Compiler construction tools

Lexical Analyzer

- scanner generators
- input: source program
- output: lexical analyzer
- task of reading characters from source program and recognizing tokens or basic syntactic components
- maintains a list of reserved words

Lexical Analyzer

- Flex (fast lexical analyzer generator) or LEX – Rule Based programming language
- Example - specifies a scanner which replaces the string “username” with the user’s login name

```
%%
username printf(“%s”, getlogin());
```

Syntax Analyzer

- parser generators
- input: context-free grammar
- output: syntax analyzer
- the task of the syntax analyzer is to produce a representation of the source program in a form directly representing its syntax structure.

Syntax Analyzer

- Bison (Yacc-compatible parser gen.)
- a general purpose parser generator that converts grammar description for an LALR(1) CFG into a C program

Syntax Analyzer

- Bison grammar example (reverse polish notation)

```
 %{
   #define YYSTYPE double
   #include <math.h>
   %}
 %token NUM
%% /* grammar rules and actions below */

  %% C program
```

Semantic Analyzer

- syntax-directed translators
- input: parse tree
- output: routines to generate Intermediate code
- “The role of the semantic analyzer is to derive methods by which the structures constructed by the syntax analyzer may be evaluate or executed.“

- type checker
- two common tactics:
  - ~ flatten the semantic analyzer’s parse tree
  - ~ embed semantic analyzer with syntax analyzer (syntax-driven translation)

Intermediate Code Generator

- Automatic code generators
- input: Intermediate code rules
- output: crude target machine program
- “The task of the code generator is to traverse this tree, producing functionally equivalent object code.”
- three address code is one type

Code Optimizer

- Data flow engines
- input: I-code
- output: transformed code
- there is rarely a guarantee that the resulting code is the best possible.

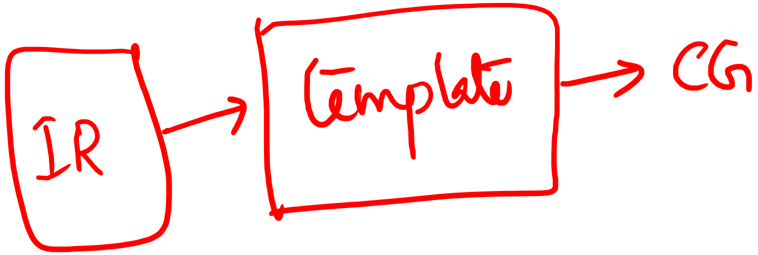
Code Generator

- Automatic code generators
- input: optimized (transformed) I-code
- output: target machine program
- Example (8 * x) / 2

  Load a, x

  Mult a, 8

  Div a, 2

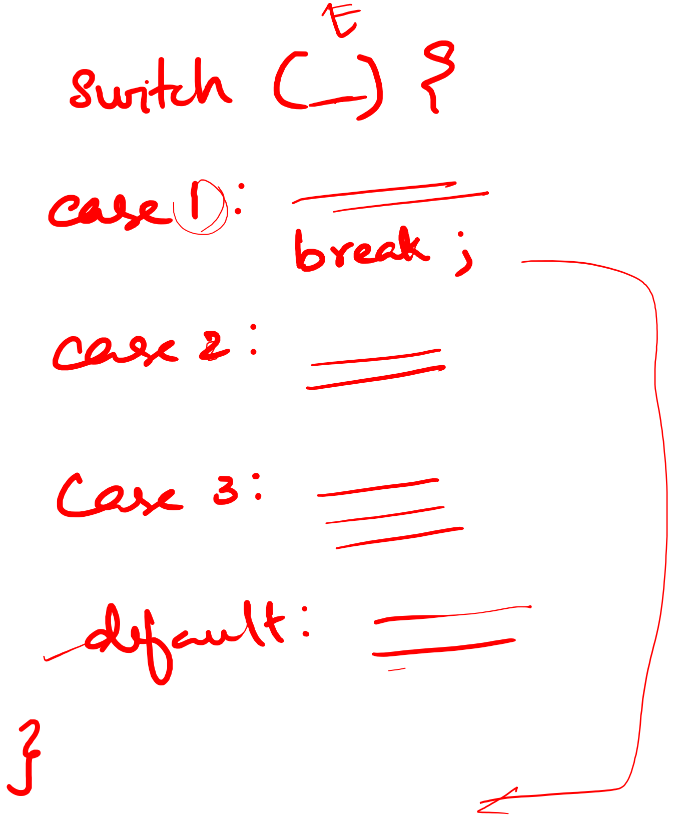


Challenges in Compiler Design

- Language Semantics
- Hardware Platform
- OS and system software
- Error Handling
- Aid in debugging
- Optimization
- Runtime Environment
- Speed of compilation

