Intermediate Representation

Abstract syntax tree, control-flow graph, three-address code

Intermediate Code Generation

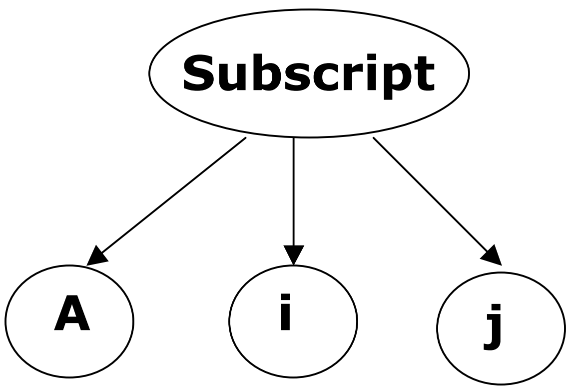
- Intermediate language between source and target
- Multiple machines can be targeted
  - Attaching a different backend for each machine
  - Intel, AMD, IBM machines can all share the same parser for C/C++
- Multiple source languages can be supported
  - Attaching a different frontend (parser) for each language
  - Eg. C and C++ can share the same backend
- Allow independent code optimizations
  - Multiple levels of intermediate representation
    - Supporting the needs of different analyses and optimizations

IR In Compilers

- Internal representation of input program by compilers
  - Source code of the input program
  - Results of program analysis
    - Control-flow graphs, data-flow graphs, dependence graphs
  - Symbol tables
    - Book-keeping information for translation (eg., types and addresses of variables and subroutines)
- Selecting IR --- depends on the goal of compilation
  - Source-to-source translation: close to source language
    - Parse trees and abstract syntax trees
  - Translating to machine code: close to machine code
    - Linear three-address code
- External format of IR
  - Support independent passes over IR

Abstraction Level in IR

- Source-level IR
  - High-level constructs are readily available for optimization
    - Array access, loops, classes, methods, functions
- Machine-level IR
  - Expose low-level instructions for optimization
    - Array address calculation, goto branches



Source-level tree

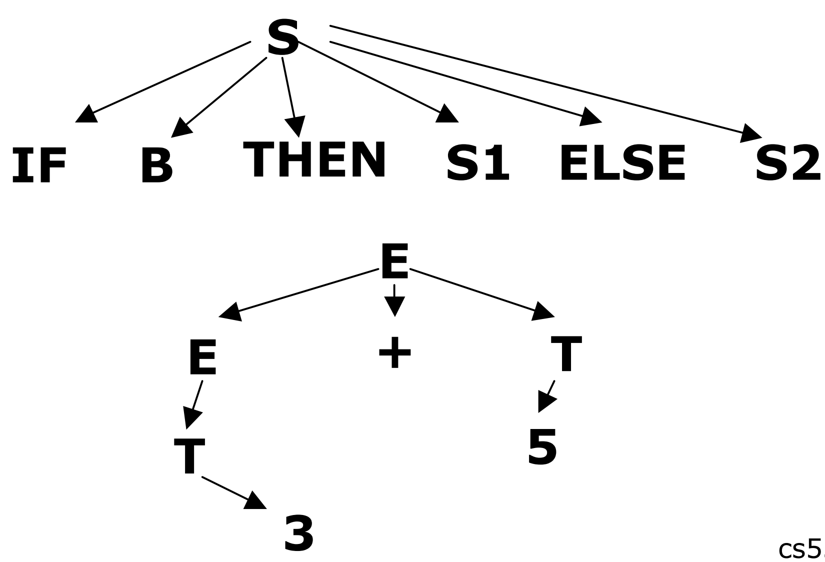
```
loadI 1   => r1
sub  rj,  r1 => r2
loadI 10   => r3
mult  r2, r3  => r4
sub  ri, r1 => r5
add  r4, r5  => r6
loadI @A    => r7
add  r7, r6  => r8
load r8   => rAij
```

