CSE 504

Course Summary

Organization of a Compiler

- Lexical analysis
- Parsing (syntax analysis)
- Abstract Syntax Tree (AST)
- Semantic Analysis (type checking etc.)
- Syntax-directed definitions (attribute grammars)
- Intermediate code generation
- Code optimization
- Final code generation
- Runtime Environment

Lexical Analysis: Foundations

- Token, Lexeme, Pattern, String
- Regular expressions
  - Syntax, semantics
  - Finite-state automata
    - NFA vs DFA
    - Recognition using NFA
    - NFA to DFA translation
    - Optimization of DFAs
  - Properties of regular languages
    - Closed under complementation, union, intersection
  - RE to FSA translation
    - RE → NFA → DFA → optimal DFA
    - Direct construction of DFA

Lexical Analysis

- Goal: convert character stream to token stream
  - Recognize “words” in language
    - Keywords, identifiers, constants (literals), ..
  - Ignore “irrelevant” input
    - White spaces, comments, …
  - Maintain association between lexer output and input
    - Line numbers, column numbers, …
- Flex: A lexical analyzer generator
  - Use of Flex in compilers
  - Use of regular expressions as well as ***start*** states
    - Ability to freely intermix automata-based and RE based specifications of lexical analysis
    - Very powerful capability, makes Flex a very versatile tool for any application requiring efficient recognition of REs

Syntax Analysis: CFGs

- Types of grammars
  - Regular, context-free, context-sensitive, unrestricted
- CFGs
  - Terminals, Nonterminals, Productions, Start symbol
  - Sentence, Sentential form, String
  - Notational conventions
  - L(G)
  - Equivalence of grammars
  - Two sides of grammars: generation and acceptance

CFGs

- Derivations
  - Single-step, multistep
  - Left-most, right-most, canonical
- Parse trees
- Ambiguity
- Disambiguation rules
  - Operator precedence
  - dangling-else and shift/reduce conflict

CFGs (continued)

- Equivalence of grammars (and how to establish this)
- Recognizing grammars
  - Push-down automata (PDA)
  - NPDA Vs DPDA
- Properties
  - Closed under union, but not complementation or intersection
  - Note: CFGs recognizable using DPDAs are closed under all these operations.

Top-Down Parsing

- **Derive sentence from start symbol**
  - Next step in derivation is guided by input
- Predictive Parsing
  - Left-recursion elimination and left-factoring
  - Parsing with back-tracking
  - Recursive descent parsing
- Non-recursive parsing
  - Table-driven
  - FIRST and FOLLOW
- LL(1) grammars

Bottom-Up Parsing

- **Reduce sentence to start symbol**
  - Next reduction is guided by PDA stack and input
- Handles
- Shift-Reduce parsing
  - Structure and operation of an SR parser
- Identification of handles
- Viable prefixes

LR Parsing

- Structure and operation of an LR parser
- Action and Goto tables
- LR Vs LL parsing
- Construction of SLR(1) parsing tables
  - Items and Item sets
  - Viable prefixes
  - DFA for recognizing viable prefixes
  - Generation of LR parsing tables from DFA
- LR(1) and LALR(1) parsing

Parser Generators

- Bison/Yacc
  - LALR(1) Parser generator
  - Integrates nicely with Lex/Flex
- Use of Bison to specify a parser
- Conflicts
  - How to interpret them
  - How to fix them
    - Operator precedence
- Bison is a versatile tool
  - Can be used for a variety of language processing applications
- Error recovery

Syntax-Directed Translation

- The concept and its use
- Syntax-directed translation using Bison
- Attribute grammars --- acceptance by AG
- Synthesized Vs inherited attributes
  - Flow of attribute information
- L-attributed definitions
- S-attributed definitions
- Maintaining attributes during parsing
  - Top-down parsing
  - Bottom-up parsing

Symbol Tables

- Bindings
- Attributes
- Binding Time
- Scopes
- Visibility
- Lexical scoping
- Implementation of symbol tables
- Static Vs Dynamic scoping

Semantic Analysis

- Semantic analysis takes place during
  - AST construction
  - Type-checking
  - Intermediate code generation
- ASTs vs Parse trees
- Syntax-directed construction of AST using Bison/C++

Types

- What is a type
- Data types in modern languages
  - Simple types
  - Compound types
    - Products, unions (tagged Vs untagged), arrays, functions, pointers
  - Type expressions
- Polymorphism
  - Parametric polymorphism Vs overloading
  - Code reuse
- Type equivalence
  - Structural Vs Name based Vs declaration based
- Type compatibility
- Type checking Vs type inference
- Type conversions
  - Explicit, implicit, coercion
- Static Vs Dynamic typing
- Strong Vs Weak typing

Type-Checking

- Syntax-directed definitions for type-checking
  - Expressions
  - Assignment
  - Function calls/returns
  - Other statements
- Subtype principle
- Name resolution
  - Overloading resolution
  - Resolution of methods in OO languages

Expression Evaluation

- Semantics of Expressions
  - Order of evaluation
  - Use of properties of arithmetic operators
  - Problems with side-effects
- Boolean expression evaluation
  - Short-circtuit evaluation
- Control-flow statement evaluation
  - Switch-statement
  - While statement
  - For statement

Procedure calls

- Parameter-passing mechanisms
  - Call-by-Value
  - Call-by-Reference
  - Call-by-Name
  - Call-by-Need
  - Macros
  - Difficulties with parameter passing mechanisms
- Semantics of parameter passing
- Implementation of procedure calls
  - Stack, activation records
  - Caller Vs Callee responsibilities
- Exception-handling

Memory allocation

- Simple types Vs structures and arrays
- Global/static variables
- Stack allocation
  - How local variables and parameters are accessed
  - Accessing nonlocal variables
- Structure of activation records
- Heap allocation
  - Explicit Vs Automatic management
  - Fragmentation
  - Garbage collection
    - Reference-counting Vs mark/sweep Vs copying collection
    - Conservative GC

Implementation Aspects OO Languages

- Layout of structures and objects
  - Accessing data members
- Efficient implementation of virtual functions
- Subtype principle and how it dictates the implementation of OO languages

Code Generation

- Intermediate code formats
- Syntax-directed definition for IC generation
  - Declarations
  - Expressions
  - Assignments
    - l- and r-values
    - accessing arrays and other complex data types
  - Function calls
  - Conditionals
    - Short-circuit evaluation of boolean expressions and handling of conditionals
    - Loops

Machine Code Generation

- Assembly code versus machine code generation issues
  - Linkers, shared libraries, executables, symbol tables, etc.
- Register allocation
  - Cost savings due to use of registers
  - Graph-coloring based algorithm and heuristics
  - Works well in practice, no sense in using “register” declarations in your program, which will likely lead to less efficient code
- Instruction selection
  - Instruction set specification
  - Automated generation of assembly code from specifications
  - Optimal code generation using dynamic programming
    - Combines register allocation with assembly code generation

Code Optimization

- High-level, intermediate code and low-level optimizations
- High-level optimizations
  - Inlining, partial evaluation, tail call elimination, loop reordering, ...
- Intermediate code optimizations
  - CSE
  - constant and copy propagation
  - strength reduction, loop-invariant code motion
  - dead-code elimination
  - jump-threading

Dataflow Analysis

- Formulation
- Setting-up dataflow equations
- Approximation, direction of approximation, and soundness
- Recursion and fixpoint iteration
- Applications
  - Reaching definitions
  - Available expressions (CSE)
  - Live variables
- Difficulties
  - Procedure calls
  - Aliasing

Translation Strategy

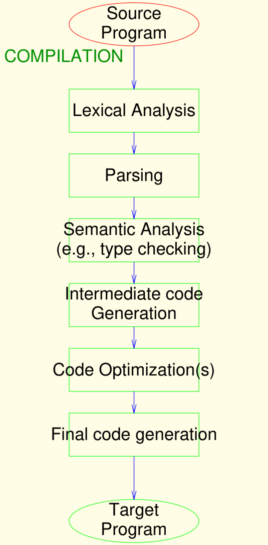
Classic Software Engineering Problem

- **Objective:** Translate a program in a high level language into *efficient* executable code.
- **Strategy:** Divide translation process into a series of phases.

  Each phase manages some particular aspect of translation.

Interfaces between phases governed by specific intermediate forms.

Translation Steps



Syntax Analysis Phase: Recognizes "sentences" in the program using the *syntax* of the language

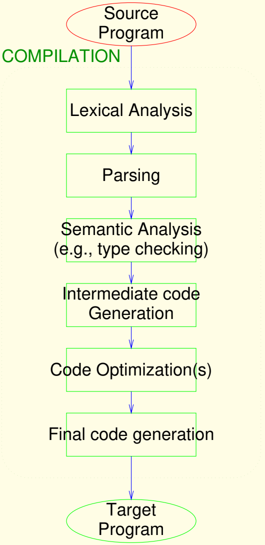
Semantic Analysis Phase: Infers information about the program using the *semantics* of the language

Intermediate Code Generation Phase: Generates "abstract" code based on the syntactic structure of the program and the semantic information from Phase 2.

Optimization Phase: Refines the generated code using a series of *optimizing* transformations.

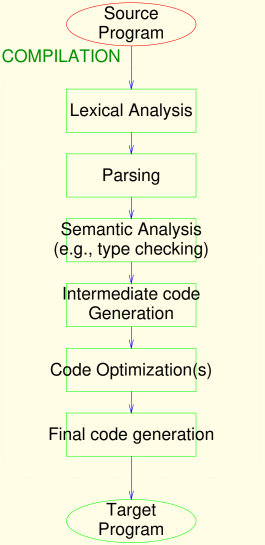
Final Code Generation Phase: Translates the abstract intermediate code into specific machine instructions.

Translation Steps: Lexical Analysis (Scanning Phase)



- Convert the *stream of characters representing input program* into a sequence of *tokens*.
- Tokens are the "words" of the programming language.
- For instance, the sequence of characters "`static int`" is recognized as two tokens, representing the two words "static" and "int".
- The sequence of characters "`* x++`" is recognized as three tokens, representing "`*`", "`x`" and "`++`".

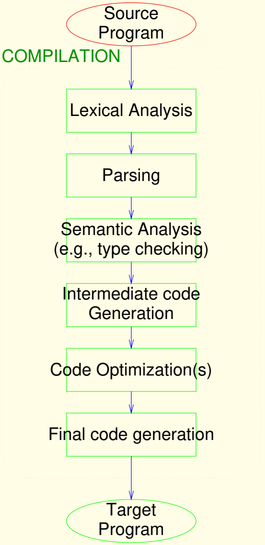
Translation Steps: Parsing (Syntax Analysis Phase)



- Uncover the *structure* of a sentence in the program from a stream of *tokens*.
- For instance, the phrase "x = -y", which is recognized as four tokens, representing "x", "=" and "-" and "y", has the structure **=(x, -(y))**, i.e., an assignment expression, that operates on "x" and the expression "-(y)".
- Build a *tree* called a *parse tree* that reflects the structure of the input sentence.

Typically, compilers build an *abstract syntax tree* directly, skipping the construction of parse trees.

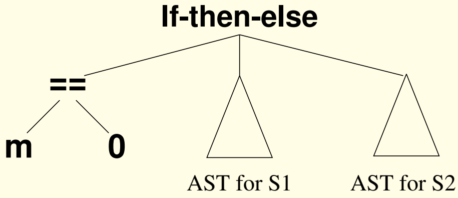
Translation Steps: Abstract Syntax Tree (AST)



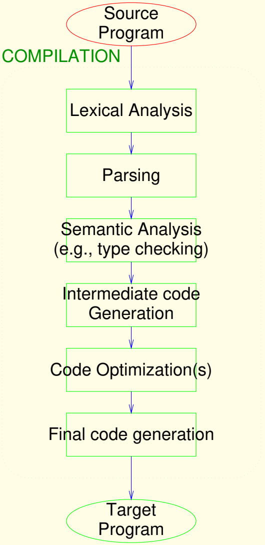
- Represents the syntactic structure of the program, hiding a few details that are irrelevent to later phases of compilation.
- For instance, consider a statement of the form:

```
if (m == 0) S1 else S2
```

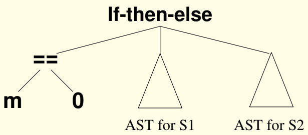
where `S1` and `S2` stand for some block of statements. A possible AST for this statement is:



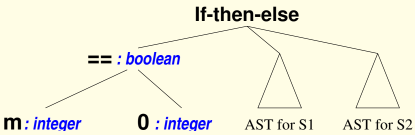
Translation Steps: Type Checking (Semantic Analysis)



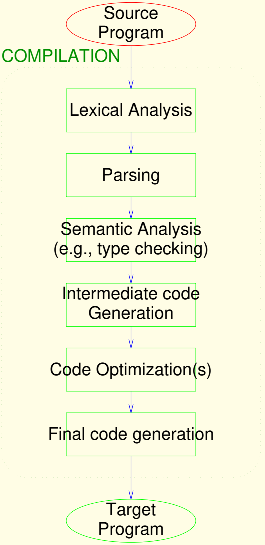
- Decorate the AST with semantic information that is necessary in later phases of translation.
- For instance, the AST



becomes

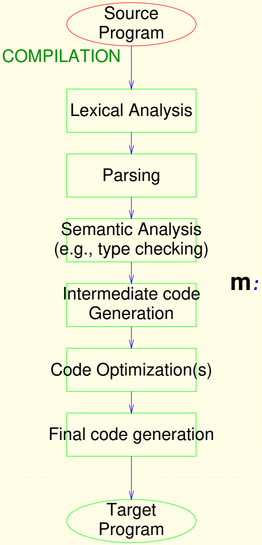


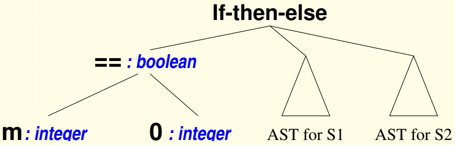
Translation Steps: Intermediate Code Generation



- Translate each sub-tree of the decorated AST into *intermediate code*.
- Intermediate code hides many machine-level details, but has instruction-level mapping to many assembly languages.
- Main motivation: portability.

Translation Steps: Intermediate Code Generation Example

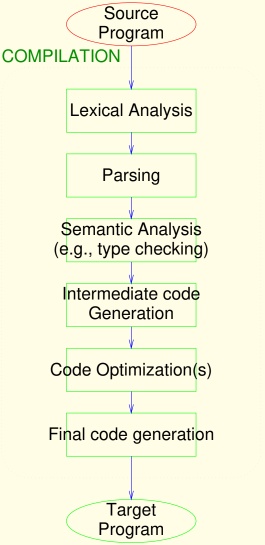




becomes

```
    R1 ← mem(m)
    cmp R1, 0
    jz .L1
    jmp .L2
.L1:
    ....  code for S1
    jmp .L3
.L2:
    ....  code for S2
    jmp .L3
.L3:
```

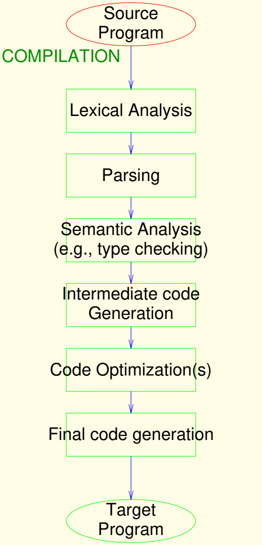
Translation Steps: Code Optimization



Apply a series of transformations to improve the time and space efficiency of the generated code.

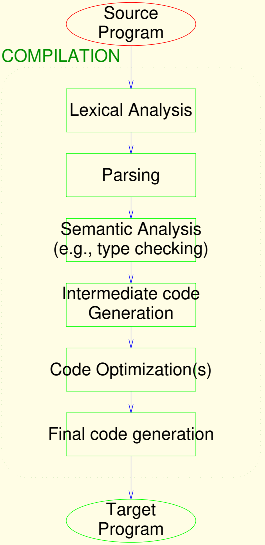
- *Peephole optimizations:* generate new instructions by combining/expanding on a small number of consecutive instructions.
- *Intraprocedural optimizations:* reorder, remove or add instructions to change the structure of generated code *within each function*. Code transformations guided by *static analysis.*
- *Interprocedural optimizations:* Guided by interprocedural static analysis.

Translation Steps: Final Code Generation



- Map instructions in the intermediate code to specific machine instructions.
- Supports standard object file formats.
- Generates sufficient information to enable symbolic debugging.

Translation Steps: Final Code Generation Example




```
      R1 ← mem(m)
      cmp R1, 0
      jz .L1
      jmp .L2
.L1:
      ....  code
for S1
      jmp .L3
.L2:
      ....  code
for S2
      jmp .L3
.L3:
```

⟹

```
      movl 8(%ebp), %esi
      testl %esi, %esi
      jne .L2
.L1:
      ....  code for S1
      jmp .L3
.L2:
      ....  code for S2
.L3:
```

Broader Applications of Languages

- Command Interpreters: bash, ksh, Powershell, ...
- Programming: Java, Python, C++, Rust, Go, Haskell, Scala, OCaml, ...
- Document Structuring: LaTeX, HTML, RTF, troff, ...
- Page Definition: PDF, PostScript, ...
- Databases: SQL, ...
- Hardware Design: VHDL, VeriLog, ...
- Domain-Specific Languages (DSL)

Phases of Syntax Analysis

1. Identify the words: **Lexical Analysis**.

   Converts a stream of characters (input program) into a stream of tokens.

   Also called *Scanning* or *Tokenizing*.

2. Identify the sentences: **Parsing**.

   Derive the structure of sentences: construct *parse trees* from a stream of tokens.

Lexical Analysis

Convert a stream of characters into a stream of *tokens*.

- Simplicity: Conventions about "words" are often different from conventions about "sentences".
- Efficiency: Word identification problem has a much more efficient solution than sentence identification problem.
- Portability: Character set, special characters, device features.

Terminology

- Token: Name given to a family of words. e.g., integer_constant
- Lexeme: Actual sequence of characters representing a word. e.g., 32894
- Pattern: Notation used to identify the set of lexemes represented by a token. e.g., $[0-9]+$

Terminology

A few more examples:

Token	Sample Lexemes	Pattern
while	while	while
integer_constant	$32894, -1093, 0$	$(-\|\epsilon)[0-9]+$
identifier	buffer_size	$[\_a-zA-Z]+$

Patterns

How do we *compactly* represent the set of all lexemes corresponding to a token?

For instance:

The token integer_constant *represents the set of all integers: that is, all sequences of digits (0–9), preceded by an optional sign (+ or −).*

Obviously, we cannot simply enumerate all lexemes.

Use **Regular Expressions**.

Regular Expressions over alphabet $\sum$

Let R be the set of all regular expressions over Σ. Then,

- Empty String: $\epsilon \in R$
- Unit Strings: $\alpha \in \Sigma \Rightarrow \alpha \in R$
- Concatenation: $r_1, r_2 \in R \Rightarrow r_1 r_2 \in R$
- Alternative: $r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R$
- Kleene Closure: $r \in R \Rightarrow r^* \in R$

Semantics of Regular Expressions

Semantic Function $\mathcal{L}$: Maps regular expressions to sets of strings.

$$
\begin{array}{rcll}
\mathcal{L}(\epsilon) & = & \{\epsilon\} & \\
\mathcal{L}(\alpha) & = & \{\alpha\} & (\alpha \in \Sigma) \\
\mathcal{L}(r_1 \mid r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) & \\
\mathcal{L}(r_1\ r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) & \\
\mathcal{L}(r^*) & = & \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*)) &
\end{array}
$$

Computing the Semantics

$$
\begin{aligned}
\mathcal{L}(a) &= \{\mathrm{a}\} \\
\mathcal{L}(a \mid b) &= \mathcal{L}(a) \cup \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cup \{\mathrm{b}\} \\
&= \{\mathrm{a}, \mathrm{b}\}
\end{aligned}
$$

Computing the Semantics

$$
\begin{aligned}
\mathcal{L}(a) &= \{\mathrm{a}\} \\
\mathcal{L}(a \mid b) &= \mathcal{L}(a) \cup \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cup \{\mathrm{b}\} \\
&= \{\mathrm{a}, \mathrm{b}\} \\
\mathcal{L}(ab) &= \mathcal{L}(a) \cdot \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cdot \{\mathrm{b}\} \\
&= \{\mathrm{ab}\}
\end{aligned}
$$

Computing the Semantics

$$
\begin{aligned}
\mathcal{L}(a) &= \{\mathrm{a}\} \\
\mathcal{L}(a \mid b) &= \mathcal{L}(a) \cup \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cup \{\mathrm{b}\} \\
&= \{\mathrm{a}, \mathrm{b}\} \\
\mathcal{L}(ab) &= \mathcal{L}(a) \cdot \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cdot \{\mathrm{b}\} \\
&= \{\mathrm{ab}\} \\
\mathcal{L}((a \mid b)(a \mid b)) &= \mathcal{L}(a \mid b) \cdot \mathcal{L}(a \mid b) \\
&= \{\mathrm{a}, \mathrm{b}\} \cdot \{\mathrm{a}, \mathrm{b}\} \\
&= \{\mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}\}
\end{aligned}
$$

Computing the Semantics of Closure

$$\mathcal{L}(r^*) \quad = \quad \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))$$

Computing the Semantics of Closure

Example: $\mathcal{L}((a \mid b)^*)$

$$= \{\epsilon\} \cup (\mathcal{L}(a \mid b) \cdot \mathcal{L}((a \mid b)^*))$$

$$\begin{aligned}
L_0 &= \{\epsilon\} && \textit{Base case} \\
L_1 &= \{\epsilon\} \cup (\{\mathrm{a}, \mathrm{b}\} \cdot L_0) \\
&= \{\epsilon\} \cup (\{\mathrm{a}, \mathrm{b}\} \cdot \{\epsilon\}) \\
&= \{\epsilon, \mathrm{a}, \mathrm{b}\} \\
L_2 &= \{\epsilon\} \cup (\{\mathrm{a}, \mathrm{b}\} \cdot L_1) \\
&= \{\epsilon\} \cup (\{\mathrm{a}, \mathrm{b}\} \cdot \{\epsilon, \mathrm{a}, \mathrm{b}\}) \\
&= \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}\} \\
&\vdots
\end{aligned}$$

Another Example: $\mathcal{L}((a^*b^*)^*)$

$$
\begin{aligned}
\mathcal{L}(a^*) &= \{\epsilon, \mathrm{a}, \mathrm{aa}, \ldots\} \\
\mathcal{L}(b^*) &= \{\epsilon, \mathrm{b}, \mathrm{bb}, \ldots\} \\
\mathcal{L}(a^*b^*) &= \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{bb}, \mathrm{aaa}, \mathrm{aab}, \mathrm{abb}, \mathrm{bbb}, \ldots\} \\
\mathcal{L}((a^*b^*)^*) &= \{\epsilon\} \\
& \quad \cup \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{bb}, \mathrm{aaa}, \mathrm{aab}, \mathrm{abb}, \mathrm{bbb}, \ldots\} \\
& \quad \cup \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}, \mathrm{aaa}, \mathrm{aab}, \mathrm{aba}, \mathrm{abb}, \mathrm{baa}, \mathrm{bab}, \mathrm{bba}, \mathrm{bbb}, \ldots\} \\
& \quad \vdots \\
&= \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}, \ldots\}
\end{aligned}
$$

Regular Definitions

Assign “names” to regular expressions.

For example,

$$\begin{array}{rcl} \text{digit} & \longrightarrow & 0 \mid 1 \mid \cdots \mid 9 \\ \text{natural} & \longrightarrow & \text{digit digit}^* \end{array}$$

SHORTHANDS:

- a^+: Set of strings with one or more occurrences of a.
- $a^?$: Set of strings with zero or one occurrences of a.

Example:

$$\text{integer} \longrightarrow (+|-)^?\text{digit}^+$$

Regular Definitions: Examples

$$
\begin{aligned}
\text{float} &\longrightarrow \text{integer} \;\textbf{.}\; \text{fraction} \\
\text{integer} &\longrightarrow (+|-)^{?}\ \text{no\_leading\_zero} \\
\text{no\_leading\_zero} &\longrightarrow (\text{nonzero\_digit digit}^{*}) \mid 0 \\
\text{fraction} &\longrightarrow \text{no\_trailing\_zero exponent}^{?} \\
\text{no\_trailing\_zero} &\longrightarrow (\text{digit}^{*}\ \text{nonzero\_digit}) \mid 0 \\
\text{exponent} &\longrightarrow (\mathrm{E} \mid \mathrm{e})\ \text{integer} \\
\text{digit} &\longrightarrow 0 \mid 1 \mid \cdots \mid 9 \\
\text{nonzero\_digit} &\longrightarrow 1 \mid 2 \mid \cdots \mid 9
\end{aligned}
$$

Regular Definitions and Lexical Analysis

Regular Expressions and Definitions *specify* sets of strings over an input alphabet.

- They can hence be used to specify the set of *lexemes* associated with a *token*.
  - Used as the *pattern* language

How do we decide whether an input string belongs to the set of strings specified by a regular expression?

Lexical Analysis

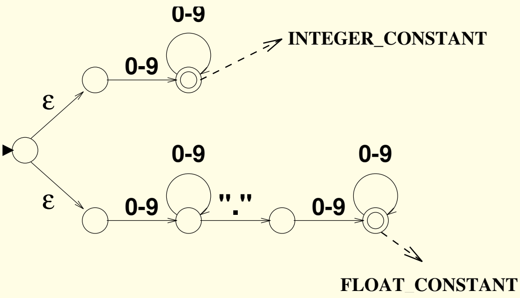
- Regular Expressions and Definitions are used to specify the set of strings (lexemes) corresponding to a *token*.
- An automaton (DFA/NFA) is built from the above specifications.
- Each final state is associated with an *action*: emit the corresponding token.

Specifying Lexical Analysis

Consider a recognizer for integers (sequence of digits) and floats (sequence of digits separated by a decimal point).

```
[0-9]+                  { emit(INTEGER_CONSTANT); }

[0-9]+"."[0-9]+         { emit(FLOAT_CONSTANT); }
```



Lex

Tool for building lexical analyzers.

Input: lexical specifications (`.l` file)

Output: C function (`yylex`) that returns a token on each invocation.

```
%%
[0-9]+                { return(INTEGER_CONSTANT); }

[0-9]+"."[0-9]+       { return(FLOAT_CONSTANT); }
```

Tokens are simply integers (`#define`'s).

Lex Specifications

```
%{
    C/C++ header statements for inclusion
%}
  Regular Definitions   e.g.:
    digit    [0-9]
%%

  Token Specifications   e.g.:
    {digit}+            { return(INTEGER_CONSTANT); }

%%
  Support functions in C
```

Regular Expressions in Lex

Adds "syntactic sugar" to regular expressions:

- Range: `[0-7]`: Integers from 0 through 7 (inclusive)

  `[a-nx-zA-Q]`: Letters a thru n, x thru z and A thru Q.

- Exception: `[^/]`: Any character other than `/`.
- Definition: `{digit}`: Use the previously specified regular definition `digit`.
- Special characters: Connectives of regular expression, convenience features.

  e.g.: `|` `*` `^`

Special Characters in Lex

| * + ? ( )	Same as in regular expressions
[ ]	Enclose ranges and exceptions
{ }	Enclose "names" of regular definitions
^	Used to negate a specified range (in Exception)
.	Match any single character except newline
\	Escape the next character
\n, \t	Newline and Tab

For literal matching, enclose special characters in double quotes (") *e.g.:* "*"

Or use \ to escape. *e.g.:* \"

Examples

`for`	Sequence of f, o, r
`"||"`	C-style OR operator (two vert. bars)
`.*`	Sequence of non-newline characters
`[^*/]+`	Sequence of characters except * and /
`\"[^"]*\"`	Sequence of non-quote characters beginning and ending with a quote
`({letter}|"_")({letter}|{digit}|"_")*`	C-style identifiers

A Complete Example

```
%{
#include <stdio.h>
#include "tokens.h"
%}
digit  [0-9]
hexdigit  [0-9a-f]
%%

"+"                  { return(PLUS); }
"-"                  { return(MINUS); }
{digit}+             { return(INTEGER_CONSTANT); }
{digit}+"."{digit}+  { return(FLOAT_CONSTANT); }
.                    { return(SYNTAX_ERROR); }
%%
```

Actions

Actions are attached to final states.

- Distinguish the different final states.
- Used to return *tokens*.
- Can be used to set *attribute values*.
- Fragment of C code (blocks enclosed by '{' and '}').

Attributes

Additional information about a token's lexeme.

- Stored in variable `yylval`
- Type of attributes (usually a union) specified by `YYSTYPE`
- Additional variables:
  - `yytext`: Lexeme (*Actual text string*)
  - `yyleng`: length of string in `yytext`
  - ▷ `yylineno`: Current line number (number of '\n' seen thus far)
    - enabled by `%option yylineno`

Priority of matching

What if an input string matches more than one pattern?

```
"if"                 { return(TOKEN_IF); }
{letter}+            { return(TOKEN_ID); }
"while"              { return(TOKEN_WHILE); }
```

- A pattern that matches the longest string is chosen.

  Example: `ifs` is matched with an identifier, not the keyword `if`.

- Of patterns that match strings of same length, the first (from the top of file) is chosen.
  - `while` is matched as an identifier, not the keyword `while`.
  - Given `if1`, a match will be announced for the keyword `if`, with `1` being considered as part of the next token.

Constructing Scanners using (f)lex

- Scanner specifications: *specifications*.l

$$\textit{specifications}.\texttt{l} \xrightarrow{\texttt{(f)lex}} \texttt{lex.yy.c}$$

- Generated scanner in lex.yy.c

$$\texttt{lex.yy.c} \xrightarrow{\texttt{(g)cc}} \textit{executable}$$

  - yywrap(): hook for signalling end of file.
  - Use -lfl (flex) or -ll (lex) flags at link time to include default function yywrap() that always returns 1.

Recognizers

Construct *automata* that recognize strings belonging to a language.

- Finite State Automata $\Rightarrow$ Regular Languages
  - Finite State $\rightarrow$ cannot maintain arbitrary counts.
- Push Down Automata $\Rightarrow$ Context-free Languages
  - Stack is used to maintain counter, but only one counter can go arbitrarily high.

Finite State Automata

Represented by a labeled directed graph.

- A finite set of *states* (vertices).
- *Transitions* between states (edges).
- *Labels* on transitions are drawn from $\Sigma \cup \{\epsilon\}$.
- One distinguished *start* state.
- One or more distinguished *final* states.

Finite State Automata: An Example

Consider the Regular Expression $(\boldsymbol{a} \mid \boldsymbol{b})^*\boldsymbol{a}(\boldsymbol{a} \mid \boldsymbol{b})$.

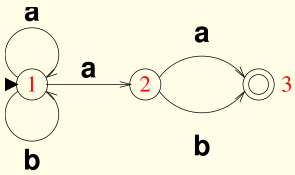
$$\mathcal{L}((a \mid b)^*a(a \mid b)) = \{\mathrm{aa}, \mathrm{ab}, \mathrm{aaa}, \mathrm{aab}, \mathrm{baa}, \mathrm{bab}, \mathrm{aaaa}, \mathrm{aaab}, \mathrm{abaa}, \mathrm{abab}, \mathrm{baaa}, \ldots\}.$$

Finite State Automata: An Example

Consider the Regular Expression $(\boldsymbol{a} \mid \boldsymbol{b})^*\boldsymbol{a}(\boldsymbol{a} \mid \boldsymbol{b})$.

