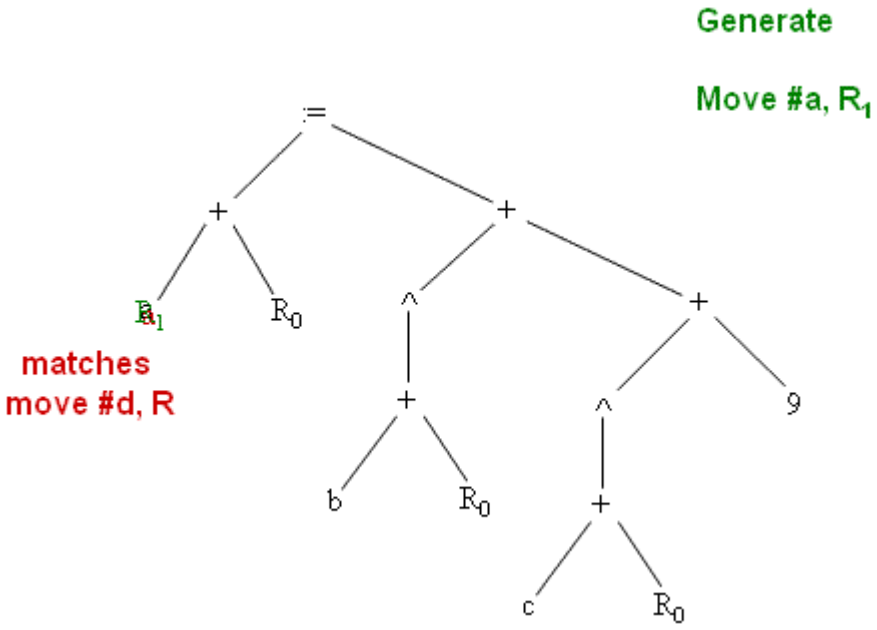
Instruction set for a hypothetical machine



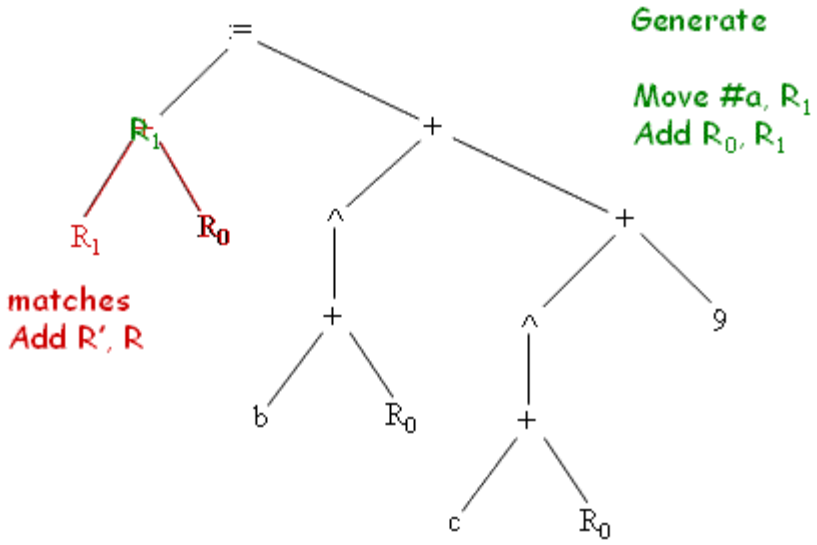
Example



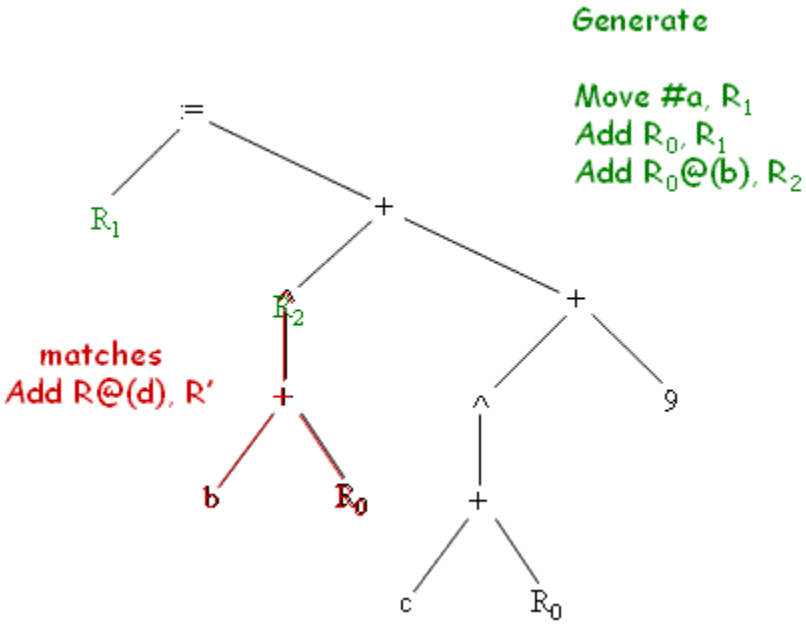
Example .