ILOC code

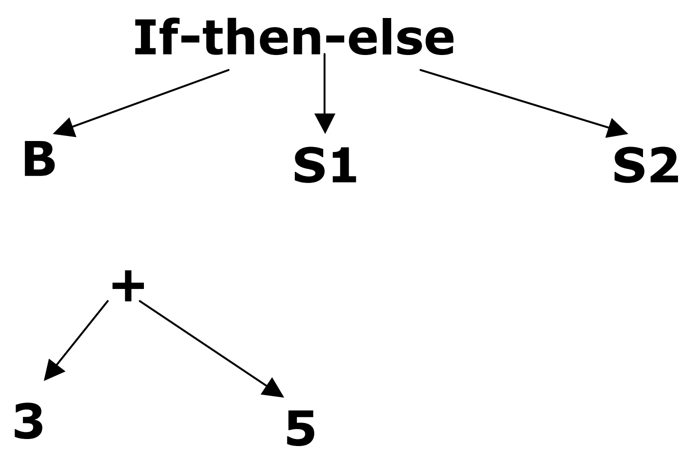
Parse Tree And AST

- Graphically represent grammatical structure of input program
  - Parse tree: tree representation of syntax derivations
  - AST: condensed form of parse tree
    - Operators and keywords do not appear as leaves
    - Chains of single productions are collapsed

Parse trees



Abstract syntax trees



Implementing AST in C

Grammar:

E ::= E + T | E – T | T
T ::= (E) | id | num

- Define different kinds of AST nodes
  - typedef enum {PLUS, MINUS, ID, NUM} ASTNodeTag;
- Define AST node types

```
typedef struct ASTnode {
    AstNodeTag kind;
    union { symbol_table_entry* id_entry;
            int  num_value;
            struct ASTnode* opds[2];
          } description;
};
```

- Define AST node construction routines
  - ASTnode* mkleaf_id(symbol_table_entry* e);
  - ASTnode* mkleaf_num(int n);
  - ASTnode* mknode_plus(struct ASTnode* opd1, struct ASTNode* opd2);
  - ASTnode* mknode_minus(struct ASTnode* opd1, struct ASTNode* opd2);

Implementing AST in Java

Grammar:

E ::= E + T | E – T | T
T ::= (E) | id | num

- Define AST node

```
abstract class ASTexpression {
    public System.String toString();
}
class ASTidentifier extends ASTexpression {  private symbol_table_entry id_entry;  … }
class ASTvalue extends ASTexpression { private int num_value; … }
class ASTplus extends ASTexpression { private ASTnode opds[2]; … }
Class ASTminus extends ASTexpression { private ASTnode opds[2]; ... }
```

- Define AST node construction routines
  - ASTexpression mkleaf_id(symbol_table_entry e)
    { return new ASTidentifier(e); }
  - ASTexpression mkleaf_num(int n)
    { return new ASTvalue(n); }
  - ASTexpression mknode_plus(ASTnode opd1, struct ASTNode opd2)
    { return new ASTplus(opd1, opd2);
  - ASTexpression mknode_minus(ASTnode opd1, struct ASTNode opd2)
    { return new ASTminus(opd1, opd2);

Constructing AST

- Use syntax-directed definitions
  - Associate each non-terminal with an AST
    - A pointer to an AST node: E.nptr T.nptr
  - Evaluate synthesized attribute bottom-up
    - From children ASTs, compute AST of the parent

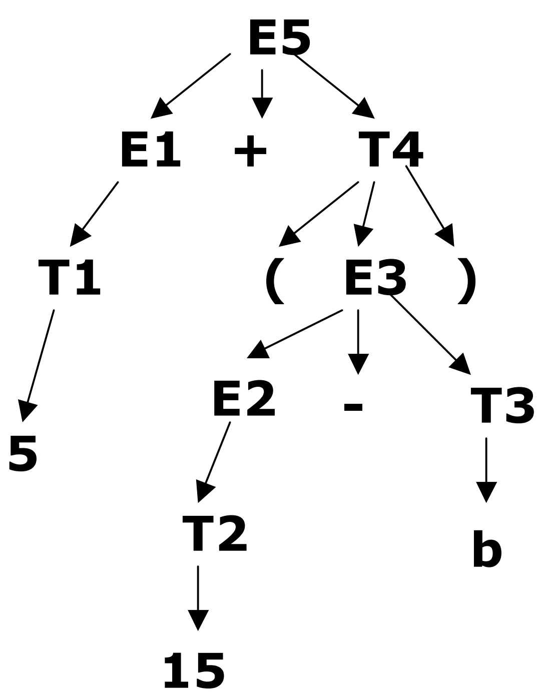
E ::= E1 + T	{ E.nptr=mknode_plus(E1.nptr,T.nptr); }
E ::= E1 – T	{ E.nptr=mknode_minus(E1.nptr,T.nptr); }
E ::= T	{ E.nptr=T.nptr; }
T ::= (E)	{T.nptr=E.nptr; }
T ::= id	{ T.nptr=mkleaf_id(id.entry); }
T ::= num	{ T.nptr=mkleaf_num(num.val); }

Exercise: what is the AST for 5 + (15-b)?
What if top-down parsing is used
(need to eliminate left-recursion)?