$\mathcal{L}((a \mid b)^*a(a \mid b)) = \{\mathrm{aa}, \mathrm{ab}, \mathrm{aaa}, \mathrm{aab}, \mathrm{baa}, \mathrm{bab},$
$\mathrm{aaaa}, \mathrm{aaab}, \mathrm{abaa}, \mathrm{abab}, \mathrm{baaa}, \ldots\}.$

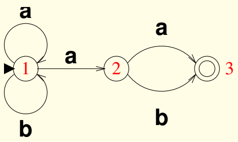
The following automaton determines whether an input string belongs to $\mathcal{L}((a \mid b)^*a(a \mid b)$:



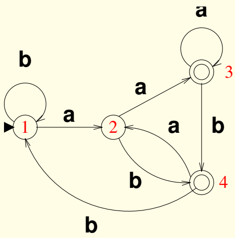
Deterministic Vs Nondeterministic FSA

$(a \mid b)^*a(a \mid b)$:

Nondeterministic:
(NFA)



Deterministic:
(DFA)



Acceptance Criterion

A finite state automaton (NFA or DFA) *accepts* an input string x

... if beginning from the start state

... we can trace some path through the automaton

... such that the sequence of edge labels spells x

... and end in a final state.

Or, there exists a path in the graph from the start state to a final state such that the sequence of labels on the path spells out x

NFA vs. DFA

For every NFA, there is a DFA that accepts the same set of strings.

- NFA may have transitions labeled by ϵ.

  (Spontaneous transitions)

- All transition labels in a DFA belong to Σ.
- For some string x, there may be *many* accepting paths in an NFA.
- For all strings x, there is *one unique* accepting path in a DFA.
- Usually, an input string can be recognized *faster* with a DFA.
- NFAs are typically *smaller* than the corresponding DFAs.

NFA vs. DFA

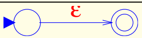
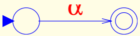
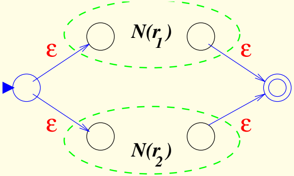
R = Size of Regular Expression

N = Length of Input String

	NFA	DFA
Size of Automaton	$O(R)$	$O(2^R)$
Recognition time per input string	$O(N \times R)$	$O(N)$

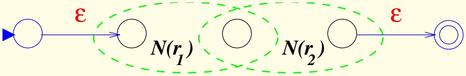
Regular Expressions to NFA

Thompson's Construction: For every regular expression r, derive an NFA $N(r)$ with unique start and final states.

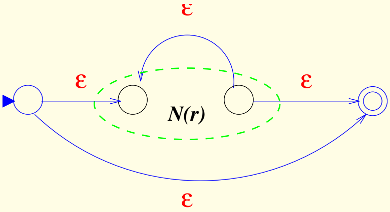
ϵ	
$\alpha \in \Sigma$	
$(r_1 \mid r_2)$	

Regular Expressions to NFA (contd.)

$r_1 r_2$

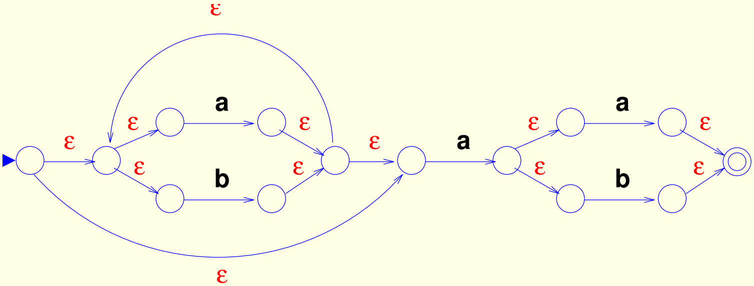


r^*



Example

$(a \mid b)^* a (a \mid b)$:



Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs

Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs

Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs
- *Implication:* REs and NFAs have the same expressive power

Expressive Power of RE Vs FSA

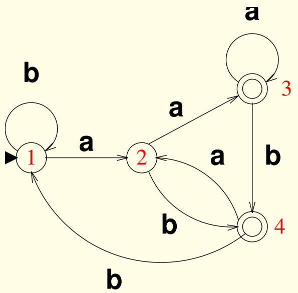
- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs
- *Implication:* REs and NFAs have the same expressive power
- Where do DFAs stand?
  - Every DFA is an NFA
  - We will show that every NFA can be converted into an equivalent DFA

Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs
- *Implication:* REs and NFAs have the same expressive power
- Where do DFAs stand?
  - Every DFA is an NFA
  - We will show that every NFA can be converted into an equivalent DFA
- **Implication:** RE, NFA and DFA are equivalent

Recognition with a DFA

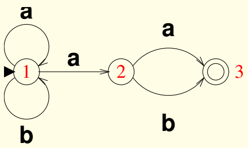
Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	b	a	b	
Path:	1	2	4	2	4	Accept

Recognition with an NFA

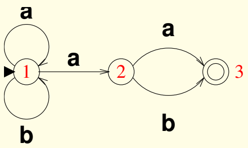
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:	a	b	a	b
Path 1: 1				

Recognition with an NFA

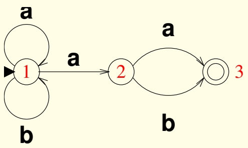
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1			

Recognition with an NFA

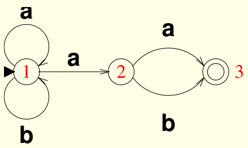
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1		

Recognition with an NFA

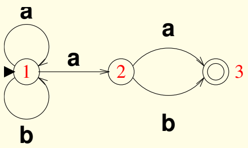
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1	1	

Recognition with an NFA

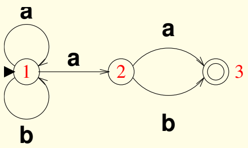
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1	1	1

Recognition with an NFA

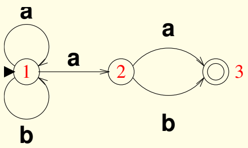
Is abab $\in \mathcal{L}((a \mid b)^*a(a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1	1	1
Path 2:	1	1	1		

Recognition with an NFA

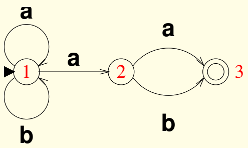
Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1	1	1
Path 2:	1	1	1	2	

Recognition with an NFA

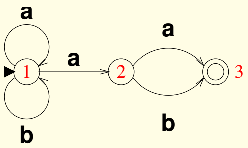
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept

Recognition with an NFA

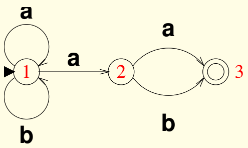
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	

Recognition with an NFA

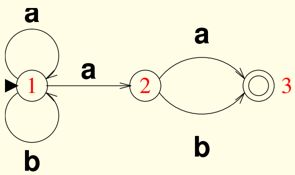
Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\bot$	$\bot$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,3\}$	$\{1,2\}$	$\{1,3\}$	Accept

Recognition with an NFA (contd.)

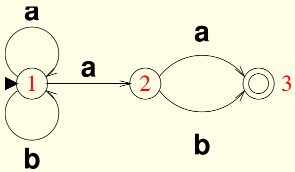
Is aaab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	a	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	1	2	
Path 3:	1	1	1	2	3	Accept
Path 4:	1	1	2	3	$\perp$	
Path 5:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	Accept

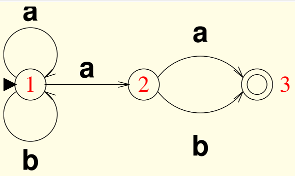
Recognition with an NFA (contd.)

Is aabb $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	a	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	2	3	$\perp$	
Path 3:	1	2	3	$\perp$	$\perp$	
All Paths	{1}	{1, 2}	{1, 2, 3}	{1, 3}	{1}	REJECT

Converting NFA to DFA



Converting NFA to DFA (contd.)

Subset construction

Given a set S of NFA states,

- compute $S_\epsilon = \epsilon\text{-closure}(S)$: S_ϵ is the set of all NFA states reachable by zero or more ϵ-transitions from S.
- compute $S_\alpha = \text{goto}(S, \alpha)$:
  - S' is the set of all NFA states reachable from S by taking a transition labeled α.
  - $S_\alpha = \epsilon\text{-closure}(S')$.

Converting NFA to DFA (contd).

Each state in DFA corresponds to a *set of states* in NFA.

Start state of DFA = ϵ-closure(start state of NFA).

From a state s in DFA that corresponds to a set of states S in NFA:

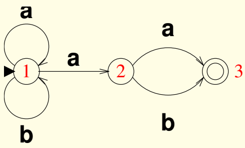
add a transition labeled α to state s' that corresponds to a non-empty S' in NFA,

such that $S' = \text{goto}(S, \alpha)$.

s is a state in DFA such that the corresponding set of states S in NFA contains a final state of NFA,

$\Leftarrow$ s is a final state of DFA

NFA → DFA: An Example



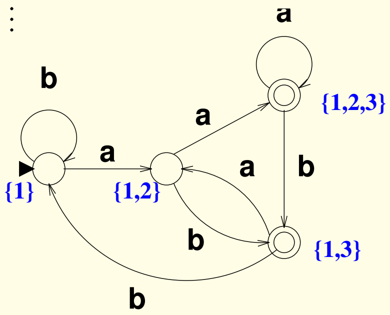
$$
\begin{aligned}
\epsilon\text{-closure}(\{1\}) &= \{1\} \\
\text{goto}(\{1\}, a) &= \{1, 2\} \\
\text{goto}(\{1\}, b) &= \{1\} \\
\text{goto}(\{1, 2\}, a) &= \{1, 2, 3\} \\
\text{goto}(\{1, 2\}, b) &= \{1, 3\} \\
\text{goto}(\{1, 2, 3\}, a) &= \{1, 2, 3\} \\
&\vdots
\end{aligned}
$$

NFA → DFA: An Example (contd.)

$$
\begin{array}{lcl}
\epsilon\text{-closure}(\{1\}) & = & \{1\} \\
\text{goto}(\{1\}, a) & = & \{1, 2\} \\
\text{goto}(\{1\}, b) & = & \{1\} \\
\text{goto}(\{1, 2\}, a) & = & \underline{\{1, 2, 3\}} \\
\text{goto}(\{1, 2\}, b) & = & \underline{\{1, 3\}} \\
\text{goto}(\{1, 2, 3\}, a) & = & \underline{\{1, 2, 3\}} \\
\text{goto}(\{1, 2, 3\}, b) & = & \{1\} \\
\text{goto}(\{1, 3\}, a) & = & \{1, 2\} \\
\text{goto}(\{1, 3\}, b) & = & \{1\}
\end{array}
$$

NFA → DFA: An Example (contd.)

$$
\begin{aligned}
\text{goto}(\{1\}, a) &= \{1, 2\} \\
\text{goto}(\{1\}, b) &= \{1\} \\
\text{goto}(\{1, 2\}, a) &= \{1, 2, 3\} \\
\text{goto}(\{1, 2\}, b) &= \{1, 3\} \\
\text{goto}(\{1, 2, 3\}, a) &= \{1, 2, 3\} \\
&\vdots
\end{aligned}
$$



Converting RE to FSA

NFA: Compile RE to NFA (Thompson's construction [1968]), then match.

DFA: Compile to DFA, then match

(A) Convert NFA to DFA (Rabin-Scott construction), minimize

(B) Direct construction: RE derivatives [Brzozowski 1964].

- More convenient and a bit more general than (A).

(C) Direct construction of [McNaughton Yamada 1960]

- Can be seen as a (more easily implemented) specialization of (B).
- Used in Lex and its derivatives, i.e., most compilers use this algorithm.

Converting RE to FSA

- NFA approach takes $O(n)$ NFA construction plus $O(nm)$ matching, so has worst case $O(nm)$ complexity.
- DFA approach takes $O(2^n)$ construction plus $O(m)$match, so has worst case $O(2^n + m)$ complexity.
- So, why bother with DFA?
  - In many practical applications, the pattern is fixed and small, while the subject text is very large. So, the $O(mn)$ term is dominant over $O(2^n)$
  - For many important cases, DFAs are of polynomial size
  - In many applications, exponential blow-ups don't occur, e.g., compilers.

Derivative of Regular Expressions

The derivative of a regular expression R w.r.t. a symbol x, denoted $\partial_x[R]$ is another regular expression R' such that $\mathcal{L}(R) = \mathcal{L}(xR')$

Basically, $\partial_x[R]$ captures the suffixes of those strings that match R and start with x.

Examples

- $\partial_a[a(b|c)] = b|c$
- $\partial_a[(a|b)cd] = cd$
- $\partial_a[(a|b)*cd] = (a|b)*cd$
- $\partial_c[(a|b)*cd] = d$
- $\partial_d[(a|b)*cd] = \emptyset$

Definition of RE Derivative (1)

inclEps(R): A predicate that returns true if $\epsilon \in \mathcal{L}(R)$

$$\begin{aligned}
inclEps(a) &= false, \quad \forall a \in \Sigma \\
inclEps(R_1|R_2) &= inclEps(R_1) \vee inclEps(R_2) \\
inclEps(R_1R_2) &= inclEps(R_1) \wedge inclEps(R_2) \\
inclEps(R*) &= true
\end{aligned}$$

Note *inclEps* can be computed in linear-time.

Definition of RE Derivative (2)

$$
\begin{aligned}
\partial_a[a] &= \epsilon \\
\partial_a[b] &= \emptyset \\
\partial_a[R_1|R_2] &= \partial_a[R_1]|\partial_a[R_2] \\
\partial_a[R*] &= \partial_a[R]R* \\
\partial_a[R_1R_2] &= \partial_a[R_1]R_2|\partial_a[R_2] \quad \text{if } \mathit{inclEps}(R_1) \\
&= \partial_a[R_1]R_2 \qquad \text{otherwise}
\end{aligned}
$$

Note: $\boxed{\mathcal{L}(\epsilon) = \{\epsilon\}} \neq \boxed{\mathcal{L}(\emptyset) = \{\}}$

DFA Using Derivatives: Illustration

Consider $R_1 = (a|b)* a(a|b)$

$\partial_a[R_1] = R_1|(a|b) = R_2$

$\partial_b[R_1] = R_1$

$\partial_a[R_2] = R_1|(a|b)|\epsilon = R_3$

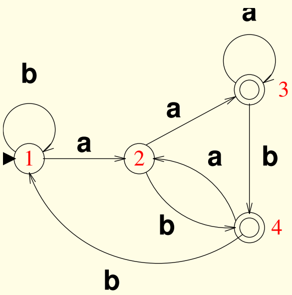
$\partial_b[R_2] = R_1|\epsilon = R_4$

$\partial_a[R_3] = R_1|(a|b)|\epsilon = R_3$

$\partial_b[R_3] = R_1|\epsilon = R_4$

$\partial_a[R_4] = R_1|(a|b) = R_2$

$\partial_b[R_4] = R_1$

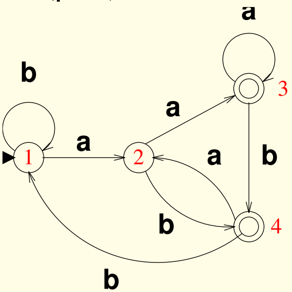


McNaughton-Yamada Construction

Can be viewed as a simpler way to represent derivatives

- Positions in RE are numbered, e.g., $^0(a^1|b^2)*\, a^3(a^4|b^5)\$^6$.
- A derivative is identified by its beginning position in the RE
  - Or more generally, a derivative is identified by a set of positions
- Each DFA state corresponds to a position set (pset)

$$R_1 \equiv \{1,2,3\}$$
$$R_2 \equiv \{1,2,3,4,5\}$$
$$R_3 \equiv \{1,2,3,4,5,6\}$$
$$R_4 \equiv \{1,2,3,6\}$$



McNaughton-Yamada: Definitions

first(P): Yields the set of first symbols of RE denoted by pset P

Determines the transitions out of DFA state for P

Example: For the RE $(a^1|b^2)*\,a^3(a^4|b^5)\$^6$, $\quad first(\{1,2,3\}) = \{a, b\}$

$P|_s$: Subset of P that contain s, i.e., $\{p \in P \mid R \text{ contains } s \text{ at } p\}$

Example: $\{1,2,3\}|_a = \{1,3\}$, $\{1,2,4,5\}|_b = \{2,5\}$

follow(P): set of positions immediately after P, i.e., $\bigcup_{p\in P} follow(\{p\}))$

Definition is very similar to derivatives

Example: $follow(\{3,4\}) = \{4,5,6\}$

$follow(\{1\}) = \{1,2,3\}$

McNaughton-Yamada Construction (2)

$BuildMY(R, pset)$

Create an automaton state S labeled $pset$

Mark this state as final if \$ occurs in R at $pset$

foreach symbol $x \in first(pset) - \{\$\}$ **do**

  Call $BuildMY(R, follow(pset|_x))$ if hasn't previously been called

  Create a transition on x from S to

    the root of this subautomaton

DFA construction begins with the call $BuildMY(R, follow(\{0\}))$. The root of the resulting automaton is marked as a start state.

BuildMY Illustration on $R = {}^0(a^1|b^2)* a^3(a^4|b^5)\$^6$

Computations Needed

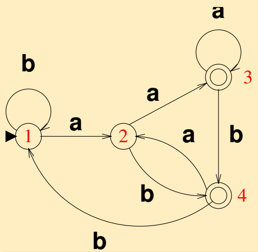
$follow(\{0\}) = \{1, 2, 3\}$
$follow(\{1\}) = follow(\{2\}) = \{1, 2, 3\}$
$follow(\{3\}) = \{4, 5\}$
$follow(\{4\}) = follow(\{5\}) = \{6\}$

$\{1, 2, 3\}|_a = \{1, 3\}, \quad \{1, 2, 3\}|_b = \{2\}$
$follow(\{1, 3\}) = \{1, 2, 3, 4, 5\}$

$\{1, 2, 3, 4, 5\}|_a = \{1, 3, 4\}$
$\{1, 2, 3, 4, 5\}|_b = \{2, 5\}$
$follow(\{1, 3, 4\}) = \{1, 2, 3, 4, 5, 6\}$
$follow(\{2, 5\}) = \{1, 2, 3, 6\}$

$\{1, 2, 3, 4, 5, 6\}|_a = \{1, 3, 4\}$
$\{1, 2, 3, 4, 5, 6\}|_b = \{2, 5\}$
$\{1, 2, 3, 6\}|_a = \{1, 3\} \quad \{1, 2, 3, 6\}|_b = \{2\}$

Resulting Automaton



State	Pset
1	{1,2,3}
2	{1,2,3,4,5}
3	{1,2,3,4,5,6}
4	{1,2,3,6}

McNaughton-Yamada (MY) Vs Derivatives

- Conceptually very similar
- MY takes a bit longer to describe, and its correctness a bit harder to follow.
- MY is also more mechanical, and hence is found in most implementations
- Derivatives approach is more general
  - Can support some extensions to REs, e.g., complement operator
  - Can avoid some redundant states during construction
    - Example: For $ac|bc$, DFA built by derivative approach has 3 states, but the one built by MY construction has 4 states

      The derivative approach merges the two c's in the RE, but with MY, the two c's have different positions, and hence operations on them are not shared.

Avoiding Redundant States

- Automata built by MY is not optimal
  - Automata minimization algorithms can be used to produce an optimal automaton.
- Derivatives approach associates DFA states with derivatives, but does not say how to determine equality among derivatives.
- There is a spectrum of techniques to determine RE equality
  - MY is the simplest: relies on syntactic identity
  - At the other end of the spectrum, we could use a complete decision procedure for RE equality.
    - In this case, the derivative approach yields the optimal RE!
  - In practice we would tend to use something in the middle
    - Trade off some power for ease/efficiency of implementation

RE to DFA conversion: Complexity

- Given DFA size can be exponential in the worst case, we obviously must accept worst-case exponential complexity.
- For the derivatives approach, it is not immediately obvious that it even terminates!
  - More obvious for McNaughton-Yamada approach, since DFA states correspond to position sets, of which there are only 2^n.
- Derivative computation is linear in RE size in the general case.
- So, overall complexity is $O(n2^n)$
- Complexity can be improved, but the worst-case 2^n takes away some of the rationale for doing so.
  - Instead, we focus on improving performance in many frequently occurring special cases where better complexity is achievable.

Using States in Lex

- Some regular languages are more easily expressed as FSA
  - Set of all strings representing binary numbers divisible by 3
- Lex allows you to use FSA concepts using *start states*

```
%x MOD1 MOD2
"0" { }
"1" {BEGIN MOD1}
<MOD1> "0"  {BEGIN MOD2}
<MOD1> "1"  {BEGIN 0}
```

Other Special Directives

- ECHO causes Lex to echo current lexeme
- REJECT causes abandonment of current match in favor of the next.
- Example

```
a|
ab|
abc|
abcd {ECHO; REJECT;}
.|\n {/* eat up the character */}
```

Implementing a Scanner

$$transition : state \times \Sigma \rightarrow state$$

```
algorithm scanner() {
    current_state = start state;
    while (1) {
        c = getc(); /* on end of file, ... */
        if defined(transition(current_state, c))
            current_state = transition(current_state, c);
        else
            return s;
    }
}
```

Implementing a Scanner (contd.)

Implementing the *transition* function:

- Simplest: 2-D array.

  Space inefficient.

- Traditionally compressed using row/colum equivalence. (default on `(f)lex`)

  Good space-time tradeoff.

- Further table compression using various techniques:
  - Example: RDM (Row Displacement Method):

    Store rows in overlapping manner using 2 1-D arrays.

  Smaller tables, but longer access times.

Lexical Analysis: A Summary

Convert a stream of characters into a stream of tokens.

- Make rest of compiler independent of character set
- Strip off comments
- Recognize line numbers
- Ignore white space characters
- Process macros (definitions and uses)
- Interface with **symbol** (name) **table**.

Parsing

A.k.a. Syntax Analysis

- Recognize *sentences* in a language.
- Discover the structure of a document/program.
- Construct (implicitly or explicitly) a tree (called as a parse tree) to represent the structure.
- The above tree is used later to guide the translation.

Grammars

The syntactic structure of a language is defined using *grammars*.

- Grammars (like regular expressions) specify a set of strings over an alphabet.
- Efficient *recognizers* (like DFA) can be constructed to efficiently determine whether a string is in the language.
- Language hierarchy:
  - Finite Languages (FL)

    Enumeration
  - Regular Languages (RL $\supset$ FL)

    Regular Expressions
  - Context-free Languages (CFL $\supset$ RL)

    Context-free Grammars

Regular Languages

Languages represented by regular expressions	$\equiv$	Languages recognized by finite automata

Examples:

- $\surd$ $\{a, b, c\}$
- $\surd$ $\{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$
- $\surd$ $\{(ab)^n \mid n \geq 0\}$
- $\times$ $\{a^n b^n \mid n \geq 0\}$

Grammars

Notation where recursion is explicit. Examples

- $\{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}, \ldots\}$:

$$\begin{aligned} E &\longrightarrow \mathrm{a} \\ E &\longrightarrow \mathrm{b} \\ S &\longrightarrow \epsilon \\ S &\longrightarrow ES \end{aligned}$$

  Notational shorthand:

$$\begin{aligned} E &\longrightarrow \mathrm{a} \mid \mathrm{b} \\ S &\longrightarrow \epsilon \mid ES \end{aligned}$$

- $\{\mathrm{a}^n\mathrm{b}^n \mid n \geq 0\}$:

$$\begin{aligned} S &\longrightarrow \epsilon \\ S &\longrightarrow \mathrm{a}S\mathrm{b} \end{aligned}$$

- $\{w \mid \textit{no. of } \mathrm{a}\textit{'s in } w = \textit{no. of } \mathrm{b}\textit{'s in } w\}$

Context-free Grammars

- Terminal Symbols: Tokens
- Nonterminal Symbols: set of strings made up of tokens
- Productions: Rules for constructing the set of strings associated with non-terminal symbols.

  Example: $\mathit{Stmt} \longrightarrow$ while Expr do Stmt

Start symbol: nonterminal symbol that represents the set of all strings in the language.

Example

$$
\begin{aligned}
E &\longrightarrow E + E \\
E &\longrightarrow E - E \\
E &\longrightarrow E * E \\
E &\longrightarrow E \,/\, E \\
E &\longrightarrow (\, E \,) \\
E &\longrightarrow \mathrm{id}
\end{aligned}
$$

$$\mathcal{L}(E) = \{\mathrm{id}, \mathrm{id} + \mathrm{id}, \mathrm{id} - \mathrm{id}, \ldots, \mathrm{id} + (\mathrm{id} * \mathrm{id}) - \mathrm{id}, \ldots\}$$

Context-free Grammars

Production: rule with *non-terminal* symbol on left hand side, and a (possibly empty) sequence of terminal or non-terminal symbols on the right-hand side.

Notations:

- Terminals: lower case letters, digits, punctuation
- Nonterminals: Upper case letters
- Arbitrary Terminals/Nonterminals: X, Y, Z
- Strings of Terminals: u, v, w
- Strings of Terminals/Nonterminals: α, β, γ
- Start Symbol: S

Context-Free Vs Other Types of Grammars

- Context-free grammar (CFG): Productions of the form $NT \longrightarrow [NT|T]*$
- Context-sensitive grammar (CSG): Productions of the form $[t|NT]*NT[t|NT]* \longrightarrow [t|NT]*$
- Unrestricted grammar: Productions of the form $[t|NT]* \longrightarrow [t|NT]*$

Examples of Non-Context-Free Languages

- Checking that variables are declared before use. If we simplify and abstract the problem, we see that it amounts to recognizing strings of the form wsw
- Checking whether the number of actual and formal parameters match. Abstracts to recognizing strings of the form $a^n b^m c^n d^m$
- In both cases, the rules are not enforced in grammar but deferred to type-checking phase
- Note: Strings of the form wsw^R and $a^n b^n c^m d^m$ can be described by a CFG

What types of Grammars Describe These Languages?

- Strings of 0's and 1's of form xx
- Strings of 0's and 1's in which 011 doesn't occur
- Strings of 0's and 1's in which each 0 is immediately followed by a 1
- Strings of 0's and 1's with ithe equal number of 0's and 1's.

Language Generated by Grammars, Equivalence of Grammars

- How to show that a grammar G generates a language $\mathcal{M}$? Show that
  - $\forall s \in \mathcal{M}$, show that $s \in \mathcal{L}(G)$
  - $\forall s \in \mathcal{L}(G)$, show that $s \in \mathcal{M}$
- How to establish that two grammars G_1 and G_2 are equivalent?
  Show that $\mathcal{L}(G_1) = \mathcal{L}(G_2)$

Grammar Examples

$$S \longrightarrow 0S1S|1S0S|\epsilon$$

What is the language generated by this grammar?

Grammar Examples

$$S \longrightarrow 0A|1B|\epsilon$$

$$A \longrightarrow 0AA|1S$$

$$B \longrightarrow 1BB|0S$$

What is the language generated by this grammar?

The Two Sides of Grammars

Specify a set of strings in a language.

Recognize strings in a given language:

- Is a given string x in the language?

  Yes, if we can construct a *derivation* for x

- Example: Is $\mathrm{id} + \mathrm{id} \in \mathcal{L}(E)$?

$$\begin{aligned} \mathrm{id} + \mathrm{id} &\Longleftarrow E + \mathrm{id} \\ &\Longleftarrow E + E \\ &\Longleftarrow E \end{aligned}$$

Derivations

Grammar:

$$\begin{array}{lcl} E & \longrightarrow & E + E \\ E & \longrightarrow & \mathrm{id} \end{array}$$

E **derives** id + id:

$$\begin{array}{lcl} E & \Longrightarrow & E + E \\ & \Longrightarrow & E + \mathrm{id} \\ & \Longrightarrow & \mathrm{id} + \mathrm{id} \end{array}$$

- $\alpha A \beta \Longrightarrow \alpha \gamma \beta$ iff $A \longrightarrow \gamma$ is a production in the grammar.
- $\alpha \stackrel{*}{\Longrightarrow} \beta$ if α derives β in zero or more steps.

  Example: $E \stackrel{*}{\Longrightarrow} \mathrm{id} + \mathrm{id}$
- Sentence: A sequence of terminal symbols w such that $S \stackrel{+}{\Longrightarrow} w$ (where S is the start symbol)
- Sentential Form: A sequence of terminal/nonterminal symbols α such that $S \stackrel{*}{\Longrightarrow} \alpha$

Derivations

- Rightmost derivation: Rightmost non-terminal is replaced first:

$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow E + \text{id} \\ &\Longrightarrow \text{id} + \text{id} \end{aligned}$$

  Written as $E \overset{*}{\Longrightarrow}_{rm} \text{id} + \text{id}$

- Leftmost derivation: Leftmost non-terminal is replaced first:

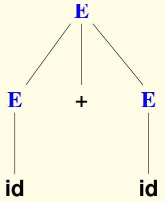
$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow \text{id} + E \\ &\Longrightarrow \text{id} + \text{id} \end{aligned}$$

  Written as $E \overset{*}{\Longrightarrow}_{lm} \text{id} + \text{id}$

Parse Trees

Graphical Representation of Derivations

$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow \mathrm{id} + E \\ &\Longrightarrow \mathrm{id} + \mathrm{id} \end{aligned}$$



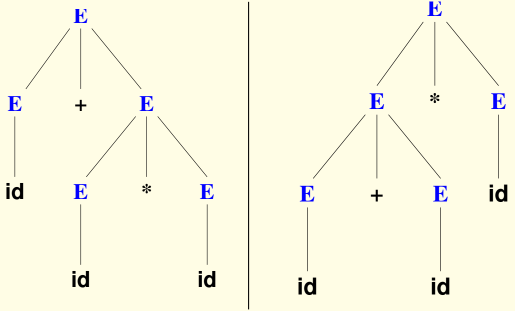
$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow E + \mathrm{id} \\ &\Longrightarrow \mathrm{id} + \mathrm{id} \end{aligned}$$

A **Parse Tree** succinctly captures the *structure* of a sentence.

Ambiguity

A Grammar is *ambiguous* if there are multiple parse trees for the same sentence.

Example: $\mathrm{id} + \mathrm{id} * \mathrm{id}$

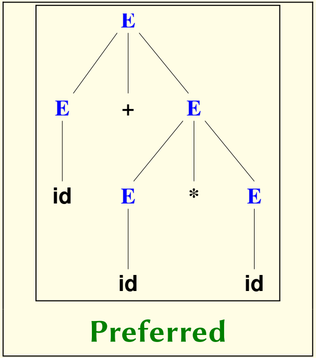


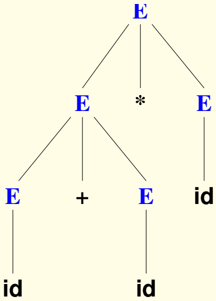
Disambiguition

Express Preference for one parse tree over others.

