Lecture 19

Intermediate Code Generation

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

March 19, 2025

Take aways from the last class

- Equivalence of type expression

Take aways from the last class

- Equivalence of type expression
- Type conversion

Take aways from the last class

- Equivalence of type expression
- Type conversion
- Type checking for expression

Take aways from the last class

- Equivalence of type expression
- Type conversion
- Type checking for expression
- Overloaded functions

Take aways from the last class

- Equivalence of type expression
- Type conversion
- Type checking for expression
- Overloaded functions
- Type resolution

Intermediate Code Generation

- Abstraction at the source level:

Intermediate Code Generation

- Abstraction at the source level: identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)

Intermediate Code Generation

- Abstraction at the source level: identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
- Abstraction at the target level:

Intermediate Code Generation

- Abstraction at the source level: identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
- Abstraction at the target level: memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems

Intermediate Code Generation

- Abstraction at the source level: identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
- Abstraction at the target level: memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems
- mapping from source level abstractions to target machine abstractions

Intermediate Code Generation

- Abstraction at the source level: identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
- Abstraction at the target level: memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems
- mapping from source level abstractions to target machine abstractions
- Front end translates a source program into an intermediate representation

Intermediate Code Generation

- Abstraction at the source level: identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
- Abstraction at the target level: memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems
- mapping from source level abstractions to target machine abstractions
- Front end translates a source program into an intermediate representation
- Back end generates target code from intermediate representation

IR Design

- More of a magic rather than science

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs
  - HIR (high level IR) preserves loop structure and array bounds (AST)
    $clang - c \quad - Xclang \quad - ast - dump \quad next/b.c$

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs
  - HIR (high level IR) preserves loop structure and array bounds (AST)
    $clang - c \quad - Xclang \quad - ast - dump \quad next/b.c$
  - MIR (medium level IR) reflects range of features in a set of source languages

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs
  - HIR (high level IR) preserves loop structure and array bounds (AST)
    $clang - c \quad - Xclang \quad - ast - dump \quad next/b.c$
  - MIR (medium level IR) reflects range of features in a set of source languages
  - LIR (low level IR) low level similar to the machines

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs
  - HIR (high level IR) preserves loop structure and array bounds (AST)
    $clang - c \quad - Xclang \quad - ast - dump \quad next/b.c$
  - MIR (medium level IR) reflects range of features in a set of source languages
  - LIR (low level IR) low level similar to the machines
- As the translation takes place, IR is repeatedly analyzed and transformed

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs
  - HIR (high level IR) preserves loop structure and array bounds (AST)
    $clang - c \quad - Xclang \quad - ast - dump \quad next/b.c$
  - MIR (medium level IR) reflects range of features in a set of source languages
  - LIR (low level IR) low level similar to the machines
- As the translation takes place, IR is repeatedly analyzed and transformed
- Compiler users want analysis and translation to be fast and correct

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs
    - HIR (high level IR) preserves loop structure and array bounds (AST)
      $clang - c \quad - Xclang \quad - ast - dump \quad next/b.c$
    - MIR (medium level IR) reflects range of features in a set of source languages
    - LIR (low level IR) low level similar to the machines
- As the translation takes place, IR is repeatedly analyzed and transformed
- Compiler users want analysis and translation to be fast and correct
- Compiler writers want optimizations to be simple to write, easy to understand and easy to extend

IR Design

- More of a magic rather than science
- Generally compiler uses 2-3 IRs
  - HIR (high level IR) preserves loop structure and array bounds (AST)
    $clang - c \quad - Xclang \quad - ast - dump \quad next/b.c$
  - MIR (medium level IR) reflects range of features in a set of source languages
  - LIR (low level IR) low level similar to the machines
- As the translation takes place, IR is repeatedly analyzed and transformed
- Compiler users want analysis and translation to be fast and correct
- Compiler writers want optimizations to be simple to write, easy to understand and easy to extend
- IR should be simple and light weight while allowing easy expression of optimizations and transformations.

Issues in IR Design

- How much machine dependent

Issues in IR Design

- How much machine dependent
- Expressiveness: how many languages are covered

Issues in IR Design

- How much machine dependent
- Expressiveness: how many languages are covered
- Appropriateness for code optimization

Issues in IR Design

- How much machine dependent
- Expressiveness: how many languages are covered
- Appropriateness for code optimization
- Appropriateness for code generation

Three Address Code

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

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

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

- 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
- Only one operator on the right hand side is allowed

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
- Only one operator on the right hand side is allowed
- Source expression like $x + y * z$ might be translated into

```
t1 := y * z
t2 := x + t1
```

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
- Only one operator on the right hand side is allowed
- Source expression like $x + y * z$ might be translated into

  ```
  t1 := y * z
  t2 := x + t1
  ```

  where t1 and t2 are compiler generated temporary names
- Three address code are 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

Three address instructions

- Assignment
  - x = y op z
  - x = op y
  - x=y
- Jump
  - goto L
  - if x relop y goto L

Three address instructions

- Assignment
  - x = y op z
  - x = op y
  - x=y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y

Three address instructions

- Assignment
  - x = y op z
  - x = op y
  - x=y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y
- Function
  - param x
  - call p,n
  - return y

Three address instructions

- Assignment
  - x = y op z
  - x = op y
  - x=y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - x = y
  - x = *y
  - *x = y

this can be written directly in 3AC in exam.
No need to generate a separate temporary for x relop y.

Other Represenation

- SSA: Single Static Assignment

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements
- Java intermediate bytecodes

Other Represenation

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements
- Java intermediate bytecodes
- The list goes on ······

whenever assigning something, use a different variable name.

is a low-level language that is used to describe the functioning of a digital circuit and, more specifically, the transfer of information between registers.

P-code is the Assembly language for a hypothetical stack machine, the P-machine, said to have been an imitation of the instruction set for the Burroughs Large System