Example: AST for 5+(15-b)

Bottom-up parsing: evaluate attribute at each reduction

Parse tree for 5+(15-b)



1. reduce 5 to T1 using T::=num:
   T1.nptr = leaf(5)
2. reduce T1 to E1 using E::=T:
   E1.nptr = T1.nptr = leaf(5)
3. reduce 15 to T2 using T::=num:
   T2.nptr=leaf(15)
4. reduce T2 to E2 using E::=T:
   E2.nptr=T2.nptr = leaf(15)
5. reduce b to T3 using T::=num:
   T3.nptr=leaf(b)
6. reduce E2-T3 to E3 using E::=E-T:
   E3.nptr=node('-',leaf(15),leaf(b))
7. reduce (E3) to T4 using T::=(E):
   T4.nptr=node('-',leaf(15),leaf(b))
8. reduce E1+T4 to E5 using E::=E+T:
   E5.nptr=node('+',leaf(5),
   node('-',leaf(15),leaf(b)))

Symbol tables

- Symbol tables
  - Record information about names defined in programs
    - Types of variables and functions
    - Additional properties (eg., static, global, scope)
  - Contain information about context of program fragment
    - Can use different symbol tables for different purposes
- Naming conflicts
  - The same name may represent different things in different places
    - Use separate symbol tables for names in different scopes
    - Multiple layers of symbol tables for nested scopes
- Implementation of symbol tables
  - Map names to additional information (types,values,etc.)
  - Efficient implementation: using hash tables

Implementing symbol tables

- Interface
  - Lookup(name)
    - Returns the record for name if one exists in the table; otherwise, indicates that name is not found
  - Insert(name, record)
    - Stores the information in record in the table for name.
- Symbol tables in nested scopes
  - StartNewScope()
    - Increment the current scope level and creates a new symbol table
  - ExitScope()
    - Changes the current-level symbol table pointer so that it points to the symbol table of surrounding scope
- Use a global symbol table pointer to keep track of the current scope

Linear IR

- Low level IL before final code generation
  - A linear sequence of low-level instructions
  - Implemented as a collection (table or list) of tuples
- Similar to assembly code for an abstract machine
  - Explicit conditional branches and goto jumps
- Reflect instruction sets of the target machine
  - Stack-machine code and three-address code

Stack-machine code	**two-address code**	**three-address code**
Push 2	**MOV 2 => t1**	**t1 := 2**
Push y	**MOV y => t2**	**t2 := y**
Multiply	**MULT t2 => t1**	**t3 := t1*t2**
Push x	**MOV x => t4**	**t4 := x**
subtract	**SUB t1 => t4**	**t5 := t4-t3**

Linear IR for x – 2 * y

Stack-machine code

- Also called one-address code
  - Assumes an operand stack
  - Take operands from top of stack; push results onto the stack
  - Need special operations such as
    - Swapping two operands on top of the stack
- Compact in space, simple to generate and execute
  - Most operands do not need names
  - Results are transitory unless explicitly moved to memory
- Used as IR for Smalltalk and Java

Stack-machine code for x – 2 * y

Push 2
Push y
Multiply
Push x
subtract

Three address code

- Each instruction contains at most two operands and one result.
- Typical forms include
  - Arithmetic operations: x := y op z | x := op y
  - Data movement: x := y [z] | x[z] := y | x := y
  - Control flow: if y op z goto x | goto x
  - Function call: param x | return y | call foo
- Each instruction maps to at most a few machine instructions
- Additional constraints depend on target machine instructions
  - Eg., for x := y op z and x := op y
    all operands must be in registers ➔ all operands must be temporaries?
- Reasonably compact, while allowing reuse of names and values

Three-address code for x – 2 * y

```
t1 := 2
t2 := y
t3 := t1*t2
t4 := x
t5 := t4-t3
```

Storing Three-Address Code

- Store all instructions in a quadruple table
  - Every instruction has four fields: op, arg1, arg2, result
  - The label of instructions ➔ index of instruction in table