Example: id $+$ id $*$ id

The usual precedence of $*$ over $+$ means:

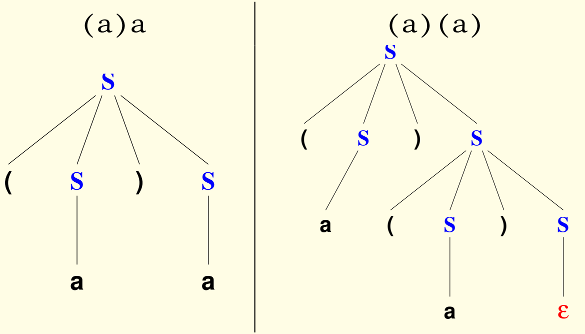




Parsing

Construct a parse tree for a given string.

$$S \longrightarrow (S)S$$
$$S \longrightarrow a$$
$$S \longrightarrow \epsilon$$



A Procedure for Parsing

Grammar: $S \longrightarrow a$

```
procedure parse_S() {
    switch (input_token) {
        case TOKEN_a:
            consume(TOKEN_a);
            return;
        default:
            /* Parse Error */
    }
}
```

Predictive Parsing

Grammar:

$$S \longrightarrow a$$
$$S \longrightarrow \epsilon$$

```
procedure parse_S() {
    switch (input_token) {
        case TOKEN_a:   /* Production 1 */
            consume(TOKEN_a);
            return;
        case TOKEN_EOF:   /* Production 2 */
            return;
        default:
            /* Parse Error */
    }
}
```

Predictive Parsing (contd.)

Grammar:

$$
\begin{aligned}
S &\longrightarrow (S)S \\
S &\longrightarrow a \\
S &\longrightarrow \epsilon
\end{aligned}
$$

```
procedure parse_S() {
    switch (input_token) {
        case TOKEN_OPEN_PAREN:   /* Production 1 */
            consume(TOKEN_OPEN_PAREN);
            parse_S();
            consume(TOKEN_CLOSE_PAREN);
            parse_S();
            return;
```

Predictive Parsing (contd.)

Grammar:

$$
\begin{aligned}
S &\longrightarrow (S)S \\
S &\longrightarrow a \\
S &\longrightarrow \epsilon
\end{aligned}
$$

```
case TOKEN_a:   /* Production 2 */
    consume(TOKEN_a);
    return;
case TOKEN_CLOSE_PAREN:
case TOKEN_EOF:   /* Production 3 */
    return;
default:
    /* Parse Error */
```

Predictive Parsing: Restrictions

Grammar cannot be left-recursive

Example: $E \longrightarrow E + E \mid a$

```
procedure parse_E() {
    switch (input_token) {
        case TOKEN_a:  /* Production 1 */
            parse_E();
            consume(TOKEN_PLUS);
            parse_E();
            return;
        case TOKEN_a:  /* Production 2 */
            consume(TOKEN_a);
            return;
    }
}
```

Removing Left Recursion

$$A \longrightarrow A\,a$$
$$A \longrightarrow b$$

$$\mathcal{L}(A) = \{b, ba, baa, baaa, baaaa, \ldots\}$$

$$A \longrightarrow bA'$$

$$A' \longrightarrow aA'$$
$$A' \longrightarrow \epsilon$$

Removing Left Recursion

More generally,

$$\begin{aligned} A &\longrightarrow A\alpha_1 | \cdots | A\alpha_m \\ A &\longrightarrow \beta_1 | \cdots | \beta_n \end{aligned}$$

Can be transformed into

$$\begin{aligned} A &\longrightarrow \beta_1 A' | \cdots | \beta_n A' \\ A' &\longrightarrow \alpha_1 A' | \cdots | \alpha_m A' | \epsilon \end{aligned}$$

Removing Left Recursion: An Example

$$E \longrightarrow E + E$$
$$E \longrightarrow \text{id}$$

$$\Downarrow$$

$$E \longrightarrow \text{id}\ E'$$
$$E' \longrightarrow +\ E\ E'$$
$$E' \longrightarrow \epsilon$$

Predictive Parsing: Restrictions

May not be able to choose a *unique* production

$$S \longrightarrow a\,B\,d$$
$$B \longrightarrow b$$
$$B \longrightarrow bc$$

Left-factoring can help:

$$S \longrightarrow a\,B\,d$$
$$B \longrightarrow bC$$
$$C \longrightarrow c|\epsilon$$

Predictive Parsing: Restrictions

In general, though, we may need a backtracking parser:

Recursive Descent Parsing

$$
\begin{aligned}
S &\longrightarrow a\ B\ d \\
B &\longrightarrow b \\
B &\longrightarrow bc
\end{aligned}
$$

Recursive Descent Parsing

Grammar:

$$S \longrightarrow a\ B\ d$$
$$B \longrightarrow b$$
$$B \longrightarrow bc$$

```
procedure parse_B() {
    switch (input_token) {
        case TOKEN_b:   /* Production 2 */
            consume(TOKEN_b);
            return;
        case TOKEN_b:   /* Production 3 */
            consume(TOKEN_b);
            consume(TOKEN_c);
            return;
}}
```

Non-recursive Parsing

Instead of recursion,

use an explicit *stack* along with the parsing table.

Data objects:

- **Parsing Table**: $M(A, a)$, a two-dimensional array, dimensions indexed by nonterminal symbols (A) and terminal symbols (a).
- A **Stack** of terminal/nonterminal symbols
- **Input stream** of tokens

The above data structures manipulated using a *table-driven parsing program*.

Table-driven Parsing

Grammar:

$$A \longrightarrow a \qquad S \longrightarrow A\ S\ B$$
$$B \longrightarrow b \qquad S \longrightarrow \epsilon$$

Parsing Table:

Nonterminal	Input Symbol		
	a	b	EOF
S	$S \longrightarrow A\ S\ B$	$S \longrightarrow \epsilon$	$S \longrightarrow \epsilon$
A	$A \longrightarrow a$		
B		$B \longrightarrow b$	

Table-driven Parsing Algorithm

```
stack initialized to EOF.
while (stack is not empty) {
    X = top(stack);
    if (X is a terminal symbol)
        consume(X);
    else /* X is a nonterminal */
        if (M[X, input_token] = X ⟶ Y_1, Y_2, ..., Y_k) {
            pop(stack);
            for i = k downto 1 do
                push(stack, Y_i);
        }
        else /* Syntax Error */
}
```

FIRST and FOLLOW

Grammar: $S \longrightarrow (S)S \mid a \mid \epsilon$

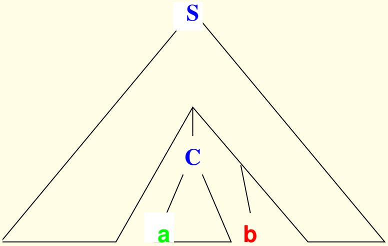
- FIRST(X) = First character of any string that can be derived from X

  $\text{FIRST}(S) = \{(, a, \epsilon\}$.

- FOLLOW(A) = First character that, in any derivation of a string in the language, appears immediately after A.

  $\text{FOLLOW}(S) = \{), \text{EOF}\}$

FIRST and FOLLOW (contd.)



$$a \in \text{FIRST}(C)$$
$$b \in \text{FOLLOW}(C)$$

FIRST and FOLLOW

FIRST(X): First terminal in some α such that $X \stackrel{*}{\Longrightarrow} \alpha$.

FOLLOW(A): First terminal in some β such that $S \stackrel{*}{\Longrightarrow} \alpha A \beta$.

Grammar:

$$
\begin{array}{lllllll}
A & \longrightarrow & a & \quad & S & \longrightarrow & A\ S\ B \\
B & \longrightarrow & b & \quad & S & \longrightarrow & \epsilon
\end{array}
$$

$$
\begin{array}{lll}
First(S) & = & \{\,a, \epsilon\,\} \\
First(A) & = & \{\,a\,\} \\
First(B) & = & \{\,b\,\}
\end{array}
\qquad
\begin{array}{lll}
Follow(S) & = & \{\,b, EOF\,\} \\
Follow(A) & = & \{\,a, b\,\} \\
Follow(B) & = & \{\,b, EOF\,\}
\end{array}
$$

Definition of FIRST

Grammar:

$$\begin{array}{lll} A & \longrightarrow & a \\ B & \longrightarrow & b \end{array} \qquad \begin{array}{lll} S & \longrightarrow & A\ S\ B \\ S & \longrightarrow & \epsilon \end{array}$$

$FIRST(\alpha)$ is the smallest set such that

$\alpha =$	Property of $FIRST(\alpha)$
a, a terminal	$a \in FIRST(\alpha)$
A, a nonterminal	$A \longrightarrow \epsilon \in G \Longrightarrow \epsilon \in FIRST(\alpha)$ $A \longrightarrow \beta \in G, \beta \neq \epsilon \Longrightarrow FIRST(\beta) \subseteq FIRST(\alpha)$
$X_1X_2 \cdots X_k$, a string of terminals and non-terminals	$FIRST(X_1) - \{\epsilon\} \subseteq FIRST(\alpha)$ $FIRST(X_i) \subseteq FIRST(\alpha)$ if $\forall j < i \quad \epsilon \in FIRST(X_j)$ $\epsilon \in FIRST(\alpha)$ if $\forall j < k \quad \epsilon \in FIRST(X_j)$

Definition of FOLLOW

Grammar:

$$A \longrightarrow a \qquad S \longrightarrow A\ S\ B$$
$$B \longrightarrow b \qquad S \longrightarrow \epsilon$$

$FOLLOW(A)$ is the smallest set such that

A	Property of $FOLLOW(A)$
$= S$, the start symbol	EOF $\in FOLLOW(S)$ Book notation: $\$ \in FOLLOW(S)$
$B \longrightarrow \alpha A \beta \in G$	$FIRST(\beta) - \{\epsilon\} \subseteq FOLLOW(A)$
$B \longrightarrow \alpha A$, or $B \longrightarrow \alpha A \beta$, $\epsilon \in FIRST(\beta)$	$FOLLOW(B) \subseteq FOLLOW(A)$

A Procedure to Construct Parsing Tables

procedure $table\_construct(G)$ {

for each $A \longrightarrow \alpha \in G$ {

for each $a \in FIRST(\alpha)$ such that $a \neq \epsilon$

add $A \longrightarrow \alpha$ to $M[A, a]$;

if $\epsilon \in FIRST(\alpha)$

for each $b \in FOLLOW(A)$

add $A \longrightarrow \alpha$ to $M[A, b]$;

}}

LL(1) Grammars

Grammars for which the parsing table constructed earlier has no multiple entries.

$$
\begin{array}{lcl}
E & \longrightarrow & \mathrm{id}\ E' \\
E' & \longrightarrow & +\ E\ E' \\
E' & \longrightarrow & \epsilon
\end{array}
$$

Nonterminal	Input Symbol		
	id	+	EOF
E	$E \longrightarrow \mathrm{id}\ E'$		
E'		$E' \longrightarrow +\ E\ E'$	$E' \longrightarrow \epsilon$

Parsing with LL(1) Grammars

Nonterminal	Input Symbol		
	id	+	EOF
E	$E \longrightarrow$ id E'		
E'		$E' \longrightarrow + \; E \; E'$	$E' \longrightarrow \epsilon$

$\$E$	id + id$	E	$\Longrightarrow$	idE'
$\$E'$id	id + id$			
$\$E'$	+ id$		$\Longrightarrow$	id+EE'
$\$E'E+$	+ id$			
$\$E'E$	id$		$\Longrightarrow$	id+id$E'E'$
$\$E'E'$id	id$			
$\$E'E'$	$		$\Longrightarrow$	id+idE'
$\$E'$	$		$\Longrightarrow$	id+id
$	$			

LL(1) Derivations

Left to Right Scan of input

Leftmost Derivation

(1) look ahead 1 token at each step

Alternative characterization of LL(1) Grammars:

Whenever $A \longrightarrow \alpha \mid \beta \in G$

1. $FIRST(\alpha) \cap FIRST(\beta) = \{\ \}$, and
2. if $\alpha \overset{*}{\Longrightarrow} \epsilon$ then $FIRST(\beta) \cap FOLLOW(A) = \{\ \}$.

Corollary: No Ambiguous Grammar is LL(1).

Leftmost and Rightmost Derivations

$$
\begin{array}{ccc}
\hline
E & \longrightarrow & E+T \\
E & \longrightarrow & T \\
T & \longrightarrow & \mathrm{id} \\
\hline
\end{array}
$$

Derivations for id + id:

$$
\begin{array}{ccc|ccc}
E & \Longrightarrow & E+T & E & \Longrightarrow & E+T \\
 & \Longrightarrow & T+T & & \Longrightarrow & E+\mathrm{id} \\
 & \Longrightarrow & \mathrm{id}+T & & \Longrightarrow & T+\mathrm{id} \\
 & \Longrightarrow & \mathrm{id}+\mathrm{id} & & \Longrightarrow & \mathrm{id}+\mathrm{id}
\end{array}
$$

LEFTMOST | RIGHTMOST

Bottom-up Parsing

Given a stream of tokens w, *reduce* it to the start symbol.

$$E \longrightarrow E + T$$
$$E \longrightarrow T$$
$$T \longrightarrow \mathrm{id}$$

Parse input stream: id + id:

$$\mathrm{id} + \mathrm{id}$$
$$T + \mathrm{id}$$
$$E + \mathrm{id}$$
$$E + T$$
$$E$$

Reduction ≡ Derivation^{-1}.

Handles

Informally, a "handle" of a sentential form is a substring that matches the right side of a production, and

whose reduction to the non-terminal on the left hand side of the production represents one step along the reverse rightmost derivation.

Handles

A structure that furnishes a means to perform reductions.

E	$\longrightarrow$	$E + T$
E	$\longrightarrow$	T
T	$\longrightarrow$	id

Parse input stream: id + id:

$$\boxed{\text{id}} + \text{id}$$
$$\boxed{T} + \text{id}$$
$$E + \boxed{\text{id}}$$
$$\boxed{E + T}$$
$$E$$

Handles

Handles are substrings of sentential forms:

1. A substring that matches the right hand side of a production
2. Reduction using that rule can lead to the start symbol

$$\begin{aligned} E &\Longrightarrow \boxed{E + T} \\ &\Longrightarrow E + \boxed{\mathrm{id}} \\ &\Longrightarrow \boxed{T} + \mathrm{id} \\ &\Longrightarrow \boxed{\mathrm{id}} + \mathrm{id} \end{aligned}$$

Handle Pruning: replace handle by corresponding LHS.

Shift-Reduce Parsing

Bottom-up parsing.

- **Shift:** Construct leftmost handle on top of stack
- **Reduce:** Identify handle and replace by corresponding RHS
- **Accept:** Continue until string is reduced to start symbol and input token stream is empty
- **Error:** Signal parse error if no handle is found.

Implementing Shift-Reduce Parsers

- Stack to hold grammar symbols (corresponding to tokens seen thus far).
- Input stream of yet-to-be-seen tokens.
- Handles appear on top of stack.
- Stack is initially empty (denoted by $).
- Parse is successful if stack contains only the start symbol when the input stream ends.

Shift-Reduce Parsing: An Example

$$
\begin{array}{rcl}
S & \longrightarrow & aABe \\
A & \longrightarrow & Abc|b \\
B & \longrightarrow & d
\end{array}
$$

To parse: $a\ b\ b\ c\ d\ e$

Shift-Reduce Parsing: An Example

$$E \longrightarrow E+T$$
$$E \longrightarrow T$$
$$T \longrightarrow \text{id}$$

Stack	Input Stream	Action
\$	id + id \$	shift
\$ id	+ id \$	reduce by $T \longrightarrow$ id
\$ T	+ id \$	reduce by $E \longrightarrow T$
\$ E	+ id \$	shift
\$ E +	id \$	shift
\$ E + id	\$	reduce by $T \longrightarrow$ id
\$ E + T	\$	reduce by $E \longrightarrow E+T$
\$ E	\$	**ACCEPT**

More on Handles

Handle: Let $S \Longrightarrow^*_{rm} \alpha A w \Longrightarrow_{rm} \alpha\beta w$.
Then $A \longrightarrow \beta$ is a handle for $\alpha\beta w$ at the position imeediately following α.

Notes:

- For unambiguous grammars, every right-sentential form has a unique handle.
- In shift-reduce parsing, handles always appear on top of stack, i.e., $\alpha\beta$ is in the stack (with β at top), and w is unread input.

Identification of Handles and Relationship to Conflicts

Case 1: With $\alpha\beta$ on the stack, don't know if we have a handle on top of the stack, or we need to shift some more input to get βx which is a handle.

- Shift-reduce conflict
- Example: if-then-else

Case 2: With $\alpha\beta_1\beta_2$ on the stack, don't know if $A \longrightarrow \beta_2$ is the handle, or $B \longrightarrow \beta_1\beta_2$ is the handle

- Reduce-reduce conflict
- Example: $E \longrightarrow E - E | - E | id$

Viable Prefix

- Prefix of a right-sentential form that does not continue beyond the rightmost handle.
- With $\alpha\beta w$ example of the previous slides, a viable prefix is something of the form $\alpha\beta_1$ where $\beta = \beta_1\beta_2$

LR Parsing

- Stack contents as $s_0 X_1 s_1 X_2 \cdots X_m s_m$
- Its actions are driven by two tables, *action* and *goto*

Parser Configuration: $(\underbrace{s_0 X_1 s_1 X_2 \cdots X_m s_m}_{\text{stack}}, \quad \underbrace{a_i a_{i+1} \cdots a_n \$}_{\text{unconsumed input}} \quad)$

$action[s_m, a_i]$ can be:

- shift s: new config is $(s_0 X_1 s_1 X_2 \cdots X_m s_m a_i s, a_{i+1} \cdots a_n \$)$
- reduce $A \longrightarrow \beta$: Let $|\beta| = r$, $goto[s_{m-r}, A] = s$: new config is $(s_0 X_1 s_1 X_2 \cdots X_{m-r} s_{m-r} A s, a_i a_{i+1} \cdots a_n \$)$
- error: perform recovery actions
- accept: Done parsing

LR Parsing

- *action* and *goto* depend only on the state at the top of the stack, not on all of the stack contents
  - The s_i states compactly summarize the "relevant" stack content that is at the top of the stack.
- You can think of *goto* as the action taken by the parser on "consuming" (and shifting) nonterminals
  - similar to the shift action in the *action* table, except that the transition is on a nonterminal rather than a terminal
- The *action* and *goto* tables define the transitions of an FSA that accepts RHS of productions!

Example of LR Parsing Table and its Use

- See Text book Algorithm 4.7: (follows directly from description of LR parsing actions 2 slides earlier)
- See expression grammar (Example 4.33), its associated parsing table in Fig 4.31, and the use of the table to parse $id * id + id$ (Fig 4.32)

LR Versus LL Parsing

Intuitively:

- LL parser needs to guess the production based on the first symbol (or first few symbols) on the RHS of a production
- LR parser needs to guess the production *after* seeing all of the RHS

Both types of parsers can use next k input symbols as look-ahead symbols (LL(k) and LR(k) parsers)

- Implication: $LL(k) \subset LR(k)$

How to Construct LR Parsing Table?

Key idea: Construct an FSA to recognize RHS of productions

- States of FSA remember which parts of RHS have been seen already.
- We use " · " to separate seen and unseen parts of RHS

LR(0) item: A production with " · " somewhere on the RHS. Intuitively,

- ▷ grammar symbols before the " · " are on stack;
- ▷ grammar symbols after the " · " represent symbols in the input stream.

$$
I_0: \quad \begin{aligned}
E' &\longrightarrow \cdot\, E \\
E &\longrightarrow \cdot\, E + T \\
E &\longrightarrow \cdot\, T \\
T &\longrightarrow \cdot\, \text{id}
\end{aligned}
$$

How to Construct LR Parsing Table?

- If there is no way to distinguish between two different productions at some point during parsing, then the same state should represent both.
  - *Closure* operation: If a state s includes LR(0) item $A \longrightarrow \alpha \cdot B\beta$, and there is a production $B \longrightarrow \gamma$, then s should include $B \longrightarrow \cdot \gamma$
  - *goto* operation: For a set I of items, $goto[I, X]$ is the closure of all items $A \longrightarrow \alpha X \cdot \beta$ for each $A \longrightarrow \alpha \cdot X\beta$ in I

Item set: A set of items that is closed under the *closure* operation, corresponds to a state of the parser.

Constructing Simple LR (SLR) Parsing Tables

Step 1: Construct LR(0) items (Item set construction)

Step 2: Construct a DFA for recognizing items

Step 3: Define *action* and *goto* based on the DFA

Item Set Construction

1. Augment the grammar with a rule $S' \longrightarrow S$, and make S' the new start symbol
2. Start with initial set I_0 corresponding to the item $S' \longrightarrow \cdot S$
3. apply *closure* operation on I_0.
4. For each item set I and grammar symbol X, add $goto[I, X]$ to the set of items
5. Repeat previous step until no new item sets are generated.

Item Set Construction

$E' \longrightarrow E$ $\quad E \longrightarrow E + T \mid T$ $\quad T \longrightarrow T * F \mid F$ $\quad F \longrightarrow (E) \mid id$

$I_0 : E' \longrightarrow \cdot E$

$I_1 : E' \longrightarrow E \cdot$

$I_2 : E \longrightarrow T \cdot$

$I_3 : T \longrightarrow F \cdot$

Item Set Construction (Contd.)

$E' \longrightarrow E$ $\qquad E \longrightarrow E + T \mid T$ $\qquad T \longrightarrow T * F \mid F$ $\qquad F \longrightarrow (E) \mid id$

$I_4 : F \longrightarrow (\cdot E)$

$I_5 : F \longrightarrow id \cdot$

$I_6 : E \longrightarrow E + \cdot T$

$I_7 : T \longrightarrow T * \cdot F$

Item Set Construction (Contd.)

$E' \longrightarrow E$ $\qquad E \longrightarrow E + T \mid T$ $\qquad T \longrightarrow T * F \mid F$ $\qquad F \longrightarrow (E) \mid id$

$I_8 : F \longrightarrow (E \cdot)$

$I_9 : E \longrightarrow E + T \cdot$

$I_{10} : T \longrightarrow T * F \cdot$

$I_{11} : F \longrightarrow (E) \cdot$

Item Sets for the Example

I_0: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + T$

I_2: $E \to T \cdot$
$T \to T \cdot * F$

I_3: $T \to F \cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_5: $F \to \mathbf{id} \cdot$

I_6: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_7: $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

I_8: $F \to (E \cdot)$
$E \to E \cdot + T$

I_9: $E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}: $T \to T * F \cdot$

I_{11}: $F \to (E) \cdot$

SLR(1) Parse Table for the Example Grammar

STATE	action						goto		
	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			

Defining *action* and *goto* tables

- Let $I_0, I_1, \ldots, I_n$ be the item sets constructed before
- Define *action* as follows
  - If $A \longrightarrow \alpha \cdot a\beta$ is in I_i and there is a DFA transition to I_j from I_i on symbol a then $action[i, a] =$ "shift j"
  - If $A \longrightarrow \alpha\,\cdot$ is in I_i then $action[i, a] =$ "reduce $A \longrightarrow \alpha$" for every $a \in FOLLOW(A)$
  - If $S' \longrightarrow S\,\cdot$ is in I_i then $action[I_i, \$] =$ "accept"
- If any conflicts arise in the above procedure, then the grammar is *not* SLR(1).
- *goto* transition for LR parsing defined directly from the DFA transitions.
- All undefined entries in the table are filled with "error"

Deficiencies of SLR Parsing

SLR(1) treats all occurrences of a RHS on stack as identical.

Only a few of these reductions may lead to a successful parse.

Example:

S	$\longrightarrow$	$A\mathrm{a}A\mathrm{b}$	$A \longrightarrow \epsilon$
S	$\longrightarrow$	$B\mathrm{b}B\mathrm{a}$	$B \longrightarrow \epsilon$

$I_0 = \{[S' \rightarrow \cdot S], [S \rightarrow \cdot A\mathrm{a}A\mathrm{b}], [S \rightarrow \cdot Bi\mathrm{b}B\mathrm{a}], [A \rightarrow \cdot], [B \rightarrow \cdot]\}$.

Since $FOLLOW(A) = FOLLOW(B)$, we have reduce/reduce conflict in state 0.

LR(1) Item Sets

Construct LR(1) items of the form $A \longrightarrow \alpha \cdot \beta, \mathrm{a}$, which means:

The production $A \longrightarrow \alpha\beta$ can be applied when the next token on input stream is a.

S	$\longrightarrow$	$A\mathrm{a}A\mathrm{b}$	$A \longrightarrow \epsilon$
S	$\longrightarrow$	$B\mathrm{b}B\mathrm{a}$	$B \longrightarrow \epsilon$

An example LR(1) item set:

$$I_0 = \{[S' \to \cdot S, \$], [S \to \cdot A\mathrm{a}A\mathrm{b}, \$], [S \to \cdot B\mathrm{b}B\mathrm{a}, \$],$$
$$[A \to \cdot , \mathrm{a}], [B \to \cdot , \mathrm{b}]\}.$$

LR(1) and LALR(1) Parsing

LR(1) parsing: Parse tables built using LR(1) item sets.

LALR(1) parsing: *Look Ahead* LR(1)

Merge LR(1) item sets; then build parsing table.

Typically, LALR(1) parsing tables are much smaller than LR(1) parsing table.

YACC

Yet **A**nother **C**ompiler **C**ompiler:

LALR(1) parser generator.

- Grammar rules are written in a specification (`.y`) file, analogous to the regular definitions in a `lex` specification file.
- Yacc translates the specifications into a parsing function `yyparse()`.

$$\texttt{spec.y} \xrightarrow{\texttt{yacc}} \texttt{spec.tab.c}$$

- `yyparse()` calls `yylex()` whenever input tokens need to be consumed.
- `bison`: GNU variant of `yacc`.

Using Yacc

```
%{
  ... C headers (#include)
%}
... Yacc declarations:
       %token ...
       %union{...}
       precedences
%%
... Grammar rules with actions:
Expr:  Expr TOK_PLUS Expr
    |  Expr TOK_MINUS Expr
    ;
%%
... C support functions
```

YACC

Yet **A**nother **C**ompiler **C**ompiler:

LALR(1) parser generator.

- Grammar rules are written in a specification (`.y`) file, analogous to the regular definitions in a `lex` specification file.
- Yacc translates the specifications into a parsing function `yyparse()`.

$$\texttt{spec.y} \xrightarrow{\texttt{yacc}} \texttt{spec.tab.c}$$

- `yyparse()` calls `yylex()` whenever input tokens need to be consumed.
- `bison`: GNU variant of `yacc`.

Using Yacc

```
%{
  ... C headers (#include)
%}
... Yacc declarations:
       %token ...
       %union{...}
       precedences
%%
... Grammar rules with actions:
Expr:  Expr TOK_PLUS Expr
    |  Expr TOK_MINUS Expr
    ;
%%
... C support functions
```

Conflicts and Resolution

- Operator precedence works well for resolving conflicts that involve operators
  - But use it with care – only when they make sense, not for the sole purpose of removing conflict reports
- Shift-reduce conflicts: Bison favors shift
  - Except for the dangling-else problem, this strategy does not ever seem to work, so don't rely on it.

Reduce-Reduce Conflicts

```
sequence: /* empty */
          { printf ("empty sequence\n"); }
        | maybeword
        | sequence word
          { printf ("added word %s\n", $2); };
maybeword: /* empty */
           { printf ("empty maybeword\n"); }
        | word
          { printf ("single word %s\n", $1); };
```

In general, grammar needs to be rewritten to eliminate conflicts.

Sample Bison File: Postfix Calculator

```
input:    /* empty */
        | input line
;
line:     '\n'
        | exp '\n'      { printf ("\t%.10g\n", $1); }
;
exp:      NUM           { $$ = $1;           }
        | exp exp '+'   { $$ = $1 + $2;      }
        | exp exp '-'   { $$ = $1 - $2;      }
        | exp exp '*'   { $$ = $1 * $2;      }
        | exp exp '/'   { $$ = $1 / $2;      }
         /* Exponentiation */
        | exp exp '^'   { $$ = pow ($1, $2); }
         /* Unary minus    */
        | exp 'n'       { $$ = -$1;          };
%%
```

Infix Calculator

```
%{
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
int yylex (void);
void yyerror (char const *);
%}
/* Bison Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation */
```

Infix Calculator (Continued)

```
%% /* The grammar follows.  */
input:    /* empty */
        | input line
;
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;
exp:      NUM                { $$ = $1;           }
        | exp '+' exp        { $$ = $1 + $3;      }
        | exp '-' exp        { $$ = $1 - $3;      }
        | exp '*' exp        { $$ = $1 * $3;      }
        | exp '/' exp        { $$ = $1 / $3;      }
        | '-' exp  %prec NEG { $$ = -$2;          }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;           }
;
%%
```

Error Recovery

```
line:      '\n'
         | exp '\n'   { printf ("\t%.10g\n", $1); }
         | error '\n' { yyerrok;                  };
```

- Pop stack contents to expose a state where an `error` token is acceptable
- Shift error token onto the stack
- Discard input until reaching a token that can follow this error token

Error recovery strategies are never perfect — some times they lead to cascading errors, unless carefully designed.

Left Versus Right Recursion

```
expseq1:   exp | expseq1 ',' exp;
```

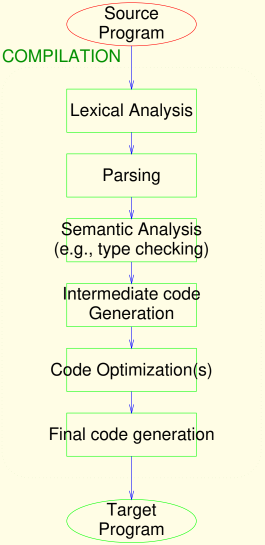
is a left-recursive definition of a sequence of exp's, whereas

```
expseq1:   exp | exp ',' expseq1;
```

is a right-recursive definition

- Left-recursive definitions are a no-no for LL parsing, but yes-yes for LR parsing
- Right-recursive definition is bad for LR parsing as it needs to shift ithe entire list on stack before any reduction — increases stack usage

Compilation



Syntax-Directed Translation

Technique used to build semantic information for large structures, based on its syntax.

In a compiler, *Syntax-Directed Translation* is used for

- Constructing Abstract Syntax Tree
- Type checking
- Intermediate code generation

The Essence of Syntax-Directed Translation

The semantics (*meaning*) of the various constructs in the language is viewed as *attributes* of the corresponding grammar symbols.

Example: Sequence of characters 495

- grammar symbol TOK_INT
- meaning $\equiv$ integer *495*
- is an attribute of TOK_INT(yylval.int_val).

Attributes are associated with ***Terminal*** as well as ***Nonterminal*** symbols.