Language Semantics

- functionality of the programming language has to be supported
- Example
  - Case statements
  - Loop index
  - Break statements



```
        E
switch (_) {
case 1:  ____
         break;
case 2:  ____
case 3:  ____
default: ____
}
```

Hardware Platform

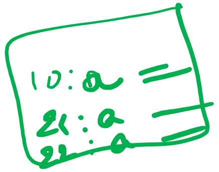
- Hardware platform vary from one machine to another machine that the architecture itself changes
- Code generation strategy for accumulator based machine cannot be similar to a stack based machine
- CISC or RISC instructions set

OS and system software

- Format of file to be executed is depicted by the operating system
- Linking process or the linker tool will combine many object file generated by different compilers into some executable file

Error Handling

- Show appropriate error messages
- Compiler designer has to imagine the probable types of mistakes, and design suitable detection, and recovery mechanism
- Some compilers even go to the extent of modifying source program partially, in order to correct it



10 : a = b + c;

21 : d = a * 2;

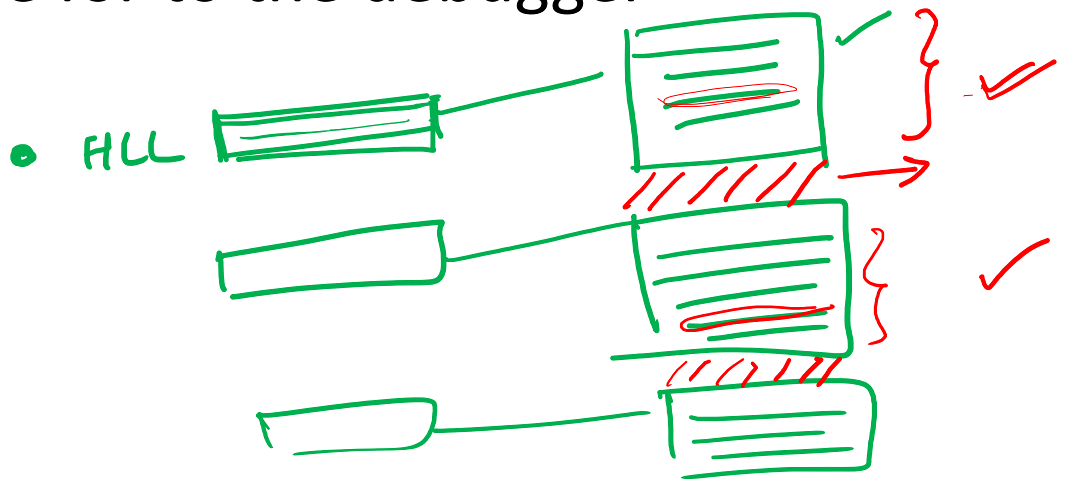
22 : c = d * a;

Aid in debugging

$a = b + c * 60$

```
MOV R1, B
MOV R2, C
ADD R1, R2
MUL R2, #60.0
MOV R2, a
```

- Helps in detecting logical errors in the program
- user needs to control the execution of machine language program, but sitting at the source language level
- compiler has to generate extra information regarding the correspondence between source and machine instructions
- Symbol table also needs to be available for to the debugger



Optimization

- Have to identify the set of transformation that may be beneficial for most of the programs in a language
- transformation should be safe
- trade-off between the time spent to optimize a program vs improvement in the execution time
- several levels of optimizations are used
- selecting a debugging mode or debugging option may disable any optimizations that disturbs the correspondence between the source program and object code

Runtime Environment

- deals with creating space for parameters and local variables
- Static memory locations may be used
- Stack frames are used to support recursion

Speed of compilation

- initial phase of program development contains lots of bugs, hence quick compilation may be the objective rather than optimized code
- towards the final stages, execution efficiency becomes the prime concern, more compilation time may be afforded to optimize the machine code