Three-address code

```
t1 := - c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Quadruple entries

	op	arg1	arg2	result
(0)	Uminus	c		t1
(1)	Mult	b	t1	t2
(2)	Uminus	c		t3
(3)	Mult	b	t3	t4
(4)	Plus	t2	t4	t5
(5)	Assign	t5		a

Alternative: store all the instructions in a singly/doubly linked list

What is the tradeoff?

Mapping Storages To Variables

- Variables are placeholders for values
  - Every variable must have a location to store its value
    - Register, stack, heap, static storage
  - Values need to be loaded into registers before operation

Three-address code for x – 2 * y:

	x and y are in registers	x and y are in memory
Three-address code for x – 2 * y:	t1 := 2 t2 := t1*y t3 := x-t2	t1 := 2 t2 := y t3 := t1*t2 t4 := x t5 := t4-t3

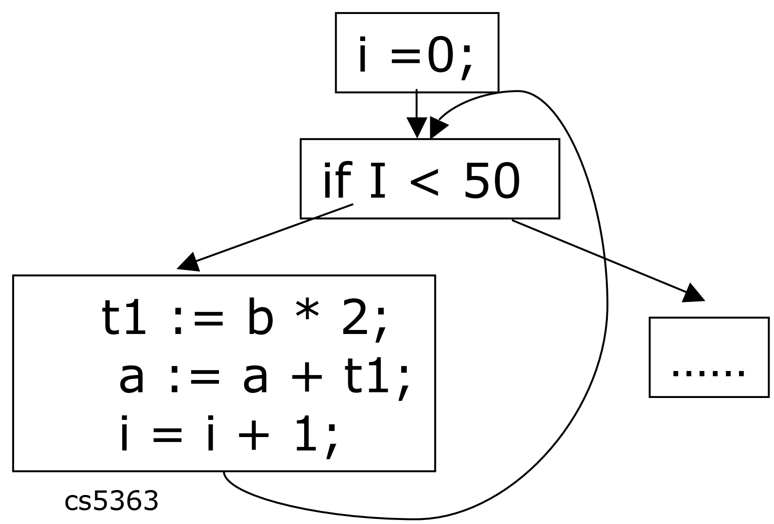
Which variables can be kept in registers?
Which variables must be stored in memory?

```
void A(int b, int *p)
{
  int a, d;
  a = 3;    d = foo(a);   *p =b+d;
}
```

Appendix: Control-flow graph

- Graphical representation of runtime control-flow paths
  - Nodes of graph: basic blocks (straight-line computations)
  - Edges of graph: flows of control
- Useful for collecting information about computation
  - Detect loops, remove redundant computations, register allocation, instruction scheduling…
- Alternative CFG: Each node contains a single statement

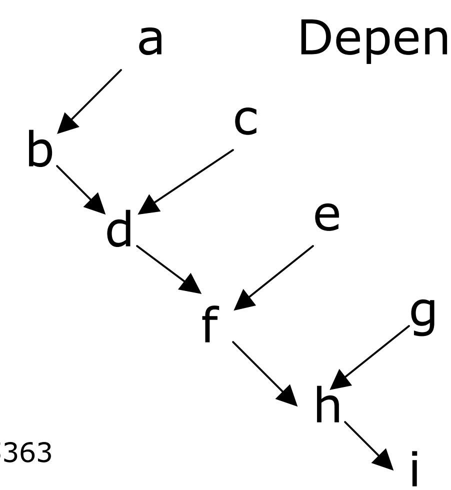
```
……
   i = 0
   while (i < 50) {
     t1 = b * 2;
     a = a + t1;
     i = i + 1;
   }
….
```



Appendix: Dependence graph

- Graphical representation of reordering constraints between statements
  - Each node n is a single operation/statement
  - Edge (n1,n2) indicates n2 uses result of n1
    - The order of evaluating n1,n2 cannot be reversed
  - Graph is acyclic within each basic block; is cyclic if loops exist
- Used in reordering transformations
  - Instruction scheduling, loop transformations
- Construction
  - For each pair of statements, evaluate ordering constraint

```
a: r1 := w
b: r1 := r1 + r1
c: r2 := x
d: r1 := r1 * r2
e: r2 := y
f:  r1 := r1 * r2
g: r2 := z
h: r1 := r1 * r2
i:  return r1
```



Dependence graph

Appendix: Static Single-Assignment

- A variable can hold multiple values throughout its lifetime
  - Mapping multiple values to a name can hide opportunities of optimization
- Static single-assignment form (SSA)
  - Each variable is defined by a single operation in the code
  - Each use of variable refers to a single definition
  - Use ∅-functions to merge definitions from different control-flow paths

```
x := …
y := …
while (x < 100)
   x := x + 1
    y := y + x
```

SSA:

```
         x0 := …
         y0 := …
         if (x0 < 100) goto loop
         goto next
loop:     x1 := ∅(x0,x2)
          y1 := ∅(y0,y2)
          x2 := x1 + 1
          y2 := y1 + x2
          if (x2 < 100) goto loop
next:    x3 := ∅(x0,x2)
          y3 := ∅(y0,y2)
```