An Example of Syntax-Directed Translation

$$\boxed{\begin{array}{lll} E & \longrightarrow & E \texttt{ * } E \\ E & \longrightarrow & E + E \\ E & \longrightarrow & \mathrm{id} \end{array}}$$

$$\begin{array}{llll} E & \longrightarrow & E_1 \texttt{ * } E_2 & \{E.val := E_1.val * E_2.val\} \\ E & \longrightarrow & E_1 + E_2 & \{E.val := E_1.val + E_2.val\} \\ E & \longrightarrow & \mathrm{int} & \{E.val := \mathrm{int}.val\} \end{array}$$

Syntax-Directed Definitions with yacc

E	$\longrightarrow$	E_1 * E_2	$\{E.val := E_1.val * E_2.val\}$
E	$\longrightarrow$	E_1 + E_2	$\{E.val := E_1.val + E_2.val\}$
E	$\longrightarrow$	int	$\{E.val :=$ int$.val\}$

E :	E MULT E	$\{\$\$.val = \$1.val * \$3.val\}$
E :	E PLUS E	$\{\$\$.val = \$1.val + \$3.val\}$
E :	INT	$\{\$\$.val = \$1.val\}$

Another Example of Syntax-Directed Translation

$$
\begin{array}{rcl}
\mathit{Decl} & \longrightarrow & \mathit{Type}\ \mathit{VarList} \\
\mathit{Type} & \longrightarrow & \ldots \\
\mathit{VarList} & \longrightarrow & \mathrm{id}\ ,\ \mathit{VarList} \\
\mathit{VarList} & \longrightarrow & \mathrm{id}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Decl} & \longrightarrow & \mathit{Type}\ \mathit{VarList} & \{\mathit{VarList}.\mathit{type} := \mathit{Type}.\mathit{type}\} \\
\mathit{Type} & \longrightarrow & \ldots & \{\mathit{Type}.\mathit{type} := \ldots\} \\
\mathit{VarList} & \longrightarrow & \mathrm{id}\ ,\ \mathit{VarList}_1 & \{\mathit{VarList}_1.\mathit{type} := \mathit{VarList}.\mathit{type}; \\
 & & & \quad \mathrm{id}.\mathit{type} := \mathit{VarList}.\mathit{type}\} \\
\mathit{VarList} & \longrightarrow & \mathrm{id} & \{\mathrm{id}.\mathit{type} := \mathit{VarList}.\mathit{type}\}
\end{array}
$$

Attributes

- *Synthesized* Attribute: Value of the attribute computed from the values of attributes of grammar symbols on RHS.
  - Example: *val* in Expression grammar
- *Inherited* Attribute: Value of attribute computed from values of attributes of the LHS grammar symbol.
  - Example: *type* of *VarList* in declaration grammar

Syntax-Directed Definition

Actions associated with each production in a grammar.

For a production $A \longrightarrow X\ Y$, actions may be of the form:

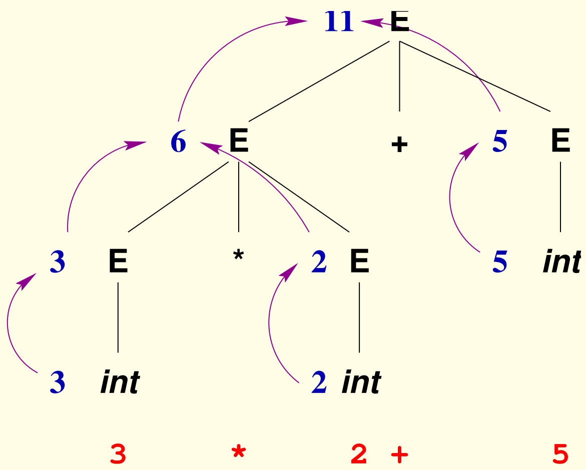
- $A.attr := f(X.attr', Y.attr'')$ for synthesized attributes
- $Y.attr := f(A.attr', X.attr'')$ for inherited attributes

Synthesized Attributes: An Example

$$
\boxed{\begin{array}{lcl} E & \longrightarrow & E \texttt{ * } E \\ E & \longrightarrow & E \texttt{ + } E \\ E & \longrightarrow & \texttt{int} \end{array}}
$$

$$
\begin{array}{lcll} E & \longrightarrow & E_1 \texttt{ * } E_2 & \{E.val := E_1.val * E_2.val\} \\ E & \longrightarrow & E_1 \texttt{ + } E_2 & \{E.val := E_1.val + E_2.val\} \\ E & \longrightarrow & \texttt{int} & \{E.val := \texttt{int}.val\} \end{array}
$$

Information Flow for Synthesized Attributes

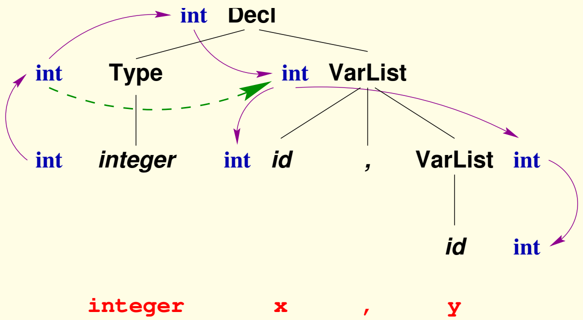


Another Example of Syntax-Directed Translation

$$
\begin{array}{rcl}
\mathit{Decl} & \longrightarrow & \mathit{Type}\ \mathit{VarList} \\
\mathit{Type} & \longrightarrow & \mathsf{integer} \\
\mathit{Type} & \longrightarrow & \mathsf{float} \\
\mathit{VarList} & \longrightarrow & \mathsf{id}\ ,\ \mathit{VarList} \\
\mathit{VarList} & \longrightarrow & \mathsf{id}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Decl} & \longrightarrow & \mathit{Type}\ \mathit{VarList} & \{\mathit{VarList.type} := \mathit{Type.type}\} \\
\mathit{Type} & \longrightarrow & \mathsf{integer} & \{\mathit{Type.type} := \mathit{int}\} \\
\mathit{Type} & \longrightarrow & \mathsf{float} & \{\mathit{Type.type} := \mathit{float}\} \\
\mathit{VarList} & \longrightarrow & \mathsf{id}\ ,\ \mathit{VarList}_1 & \{\mathit{VarList}_1\mathit{.type} := \mathit{VarList.type}; \\
 & & & \quad \mathsf{id}.\mathit{type} := \mathit{VarList.type}\} \\
\mathit{VarList} & \longrightarrow & \mathsf{id} & \{\mathsf{id}.\mathit{type} := \mathit{VarList.type}\}
\end{array}
$$

Information Flow for Inherited Attributes



Attributes and Definitions

- **S-Attributed Definitions:** Where all attributes are *synthesized*.
- **L-Attributed Definitions:** Where all *inherited* attributes are such that their values depend only on
  - inherited attributes of the parent, and
  - attributes of left siblings

Attributes and Top-down Parsing

- *Inherited:* analogous to function arguments
- *Synthesized:* analogous to return values

L-attributed definitions mean that argument to a parsing function is

- argument of the calling function, or
- return value/argument of a previously called function

Synthesized Attributes and Bottom-up Parsing

Keep track of attributes of symbols while parsing.

- Keep a stack of attributes corresponding to stack of symbols.
- Compute attributes of LHS symbol while performing reduction (*i.e.*, while pushing the symbol on symbol stack)

Synthesized Attributes and Bottom-Up Parsing

E	$\longrightarrow$	$E + E$
E	$\longrightarrow$	$E * E$
E	$\longrightarrow$	int

STACK	INPUT STREAM	ATTRIBUTES
\$	3 * 2 + 5 \$	\$
\$ int	* 2 + 5 \$	\$ 3
\$ E	* 2 + 5 \$	\$ 3
\$ E *	2 + 5 \$	\$ 3 $\bot$
\$ E * int	+ 5 \$	\$ 3 $\bot$ 2
\$ E	+ 5 \$	\$ 6
\$ E +	5 \$	\$ 6 $\bot$
\$ E + int	\$	\$ 6 $\bot$ 5
\$ E + E	\$	\$ \$ 6 $\bot$ 5
\$ E	\$	\$ 11

Inherited Attributes and Bottom-up Parsing

- Inherited attributes depend on the *context* in which a symbol is used.
- For inherited attributes, we cannot assign a value to a node's attributes unless the parent's attributes are known.
- When building parse trees bottom-up, parent of a node is not known when the node is created!
- Solution:
  - Ensure that all attributes are inherited only from left siblings.
  - Use "global" variables to capture inherited values,
  - and introduce "marker" nonterminals to manipulate the global variables.

Inherited Attributes & Bottom-up parsing

$$
\boxed{
\begin{array}{rcl}
Ss & \longrightarrow & S \; ; \; Ss \;\mid\; \epsilon \\
S & \longrightarrow & B \;\mid\; \text{other} \\
B & \longrightarrow & \{ \; Ss \; \}
\end{array}
}
$$

$$
\begin{array}{rcll}
B & \longrightarrow & \{ \; M_1 \; Ss \; M_2 \; \} & \\
M_1 & \longrightarrow & \epsilon & \{ current\_block{+}{+}; \} \\
M_2 & \longrightarrow & \epsilon & \{ current\_block{-}; \}
\end{array}
$$

Attribute Grammars

- syntax-directed definitions without side-effects
- attribute definitions can be thought of as *logical assertions* rather than as things that need to be computed
  - distinction between synthesized and inherited attributes disappears

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 * E_2 & \{E.type = E_1.type = E_2.type\} \\
E & \longrightarrow & E_1 + E_2 & \{E.type = E_1.type = E_2.type\} \\
E & \longrightarrow & \text{int} & \{E.type = \text{integer}\}
\end{array}
$$

Attribute Grammars

An attribute grammar AG is given by (G, V, F), where:

- G is a context-free grammar
- V is the set of attributes, each of which is associated with a terminal or a nonterminal
- F is the set of attribute assertions, each of which is associated with a production in the grammar

A string $s \in L(AG)$ iff $s \in L(G)$ and the attribute assertions hold for production used to derive s, i.e., $\exists$ a parse tree for s w.r.t. G where assertions associated with each edge in the parse tree are satisfied.

Semantic Analysis Phases of Compilation

- Build an Abstract Syntax Tree (AST) while parsing
- Decorate the AST with type information (type checking/inference)
- Generate intermediate code from AST
- Optimize intermediate code
- Generate final code

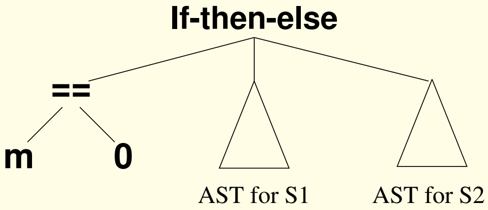
Abstract Syntax Tree (AST)

- Represents syntactic structure of a program
- Abstracts out irrelevant grammar details

An AST for the statement:

“`if (m == 0) S1 else S2`”

is



Construction of Abstract Syntax Trees

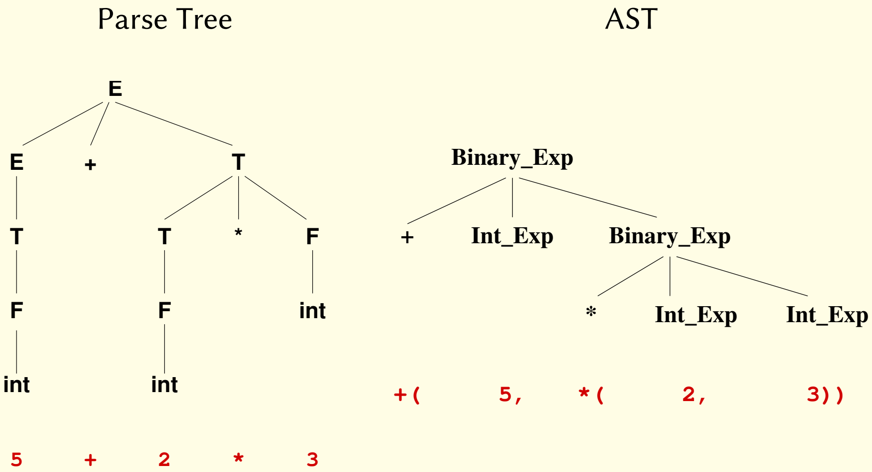
Typically done simultaneously with parsing

... as another instance of syntax-directed translation

... for translating *concrete* syntax (the parse tree) to *abstract* syntax (AST).

... with AST as a *synthesized attribute* of each grammar symbol.

Abstract Syntax Trees



Actions and AST

$$E \longrightarrow E_1 + T$$

$$\{ E.\texttt{ast} = \texttt{new BinaryExpr(OP\_PLUS,}\ E_1.\texttt{ast}, T.\texttt{ast);}\ \}$$

$$E \longrightarrow T \quad \{ E.\texttt{ast} = T.\texttt{ast;} \}$$

$$\vdots$$

$$F \longrightarrow (\ E\ ) \quad \{ F.\texttt{ast} = E.\texttt{ast;} \}$$

$$F \longrightarrow \texttt{int}$$

$$\{ F.\texttt{ast} = \texttt{new IntValNode(int.val);}\ \}$$

Actions and AST: Another Example

$$S \longrightarrow \textsf{if}\ E\ S_1\ \textsf{else}\ S_2$$
$$\{\, S\texttt{.ast} = \texttt{new IfStmtNode(}E\texttt{.ast,}\ S_1\texttt{.ast,}\ S_2\texttt{.ast);}\ \}$$

$$S \longrightarrow \textsf{return}\ E$$
$$\{\, S\texttt{.ast} = \texttt{new ReturnNode(}E\texttt{.ast)}\}$$

Bindings: Names and Attributes

- Names are a fundamental abstraction in languages to denote entities
- Meanings associated with these entities is captured via attributes associated with the names
- Attributes differ depending on the entity:
  - location (for variables)
  - value (for constants)
  - formal parameter types (functions)
- Binding: Establishing an association between name and an attribute.

Names

- **Names** or **Identifiers** denote various language *entities*:
  - Constants
  - Variables
  - Procedures and Functions
  - Types, . . .
- Entities have *attributes*

Entity	*Example Attributes*
Constants	type, value, . . .
Variables	type, location, . . .
Functions	signature, implementation, . . .

Attributes

- Attributes are associated with names (to be more precise, with the entities they denote).
- Attributes describe the *meaning* or *semantics* of these entities.

`int x;`	There is a variable, named `x`, of type integer.
`int y = 2;`	Variable named `x`, of type integer, with initial value 2.
`Set s=new Set();`	Variable named `s`, of type `Set` that refers to an object of class `Set`

- An *attribute* may be
  - *static:* can be determined at translation (compilation) time, or
  - *dynamic:* can be determined only at execution time.

Static and Dynamic Attributes

- `int x;`
  - The *type* of `x` can be statically determined;
  - The *value* of `x` is dynamically determined;
  - The *location* of `x` (the element in memory will be associated with `x`) can be statically determined if `x` is a global variable.
- `Set s = new Set();`
  - The *type* of `s` can be statically determined.
  - The *value* of `s`, i.e. the object that `s` refers to, is dynamically determined.

Static vs. Dynamic specifies the *earliest* time the attribute <u>*can*</u> be computed

. . . not when it <u>*is*</u> computed in any particular implementation.

Binding

"Binding" is the process of associating attributes with names.

- **Binding time** of an attribute: whether an attribute can be computed at translation time or only at execution time.
- A more refined classification of binding times:
  - **Static:**
    - Language definition time (e.g. `boolean`, `char`, etc.)
    - Language implementation time (e.g. `maxint`, `float`, etc.)
    - Translation time ("compile time") (e.g. value of `n` in `const int n = 5;`)
    - Link time (e.g. the definition of function `f` in `extern int f();`)
    - Load time (e.g. the location of a global variable, i.e., where it will be stored in memory)
  - **Dynamic:**
    - Execution time

Binding Time (Continued)

- Examples
  - type is statically bound in most langs
  - value of a variable is dynamically bound
  - location may be dynamically or statically bound
- Binding time also affects where bindings are stored
  - Name $\rightarrow$ type: symbol table
  - Name $\rightarrow$ location: environment
  - Location $\rightarrow$value: memory

Declarations and Definitions

- **Declaration** is a syntactic structure to establish bindings.
  - `int x;`
  - `const int n = 5;`
  - `extern int f();`
  - `struct foo;`
- **Definition** is a declaration that usually binds *all* static attributes.
  - `int f() { return x;}`
  - `struct foo { char *name; int age;};`
- Some bindings may be implicit, i.e., take effect without a declaration.
  - FORTRAN: All variables beginning with [i-nI-N] are integers; others are real-valued.
  - PROLOG: All identifiers beginning with [A-Z_] are variables.

Scopes

- Region of program over which a declaration is in effect
  - i.e. bindings are maintained
- Possible values
  - Global
  - Package or module
  - File
  - Class
  - Procedure
  - Block

Visibility

- Redefinitions in inner scopes supercede outer definitions
- Qualifiers may be needed to make otherwise invisible names to be visible in a scope.
- Examples
  - local variable superceding global variable
  - names in other packages.
  - private members in classes.

Symbol Table

Maintains bindings of attributes with names:

$$\mathit{SymbolTable} : \mathit{Names} \longrightarrow \mathit{Attributes}$$

- In a compiler, only *static attributes* can be computed; thus:

$$\mathit{SymbolTable} : \mathit{Names} \longrightarrow \mathit{StaticAttributes}$$

- While execution, the names of entities no longer are necessary: only locations in memory representing the variables are important.

$$\mathit{Store} : \mathit{Locations} \longrightarrow \mathit{Values}$$

  (*Store* is also called as *Memory*)

- A compiler then needs to map variable names to locations.

$$\mathit{Environment} : \mathit{Names} \longrightarrow \mathit{Locations}$$

Blocks and Scope

- Usually, a name refers to an entity within a given *context*.

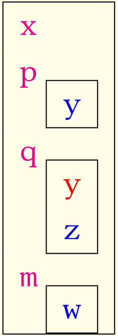
```
class A {
  int x;
  double y;
  int f(int x) { // Parameter "x" is different from field "x"
     B b = new B();
     y = b.f(); // method "f" of object "b"
     this.x = x;
     ...
  }
}
```

- The context is specified by “Blocks”
  - Delimited by “{” and “}” in C, C++ and Java
  - Delimited by “`begin`” and “`end`” in Pascal, Algol and Ada.

Scope

Scope: Region of the program over which a binding is maintained.

```
int x;
void p(void) {
  char y;
  ...
}
void q(int y) {
  double z;
  ...
}
m() {
  int w;
  ...
}
```



Lexical Scope

Lexical scope: the scope of a binding is limited to the block in which its declaration appears.

- The bindings of local variables in C, C++, Java follow lexical scope.
- Some names in a program may have a “meaning” outside its lexical scope.

  e.g. field/method names in Java
  - Names must be *qualified* if they cannot be resolved by lexical scope.

    e.g. `a.x` denotes the field `x` of object referred by `a`.

    `a.x` can be used even outside the lexical scope of `x`.
- Visibility of names outside the lexical scope is declared by *visibilty modifiers* (e.g. public, private, etc.)

Namespaces

- Namespaces are a way to specify "contexts" for names.
  - `www.google.com`:
    - The trailing `com` refers to a set of machines
    - `google` is subset of machines in the set `com`

      `google` is interpreted here in the context of `com`
    - `www` is a subset of machines in the set `google`

      `www` is interpreted here in the context of `google.com`
  - Other common use of name spaces: directory/folder structure.
- Names should be fully qualified if they are used outside their context.

  e.g. `Stack::top()` in C++, `List.hd` in OCAML.
- Usually there are ways to declare the context *a priori* so that names can be specified without qualifying them.

Lifetimes

The lifetime of a binding is the interval during which it is effective.

```
int fact(int n) {
  int x;
  if (n == 0)
     return 1;
  else {
     x = fact(n-1);
     return x * n;
  }
}
```

fact: n = 2
fact: n = 2 → fact: n = 1
fact: n = 2 → fact: n = 1 → fact: n = 0
fact: n = 2 → fact: n = 1, x = 1
fact: n = 2, x = 1
2

- Each invocation of `fact` defines new variables `n` and `x`.
- The lifetime of a binding may exceed the scope of the binding.
  - e.g., consider the binding `n=2` in the first invocation of `fact`.
  - Call to `fact(1)` creates a new local variable `n`.
  - But the first binding is still effective.

Symbol Table

- Uses data structures that allow efficient name lookup operations in the presence of scope changes.
- We can use
  - hash tables to lookup attributes for each name
  - a scope stack that keeps track of the current scope and its surrounding scopes
    - the top most element in the scope stack corresponds to the current scope
    - the bottommost element will correspond to the outermost scope.

Support for Scopes

- Lexical scopes can be supported using a scope stack as follows:
- Symbols in a program reside in multiple hash tables
  - In particular, symbols within each scope are contained in a single hash table for that scope
- At anytime, the scope stack keeps track of all the scopes surrounding that program point.
- The elements of the stack contain pointers to the corresponding hash table.

Support for Scopes (Continued)

- To lookup a name
- Symbols in a program reside in multiple hash tables
  - Start from the hash table pointed to by the top element of the stack.
  - If the symbol is not found, try hash table pointed by the next lower entry in the stack.
  - This process is repeated until we find the name, or we reach the bottom of the stack.
- Scope entry and exit operations modify the scope stack appropriately.
  - When a new scope is entered, a corresponding hash table is created. A pointer to this hash table is pushed onto the scope stack.
  - When we exit a scope, the top of the stack is popped off.

Example

```
float y = 1.0
void f(int x){
  for(int x=0;...){
     float x1 = x + y;
  }
  {
     float x = 1.0;
  }
}
main() {
  float y = 10.0;
  f(1);
}
```

illustration

- At (1)
  - We have a single hash table, which is the global hash table.
  - The scope stack contains exactly one entry, which points to this global hash table.
- When the compiler moves from (1) to (2)
  - The name y is added to the hash table for the current scope.
  - Since the top of scope stack points to the global table, “y” is being added to the global table.
- When the compiler moves from (2) to (3)
  - The name “f” is added to the global table, a new hash table for f’s scope is created.
  - A pointer to f’s table is pushed on the scope stack.
  - Then “x” is added to hash table for the current scope.

Static vs Dynamic Scoping

- Static or lexical scoping:
  - associations are determined at compile time
  - using a sequential processing of program
- Dynamic scoping:
  - associations are determined at runtime
  - processing of program statements follows the execution order of different statements

Example

- if we added a new function "g" to the above program as follows:

```
void g() {
  int y;
  f();
}
```

- Consider references to the name “y” at (4).
  - With static scoping, it always refers to the global variable “y” defined at (1).
  - With dynamic scoping
    - if “f” is called from main, “y” will refer to the float variable declared in main.
    - If “f” is invoked from within “g”, the same name will refer to the integer variable “y” defined in “g”.

Example (Continued)

- Since the type associated with “y” at (4) can differ depending upon the point of call, we cannot statically determine the type of “y” .
- Dynamic scoping does not fit well with static typing.
- Since static typing has now been accepted to be the right approach, almost all current languages (C/C++/Java/OCAML/LISP) use static scoping.

What is a Type?

- A set of values

What is a Type?

- A set of values
  - Together with a set of operations on these values that possess certain properties

Topics

- Data types in modern languages
  - simple and compound types
- Type declaration
- Type inference and type checking
- Type equivalence, compatibility, conversion and coercion
- Strongly/Weakly/Un-typed languages
- Static Vs Dynamic type checking

Simple Types

- Predefined
  - int, float, double, etc in C
- All other types are constructed, starting from predefined (aka primitive) types
  - Enumerated:
    - enum colors {red, green, blue} in C
    - type colors = Red|Green|Blue in OCAML

Detour: Evolution of Programming Languages

Compound Types

- Types constructed from other types using type constructors
  - Cartesian product (*)
  - Function types ($\rightarrow$)
  - Union types ($\cup$)
  - Arrays
  - Pointers
  - Recursive types

Cartesian Product

- Let I represent the integer type and R represent real type.
- The cross product $I \times R$ is defined in the usual manner of product of sets, i.e.,

$$I \times R = \{(i, r) | i \in I, r \in R\}$$

- Cartesian product operator is non-associative.

Labeled Product types

- In Cartesian products, components of tuples don't have names.
  - Instead, they are identified by numbers.
- In labeled products each component of a tuple is given a name.
- Labeled products are also called records (a language-neutral term)

Labeled Product types (Continued)

- `struct` is a term that is specific to C and C++

  `struct t {int a;float b;char *c;};` in C

Function Types

- $T_1 \rightarrow T_2$ is a function type
  - Type of a function that takes one argument of type T_1 and returns type T_2
- OCAML supports functions as first class values.
  - They can be created and manipulated by other functions.
- In imperative languages such as C, we can pass pointers to functions, but this does not offer the same level of flexibility.
  - E.g., no way for a C-function to dynamically create and return a pointer to a function;
  - rather, it can return a pointer to an EXISTING function
- Recent versions of C++ (as well Python, JavaScript and recent Java versions) support dynamically created functions (aka lambda abstractions)
  - See Functional Programming for Imperative Programmers for a discussion of functional programming features in C++.

Union types

- Union types correspond to set unions, just like product types corresponded to Cartesian products.
  - -> operator is right-associative, so we read the type as float -> (float -> float).
- Unions can be tagged or untagged. C/C++ support only untagged unions:

```
union v {
  int ival;
  float fval;
  char cval;
};
```

Tagged Unions

- In untagged unions, there is no way to ensure that the component of the right type is always accessed.
  - E.g., an integer value may be stored in the above union, but due to a programming error, the fval field may be accessed at a later time.
  - fval doesn't contain a valid value now, so you get some garbage.
- With tagged unions, the compiler can perform checks at runtime to ensure that the right components are accessed.
- Tagged unions are NOT supported in C/C++.

Tagged Unions (Continued)

- Pascal supports tagged unions using VARIANT RECORDs

```
RECORD
  CASE b: BOOLEAN OF
    TRUE: i: INTEGER; |
    FALSE: r: REAL END
  END
END
```

- Tagged union is also called a discriminated union

Array types

- Array construction is denoted by
  - array(<range>, <elememtType>).
- C-declaration
  - `int a[5];`
  - defines a variable a of type array(0-4, int)
- A declaration
  - `union tt b[6][7];`
  - declares a variable b of type array(0-4, array(0-6, union tt))
- We may not consider range as part of type

Pointer types

- A pointer type will be denoted using the syntax
  - ptr(<elementType>)
  - where <elementType> denote the types of the object pointed by a pointer type.
- The C-declaration
  - `char *s;`
  - defines a variable s of type ptr(char)
- A declaration
  - `int (*f)(int s, float v)`
  - defines a (function) pointer of type ptr(int*float $\rightarrow$ int)

Recursive types

- Recursive type: a type defined in terms of itself.
- Example in C:

```
struct IntList {
  int hd;
  intList tl;
};
```

- Does not work:
  - This definition corresponds to an infinite list.
  - There is no end, because there is no way to capture the case when the tail has the value "nil"

Recursive types (Continued)

- Need to express that tail can be nil or be a list.
- Try: variant records:

```
TYPE charlist = RECORD
   CASE IsEmpty:  BOOLEAN OF
     TRUE: /* empty list */ |
     FALSE:
       data:  CHAR;
       next:  charlist;
   END
END
```

- Still problematic: Cannot predict amount of storage needed.

Recursive types (Continued)

- Solution in typical imperative languages:
- Use pointer types to implement recursive type:

```
struct IntList {
  int hd;
  IntList *tl;
};
```

- Now, tl can be:
  - a NULL pointer (i.e., nil or empty list)
  - or point to a nonempty list value
- Now, IntList structure occupies only a fixed amount of storage

Recursive types In OCAML

- Direct definition of recursive types is supported in OCAML using type declarations.
- Use pointer types to implement recursive type:

```
# type intBtree =
   LEAF of int
   | NODE of int * intBtree * intBtree;;
type intBtree = LEAF of int | NODE of int * intBtree * intBtree
```

- We are defining a binary tree type inductively:
  - Base case: a binary tree with one node, called a LEAF
  - Induction case: construct a binary tree by constructing a new node that sores an integer value, and has two other binary trees as children

Polymorphism

- Ability of a function to take arguments of multiple types.
- The primary use of polymorphism is code reuse.
- Functions that call polymorphic functions can use the same piece of code to operate on different types of data.

Overloading (adhoc polymorphism)

- Same function NAME used to represent different functions
  - implementations may be different
  - arguments may have different types
- Example:
  - operator '+' is overloaded in most languages so that they can be used to add integers or floats.
  - But implementation of integer addition differs from float addition.
  - Arguments for integer addition or ints, for float addition, they are floats.
- Any function name can be overloaded in C++, but not in C.
- All virtual functions are in fact overloaded functions.

Polymorphism & Overloading

- Parametric polymorphism:
  - same function works for arguments of different types
  - same code is reused for arguments of different types.
  - allows reuse of "client" code (i.e., code that calls a polymorphic function) as well
- Overloading:
  - due to differences in implementation of overloaded functions, there is no code reuse in their implementation
  - but client code is reused

Parametric polymorphism in C++

- Example:

```
template <class C>
C min(const C* a, int size, C minval) {
  for (int i = 0; i < size; i++)
    if (a[i] < minval)
      minval = a[i];
  return minval;
}
```

- Note: same code used for arrays of any type.
  - The only requirement is that the type support the "<" and "=" operations
- The above function is parameterized wrt class C
  - Hence the term "parametric polymorphism".
- Unlike C++, C does not support templates.

Code reuse with Parametric Polymorphism

- With parametric polymorphism, same function body reused with different types.
- Basic property:
  - does not need to "look below" a certain level
  - E.g., min function above did not need to look inside each array element.
  - Similarly, one can think of length and append functions that operate on linked lists of all types, without looking at element type.

Code reuse with overloading

- No reuse of the overloaded function
  - there is a different function body corresponding to each argument type.
- But client code that calls a overloaded function can be reused.
- Example
  - Let C be a class, with subclasses C1,...,Cn.
  - Let f be a virtual method of class C
  - We can now write client code that can apply the function f uniformly to elements of an array, each of which is a pointer to an object of type C1,...,Cn.

Example

- Example:

```
void g(int size, C *a[]) {
  for (int i = 0; i < size; i++)
    a[i]->f(...);
}
```

- Now, the body of function g (which is a client of the function f) can be reused for arrays that contain objects of type C_1 or C_2 or ... or C_n,or even a mixture of these types.

Type Equivalence

- Structural equivalence: two types are equivalent if they are defined by identical type expressions.
  - array ranges usually not considered as part of the type
  - record labels are considered part of the type.
- Name equivalence: two types are equal if they have the same name.
- Declaration equivalence: two types are equivalent if their declarations lead back to the same original type expression by a series of redeclarations.

Type Equivalence (contd.)

- Structural equivalence is the least restrictive
- Name equivalence is the most restrictive.
- Declaration equivalence is in between
- TYPE t1 = ARRAY [1..10] of INTEGER; VAR v1: ARRAY [1..10] OF INTEGER;
- TYPE t2 = t1; VAR v3,v4: t1; VAR v2: ARRAY [1..10] OF INTEGER;

	Structurally equivalent?	Declaration equivalent?	Name equivalent?
t1,t2	Yes	Yes	No
v1,v2	Yes	No	No
v3,v4	Yes	Yes	Yes

Declaration equivalence

- In Pascal, Modula use decl equivalence
- In C
  - Decl equiv used for structs and unions
  - Structual equivalence for other types.

```
struct { int a ; float b ;} x ;
struct { int a; float b; }y;
```

- x and y are structure equivalent but not declaration equivalent.

```
typedef int* intp ;
typedef int** intpp ;
intpp v1 ;
intp *v2 ;
```

- v1 and v2 are structure equivalent.

Type Compatibility

- Weaker notion than type equivalence
- Notion of compatibility differs across operators
- Example: assignment operator:
  - v = expr is OK if <expr> is type-compatible with v.
  - If the type of expr is a Subtype of the type of v, then there is compatibility.
- Other examples:
  - In most languages, assigning integer value to a float variable is permitted, since integer is a subtype of float.
  - In OO-languages such as Java, an object of a derived type can be assigned to an object of the base type.

Type Compatibility (Continued)

- Procedure parameter passing uses the same notion of compatibility as assignment
  - Note: procedure call is a 2-step process
    - assignment of actual parameter expressions to the formal parameters of the procedure
    - execution of the procedure body
- Formal parameters are the parameter names that appear in the function declaration.
- Actual parameters are the expressions that appear at the point of function call.