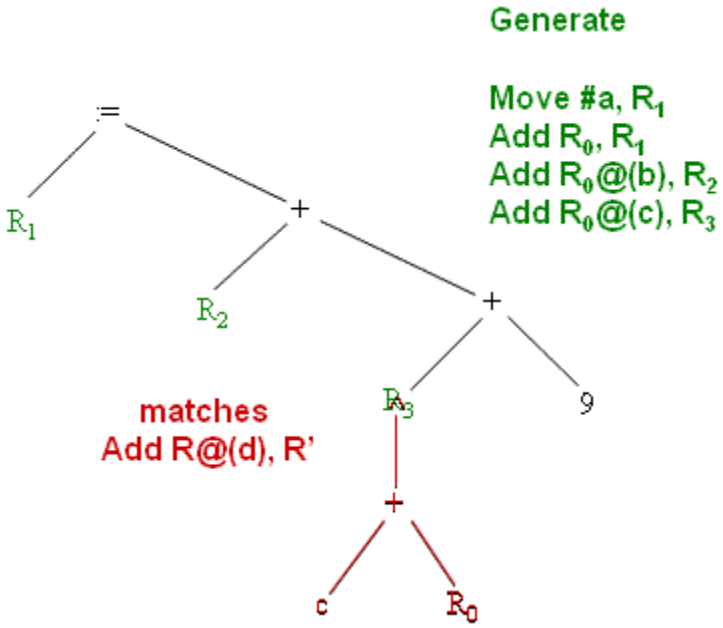
Example .



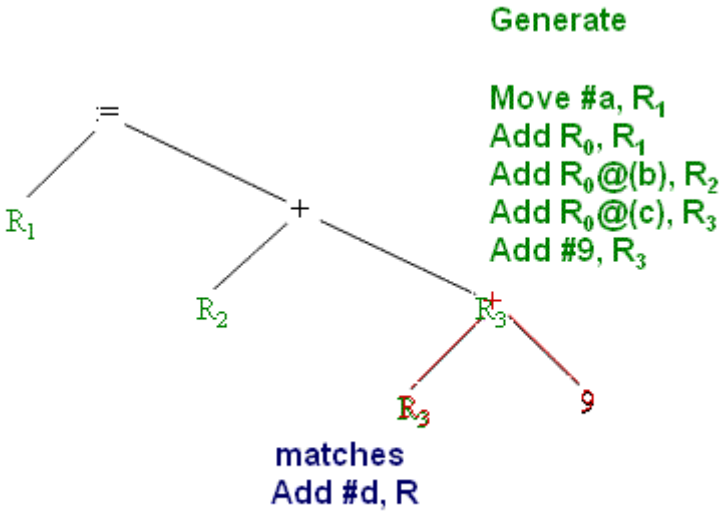
Example .



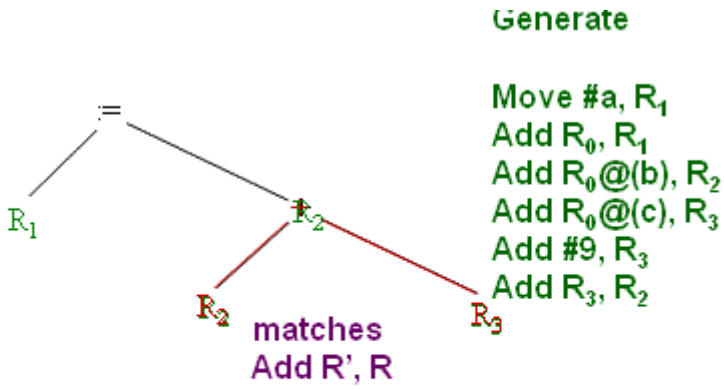
Example .



Example .



Example .



Example .

matches
Add R', R@

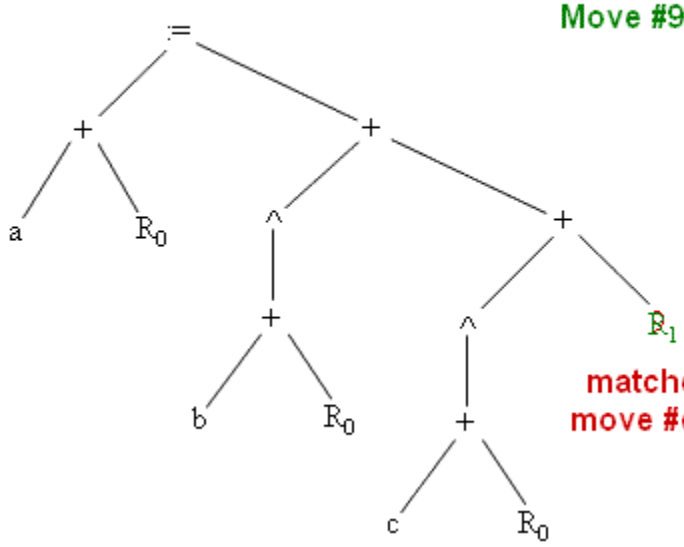
Generate

Move #a, R_1
Add R_0, R_1
Add R_0@(b), R_2
Add R_0@(c), R_3
Add #9, R_3
Add R_3, R_2
Move R_2, R_1@

Example .

Generate

Move #9, R_1



matches
move #d, R