Lecture 3 Jan 17, 2019

Optimization of intermediate code:-

- **Constant folding**:- process of evaluating and recognizing constant expression at compile time rather than computing them at runtime.

  e.g. int2real(60) can be evaluated to 60.0 compile time

- Elimination of unnecessary intermediate variables.

- **Common subexpression elimination**:-

$$a = \underbrace{b * c} * g \qquad a = tmp * g$$
$$d = \underbrace{b * c} + e \qquad d = tmp + e$$
$$tmp = b * c$$

  searches for instances of identical expressions.

- code movement of loops...

- strength reduction of operations (e.g. multiplication by 2 changed into shift of bits)

Assembler Optimization

- instruction selection : choosing the best way to implement the constructs may depend on the context
- register allocation:- decide best way to use the register available
- Load latency:- Loading a value into a register from store because instructions are pipelined in most modern processors the value is not available immediately.
  - we may rearrange statements so that some operation between when the value is loaded as when the register is used.
- Peephole optimization:- is performed over a very small set of instruction in a segment of generated code.

Qualities of a Compiler:-

- Correctness (preserve its meaning)
- Compiles quickly
- output execution speed
- how large is the code ?
  how much memory it use
  (Output foot print)
- separate compilation (relocatable code, linking)
- User friendly (front end):-
  good error recovery
- Debugging
- Cross language (interface compatibilities)
- Understandable and correct optimization.

Infix to postfix conversion

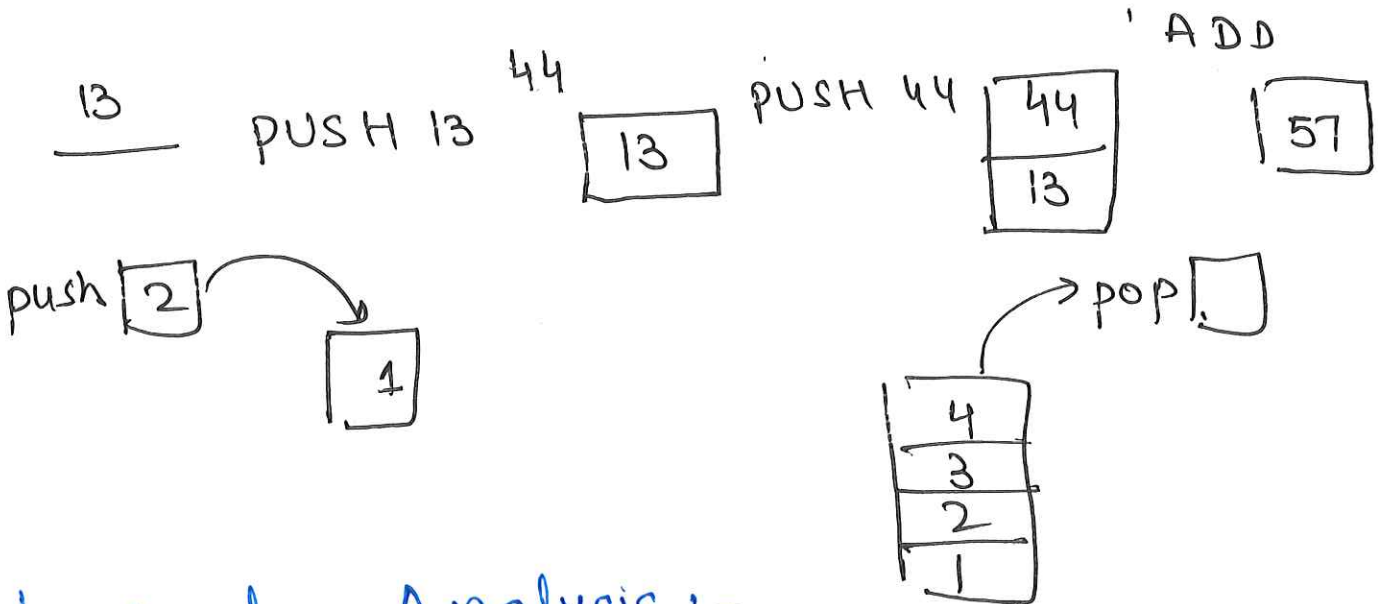
infix:- a op b

(13 + 44) * 2 + 6 → $\boxed{\text{infx 2 pofx}}$ 13 44 + 2 * 6 +

postfix ab op

Why is postfix notation popular in compiler writing ??

because it is easy to implement on the stack machine



Lexical Analysis :-

identifies tokens in input string.

Issues in lexical analysis :-

- Lookahead
- ambiguities.

Specifying Lexers :-

- Regular Expression.
- Examples of regular expressions.

```
if (i==j)
    z = 0;
else
    z = 1;
```

\t if (i==j) \n \t \t z=0; \n \t else \n \t \t z=1;

Goal :- partition input string into substring :

- Definite a finite set of tokens.
- Choice of token depends on language, design of parser.

Token : syntactic category

English : noun, verb, objective

Programming Language : identifier, int, keyword, whitespace ...

- Identifier :- strings of letters | digits, starting with a letter

Integer :- a non-empty string of digits

keyword :- if, else, begin

whitespace - non-empty set of blank, newline, tab.

Designing a Lexical Analyzer

Step 1 :- define a finite set of tokens

```
\t if (i==j) \n \t \t z = 0; \n \t else \n \t \t z = 1;
```

Tokens for this exp.
integer, keyword, relation, identifier
whitespace (,) = ;

Step 2 :- describe which strings belong to each token

identifier
int
keyword
whitespace

Step 3 :- Implementation

- Recognize substrings corresponding to tokens
- Return the value or lexeme of token.

Lexer usually discard "uninteresting tokens" that don't contribute to parsing. whitespace, comments.

Regular Languages:-

- simple and useful theory
- Easy to understand
- efficient implementation

Language:-

Let Σ be a set of characters.

A language over Σ is a set of strings of characters drawn from Σ.

English

• Alphabet = English characters	Alphabet = ASCII
• Language = English sentence	Langage - C prog.

- Regular Expression:- regexp / regex

sequence of characters that define a search pattern.

- consists of constants which denote set of strings, operator symbols, which denote operations over the sets.

Σ. finite alphabet

(empty set) ϕ denoting set ϕ

(empty string) ϵ → which has no characters at all

(literal char) a in Σ denoting set containing char 'a'

<u>Concatenation</u>

R = {"ab", "c"}

S = {"d", "ef"}

RS = {"abd", "abef", "cd", "cef"}

<u>Alteration</u> :- $R|S$ → set union of R and S

R = {"ab", "c"}

S = {"ab", "d", "ef"}

$\cancel{R^*}$

$R|S$ = {"ab", "c", "d", "ef"}

R^*