Type Checking

- Static (compile time)
  - Benefits
    - no run-time overhead
    - programs safer/more robust
- Dynamic (run-time)
  - Disadvantages
    - runtime overhead for maintaining type info at runtime
    - performing type checks at runtime
  - Benefits
    - more flexible/more expressive

Examples of Static and Dynamic Type Checking

- C++ allows

  Upcasts: casting of subclass to superclass (always type-safe)

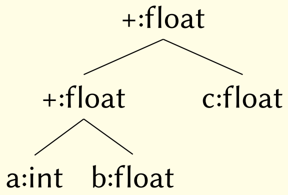
  Downcasts: superclass to subclass (not necessarily type-safe) – but no way to check since C++ is statically typed.

  - Actually, runtime checking of downcasts is supported in C++ but is typically not used due to runtime overhead

- Java uses combination of static and dynamic type-checking to catch unsafe casts (and array accesses) at runtime.

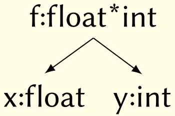
Type Checking (Continued)

- Type checking relies on type compatibility and type inference rules.
- Type inference rules are used to infer types of expressions. e.g., type of (a+b)+c is inferred from type of a, b and c and the inference rule for operator '+'.
- Type inference rules typically operate on a bottom-up fashion.
- Example: (a+b)+c



Type Checking (Continued)

- In OCAML, type inference rules capture bottom-up *and* top-down flow of type info.
- Example of Top-down: let f x y:float*int = (x, y)



- Here types of x and y inferred from return type of f.
- Note: Most of the time OCAML programs don't require type declaration.
  - But it really helps to include them: programs are more readable, and most important, you get far fewer hard-to-interpret type error messages.

Strong Vs Weak Typing

- Strongly typed language: such languages will execute without producing uncaught type errors at runtime.
  - no invalid memory access
    - no seg fault
    - array index out of range
    - access of null pointer
  - No invalid type casts
- Weakly typed: uncaught type errors can lead to undefined behavior at runtime
- In practice, these terms used in a relative sense
- Strong typing does not imply static typing

Type Conversion

- Explicit: Functions are used to perform conversion.
  - example: strtol, atoi, itoa in C; float and int etc.
- Implicit conversion (coercion)
  - example:
    - If a is float and b is int then type of a+b is float
    - Before doing the addition, b must be converted to a float value. This conversion is done automatically.
- Casting (as in C)
- Invisible “conversion:” in untagged unions

Data Types Summary

- Simple/built-in types
- Compound types (and their type expressions)
  - Product, union, recursive, array, pointer
- Parametric Vs subtype polymorphism, Code reuse
- Polymorphism in OCAML, C++,
- Type equivalence
  - Name, structure and declaration equivalence
- Type compatibility
- Type inference, type-checking, type-coercion
- Strong Vs Weak, Static Vs Dynamic typing

OOP (Object Oriented Programming)

- So far the languages that we encountered treat data and computation separately.
- In OOP, the data and computation are combined into an "object".

Benefits of OOP

- more convenient: collects related information together, rather than distributing it.
  - Example: C++ iostream class collects all I/O related operations together into one central place.
  - Contrast with C I/O library, which consists of many distinct functions such as getchar, printf, scanf, sscanf, etc.
- centralizes and regulates access to data.
  - If there is an error that corrupts object data, we need to look for the error only within its class
  - Contrast with C programs, where access/modification code is distributed throughout the program

Benefits of OOP (Continued)

- Promotes reuse.
  - by separating interface from implementation.
    - We can replace the implementation of an object without changing client code.
    - Contrast with C, where the implementation of a data structure such as a linked list is integrated into the client code
  - by permitting extension of new objects via inheritance.
    - Inheritance allows a new class to reuse the features of an existing class.
    - Example: define doubly linked list class by inheriting/ reusing functions provided by a singly linked list.

Encapsulation & Information hiding

- Encapsulation
  - centralizing/regulating access to data
- Information hiding
  - separating implementation of an object from its interface
- These two terms overlap to some extent.

Classes and Objects

- Class is an (abstract) type
  - includes data
    - class variables (aka static variables)
      - shared (global) across all objects of this class
    - instance variables (aka member variables)
      - independent copy in each object
      - similar to fields of a struct
  - and operations
    - member functions
      - always take object as implicit (first) argument
    - class functions (aka static functions)
      - don't take an implicit object argument
- Object is an instance of a class
  - variable of class type

Access to Members

- Access to members of an object is regulated in C++ using three keywords
  - Private:
    - Accessibly only to member functions of the class
    - Can't be directly accessed by outside functions
  - Protected:
    - allows access from member functions of any subclass
  - Public:
    - can be called directly by any piece of code.

Member Function

- Member functions are of two types
  - statically dispatched
  - dynamically dispatched.
- The dynamically dispatched functions are declared using the keyword "virtual" in C++
  - all member function functions are virtual in Java

C++

- Developed as an *extension* to C

  by adding object oriented constructs originally found in Smalltalk (and Simula67).
- Most legal C programs are also legal C++ programs
  - "Backwards compatibility" made it easier for C++ to be accepted by the programming community
  - . . . but made certain features problematic (leading to "dirty" programs)
- Many of C++ features have been used in Java
  - Some have been "cleaned up"
  - Some useful features have been left out

Example of C++ Class

- A typical convention is C++ is to make all data members private. Most member functions are public.
- Consider a list that consists of integers. The declaration for this could be :

```
class IntList {
   private:
      int elem; // element of the list
      IntList *next ; // pointer to next element
   public:
      IntList (int first); //"constructor"
      ~IntList () ; // "destructor".
      void insert (int i); // insert element i
      int getval () ; // return the value of elem
      IntList *getNext (); // return the value of next
}
```

Example of C++ Class (Continued)

- We may define a subclass of IntList that uses doubly linked lists as follows:

```
class IntDList: IntList {
   private:
      IntList *prev;
   public:
      IntDlist(int first);
      // Constructors need to be redefined
       ~IntDlist();
      // Destructors need not be redefined, but
      // typically this is needed in practice.
      // Most operations are inherited from IntList.
      // But some operations may have to be redefined.
      insert (int);
      IntDList *prev();
 }
```

C++ and Java: The Commonalities

- Classes, instances (objects), data members (fields) and member functions (methods).
- Overloading and inheritance.
  - base class (C++) → superclass (Java)
  - derived class (C++) → subclass (Java)
- Constructors
- Protection (visibility): `private`, `protected` and `public`
- Static binding for data members (fields)

A C++ Primer for Java Programmers

Classes, fields and methods:

Java:

```java
class A extends B {
  private int x;
  protected int y;
  public int f() {
      return x;
    }
  public void print() {
      System.out.println(x);
  }
}
```

C++:

```cpp
class A : public B {
  private: int x;
  protected: int y;
  public: int f() {
      return x;
    }
  void print() {
      std::cout << x << std::endl;
  }
}
```

A C++ Primer for Java Programmers

Declaring objects:

- In Java, the declaration `A va` declares va to be a *reference* to object of class A.
  - Object creation is always via the new operator
- In C++, the declaration `A va` declares va to be an object of class A.
  - Object creation may be automatic (using declarations) or via new operator:

```
A *va = new A;
```

Objects and References

- In Java, all objects are allocated on the heap; references to objects may be stored in local variables.
- In C++, objects are treated analogous to *C* `structs`: they may be allocated and stored in local variables, or may be dynamically allocated.
- Parameters to methods:
  - Java distinguishes between two sets of values: primitives (e.g. `ints`, `floats`, etc.) and objects (e.g `String`, `Vector`, etc.

    Primitive parameters are passed to methods *by value* (copying the value of the argument to the formal parameter)

    Objects are passed *by reference* (copying only the reference, not the object itself).
  - C++ passes all parameters *by value* unless specially noted.

Type

- **Apparent Type:** Type of an object as per the declaration in the program.
- **Actual Type:** Type of the object at run time.

Let `Test` be a subclass of `Base`. Consider the following Java program:

```
Base b = new Base();
Test t = new Test();
...
b = t;
```

Variable	*Apparent type of object referenced*
`b`	`Base`
`t`	`Test`

... throughout the scope of `b` and `t`'s declarations

Type (Continued)

Let `Test` be a subclass of `Base`. Consider the following Java program fragment:

```
Base b = new Base();
Test t = new Test();
...
b = t;
```

Variable	*Program point*	*Actual type of object referenced*
b	**before** b=t	Base
t	**before** b=t	Test
b	**after** b=t	Test
t	**after** b=t	Test

Type (Continued)

Things are a bit different in C++, because you can have both objects and object references. Consider the case where variables are objects in C++:

```
Base b();
Test t();
...
b = t;
```

Variable	*Program point*	*Actual type of object referenced*
`b`	**before** `b=t`	`Base`
`t`	**before** `b=t`	`Test`
`b`	**after** `b=t`	`Base`
`t`	**after** `b=t`	`Test`

Type (Continued)

Things are a bit different in C++, because you can have both objects and object references. Consider the case where variables are pointers in C++:

```
Base *b = new Base();
Test *t = new Test();
...
b = t;
```

Variable	*Program point*	*Actual type of object referenced*
b	**before** b=t	Base*
t	**before** b=t	Test*
b	**after** b=t	Test*
t	**after** b=t	Test*

Subtype

- A is a subtype of B if every object of type A is also a B, i.e., every object of type A has
  - (1) all of the data members of B
  - (2) supports all of the operations supported by B, with the operations taking the same argument types and returning the same type.
  - (3) AND these operations and fields have the "same meaning" in A and B.
- It is common to view data field accesses as operations in their own right. In that case, (1) is subsumed by (2) and (3).

Subtype Principle

- A key principle :
  - “For any operation that expects an object of type T, it is acceptable to supply object of type T’, where T’ is subtype of T.”
- The subtype principle enables OOL to support subtype polymorphism:
  - client code that accesses an object of class C can be reused with objects that belong to subclasses of C.

Subtype Principle (Continued)

- The following function will work with any object whose type is a subtype of IntList.

```
void q (IntList &i, int j) {
   ...
   i.insert(j) ;
}
```

- Subtype principle dictates that this work for IntList and IntDList.
  - This must be true even is the insert operation works differently on these two types.
  - Note that use of IntList::insert on IntDList object will likely corrupt it, since the prev pointer would not be set.

Subtype Principle (Continued)

- Hence, `i.insert` must refer to
  - IntList::insert when `i` is an IntList object, and
  - IntDList::insert function when i is an IntDList.
- Requires dynamic association between the name "insert" and the its implementation.
  - achieved in C++ by declaring a function be virtual.
  - definition of insert in IntList should be modified as follows: `virtual void insert(int i);`
  - all member functions are by default virtual in Java, while they are nonvirtual in C++
    - equivalent of "virtual" keyword is unavailable in Java.

Reuse of Code

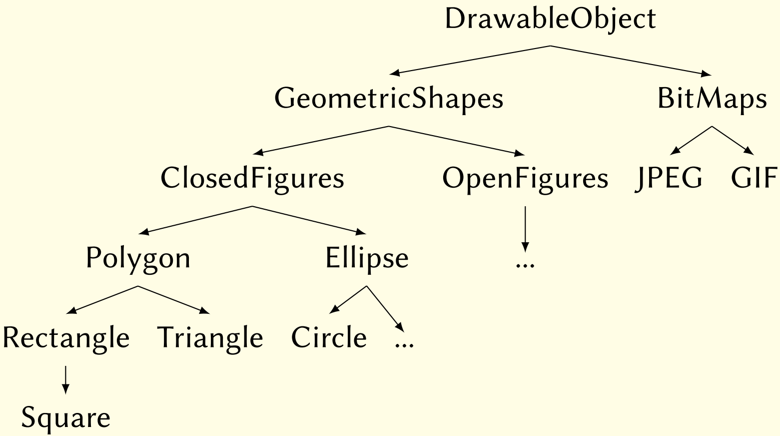
- Reuse achieved through subtype polymorphism
  - the same piece of code can operate on objects of different type, as long as:
    - Their types are derived from a common base class
    - Code assumes only the interface provided by base class.
- Polymorphism arises due to the fact that the implementation of operations may differ across subtypes.

Reuse of Code (Continued)

- Example:
  - Define a base class called DrawableObject
    - supports draw() and erase().
  - DrawableObject just defines an interface
    - no implementations for the methods are provided.
    - this is an abstract class — a class with one or more abstract methods (declared but not implemented).
    - also an interface class — contains only abstract methods subtypes.

Reuse of Code: example (Continued)

- The hierarchy of DrawableObject may look as follows:



Reuse of Code: example (Continued)

- The subclasses support the draw() and erase() operation supported by the base class.
- Given this setting, we can implement the redraw routine using the following code fragment:

```
void redraw(DrawableObject* objList[], int size){
   for (int i = 0; i < size; i++)
   objList[i]->draw();
}
```

Reuse of Code: example (Continued)

- objList[i].draw will call the appropriate method:
  - for a square object, Square::draw
  - for a circle object, Circle:draw
- The code need not be changed even if we modify the inheritance hierarchy by adding new subtypes.

Reuse of Code: example (Continued)

- Compare with implementation in C:

```
void redraw(DrawableObject *objList[], int size) {
   for (int i = 0; i < size; i++){
      switch (objList[i]->type){
         case SQUARE: square_draw((struct Square *)objList[i]);
            break;
         case CIRCLE: circle_draw((struct Circle *)objList[i]);
            break;
         ........
         default: ....
      }
   }
}
```

- Differences:
  - no reuse across types (e.g., Circle and Square)
  - need to explicitly check type, and perform casts
  - will break when new type (e.g., Hexagon) added

Reuse of Code (Continued)

- Reuse achieved through subtype polymorphism
  - the same piece of code can operate on objects of different type, as long as:
    - Their types are derived from a common base class
    - Code assumes only the interface provided by base class.
- Polymorphism arises due to the fact that the implementation of operations may differ across subtypes.

Dynamic Binding

- Dynamic binding provides overloading rather than parametric polymorphism.
  - the draw function implementation is not being shared across subtypes of DrawableObject, but its name is shared.
- Enables client code to be reused
- To see dynamic binding more clearly as overloading:
  - Instead of a.draw(),
  - view as draw(a)

Reuse of Code (Continued)

- Subtype polymorphism = function overloading
- Implemented using dynamic binding
  - i.e., function name is resolved at runtime, rather than at compile time.
- Conclusion: just as overloading enables reuse of client code, subtype polymorphism enables reuse of client code.

Inheritance

- language mechanism in OO languages that can be used to implement subtypes.
- The notion of interface inheritance corresponds conditions (1), (2) and (3) in the definition of Subtype
- but provision (3) is not checked or enforced by a compiler.

Subtyping & interface inheritance

- The notion of subtyping and interface inheritance coincide in OO languages.
  OR
- Another way to phrase this is to say that “interface inheritance captures an ’is-a’ relationship”
  OR
- If A inherits B’s interface, then it must be the case that every A is a B.

Implementation Inheritance

- If A is implemented using B, then there is an implementation inheritance relationship between A and B.
  - However A need not support any of the operations supported by B

    OR
  - There is no `is-a` relationship between the two classes.
- Implementation inheritance is thus "irrelevant" from the point of view of client code.
- Private inheritance in C++ corresponds to implementation-only inheritance, while public inheritance provides both implementation and interface inheritance.

Implementation Inheritance (Continued)

- Implementation-only inheritance is invisible outside a class
  - not as useful as interface inheritance.
  - can be simulated using composition.

```
class B{
   op1(...)
   op2(...)
}
class A: private class B {
   op1(...) /* Some operations supported by B may also be supported
             A (e.g., op1), while others (e.g., op2) may not be */
   op3(...) /* New operations supported by A */
}
```

Implementation Inheritance (Continued)

- The implementation of op1 in A has to explicitly invoke the implementation of op1 in B:

```
A::op1(...){
    B::op1(...)
}
```

- So, we might as well use composition:

```
class A{
    B b;
    op1(...) {  b.op1(...) }
    op3(...)...
}
```

Polymorphism

"*The ablilty to assume different forms*"

- A function/method is polymorphic if it can be applied to values of many types.
- Class hierarchy and inheritance provide a form of polymorphism called *subtype polymorphism*.
- As dicussed earlier, it is a form of overloading.
  - Overloading based on the first argument alone.
  - Overloading resolved dynamically rather than statically.
- Polymorphic functions increase code reuse.

Polymorphism (Continued)

- Consider the following code fragment: `(x < y)? x : y`
- "Finds the minimum of two values".
- The same code fragment can be used regardless of whether x and y are:
  - integers
  - floating point numbers
  - objects whose class implements operator "<".
- *Templates* lift the above form of polymorphism (called *parametric* polymorphism) to functions and classes.

Parametric polymorphism Vs Interface Inheritance

- In C++,
  - template classes support parametric polymorphism
  - public inheritance support interface + implementation inheritance.
- Parametric polymorphism is more flexible in many cases.

```
template class List<class ElemType>{
   private:
      ElemType *first; List<ElemType> *next;
   public:
      ElemType *get(); void insert(ElemType *e);
}
```

- Now, one can use the List class with any element type:

```
void f(List<A> alist, List<B> blist){
   A a = alist.get();
   B b = blist.get();
}
```

Parametric polymorphism Vs Inheritance (Continued)

- If we wanted to write a List class using only subtype polymorphism:
  - We need to have a common base class for A and B
  - e.g., in Java, all objects derived from base class “Object”

```
class AltList{
   private:
      Object first; AltList next;
   public:
      Object get(); void insert(Object o);
}

void f(AltList alist, AltList blist) {
   A a = (A)alist.get();
   B b = (B)blist.get();
}
```

Parametric polymorphism Vs Interface Inheritance (Continued)

- Note: get() returns an object of type Object, not A.
- Need to explicitly perform runtime casts.
  - type-checking needs to be done at runtime, and type info maintained at runtime
  - potential errors, as in the following code, cannot be caught at compile time

```
List alist, blist;
A a; A b;//Note b is of type A, not B
alist.insert(a);
blist.insert(b);
f(alist, blist);//f expects second arg to be list of B's, but we are giving a list of A's.
```

Overloading, Overriding, and Virtual Functions

- Overloading is the ability to use the same function NAME with different arguments to denote DIFFERENT functions.
- In C++
  - void add(int a, int b, int& c);
  - void add(float a, float b, float& c);
- Overriding refers to the fact that an implementation of a method in a subclass supersedes the implementation of the same method in the base class.

Overloading, Overriding, and Virtual Functions (Continued)

- Overriding of non-virtual functions in C++:

```
class B {
   public:
      void op1(int i) { /* B's implementation of op1 */ }
}
class A: public class B {
   public:
      void op1(int i) { /* A's implementation of op1 */ }
}
main() {
   B b; A a;
   int i = 5; b.op1(i); // B's implementation of op1 is used
   a.op1(i); // Although every A is a B, and hence B's implementation of
             // op1 is available to A, A's definition supercedes B's defn,
             // so we are using A's implementation of op1.
   ((B)a).op1(); // Now that a has been cast into a B, B's op1 applies.
   a.B::op1(); // Explicitly calling B's implementation of op1
}
```

Overloading, Overriding, and Virtual Functions (Continued)

- In the above example the choice of B's or A's version of op1 to use is based on compile-time type of a variable or expression. The runtime type is not used.
- Overloaded (non-member) functions are also resolved using compile-time type information.

Overriding In The Presence Of Virtual Function

```
class B {
   public:
      virtual void op1(inti){/* B's implementation of op1 */ }
}
class A: public class B {
   public:
      void op1(int i) {// op1 is virtual in base class, so it is virtual here too
      /* A's implementation of op1 */ }
}
main() {
   B b; A a;
   int i = 5;
   b.op1(i); // B's implementation of op1 is used
   a.op1(i); // A's implementation of op1 is used.
   ((B)a).op1(); // Still A's implementation is used
   a.B::op1(); // Explicitly requesting B's definition of op1
}
```

Overriding In The Presence Of Virtual Function (Continued)

```
void f(B x, int i) {
   x.op1(i);
}
```

- which may be invoked as follows:

```
B b;
A a;
f(b, 1); // f uses B's op1
f(a, 1); // f still uses B's op1, not A's
```

```
void f(B& x, int i) {
   x.op1(i);
}
```

- which may be invoked as follows:

```
B b;
A a;
f(b, 1); // f uses B's op1
f(a, 1); // f uses A's op1
```

Function Template

- Declaring function templates:

```
template <typename T>
T min (  T x,   T y  ) {
return (x < y)? x : y;
}
```

- typename parameter can be name of any type (e.g. `int`, `long`, `Base`, . . .)
- Using template functions:
  - `z = min(x, y)`
  - Compiler fills out the template's typename parameter using the types of arguments.
  - Can also be explicitly used as: min<float>(x, y)

Class Templates

- Of great importance in implementing data structures (say list of elements, where all elements have to be of the same type).
- Java does not provide templates:
  - Some uses of templates can be replaced by using Java `interfaces`.
  - Many other uses would require "type casting"

    e.g.:

    ```
    Iterator e = ...
    Int x = (Integer) e.next();
    ```
  - Inherently dangerous since it skirts around compile-time type checking.

Dynamic Binding

- A function f may take parameters of class C1
- The actual parameter passed into the function may be of class C2 that is a subclass of C1
- Methods invoked on this parameter within f will be the member function supported by C2, rather than C1
- To do this, we have to identify the appropriate member function at runtime, based on the actual type C2 of the parameter, and not the (statically) determined type C1

Dynamic Binding (Continued)

- Dynamic binding provides overloading rather than parametric polymorphism.

```
void q (IntList &i, int j) {
   ...
   i.insert(j) ;
}
```

  - the `insert` function implementation is not being shared across subtypes of IntList, but its name is shared.

- enables client code to be reused
- To see dynamic binding as overloading, we need to eliminate the "syntactic sugar" used for calling member functions in OOL:
  - Instead of viewing it as `i.insert(...)`, we would think of it as a simple function `insert(i,...)` that explicitly takes an object as an argument.

Implementation of OO-Languages

- Data
  - nonstatic data (aka instance variables) are allocated within the object
    - the data fields are laid out one after the other within the object
    - alignment requirements may result in "gaps" within the object that are unused
    - each field name is translated at compile time into a number that corresponds to the offset within the object where the field is stored
  - static data (aka class variables) are allocated in a static area, and are shared across all instances of a class.
    - Each class variable name is converted into an absolute address that corresponds to the location within the static area where the variable is stored.

Implementation of Dynamic Binding

- All virtual functions corresponding to a class C are put into a virtual method table (VMT) for class C
- Each object contains a pointer to the VMT corresponding to the class of the object
- This field is initialized at object construction time
- Each virtual function is mapped into an index into the VMT. Method invocation is done by
  - access the VMT table by following the VMT pointer in the object
  - look up the pointer for the function within this VMT using the index for the member function

Implementation of Inheritance

- Key requirement to support subtype principle:
  - a function f may expect parameter of type C1, but the actual parameter may be of type C2 that is a subclass of C1
  - the function f must be able to deal with an object of class C2 as if it is an object of class C1
    - this means that all of the fields of C2 that are inherited from C1, including the VMT pointer, must be laid out in the exact same way they are laid out in C1
    - all functions in the interface of C1 that are in C2 must be housed in the same locations within the VMT for C2 as they are located in the VMT for C1

Impact of subtype principle on Implementation (Continued)

- In order to satisfy the constraint that VMT (Virtual Method Table) ptr appear at the same position in objects of type A and B, it is necessary for the data field f in A to appear after the VMT field.
- A couple of other points:
  - non-virtual functions are statically dispatched, so they do not appear in the VMT table
  - when a virtual function f is NOT redefined in a subclass, the VMT table for that class is initialized with an entry to the function f defined its superclass.

Summary

- The key properties of OOL are:
  - encapsulation
  - inheritance+dynamic binding

Type Checking: Declarations

$$
\begin{array}{llll}
T & \longrightarrow & \text{int} & \{\ T.type = int;\ \} \\
T & \longrightarrow & \text{float} & \{\ T.type = float;\ \} \\
D & \longrightarrow & T\ \ \text{id} & \{\ D.type = T.type; \\
 & & & sym\_enter(\text{id}.name,\ D.type);\ \} \\
D & \longrightarrow & D_1,\ \text{id} & \{\ D.type = D_1.type; \\
 & & & sym\_enter(\text{id}.name,\ D.type);\ \}
\end{array}
$$

Type Checking Expressions

$$
\begin{array}{lcll}
E & \longrightarrow & \text{int\_const} & \{\ E.type = int;\ \} \\
E & \longrightarrow & \text{float\_const} & \{\ E.type = float;\ \} \\
E & \longrightarrow & \text{id} & \{\ E.type = sym\_lookup(\text{id}.name, \texttt{type});\ \} \\
E & \longrightarrow & E_1 + E_2 & \{\ \text{if } (E_1.type \notin \{int, float\}) \text{ OR} \\
 & & & \qquad\qquad (E_2.type \notin \{int, float\}) \\
 & & & \qquad E.type = error; \\
 & & & \quad \text{else if } E_1.type == E_2.type == int \\
 & & & \qquad E.type = int; \\
 & & & \quad \text{else } E.type = float; \\
 & & & \}
\end{array}
$$

Type Checking (contd.)

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 \; [\; E_2 \;] & \{ \text{if } E_1.type == \text{array}(\mathbf{S}, \mathbf{T}) \text{ AND} \\
 & & & \quad E_2.type == \text{int} \\
 & & & \quad E.type = \mathbf{T} \\
 & & & \text{else } E.type = \text{error} \} \\
E & \longrightarrow & * \; E_1 & \{ \text{if } E_1.type == \text{ptr}(\mathbf{T}) \\
 & & & \quad E.type = \mathbf{T} \\
 & & & \text{else } E.type = \text{error} \} \\
E & \longrightarrow & \& \; E_1 & \{ E.type = \text{ptr}(E_1.type) \}
\end{array}
$$

Type Checking (contd.)

$$
\begin{array}{lcll}
E & \longrightarrow & E_1\ E_2 & \{ \text{if } E_1.type \equiv \text{arrow}(\mathbf{S}, \mathbf{T}) \text{ AND} \\
 & & & \qquad\qquad E_2.type \equiv \mathbf{S} \\
 & & & \qquad E.type = \mathbf{T} \\
 & & & \text{else} \\
 & & & \qquad E.type = \text{error} \,\} \\
E & \longrightarrow & (\ E_1,\ E_2\ ) & \{\ E.type = \text{tuple}(E_1.type,\ E_2.type)\ \}
\end{array}
$$

Resolving Names

What entity is represented by `t.area()`?

- Determine the type of `t`.

  `t` has to be of type user(c).

- If c has a method of name `area`, we are done.

  Otherwise, if the superclass of c has a method of name `area`, we are done.

  Otherwise, if the superclass of superclass of c...

  $\Longrightarrow$ Determine the nearest *superclass* of class c that has a method with name `area`.

Resolving Names (contd.)

```
class Rectangle {
  int x,y; // top lh corner
  int l, w; // length and width

  Rectangle move() {
    x = x + 5;     y = y + 5;
    return this;
  }
  Rectangle move(int dx, int dy) {
    x = x + dx;    y = y + dy;
    return this;
  }
}
```

Resolving Names (contd.)

What entity is represented by `move` in `r.move(3, 10)`?

- Determine the type C of `r`.
- Determine the nearest *superclass* of class C that has a method with name `move` ***such that*** `move` ***is a method that takes two*** `int` ***parameters.***

Type Checking Statements

$$
\begin{array}{lcll}
S & \longrightarrow & id \coloneqq E & \{\ \text{if isSubType}(E.type,\ id.type) \\
 & & & \qquad\qquad S.type == \text{void} \\
 & & & \text{else } S.type = \text{error}\ \} \\
S & \longrightarrow & S_1;\ S_2 & \{\ \text{if } (S_1.type == S_2.type == void) \\
 & & & \qquad\qquad S.type == \text{void} \\
 & & & \text{else } S.type = \text{error}\ \} \\
S & \longrightarrow & \text{if } E \text{ then} & \\
 & & \quad S_1 \text{ else } S_2 & \{\ \text{if } (S_1.type == S_2.type == void) \\
 & & & \qquad \&\&\ (E.type == bool) \\
 & & & \qquad\qquad S.type == \text{void} \\
 & & & \text{else } S.type = \text{error}\ \}
\end{array}
$$

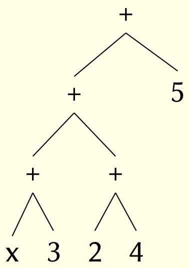
CSE 504: Compilers

Evaluation and Runtime Environments

R. Sekar

Expression evaluation

- Order of evaluation
- For the abstract syntax tree



- the equivalent expression is (x + 3) + (2 + 4) + 5

Expression evaluation (Continued)

- One possible semantics:
  - evaluate AST bottom-up, left-to-right.
- This constrains optimization that uses mathematical properties of operators
- (e.g. commutativity and associativity)
  - e.g.,it may be preferable to evaluate of e1+(e2+e3)instead of (e1+e2)+e3
  - (x+0)+(y+3)+(z+4)=>x+y+z+0+3+4=>x+y+z+7
  - the compiler can evaluate 0+3+4 at compile time, so that at runtime, we have two fewer addition operations.

Expression evaluation (Continued)

- Some languages leave order of evaluation unspecified.
  - order of evaluation of procedure parameters are also unspecified.
- Problem:
  - Semantics of expressions with side-effects, e.g., `(x++) + x`
  - If initial value of x is 5
    - left-to-right evaluation yields 11 as answer, but
    - right-to-left evaluation yields 10
- So, languages with expressions with side-effects forced to specify evaluation order
- Still, a bad programming practice to use expressions where different orders of evaluation can lead to different results
  - Impacts readability (and maintainability) of programs

Left-to-right evaluation

- Left-to-right evaluation with short-circuit semantics is appropriate for boolean expressions.

  e1&&e2: e2 is evaluated only if e1 evaluates to true.

  e1||e2: e2 is evaluated only if e1 evaluates to false.

- This semantics is convenient in programming:
  - Consider the statement: `if((i<n) && a[i]!=0)`
  - With short-circuit evaluation, `a[i]` is never accessed if `i>= n`
  - Another example: `if ((p!=NULL) && p->value>0)`

Left-to-right evaluation (Continued)

- Disadvantage:
  - In an expression like "if((a==b)||(c=d))"
  - The second expression has a statement. The value of c may or may not be the value of d, depending on if a == b is true or not.
- Bottom-up:
  - No order specified among unrelated subexpressions.
  - Short-circuit evaluation of boolean expressions.
- Delayed evaluation
  - Delay evaluation of an expressions until its value is absolutely needed.
  - Generalization of short-circuit evaluation.

Control Statements

- Structured Control Statements:
- Case Statements:
  - Implementation using if-then-else
  - Understand semantics in terms of the semantics of simple constructs
  - actual implementation in a compiler
- Loops
  - while, repeat, for

If-Then-Else

- If-then-else. It is in two forms:
  - if cond then s1 else s2
  - if cond then s1
- evaluate condition: if and only if evaluates to true, then evaluate s1 otherwise evaluate s2.

Case (Switch) Statement

- Case statement

```
switch(<expr>){
  case <value> :
  case <value> :
  ...
  default :
}
```

- Evaluate “<expr>” to get value v. Evaluate the case that corresponds to v.
- Restriction:
  - “<value>” has to be a constant of an original type e.g., int, enum
  - Why?

Implementation of case statement

- Naive algorithm:
  - Sequential comparison of value v with case labels.
  - This is simple, but inefficient. It involves O(N) comparisons

```
switch(e){
  case 0:s0;
  case 1:s1;
  case 2:s2;
  case 3:s3;
}
```

- can be translated as:

```
v = e;
if (v==0) s0;
else if (v == 1) s1;
else if (v == 2) s2;
else if (v == 3) s3;
```

Implementation of case statement (Continued)

- Binary search:
  - O(log N) comparisons, a drastic improvement
  - over sequential search for large N.
- Using this, the above case statement can be translated as

```
v = e;
if (v<=1)
   if (v==0) s0;
   else if (v==1) s1;
else if (v>=2)
   if (v==2) s2;
   else if (v==3) s3;
```

Implementation of case statement (Continued)

- Another technique is to use hash tables.
- This maps the value v to the case label that corresponds to the value v.
- This takes constant time (expected).

Control Statements (contd.)

- while:
  - let s1 = while C do S
  - then it can also be written as
  - s1 = if C then {S; s1}
- repeat:
  - let s2 = repeat S until C
  - then it can also be written as
  - s2 = S; if (!C) then s2
- loop
  - let s = loop S end
  - its semantics can be understood as S; s
  - S should contain a break statement, or else it won't terminate.

For-loop

- Semantics of for (S2; C; S3) S can be specified in terms of while:
  - S2; while C do { S; S3 }
- In some languages, additional restrictions imposed to enable more efficient code
  - Value of index variable can't change loop body, and is undefined outside the loop
  - Bounds may be evaluated only once

Unstructured Control Flow

- Unstructured control transfer statements (goto) can make programs hard to understand:

```
40:if (x > y) then goto 10
  if (x < y) then goto 20
  goto 30
10:x = x - y
  goto 40
20:y = y -x
  goto 40
30:gcd = x
```

Unstructured Control Flow (Continued)

- Unstructured control transfer statements (goto) can make programs hard to understand:

```
40:if (x > y) then goto 10
  if (x < y) then goto 20
  goto 30
10:x = x - y
  goto 40
20:y = y -x
  goto 40
30:gcd = x
```

- Equivalent program with structured control statements is easier to understand:

```
while (x!=y) {
  if (x > y) then x=x-y
  else y=y-x
}
```

Control Statements (contd.)

- goto should be used in rare circumstances
  - e.g., error handling.
- Java doesn't have goto. It uses labeled break instead:

```
l1:  for ( ...  )  {
       while (...)  {
         ....
         break l1
       }
     }
```

- break l1 causes exit from loop labeled with l1

Control Statements (contd.)

- Restrictions in use of goto:
  - jumps across procedures
  - jumps from outer blocks to inner blocks or unrelated blocks

```
goto l1;
if (...) then {
  int x;
  x = 5;
  l1: y = x*x;
}
```

- Jumps from inner to outer blocks are permitted.

Control Statements (Continued)

- Procedure calls:
  - Communication between the calling and the called procedures takes place via parameters.
- Semantics:
  - substitute formal parameters with actual parameters
  - rename local variables so that they are unique in the program
    - In an actual implementation, we will simply look up the local variables in a different environment (callee's environment)
    - Renaming captures this semantics without having to model environments.
  - replace procedure call with the body of called procedure

Parameter-passing semantics

- Call-by-value
- Call-by-reference
- Call-by-value-result
- Call-by-name
- Call-by-need
- Macros

Call-by-value

- Evaluate the actual parameters
- Assign them to corresponding formal parameters
- Execute the body of the procedure.

```
int p(int x) {
   x =x +1 ;
   return x ;
}
```

- An expression y = p(5+3) is executed as follows:
  - evaluate 5+3 = 8, call p with 8, assign 8 to x, increment x, return x which is assigned to y.

Call-by-value (Continued)

- Preprocessing
  - create a block whose body is that of the procedure being called
  - introduce declarations for each formal parameter, and initialize them with the values of the actual parameters
- Inline procedure body
  - Substitute the block in the place of procedure invocation statement.

Call-by-value (Continued)

- Example:

```
int z;
void p(int x){
   z = 2*x;
}
main(){
   int y;
   p(y);
}
```

- Replacing the invocation p(y) as described yields:

```
int z;
main(){
   int y;
   {
      int x1=y;
      z = 2*x1;
   }
}
```

"Name Capture"

- Same names may denote different entities in the called and calling procedures
- To avoid name clashes, need to rename local variables of called procedure
  - Otherwise, local variables in called procedure may be confused with local variables of calling procedure or global variables

Call-by-value (Continued)

- Example:

```
int z;
void p(int x){
   int y = 2;
   z = y*x;
}
main(){
   int y;
   p(y);
}
```

- After replacement:

```
int z;
main(){
   int y;
   {
      int x1=y;
      int y1=2;
      z = y1*x1;
   }
}
```

Call-by-reference

- Evaluate actual parameters (must have l-values)
- Assign these l-values to formal parameters
- Execute the body.

```
int z = 8;
y=p(z);
```

- After the call, y and z will both have value 9.
- Call-by-reference supported in C++, but not in C
  - Effect realized by explicitly passing l-values of parameters using “&” operator

Call-by-reference (Continued)

- Explicit simulation in C provides a clearer understanding of the semantics of call-by-reference:

```
int p(int *x){
   *x = *x + 1;
   return *x;
}
...
int z;
y= p(&z);
```

Call-By-Reference (Continued)

- Example:

```
int z;
void p(int x){
   int y = 2;
   z = y*x;
}
main(){
   int y;
   p(y);
}
```

- After replacement:

```
int z;
main(){
   int y;
   {
      int& x1=y;
      int y1=2;
      z = y1*x1;
   }
}
```

Call-by-value-result

- Works like call by value but in addition, formal parameters are assigned to actual parameters at the end of procedure.

```
void p (int x, int y) {
   x = x +1;
   y = y+ 1;
}
...
int a = 3;
p(a, a) ;
```

- After the call, a will have the value 4, whereas with call-by- reference, a will have the value 5.

Call-by-value-result (Continued)

- The following is the equivalent of call-by-value-result call above:

```
x = a; y =a ;
x = x +1 ;
y =y +1 ;
a =x ; a =y ;
```

- thus, at the end, a = 4.

Call-By-Value-Result (Continued)

- Example:

```
void p(int x, y){
   x = x + 1;
   y = y + 1;
}
main(){
   int u = 3;
   p(u,u);
}
```

- After replacement:

```
main(){
   int u = 3;
   {
      int x1 = u;
      int y1 = u;
      x1 = x1 + 1;
      y1 = y1 + 1;
      u = x1; u = y1;
   }
}
```

Call-by-Name

- Instead of assigning l-values or r-values, CBN works by substituting actual parameter expressions in place of formal parameters in the body of callee
- Preprocessing:
  - Substitute formal parameters in procedure body by actual parameter expressions.
  - Rename as needed to avoid “name capture”
- Inline:
  - Substitute the invocation expression with the modified procedure body.

Call-By-Name (Continued)

- Example:

```
void p(int x, y){
   if (x==0)
       then x=y;
   else{
      x=y+1;
   }
}
main(){
   int u=5; int v=0;
   p(v,u/v);
}
```

- After replacement:

```
main(){
   int u=5; int v=0;
   {
      if (v==0)
         then v=u/v;
      else{
         v=u/v+1;
      }
   }
}
```

Call-By-Need

- Similar to call-by-name, but the actual parameter is evaluated at most once
  - Has same semantics as call-by-name in functional languages
    - This is because the value of expressions does not change with time
  - Has different semantics in imperative languages, as variables involved in the actual parameter expression may have different values each time the expression is evaluated in C-B-Name

Macros

- Macros work like CBN, with one important difference:
  - No renaming of “local” variables
- This means that possible name clashes between actual parameters and variables in the body of the macro will lead to unexpected results.

Macros (Continued)

- given

```
#define sixtimes(y) {int z=0; z = 2*y; y = 3*z;}
main() {
   int x=5, z=3;
   sixtimes(z);
}
```

- After macro substitution, we get the program:

```
main(){
   int x=5,z=3;
   {int z=0; z = 2*z; z = 3*z;}
}
```

Macros (Continued)

- It is different from what we would have got with CBN parameter passing.
- In particular, the name confusion between the local variable z and the actual parameter z would have been avoided, leading to the following result:

```
main() {
    int x = 5, z = 3;
    {
        int z1=0;   // z renamed as z1
        z1 = 2*z;   // y replaced by z without
        z = 3*z1;   // confusion with original z
    }
}
```

Difficulties in Using Parameter Passing Mechanisms

- CBV: Easiest to understand, no difficulties or unexpected results.
- CBVR:
  - When the same parameter is passed in twice, the end result can differ depending on the order in which formals are assigned back to the actual parameters.
  - Otherwise, relatively easy to understand.

Difficulties With CBVR (Continued)

- Example:

```
int f(int x, int y) {
   x=4;
   y=5;
}
void g() {
   int z;
   f(z, z);
}
```

- If assignment of formal parameter values to actual parameters were performed left to right, then z would have a value of 5.
- If they were performed right to left, then z will be 4.

Difficulties in Using CBR

- Aliasing can create problems.

```
int rev(int a[], int b[], int size) {
   for (int i = 0; i < size; i++)
      a[i] = b[size-i-1];
}
```

- The above procedure will normally copy b into a, while reversing the order of elements in b.
- However, if a and b are the same, as in an invocation rev(c,c,4), the result is quite different.
- If c is 1,2,3,4 at the point of call, then its value on exit from rev will be 4,3,3,4.

Difficulties in Using CBN

- CBN is complicated, and can be confusing in the presence of side-effects.
  - actual parameter expression with side-effects:

```
void f(int x) {
   int y = x;
   int z = x;
}
main() {
   int y = 0;
   f(y++);
}
```

- Note that after a call to f, y's value will be 2 rather than 1.

Difficulties in Using CBN (Continued)

- If the same variable is used in multiple parameters.

```
void swap(int x, int y) {
   int tp = x;
   x = y;
   y = tp;
}

main() {
   int a[] = {1, 1, 0};
   int i = 2;
   swap(i, a[i]);
}
```

- When using CBN, by replacing the call to swap by the body of swap: i will be 0, and a will be 2, 1, 0.

Difficulties in Using Macro

- Macros share all of the problems associated with CBN.
- In addition, macro substitution does not perform renaming of local variables, leading to additional problems.

Components of Runtime Environment (RTE)

Static area: allocated at load/startup time.

- Examples: global/static variables and load-time constants.

Stack area: for execution-time data that obeys a last-in first-out lifetime rule.

- Examples: nested declarations and temporaries.

Heap: a dynamically allocated area for “fully dynamic” data, i.e. data that does not obey a LIFO rule.

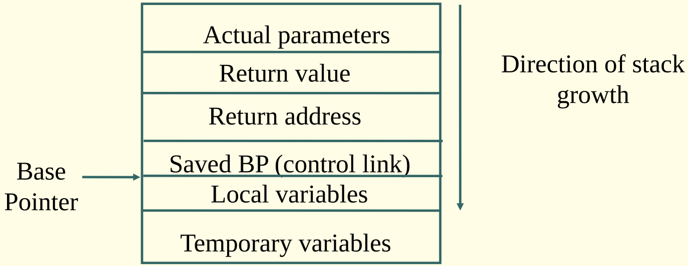
- Examples: objects in Java, lists in OCaml.

Languages and Environments

- Languages differ on where activation records must go in the environment:
  - (Old) Fortran is static: all data, including activation records, are statically allocated.
    - Each function has only one activation record — no recursion!
- Functional languages (Scheme, ML) and some OO languages (Smalltalk) are heap-oriented:
  - almost all data, including AR, allocated dynamically.
- Most languages are in between: data can go anywhere
  - ARs go on the stack.

Procedures and the environment

- An Activation Record (AR) is created for each invocation of a procedure
- Structure of AR:



Access to Local Variables

- Local variables are allocated at a fixed offset on the stack
  - Accessed using this constant offset from BP
    - Example: to load a local variable at offset 8 into the EBX register (x86 architecture)

      mov 0x8(%ebp),%ebx

- Example:

```
{int x; int y;
   { int z; }
   { int w; }
}
```

Steps involved in a procedure call

- Caller
  - Save registers
  - Evaluate actual parameters, push on the stack
    - Push l-values for CBR, r-values in the case of CBV
  - Allocate space for return value on stack (unless return is through a register)
  - Call: Save return address, jump to the beginning of called function
- Callee
  - Save BP (control link field in AR)
  - Move SP to BP
  - Allocate storage for locals and temporaries (Decrement SP)
  - Local variables accessed as [BP-k], parameters using [BP+l]

Steps in return

- Callee
  - Copy return value into its location on AR
  - Increment SP to deallocate locals/temporaries
  - Restore BP from Control link
  - Jump to return address on stack
- Caller
  - Copy return values and parameters
  - Pop parameters from stack
  - Restore saved registers

Example (C):

```
int x;
void p(int y){
   int i = x;
   char c; ...
}
void q (int a){
   int x;
   p(1);
}
main(){
   q(2);
   return 0;
}
```

Non-local variable access

- Requires that the environment be able to identify frames representing enclosing scopes.
- Using the control link results in dynamic scope (and also kills the fixed-offset property).
- If procedures can't be nested (C), the enclosing scope is always locatable:
  - it is global/static (accessed directly)
- If procedures can be nested (Ada, Pascal), to maintain lexical scope a new link must be added to each frame:
  - access link, pointing to the activation of the defining environment of each procedure.

Access Link vs Control Link

- Control Link is a reference to the AR of the caller
- Access link is a reference to the AR of the surrounding scope

Access Link vs Control Link

- Control Link is a reference to the AR of the caller
- Access link is a reference to the AR of the surrounding scope
- **Dynamic Scoping:** When an identifier is not found in the current AR, use *control link* to access caller's AR and look up the name there
  - If not found, keep walking up the control links until name is found

Access Link vs Control Link

- Control Link is a reference to the AR of the caller
- Access link is a reference to the AR of the surrounding scope
- **Dynamic Scoping:** When an identifier is not found in the current AR, use *control link* to access caller's AR and look up the name there
  - If not found, keep walking up the control links until name is found
- **Static Scoping:** When an identifier is not found in the AR of the current function, use *access link* to get to AR for the surrounding scope and look up the name there
  - If not found, keep walking up the access links until the name is found.

Access Link vs Control Link

- Control Link is a reference to the AR of the caller
- Access link is a reference to the AR of the surrounding scope
- **Dynamic Scoping:** When an identifier is not found in the current AR, use *control link* to access caller's AR and look up the name there
  - If not found, keep walking up the control links until name is found
- **Static Scoping:** When an identifier is not found in the AR of the current function, use *access link* to get to AR for the surrounding scope and look up the name there
  - If not found, keep walking up the access links until the name is found.
- **Note:** Except for top-level functions, access links correspond to function scopes, so they cannot be determined statically
  - They need to be "passed in" like a parameter.

Access Link Vs Control Link: Example

```
int q(int x) {
  int p(int y) {
    if (y==0)
      return x+y;
    else {
      int x = 2*p(y-1);
      return x;
    }
  }

  return p(3);
}
```

- If p used its caller's BP to access x, then it ends up accessing the variable x defined within p
  - This would be dynamic scoping.
  - To get static scoping, access should use q's BP
- *Access link:* q explicitly passes a link to its BP
  - Calls to self: pass access link without change.
  - Calls to immediately nested functions: pass your BP
  - Calls to outer functions: Follow your access link to find the right access link to pass
  - Other calls: these will be invalid (like `goto` to an inner block)

Supporting Closures

- *Closures* are needed for
  - Call-by-name and lazy evaluation
  - Returning dynamically constructed functions containing references to variables in surrounding scope
- Variables inside closures may be accessed long after the functions defining them have returned
  - Need to "copy" variable values into the closure, or
  - Not free the AR of functions when they return,
    - i.e., allocate ARs on heap and garbage collect them

Exception Handling

- Example:

```
int fac(int n) {
   if (n <= 0) throw (-1) ; else if (n > 15) throw ("n too large");
   else return n*fac(n-1); }
void g (int n) {
   int k;
   try { k = fac (n) ;}
   catch (int i) { cout << "negative value invalid" ; }
   catch (char *s) { cout << s; }
   catch (...) { cout << "unknown exception" ;}
```

- g(-1) will print “negative value invalid”, g(16) will print “n too large”

Exception Vs Return Codes

- Exceptions are often used to communucate error values from a callee to its caller. Return values provide alternate means of communicating errors.
- Example use of exception handler:

```
float g (int a, int b, int c) {
   float x = fac(a) + fac(b) + fac(c) ; return x ; }
main() {
   try { g(-1, 3, 25); }
   catch (char *s) { cout << "Exception ‘" << s << "’raised, exiting\n"; }
   catch (...) { cout << "Unknown exception, eixting\n";
}
```

- We do not need to concern ourselves with every point in the program where an error may arise.

Exception Vs Return Codes (Continued)

```
float g(int a, int b, int c) {
   int x1 = fac(a);
   if (x1 > 0) {
      int x2 = fac(b);
      if (x2 > 0) {
         int x3 = fac(c);
         if (x3 > 0)
            return x1 + x2 + x3;
         else return x3;
      }
      else return x2;
   }
   else return x1;
}
main() {
   int x = g(-1, 2, 25);
    if (x < 0) { /* identify where error occurred, print */ }
}
```

- Assume that `fac` returns 0 or a negative number to indicated errors
- If return codes were used to indicate errors, then we are forced to check return codes (and take appropriate action) at every point in code.

Use of Exceptions in C++ Vs Java

- In C++, exception handling was an after-thought.
  - Earlier versions of C++ did not support exception handling.
  - Exception handling not used in standard libraries
  - Net result: continued use of return codes for error-checking
- In Java, exceptions were included from the beginning.
  - All standard libraries communicate errors via exceptions.
  - Net result: all Java programs use exception handling model for error-checking, as opposed to using return codes.

Implementation of Exception Handling

- Exception handling can be implemented by adding “markers” to ARs to indicate the points in program where exception handlers are available.
- In C++, entering a try-block at runtime would cause such a marker to be put on the stack
- When exception arises, the RTE gets control and searches down from stack top for a marker.
- Exception then "handed" to the catch statement of this try-block that matches the exception
- If no matching catch statement is present, search for a marker is continued further down the stack, and the whole process is repeated.

Memory Allocation

- A variable is stored in memory at a location corresponding to the variable.
- Constants do not need to be stored in memory.
- Environment stores the binding between variable names and the corresponding locations in memory.
- The process of setting up this binding is known as storage allocation.

Static Allocation

- Allocation performed at compile time.
- Compiler translates all names to corresponding location in the code generated by it.
- Examples:
  - all variables in original FORTRAN
  - all global and static variables in C/C++/Java

Stack Allocation

- Needed in any language that supports the notion of local variables for procedures.
- Also called “automatic allocation”, but this is somewhat of a misnomer now.
- Examples: all local variables in C/C++/Java procedures and blocks.
- Implementation:
  - Compiler translates all names to relative offsets from a location called the “base pointer” or “frame pointer”.
  - The value of this pointer varies will, in general, be different for different procedure invocations

Stack Allocation (Continued)

- The pointer refers to the base of the "activation record" (AR) for an invocation of a procedure.
- The AR holds such information as parameter values, local variables, return address, etc.

```
int fact(int n){
   if n=0 then 1
   else{
      int rv = n*fact(n-1);
      return rv;
   }
}
main(){
   fact(5);
}
```

Stack Allocation (Continued)

- An activation record is created on the stack for each a call to function.
- The start of activation record is pointed to by a register called BP.
- On the first call to fact, BP is decremented to point to new activation record, n is initialized to 5, rv is pushed but not initialized.
- New activation record is created for the next recursive call and so on.
- When n becomes 0, stack is unrolled where successive rv's are assigned the value of n at that stage and the rv of previous stage.

Heap Management

- Issues
  - No LIFO property, so management is difficult
  - Fragmentation
  - Locality
- Models
  - explicit allocation, free
    - e.g., malloc/free in C, new/delete in C++
  - explicit allocation, automatic free
    - e.g., Java
  - automatic allocation, automatic free
    - e.g., Lisp, OCAML, Python, JavaScript

Fragmentation

Internal fragmentation: When asked for x bytes, allocator returns $y > x$ bytes

- $y - x$ represents internal fragmentation

External fragmentation: When (small) free blocks of memory occur in between (i.e., external to) allocated blocks

- the memory manager may have a total of $M \gg N$ bytes of free memory available, but no contiguous block larger enough to satisfy a request of size N.

Fragmentation

Approaches for Reducing Fragmentation

- Use blocks of single size (early LISP)
  - Limits data-structures to use less efficient implementations.
- Use bins of fixed sizes, e.g., 2^n for $n = 0, 1, 2, \ldots$
  - When you run out of blocks of a certain size, break up a block of next available size
  - Eliminates external fragmentation, but increases internal fragmentation
- Maintain bins as LIFO lists to increase locality
- malloc implementations (Doug Lea)
  - For small blocks, use bins of size 8k bytes, $0 < k < 64$
  - For larger blocks, use bins of sizes 2^n for $n > 9$

Coalescing

- What if a program allocates many 8 byte chunks, frees them all and then requests lots of 16 byte chunks?
  - Need to coalesce 8-byte chunks into 16-byte chunks
  - Requires additional information to be maintained
    - for allocated blocks: where does the current block end, and whether the next block is free

Coalescing

Explicit Vs Automatic Management

- Explicit memory management can be more efficient, but takes a lot of programmer effort
- Programmers often ignore memory management early in coding, and try to add it later on
  - But this is very hard, if not impossible
- Result:
  - Majority of bugs in production code is due to memory management errors
    - Memory leaks
    - Null pointer or uninitalized pointer access
    - Access through dangling pointers

Managing Manual Deallocation

- How to avoid errors due to manual deallocation of memory
  - Never free memory!!!
  - Use a convention of object ownership (owner responsible for freeing objects)
    - Tends to reduce errors, but still requires a careful design from the beginning. (Cannot ignore memory deallocation concerns initially and add it later.)
  - Smart data structures, e.g., reference counting objects
  - Region-based allocation
    - When a collection of objects having equal life time are allocated
    - Example: Apache web server's handling of memory allocations while serving a HTTP request

Garbage Collection

- Garbage collection aims to avoid problems associated with manual deallocation
  - Identify and collect garbage automatically
- What is garbage?
  - Unreachable memory
- Automatic garbage collection techniques have been developed over a long time
  - Since the days of LISP (1960s)

Garbage Collection Techniques

- Reference Counting
  - Works if there are no cyclic structures
- Mark-and-sweep
- Generational collectors
- Issues
  - Overhead (memory and space)
  - Pause-time
  - Locality

Reference Counting

- Each heap block maintains a count of the number of pointers referencing it.
- Each pointer assignment increments/decrements this count
- Deallocation of a pointer variable decrements this count
- When reference count becomes zero, the block can be freed

Reference Counting (Continued)

Disadvantages:

- Does not work with cyclic structures
- May impact locality
- Increases cost of each pointer update operation

Advantages:

- Overhead is predictable, fixed
- Garbage is collected immediately, so more efficient use of space

Reference Counting

Mark-and-Sweep

- Mark every allocated heap block as “unreachable”
- Start from registers, local and global variables
- Do a depth-first search, following the pointers
  - Mark each heap block visited as “reachable”
- At the end of the sweep phase, reclaim all heap blocks still marked as unreachable

Mark-and-Sweep

Garbage Collection Issues

- Memory fragmentation
  - Memory pages may become sparsely populated
  - Performance will be hit due to excessive virtual memory usage and page faults
  - Can be a problem with explicit memory management as well
    - But if a programmer is willing to put in the effort, the problem can be managed by freeing memory as soon as possible
- Solution:
  - Compacting GC
    - Copy live structures so that they are contiguous
  - Copying GC

Copying Garbage Collection

- Instead of doing a sweep, simply copy over all reachable heap blocks into a new area
- After the copying phase, all original blocks can be freed
- Now, memory is compacted, so paging performance will be much better
- Needs up to twice the memory of compacting collector, but can be much faster
  - Reachable memory is often a small fraction of total memory

Copying Garbage Collection

Generational Garbage Collection

- Take advantage of the fact that most objects are short-lived
- Exploit this fact to perform GC faster
- Idea:
  - Divide heap into generations
  - If all references go from younger to older generation (as most do), can collect youngest generation without scanning regions occupied by other generations
  - Need to track references from older to younger generation to make this work in all cases

Garbage collection in Java

- Generational GC for young objects
- "Tenured" objects stored in a second region
  - Use mark-and-sweep with compacting
- Makes use of multiple processors if available
- References

  http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html

  http://www.ibm.com/developerworks/java/library/j-jtp11253/

GC for C/C++: Conservative Garbage Collection

- Cannot distinguish between pointers and nonpointers
  - Need "conservative garbage collection"
- The idea: if something "looks" like a pointer, assume that it may be one!
  - Problem: works for finding reachable objects, but cannot modify a value without being sure
    - Copying and compaction are ruled out!
- Reasonable GC implementations are available, but they do have some drawbacks
  - Unpredictable performance
  - Can break some programs that modify pointer values before storing them in memory

Code Generation

- *Intermediate code generation:* Abstract (machine independent) code.
- *Code optimization:* Transformations to the code to improve time/space performance.
- *Final code generation:* Emitting machine instructions.

Syntax Directed Translation

Interpretation:

$$E \longrightarrow E_1 + E_2 \quad \{\, E.val := E_1.val + E_2.val; \,\}$$

Type Checking:

$$E \longrightarrow E_1 + E_2 \quad \{$$

$$\text{if } E_1.type \equiv E_2.type \equiv int$$

$$E.type = int;$$

$$\text{else}$$

$$E.type = float;$$

$$\}$$

Code Generation via Syntax Directed Translation

Code Generation:

$$E \longrightarrow E_1 + E_2 \quad \{ \; E.code = E_1.code \,||\, E_2.code \,||\, \text{"add")} \; \}$$

Intermediate Code

"Abstract" code generated from AST

- **Simplicity and Portability**
  - Machine independent code.
  - Enables common optimizations on intermediate code.
  - Machine-dependent code optimizations postponed to last phase.

Intermediate Forms

- *Stack machine code:*

  Code for a “postfix” stack machine.

- *Two address code:*

  Code of the form “add r_1, r_2”

- *Three address code:*

  Code of the form “add src_1, src_2, $dest$”

  Quadruples and Triples: Representations for three-address code.

Quadruples

Explicit representation of three-address code.

Eixample: `a := a + b * -c;`

Instr	Operation	Arg 1	Arg 2	Result
(0)	`uminus`	`c`		t_1
(1)	`mult`	`b`	t_1	t_2
(2)	`add`	`a`	t_2	t_3
(3)	`move`	t_3		`a`

Triples

Representation of three-address code with implicit destination argument.

Example: `a := a + b * -c;`

Instr	Operation	Arg 1	Arg 2
(0)	`uminus`	`c`	
(1)	`mult`	`b`	(0)
(2)	`add`	`a`	(1)
(3)	`move`	`a`	(2)

Intermediate Forms

Choice depends on convenience of further processing

- Stack code is simplest to generate for expressions.
- Quadruples are most general, permitting most optimizations including code motion.
- Triples permit optimizations such as *common subexpression elimination*, but code motion is difficult.

Static Single Assignment (SSA)

- Each variable is assigned at most once
- ϕ nodes used to combine values of variables after a conditional

```
if (f) x = 1; else x=2;
y=x*x;
```

Becomes

```
if (f) x₁ = 1; else x₂=2;
x₃ = φ(x₁, x₂);
y=x₃*x₃;
```

Generating 3-address code

$E \longrightarrow E_1 + E_2$ {

$E.addr = newtemp();$

$E.code = E_1.code \,||\, E_2.code \,||$

$E.addr \,||\, \text{':='} \,||\, E_1.addr \,||\, \text{' + '} \,||\, E_2.addr;$

}

$E \longrightarrow \text{int}$ {

$E.addr = newtemp();$

$E.code = E.addr \,||\, \text{':='} \,||\, \text{int}.val;$

}

$E \longrightarrow \text{id}$ {

$E.addr = \text{id}.name;$

$E.code = \text{''};$

}

Generation of Postfix Code for Boolean Expressions

$$
\begin{aligned}
E \quad & \longrightarrow E_1 \;\&\&\; E_2 \; \{ \\
& E.code = E_1.code \;||\; \\
& \qquad\qquad E_2.code \;||\; \\
& \qquad\qquad gen(\mathtt{and}) \\
\}\; & \\
E \quad & \longrightarrow \;!\; E_1 \; \{ \\
& E.code = E_1.code \;||\; \\
& \qquad\qquad gen(\mathtt{not}) \\
\}\; & \\
E \quad & \longrightarrow \mathsf{true} \; \{ E.code = gen(\mathtt{load\_immed},\, 1) \} \\
E \quad & \longrightarrow \mathsf{id} \; \{ E.code = gen(\mathtt{load},\, \mathsf{id}.addr) \}
\end{aligned}
$$

Code for Boolean Expressions

```
if ((p != NULL)
  && (p->next != q)) {
  ... then part
} else {
  ... else part
}
```

```
 load(p);
 null();
 neq();
 load(p);
 ildc(1);
  getfield();
 load(q);
 neq();
 and();
  jnz elselabel;
  ...    then part
elselabel:
  ...    else part
```

Shortcircuit Code

```
if ((p != NULL)
  && (p->next != q)) {
  ... then part
} else {
  ... else part
}
```

```
 load(p);
 null();
 neq();
  jnz elselabel;
 load(p);
 ildc(1);
  getfield();
 load(q);
 neq();
  jnz elselabel;
  ...    then part
elselabel:
  ...    else part
```

l- and *r*-Values

```
i := i + 1;
```

- ***l*-value**: location where the value of the expression is stored.
- ***r*-value**: actual value of the expression

Computing *l*-values

$E \longrightarrow$ id {

$E.lval$ = id.loc;

$E.code$ = ‘’; }

$E \longrightarrow E_1$ [E_2] {

$E.lval = newtemp();$

$x = newtemp();$

$E.lcode = E_1.lcode \,||\, E_2.code \,||$

$x \,||$ ’:=’ $||\, E_2.rval \,||$ ’*’ $||\, E_1.elemsize \,||$

$E.lval \,||$ ’:=’ $||\, E_1.lval \,||$ ’+’ $||\, x$ }

$E \longrightarrow E_1$. id { // for field access

$E.lval = newtemp();$

$E.lcode = E_1.lcode \,||$

$E.lval \,||$ ’:=’ $||\, E_1.lval \,||$ ’+’ $||$ id.$offset$ }

Computing lval and rval attributes

$E \longrightarrow E_1 = E_2 \{$

$\quad E.code = E_1.lcode \,||\, E_2.code \,||$

$\qquad gen(\text{'*'}\; E_1.lval \;\text{':='}\; E_2.rval)$

$\quad E.rval = E_2.rval \}$

$E \longrightarrow E_1 \,[\, E_2 \,]\, \{$

$\quad E.lval = newtemp();$

$\quad E.rval = newtemp();$

$\quad x = newtemp();$

$\quad E.lcode = E_1.lcode \,||\, E_2.code \,||$

$\qquad gen(x \;\text{':='}\; E_2.rval * E_1.elemsize) \,||$

$\qquad gen(E.lval \;\text{':='}\; E_1.lval \;\text{'+'}\; x)$

$\quad E.code = E.lcode \,||$

$\qquad gen(E.rval \;\text{':='}\; \text{'*'}\; E.lval)$

$\}$

Function Calls (Call-by-Value)

$$
\begin{aligned}
E \quad \longrightarrow \quad & E_1 \; (\; E_2, E_3 \;) \quad \{ \\
& \quad E.rval = newtemp(); \\
& \quad E.code = E_1.code \; || \\
& \qquad E_2.code \; || \\
& \qquad E_3.code \; || \\
& \qquad gen(\texttt{push } E_2.rval) \\
& \qquad gen(\texttt{push } E_3.rval) \\
& \qquad gen(\texttt{call } E_1.rval) \\
& \qquad gen(\texttt{pop } E.rval) \\
\}
\end{aligned}
$$

Function Calls (Call-by-Reference)

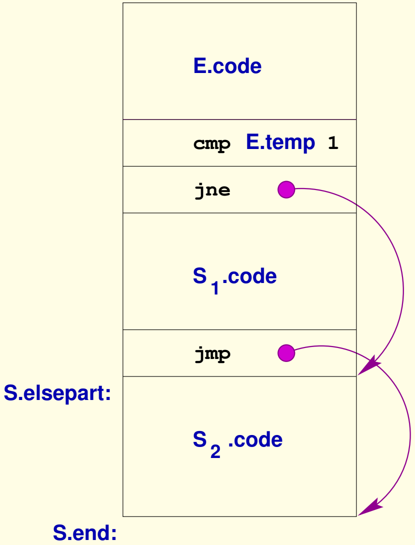
$$
\begin{aligned}
E \quad &\longrightarrow E_1\ (\ E_2,\ E_3\ )\ \{ \\
&\qquad E.rval = newtemp(); \\
&\qquad E.code = E_1.code\ || \\
&\qquad\qquad E_2.lcode\ || \\
&\qquad\qquad E_3.lcode\ || \\
&\qquad\qquad gen(\texttt{push}\ E_2.lval) \\
&\qquad\qquad gen(\texttt{push}\ E_3.lval) \\
&\qquad\qquad gen(\texttt{call}\ E_1.rval) \\
&\qquad\qquad gen(\texttt{pop}\ E.rval) \\
\}
\end{aligned}
$$

Code Generation for Statements

$$
\begin{aligned}
S \quad &\longrightarrow \quad S_1 \; ; \; S_2 \quad \{ \\
&\qquad\qquad S.code = S_1.code \; || \\
&\qquad\qquad\qquad\qquad S_2.code; \\
&\qquad\qquad \} \\
S \quad &\longrightarrow \quad E \qquad\quad \{ \; S.code = E.code; \; \}
\end{aligned}
$$

Conditional Statements

$$S \longrightarrow \text{if } E, S_1, S_2$$

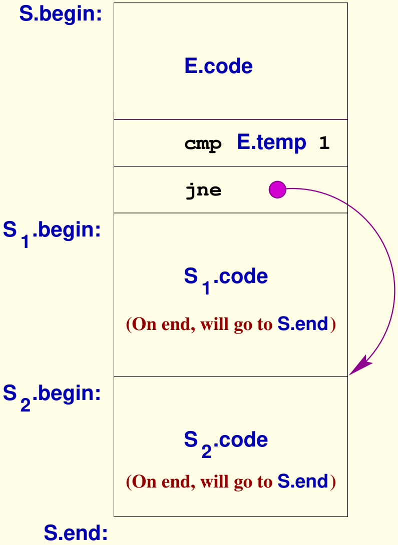


Conditional Statements

$$
\begin{aligned}
S \quad \longrightarrow \quad \text{if } E, S_1, S_2 \quad \{ & \\
& elselabel = newlabel(); \\
& endlabel = newlabel(); \\
& S.code = \quad E.code \;||\; \\
& \qquad gen(\text{if } E.temp \text{ `}\neq\text{' `1' } elselabel) \;||\; \\
& \qquad S_1.code \;||\; \\
& \qquad gen(\text{jmp } endlabel) \;||\; \\
& \quad gen(elselabel\text{:}) \;||\; \\
& \qquad S_2.code \;||\; \\
& \quad gen(endlabel\text{:}) \\
\} &
\end{aligned}
$$

If Statements: An Alternative

$$S \longrightarrow \text{if } E,\ S_1,\ S_2$$



Continuations

An attribute of a statement that specifies where control will flow to ***after*** the statement is executed.

- Analogous to the *follow* sets of grammar symbols.
- In deterministic languages, there is only one continuation for each statement.
- Can be generalized to include local variables whose values are needed to execute the following statements:

  Uniformly captures call, return *and **exceptions**.*

Conditional Statements and Continuations

$$
\begin{aligned}
S \quad \longrightarrow \quad \textsf{if}\ E, S_1, S_2 \quad \{ \\
& S.begin = newlabel(); \\
& S.end = newlabel(); \\
& S_1.end = S_2.end = S.end; \\
& S.code = gen(S.begin:)\ || \\
& \qquad E.code\ || \\
& \qquad gen(\text{if } E.rval \text{ '==' '1' } S_2.begin)\ || \\
& \qquad S_1.code\ || \\
& \qquad S_2.code;|| \\
& \qquad gen(S.end:) \\
& \}
\end{aligned}
$$

Continuations

- Each boolean expression has two possible continuations:
  - $E.true$: where control will go when expression in E evaluates to *true*.
  - $E.false$: where control will go when expression in E evaluates to *false*.
- Every statement S has one continuation, $S.next$
- Every while loop statement has an additional continuation, $S.begin$

Shortcircuit Code for Boolean Expressions

$$
\begin{array}{ll}
E & \longrightarrow E_1 \;\&\&\; E_2 \; \{ \\
& E_1.true = newlabel(); \\
& E_1.false = E_2.false = E.false; \\
& E_2.true = E.true; \\
& E.code = E_1.code \,||\, gen(E_1.true`:\text{'}) \,||\, E_2.code \\
\}
\end{array}
$$

$$
\begin{array}{ll}
E & \longrightarrow E_1 \text{ or } E_2 \; \{ \\
& E_1.true = E_2.true = E.true; \\
& E_1.false = newlabel(); \\
& E_2.false = E.false; \\
& E.code = E_1.code \,||\, gen(E_1.false`:\text{'}) \,||\, E_2.code \\
\}
\end{array}
$$

$$
\begin{array}{ll}
E & \longrightarrow \,!\, E_1 \; \{ \\
& E_1.false = E.true; \; E_1.true = E.false; \\
\}
\end{array}
$$

$$
E \quad \longrightarrow \text{true} \; \{ \; E.code = gen(\text{jmp}, E.true) \; \}
$$

Short-circuit code for Conditional Statements

$$
\begin{aligned}
S \quad & \longrightarrow S_1 \; ; \; S_2 \; \{ \\
& S_1.next = newlabel(); \\
& S.code = S_1.code \,||\, gen(S_1.next \text{ `:'}) \,||\, S_2.code; \\
\} & \\
S \quad & \longrightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \; \{ \\
& E.true = newlabel(); \\
& E.false = newlabel(); \\
& S_1.next = S_2.next = S.next; \\
& S.code = E.code \,|| \\
& \qquad gen(E.true\text{`:'}) \,||\, S_1.code \,|| \\
& \qquad gen(\text{jmp } S.next) \,|| \\
& \qquad gen(E.false\text{`:'}) \,||\, S_2.code; \\
& \}
\end{aligned}
$$

Short-circuit code for While

$$
\begin{aligned}
S \quad \longrightarrow\ & \textbf{while } E \textbf{ do } S_1 \ \{ \\
& S.begin = newlabel(); \\
& E.true = newlabel(); \\
& E.false = S.next; \\
& S_1.next = S.begin; \\
& S.code = gen(S.begin\text{‘:’}) \,||\, E.code \,|| \\
& \qquad gen(E.true\text{‘:’}) \,||\, S_1.code \,|| \\
& \qquad gen(\text{jmp } S.begin); \\
& \}
\end{aligned}
$$

Continuations and Code Generation

- Continuation of a statement is an inherited attribute.
  - It is not an L-inherited attribute!
- Code of statement is a synthesized attribute, but is dependent on its continuation.
  - ***Backpatching:*** Make two passes to generate code.
    1. Generate code, leaving "holes" where continuation values are needed.
    2. Fill these holes on the next pass.

Machine Code Generation Issues

- Register assignment
- Instruction selection
- ...

How GCC Handles Machine Code Generation

- gcc uses machine descriptions to *automatically* generate code for target machine
- machine descriptions specify:
  - memory addressing (bit, byte, word, big-endian, ...)
  - registers (how many, whether general purpose or not, ...)
  - stack layout
  - parameter passing conventions
  - semantics of instructions
  - . . .

Specifying Instruction Semantics

- gcc uses intermediate code called RTL, which uses a LISP-like syntax
- after parsing, programs are translated into RTL
- semantics of each instruction is also specified using RTL:

  `movl (r3), @8(r4)` ≡

```
    (set (mem: SI (plus: SI (reg: SI 4) (const_int 8)))
         (mem: SI (reg: SI 3)))
```

- cost of machine instructions also specified
- gcc code generation = selecting a low-cost instruction sequence that has the same semantics as the intermediate code

Optimization Techniques

- The most complex component of modern compilers
- Must always be *sound,* i.e., semantics-preserving
  - Need to pay attention to exception cases as well
  - Use a conservative approach: risk missing out optimization rather than changing semantics
- Reduce runtime resource requirements (most of the time)
  - Usually, runtime, but there are memory optimizations as well
  - Runtime optimizations focus on frequently executed code
    - How to determine what parts are frequently executed?
      - Assume: loops are executed frequently
      - Alternative: profile-based optimizations
  - Some optimizations involve trade-offs, e.g., more memory for faster execution
- Cost-effective, i.e., benefits of optimization must be worth the effort of its implementation

Code Optimizations

- High-level optimizations
  - Operate at a level close to that of source-code
  - Often language-dependent
- Intermediate code optimizations
  - Most optimizations fall here
  - Typically, language-independent
- Low-level optimizations
  - Usually specific to each architecture

High-level optimizations

- **Inlining**
  - Replace function call with the function body
- **Partial evaluation**
  - Statically evaluate those components of a program that can be evaluated
- Tail recursion elimination
- Loop reordering
- Array alignment, padding, layout

Intermediate code optimizations

- Common subexpression elimination
- Constant propagation
- Jump-threading
- Loop-invariant code motion
- Dead-code elimination
- Strength reduction

Constant Propagation

- Identify expressions that can be evaluated at compile time, and replace them with their values.
- x = 5; => x = 5; => x = 5;

```
x = 5;             =>   x =   5;          =>   x =   5;
y = 2;                  y = 2;                 y = 2;
v = u + y;              v = u + y;             v = u + 2;
z = x * y;              z = x * y;             z = 10;
w = v + z + 2;          w = v + z + 2;         w = v + 12;
...                     ...                    ...
```

Strength Reduction

- Replace expensive operations with equivalent cheaper (more efficient) ones.

```
y = 2;      =>    y = 2;
z = x^y;          z = x*x;
 ...               ...
```

- The underlying architecture may determine which operations are cheaper and which ones are more expensive.

Loop-Invariant Code Motion

- Move code whose effect is independent of the loop's iteration outside the loop.

```
for (i=0; i<N; i++) {      =>      for (i=0; i<N; i++) {
  for (j=0; j<N; i++) {              base = a + (i * dim1);
     ... a[i][j] ...                 for (j=0; j<N; i++) {
                                        ... (base + j) ...
```

Low-level Optimizations

- Register allocation
- Instruction Scheduling for pipelined machines.
- loop unrolling
- instruction reordering
- delay slot filling
- Utilizing features of specialized components, e.g., floating-point units.
- Branch Prediction

Peephole Optimization

- Optimizations that examine small code sections at a time, and transform them
- Peephole: a small, moving window in the target program
- Much simpler to implement than global optimizations
- Typically applied at machine code, and some times at intermediate code level as well
- Any optimization can be a peephole optimization, provided it operates on the code within the peephole.
- redundant instruction elimination
- flow-of control optimizations
- algebraic simplifications
- ...

Profile-based Optimization

- A compiler has difficulty in predicting:
  - likely outcome of branches
  - functions and/or loops that are most frequently executed
  - sizes of arrays
  - or more generally, any thing that depends on dynamic rogram behavior.
- Runtime profiles can provide this missing information, making it easier for compilers to decide when certain

Example Program: *Quicksort*

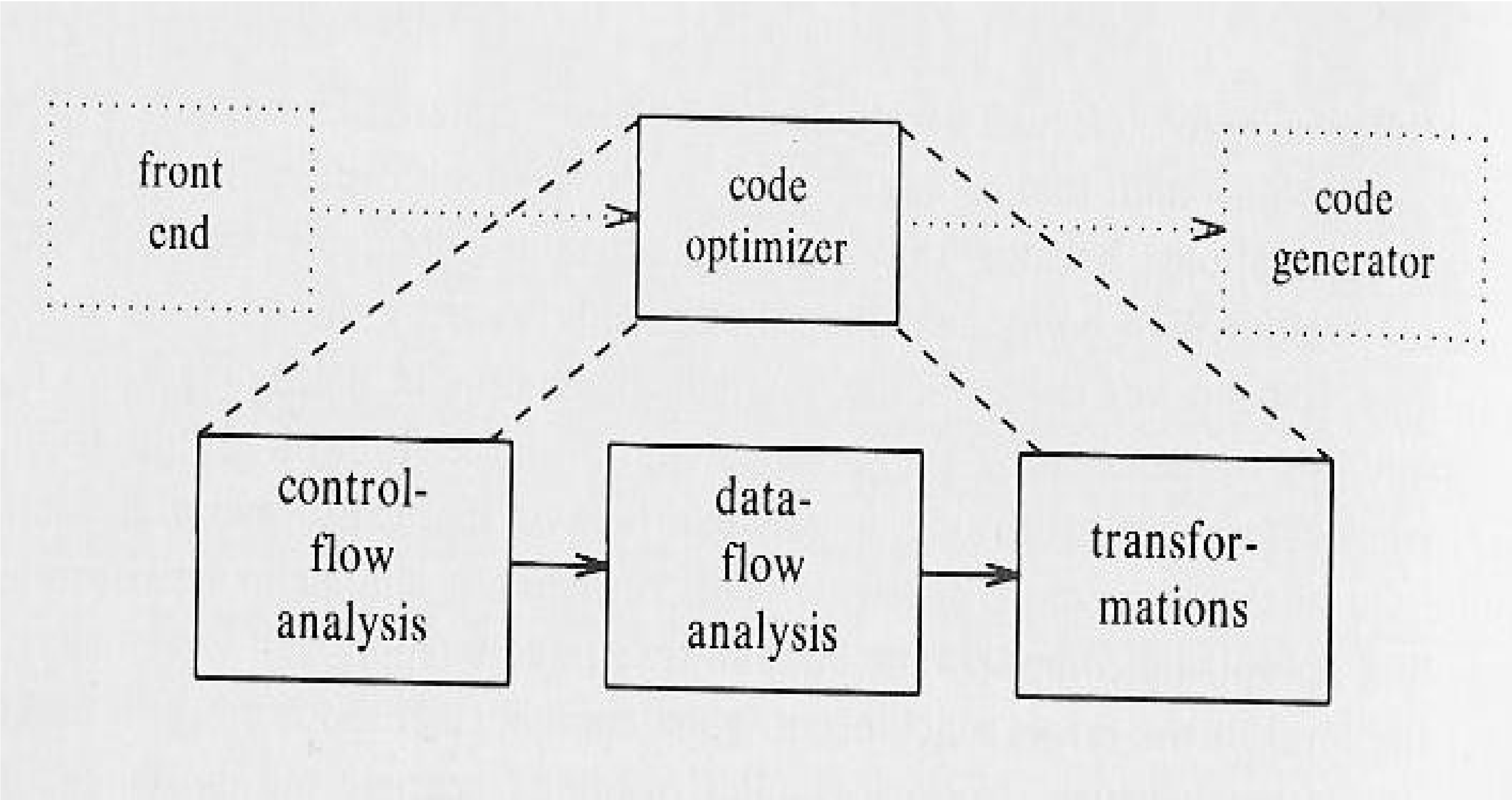
```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while(1) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if (i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

- Most optimizations opportunities arise in intermediate code
  - Several aspects of execution (e.g., address calculation for array access) aren't exposed in source code
- Explicit representations provide most opportunities for optimization
- It is best for programmers to focus on writing readable code, leaving simple optimizations to a compiler

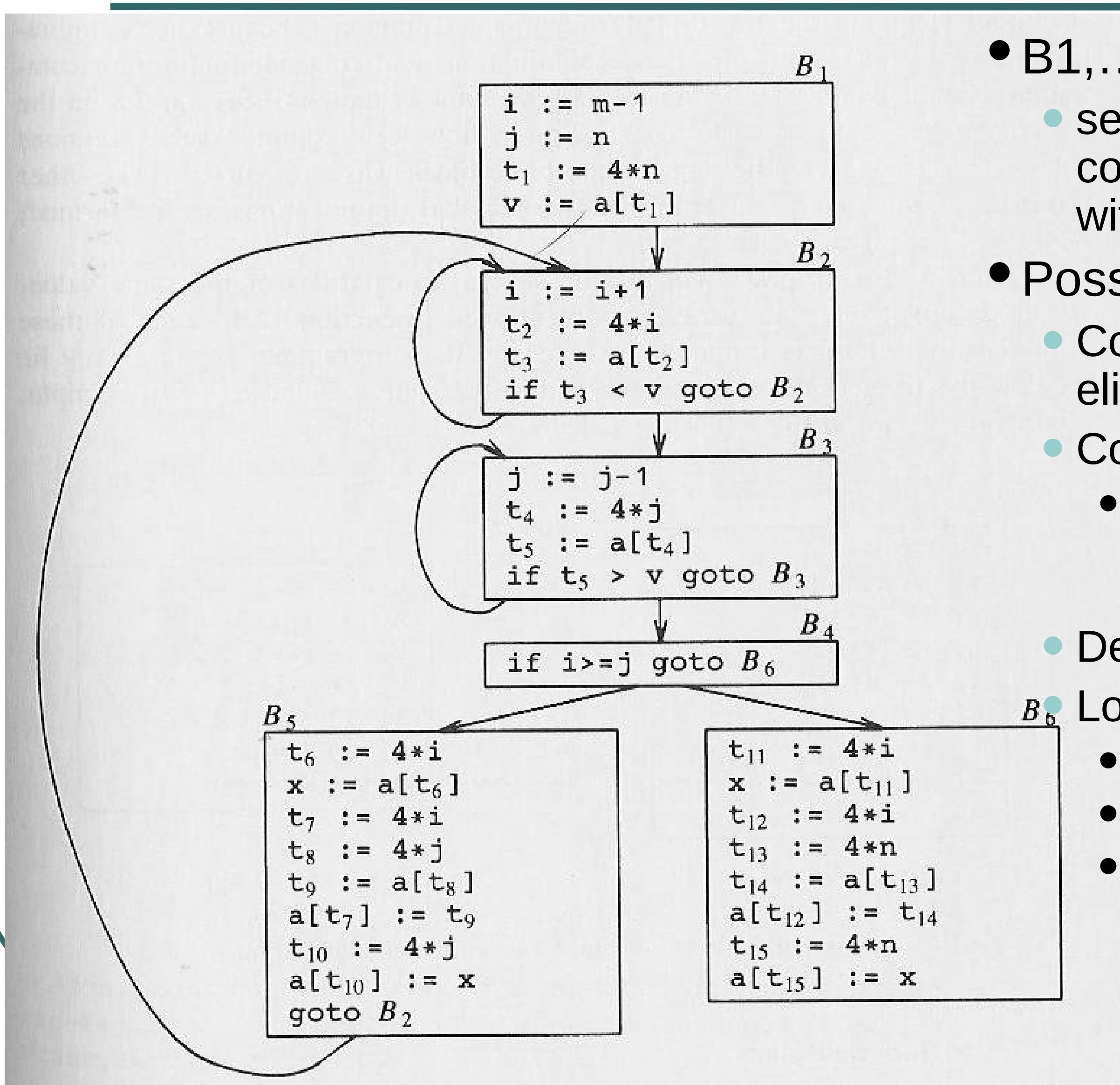
3-address code for *Quicksort*

```
(1)   i := m-1
(2)   j := n
(3)   t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4*i
(17)  t8 := 4*j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4*j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4*i
(24)  x := a[t11]
(25)  t12 := 4*i
(26)  t13 := 4*n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4*n
(30)  a[t15] := x
```

Organization of Optimizer

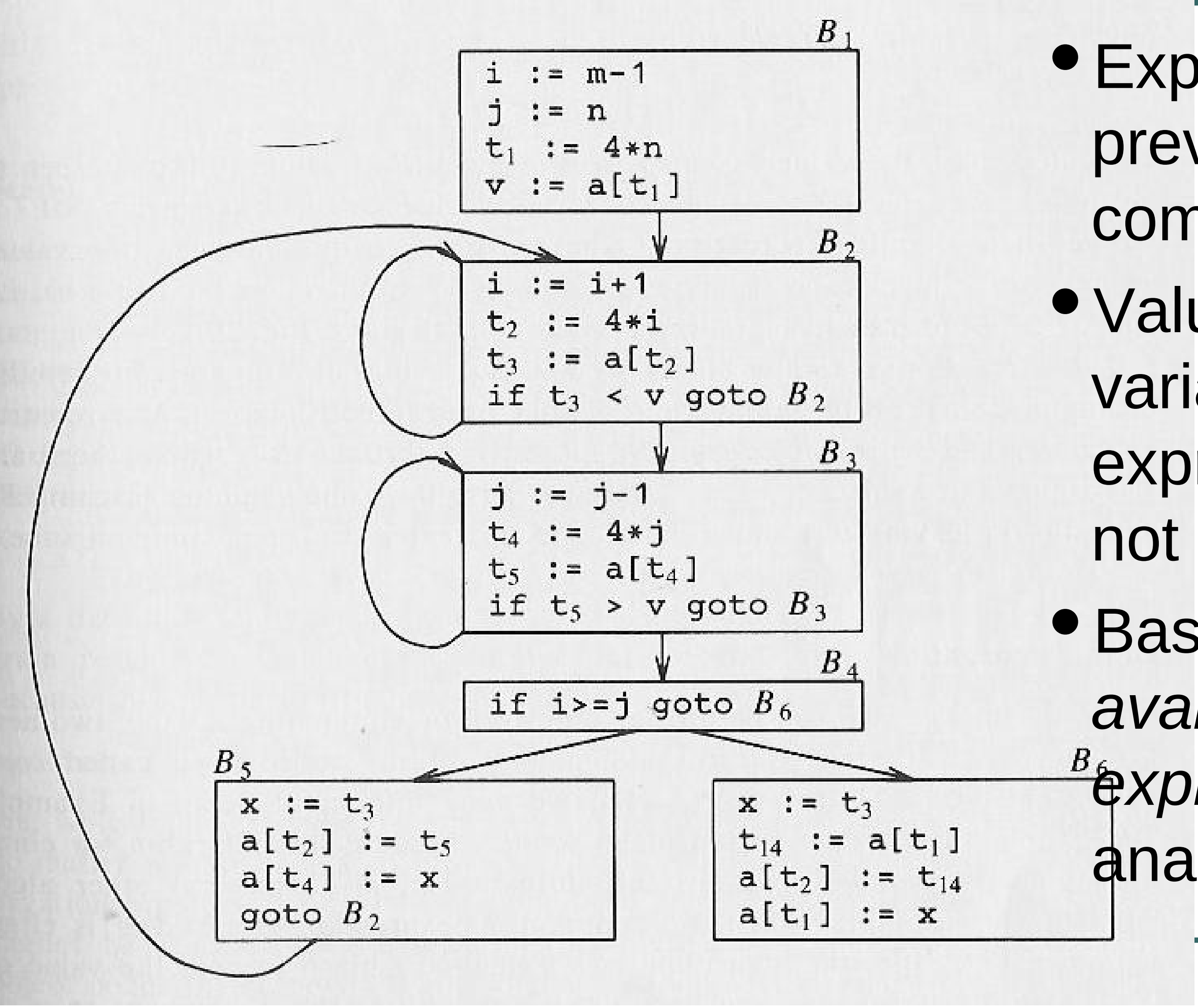


Flow Graph for *Quicksort*



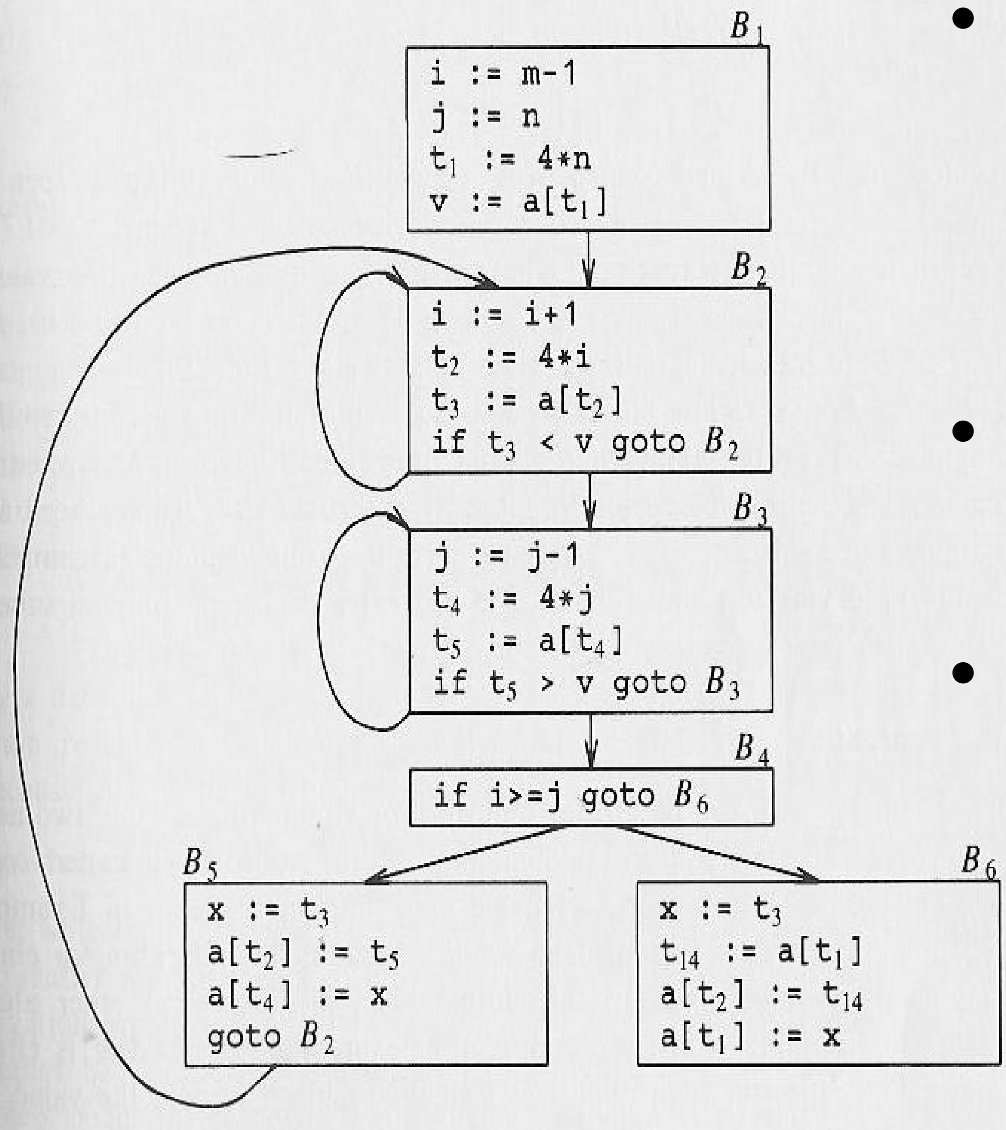
- B1,…,B6 are *basic blocks*
  - sequence of statements where control enters at beginning, with no branches in the middle
- Possible optimizations
  - Common subexpression elimination (CSE)
  - Copy propagation
    - Generalization of constant folding to handle assignments of the form x = y
  - Dead code elimination
  - Loop optimizations
    - Code motion
    - Strength reduction
    - Induction variable elimination

Common Subexpression Elimination



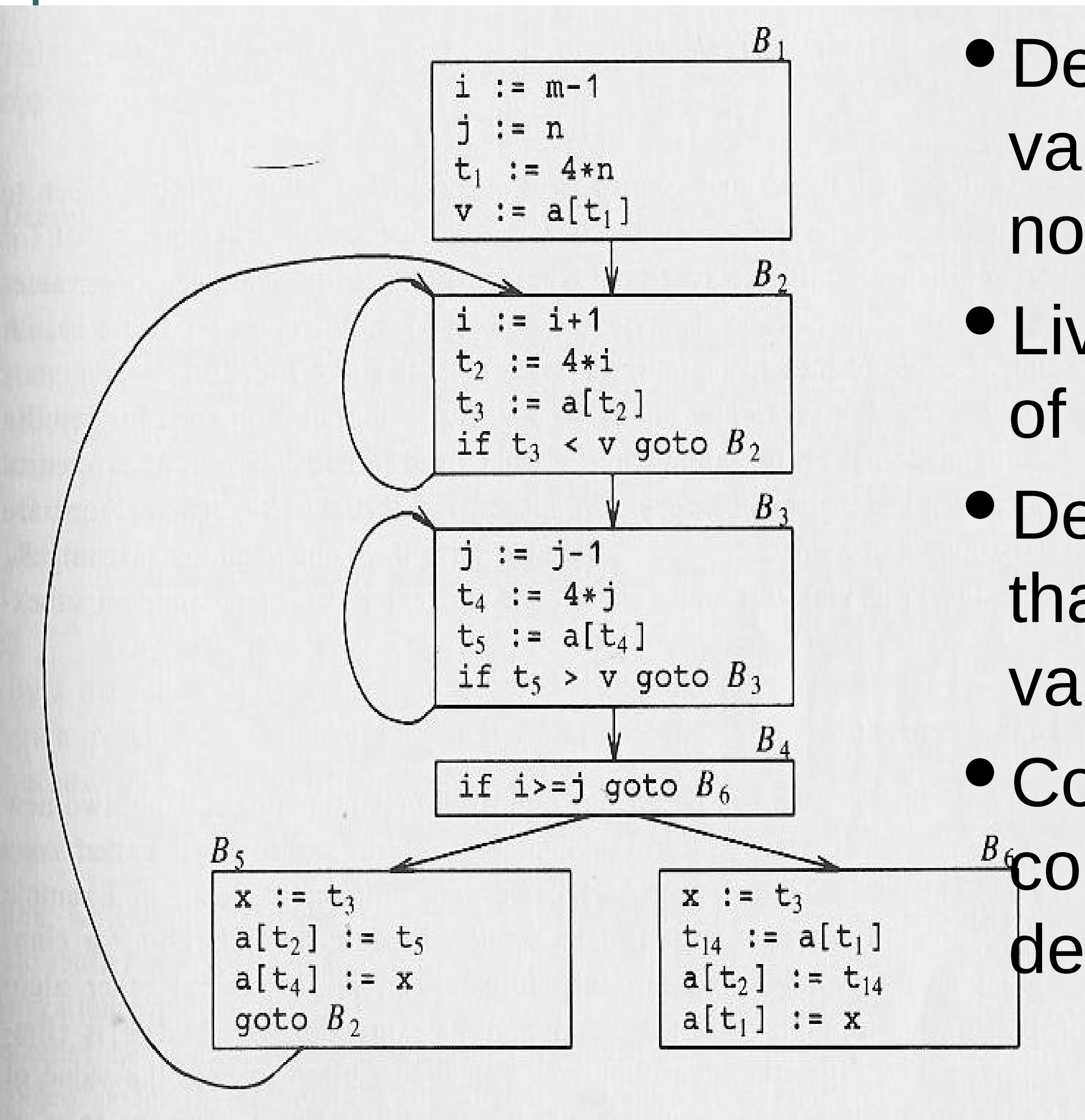
- Expression previously computed
- Values of all variables in expression have not changed.
- Based on *available expressions* analysis

Copy Propagation



- Consider
  x = y;
  z = x*u;
  w = y*u;
  Clearly, we can replace assignment on w by
  w = z
- This requires recognition of cases where multiple variables have same value (i.e., they are copies of each other)
- One optimization may expose opportunities for another
  - Even the simplest optimizations can pay off
  - Need to iterate optimizations a few times

Dead Code Elimination

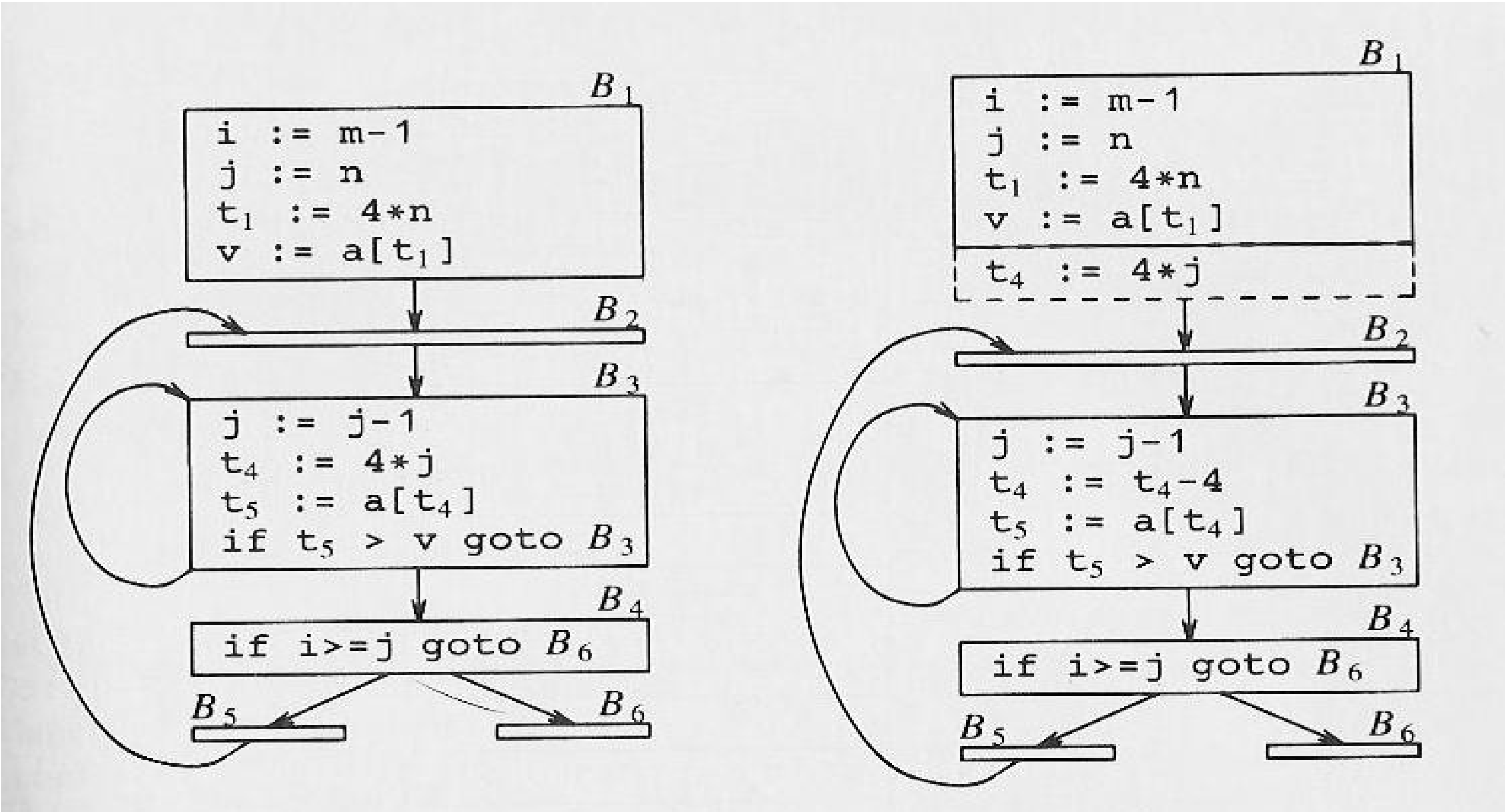


- Dead variable: a variable whose value is no longer used
- Live variable: opposite of dead variable
- Dead code: a statement that assigns to a dead variable
- Copy propagation turns copy statement into dead code.

Induction Vars, Strength Reduction and IV Elimination

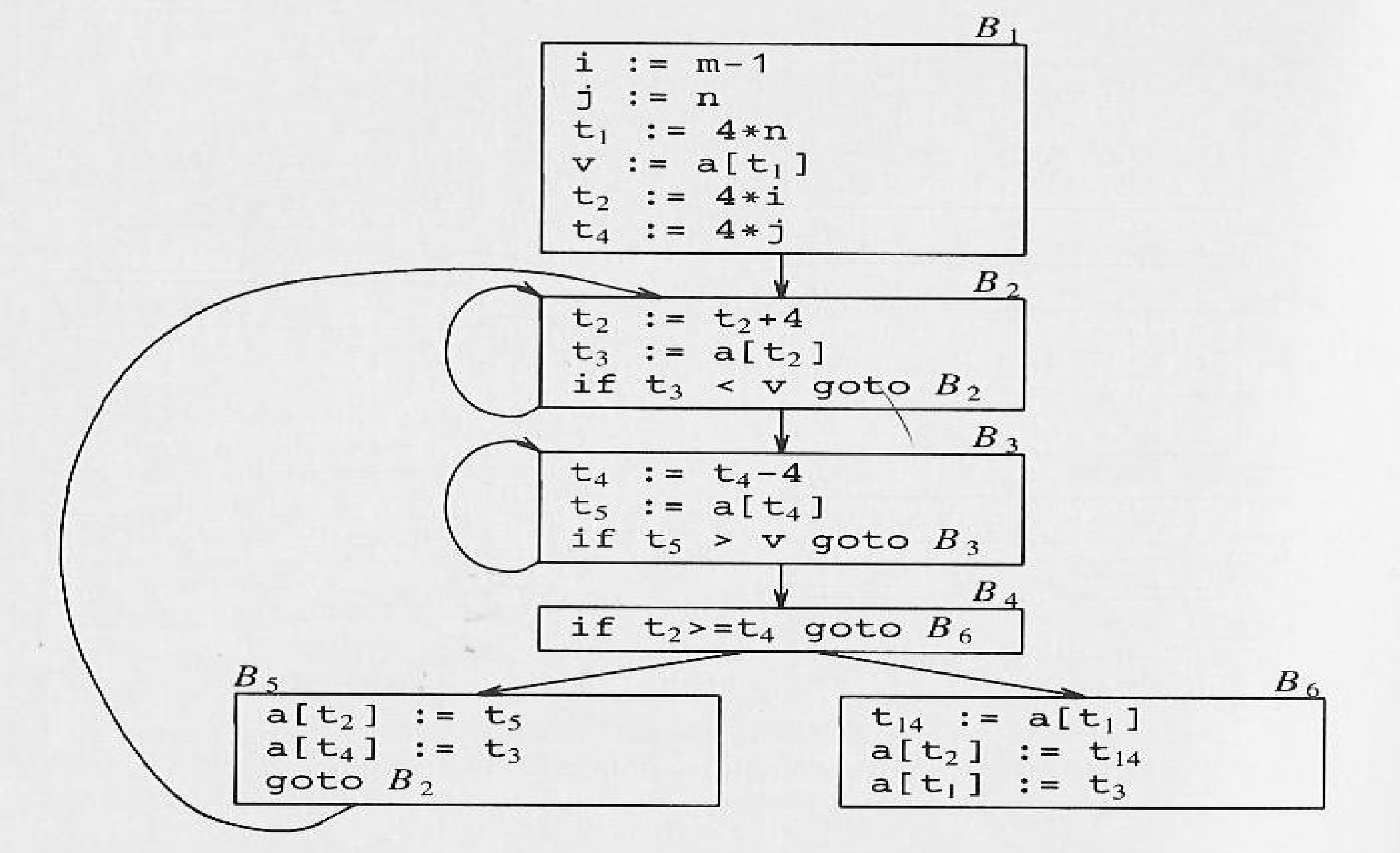
- Induction Var: a variable whose value changes in lock-step with a loop index
- If expensive operations are used for computing IV values, they can be replaced by less expensive operations
- When there are multiple IVs, some can be eliminated

Strength Reduction on IVs



(a) Before (b) After

After IV Elimination …



Program Analysis

- Optimization is usually expressed as a program transformation

  $C_1 \Leftrightarrow C_2$ when property *P* holds
- Whether property *P* holds is determined by a *program analysis*
- Most program properties are undecidable in general
  - Solution: Relax the problem so that the answer is an “yes” or “don’t know”

Applications of Program Analysis

- Compiler optimization
- Debugging/Bug-finding
  - "Enhanced" type checking
    - Use before assign
    - Null pointer dereference
    - Returning pointer to stack-allocated data
- Vulnerability analysis/mitigation
  - Information flow analysis
    - Detect propagation of sensitive data, e.g., passwords
    - Detect use of untrustworthy data in security-critical context
  - Find potential buffer overflows
- Testing – automatic generation of test cases
- Verification: Show that program satisfies a specified property, e.g., no deadlocks
  - model-checking

Dataflow Analysis

- Answers questions relating to how data flows through a program
  - What can be asserted about the value of a variable (or more generally, an expression) at a program point
- Examples
  - Reaching definitions: which assignments reach a program statement
  - Available expressions
  - Live variables
  - Dead code
  - …

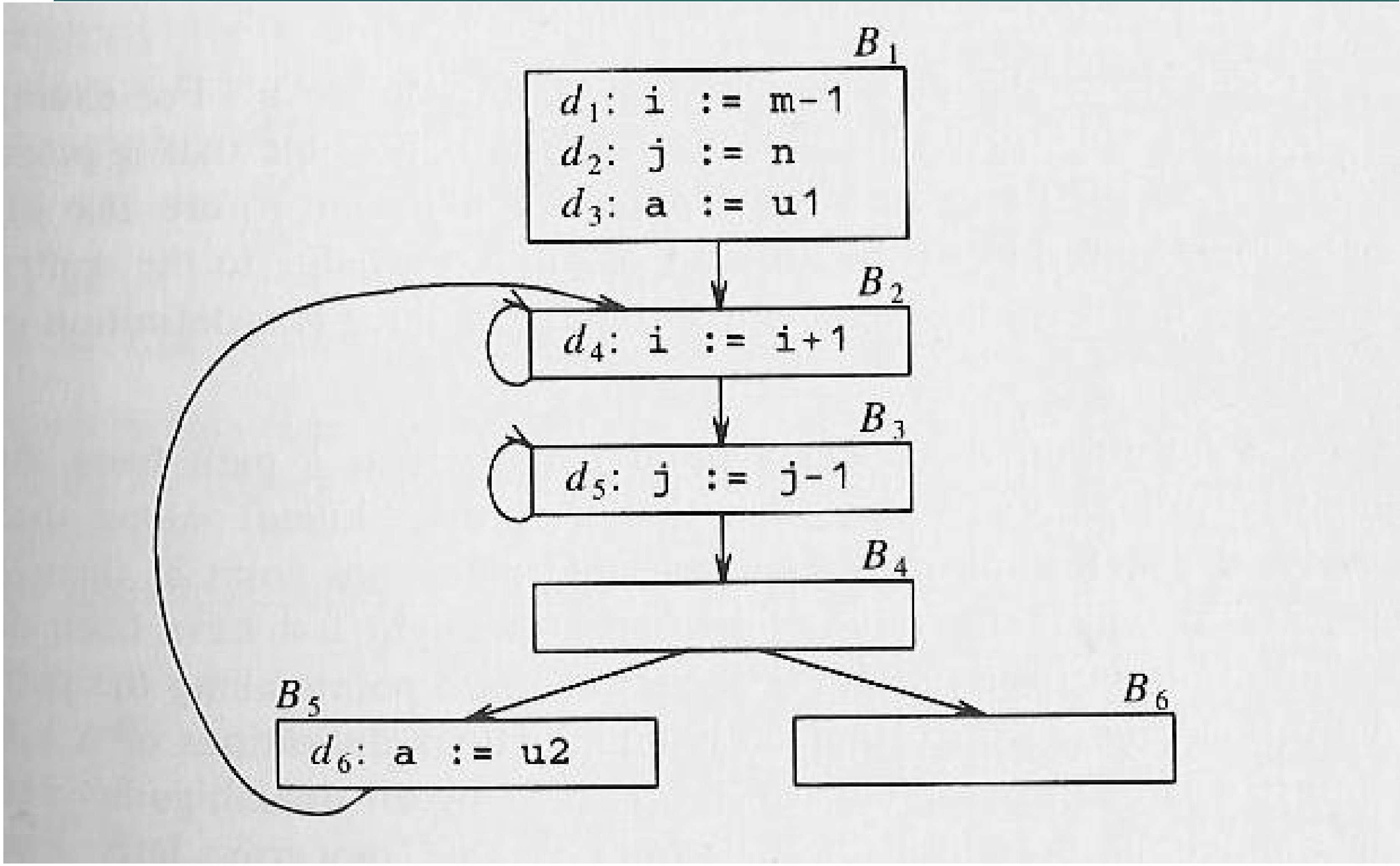
Dataflow Analysis

- Equations typically of the form

  out[S] = *gen*[S] ∪ (*in*[S] – *kill*[S])

  where the definitions of *out, gen, in* and *kill* differ for different analysis
- When statements have multiple predecessors, the equations have to be modified accordingly
- Procedure calls, pointers and arrays require careful treatment

Points and Paths



Reaching Definitions

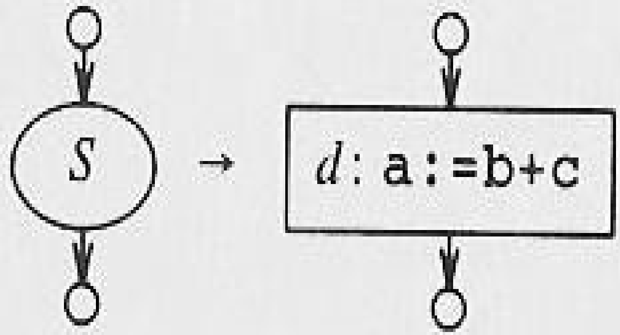
- A *definition* of a variable *x* is a statement that assigns to *x*
  - *Ambiguous definition:* In the presence of aliasing, a statement may define a variable, but it may be impossible to determine this for sure.
- A definition *d* reaches a point *p* provided:
  - There is a path from *d* to *p,* and this definition is not “killed” along *p*
    - “Kill” means an unambiguous redefinition
- Ambiguity ➔ approximation
  - Need to ensure that approximation is in the right direction, so that the analysis will be *sound*

DFA of Structured Programs

- *S* → **id** := *E*
  | *S*;*S*
  | **if** *E* **then** *S* **else** *S*
  | **do** *S* **while** *E*
- *E* → ***E* + *E***
  | **id**

DF Equations for Reaching Defns

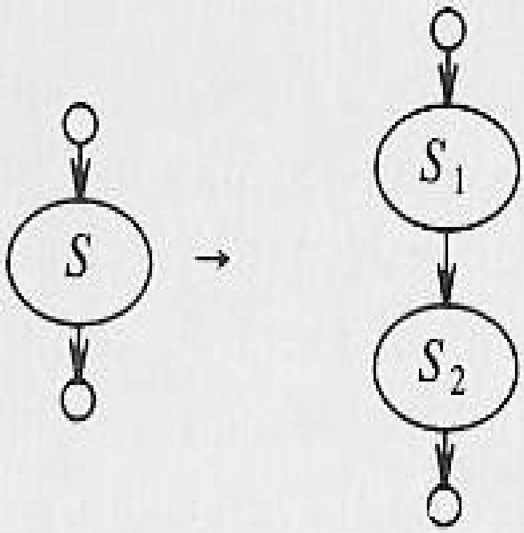
(a)



$$gen[S] = \{d\}$$
$$kill[S] = D_a - \{d\}$$

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

(b)

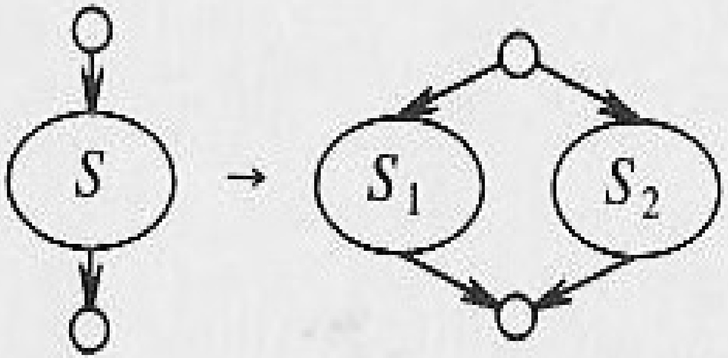


$$gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2])$$
$$kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$$

$$in[S_1] = in[S]$$
$$in[S_2] = out[S_1]$$
$$out[S] = out[S_2]$$

DF Equations for Reaching Defns

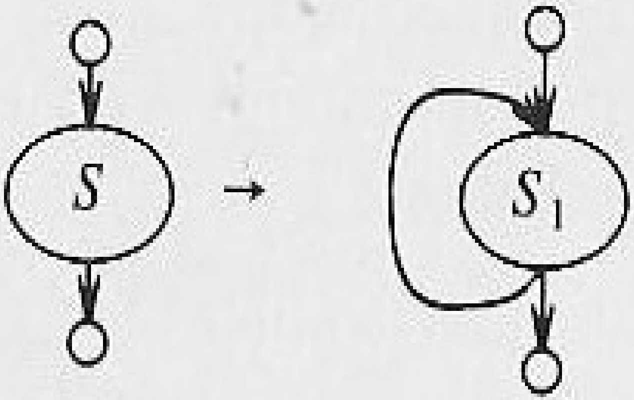
(c)



$$gen[S] = gen[S_1] \cup gen[S_2]$$
$$kill[S] = kill[S_1] \cap kill[S_2]$$

$$in[S_1] = in[S]$$
$$in[S_2] = in[S]$$
$$out[S] = out[S_1] \cup out[S_2]$$

(d)



$$gen[S] = gen[S_1]$$
$$kill[S] = kill[S_1]$$

$$in[S_1] = in[S] \cup gen[S_1]$$
$$out[S] = out[S_1]$$

Direction of Approximation

- Actual *kill* is a superset of the set computed by the dataflow equations
- Actual *gen* is a subset of the set computed by these equations
- Are other choices possible?
  - Subset approximation of kill, superset approximation of gen
  - Subset approximation of both
  - Superset approximation of both
- Which approximation is suitable depends on the intended use of analysis results

Solving Dataflow Equations

- Dataflow equations are recursive
- Need to compute so-called *fixpoints,* to solve these equations
- Fixpoint computations uses an interative procedure
  - $out^0 = \phi$
  - out^i is computed using the equations by substituting out^{i-1} for occurrences of *out* on the rhs
  - Fixpoint is a solution, i.e., $out^i = out^{i-1}$

Computing Fixpoints: Equation for Loop

- Rewrite equations using more compact notation, with:

  J standing for in[S] and

  I, G, K, and O for in[S1], gen[S1], kill[S1] and out[S1]:

  $$I = J \cup O,$$
  $$O = G \cup (I - K)$$

- Letting $I^0 = O^0 = \phi$, we have:

$I^1 = \mathbf{J}$	$O^1 = G \cup (I^0 - K) = \mathbf{G}$
$I^2 = J \cup O^1 = \mathbf{J \cup G}$	$O^2 = G \cup (I^1 - K) = \mathbf{G \cup (J - K)}$
$I^3 = J \cup O^2$	$O^3 = G \cup (I^2 - K)$
$= J \cup G \cup (J - K)$	$= G \cup (J \cup G - K)$
$= J \cup G = I^2$	$= G \cup (J - K) = O^2$

(Note that for all sets A and B, A U (A-B) = A, and

for all sets A, B and C, A U (A U C –B) = A U (C-B).)

Thus, we have a fixpoint.

Use-Definition Chains

- Convenient way to represent reaching definition information
- ud-chain for a variable links each use of the variable to its reaching definitions
  - One list for each use of a variable

Available Expressions

- An expression ***e*** is available at point ***p*** if
  - every path to ***p*** evaluates ***e***
  - none of the variables in ***e*** are assigned after last computation of ***e***
- A block *kills* ***e*** if it assigns to some variable in ***e*** and does not recompute ***e.***
- A block *generates* ***e*** if it computes ***e*** and doesn't subsequently assign to variables in ***e***
- ***Exercise:*** Set up data-flow equations for available expressions. Give an example use for which your equations are sound, and another example for which they aren't

Available expressions -- Example

a := b+c

b := a-d

c := b+c

d := a-d

Live Variable Analysis

- A variable ***x*** is *live* at a program point ***p*** if the value of ***x*** is used in some path from ***p***
- Otherwise, ***x*** is *dead.*
- Storage allocated for dead variables can be freed or reused for other purposes.
- in[B] = use[B] ∪ (out[B] – def[B])
- out[B] = ⋃ in[S], for S a successor of B
- Equation similar to reaching definitions, but the role of in and out are interchanged

Def-Use Chains

- du-chain links the definition of a variable with all its uses
  - Use of a definition of a variable ***x*** at a point ***p*** implies that there is a path from this definition to ***p*** in which there are no assignments to ***x***
- du-chains can be computed using a dataflow analysis similar to that for live variables

Optimizations and Related Analyses

- Common subexpression elimination
  - Available expressions
- Copy propagation
  - In every path that reaches a program point ***p***, the variables ***x*** and ***y*** have identical values
- Detection of loop-invariant computation
  - Any assignment ***x := e*** where the definition of every variable in ***e*** occurs outside the loop.
- Code reordering: A statement ***x := e*** can be moved
  - earlier before statements that (a) do not use ***x***, (b) do not assign to variables in ***e***
  - later after statements that (a) do not use ***x***, (b) do not assign to variables in ***e***

Difficulties in Analysis

- Procedure calls
- Aliasing

Difficulties in Analysis

- Procedure calls
  - may modify global variables
    - potentially kill all available expressions involving global variables
    - modify reaching definitions on global variables
- Aliasing
  - Create ambiguous definitions
  - a[i] = a[j] --- here, i and j may have same value, so assignment to a[i] can potentially kill a[j]
  - *p = q + r --- here, p could potentially point to q, r or any other variable
    - creates ambiguous redefinition for all variables in the program!

Low-level Code Generation

- Assembly code generation
  - Register allocation
  - Instruction selection
- Machine code generation
  - Instruction encoding
  - Linker and loader
  - Relocatable code
    - Defer assignment of locations for static objects (code, variables) to linking phase
      - Static linking
      - Dynamic linking

Machine code generation (contd.)

- Position-independent code (PIC)
  - Can be shared by different processes that map a library to different locations
  - Code does not assume knowledge of memory location of its code or variables
- Symbol tables
  - Often, code that is shipped has all symbols “stripped off”
  - For libraries, need to maintain a minimal amount of symbol info

Register Allocation: Factors

- Special-purpose registers
  - Stack pointer, Base pointer, Instruction pointer, ...
  - Reserved for specific uses across most code
    - Register allocation deals with general-purpose registers
- Application/binary interface requirements
  - Caller- Vs Callee-save registers
    - Caller-save registers need to be explicitly saved by the caller before every procedure call, and restored after
    - Callee-save registers have to be saved before use by every function, and restored if used.
- Some (most) instructions may operate only on register operands

Register Allocation: Simple Strategies

1. Load a register from memory before each operation, store immediately afterwards
   - Too inefficient
2. Avoid load/store's within a basic block
   - Load registers at entry of a BB, and store at its end.
   - Fails to discriminate between loops and other Bbs
   - May require too many registers

- "Global" register allocation
  - Consider uses across Bbs
  - Even more "pressure" on registers ...

Global Register Allocation

- Model cost of instructions
  - Cost of fetching
    - On modern processors, fetching costs can be ignored to a certain extent due to the use of dedicated pipelines for instruction fetching/decoding, plus branch prediction etc.
  - Cost of memory access
    - For loading registers
    - For saving registers
    - For accessing memory (in case of instructions that accept memory operands)
  - Take into account loops
    - e.g., treat the cost of non-loop operations to be zero

Register usage counts

- Use(x) = number of uses of variable x (before reassignment) within a block, plus 2 if x is live at the end of the loop
  - Use registers to hold variables with highest use count
- If there are nested loops, allocate registers for innermost loop, and then allocate remaining registers to outer loops
  - Alternatively, reuse registers used in inner loops in outer loops by saving/restoring registers
  - Avoid unnecessary save/restores by analyzing across BBs to find variables used in inner as well as outer loops.

Working with fixed number of Registers

- Can be modeled as a graph-coloring problem
  - Allocate a symbolic register for each variable
  - Construct a register-interference graph (RIG)
    - Edge between two symbolic registers if one is live at the point where the other is assigned
  - You can use N registers if RIG is N-colorable
    - i.e., there is a way to assign N colors to graph nodes such that neighboring nodes have different colors

Graph-coloring (contd.)

- Graph-coloring problem is NP-complete
  - But good heuristics exist:
    - Eliminate all nodes that have less than degree N
      - Eliminating one node will reduce the degree of nodes connected to it
      - Color for the eliminated node can be chosen to be one of those that is not assigned to any of its neighbors
    - If all nodes have degree >= N, pick one to “spill,” i.e., save to memory and restore later
      - Pick registers that have least cost savings
      - Avoid spills in inner loops

Instruction Selection

- Instruction selection is a complex task, especially when considering modern processors with a large number of instructions and addressing modes
- Many semantically equivalent instructions sequences may perform the same desired task
  - How to select the "minimal cost" sequence?
- Ideally, one does not have to hand-code a code generator, but have it be generated from specifications!
  - Instruction selection by tree-rewriting
  - Initially, the tree represents generated intermediate code

Target code generation in GCC

- gcc uses machine descriptions to automatically generate code for target machine
  - Enables gcc to support numerous target machines, with greatly reduced programmer effort
- machine descriptions specify:
  - memory addressing (bit, byte, word, big-endian, ...)
  - registers (how many, whether general purpose or not, ...)
  - stack layout
  - parameter passing conventions
  - semantics of instructions
  - …

Instruction Specification

- For each instruction in target language, specify:
  - Assembly representation of target machine instructions
    - Instruction parameters include registers and constants
  - Its semantics in the intermediate language
    - Parameterized in terms of registers and constants in the target instruction
    - Specify input operands as well as the location where the result is stored
  - Cost of executing the instruction
  - Additional constraints on applicability of instruction
    - e.g., a certain constant must be at most 8 bits

Code generation by rewriting

- Represent intermediate code generated by the compiler as a tree, and use rewriting using the rules in the instruction specification
- Trees can represent expressions as well as sequence of statements
  - Introduce a sequencing operation to represent sequencing
  - Don't force sequencing of unrelated statements, or else the code generator won't be able to choose evaluation orders that lead to more efficient code.
    - Example: a=b+5; c=d+5; e=a+b
    - More efficient if c=d+5 is moved later, as it would allow a and b to continue to be in registers while evaluating e=a+b

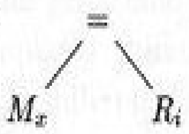
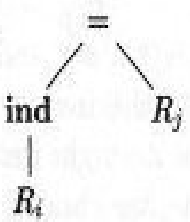
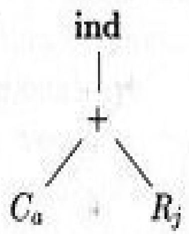
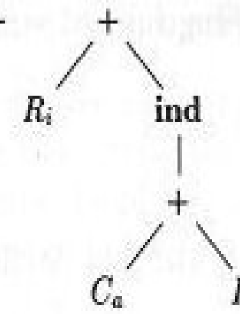
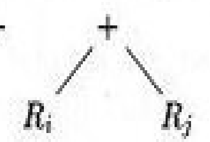
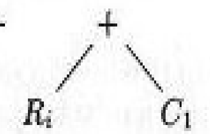
GCC target code generation

- gcc uses intermediate code called RTL, which uses a LISP-like syntax
  - Actually, gcc uses multiple intermediate languages, with RTL being the lowest level among them
- semantics of each instruction is also specified using RTL:
  - **movl (r3), @8(r4)**

```
(set (mem: SI (plus: SI (reg: SI 4) (const_int 8)))
     (mem: SI (reg: SI 3)))
```

- gcc code generation = selecting a low-cost instruction sequence that has the same semantics as the intermediate code

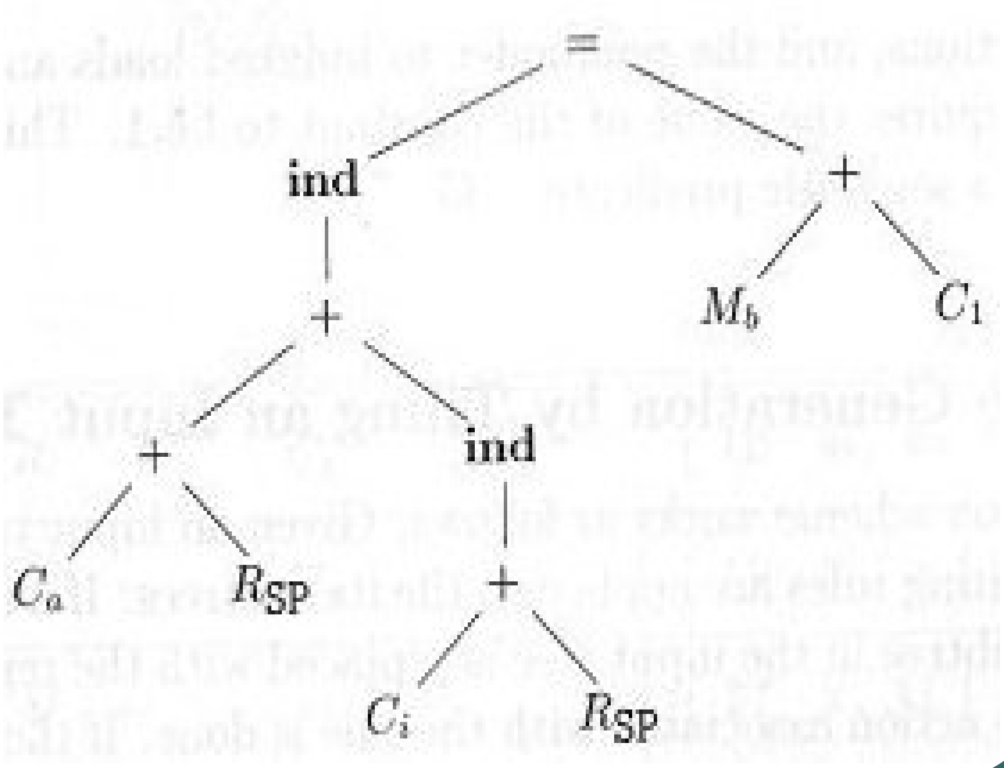
Instruction Specification

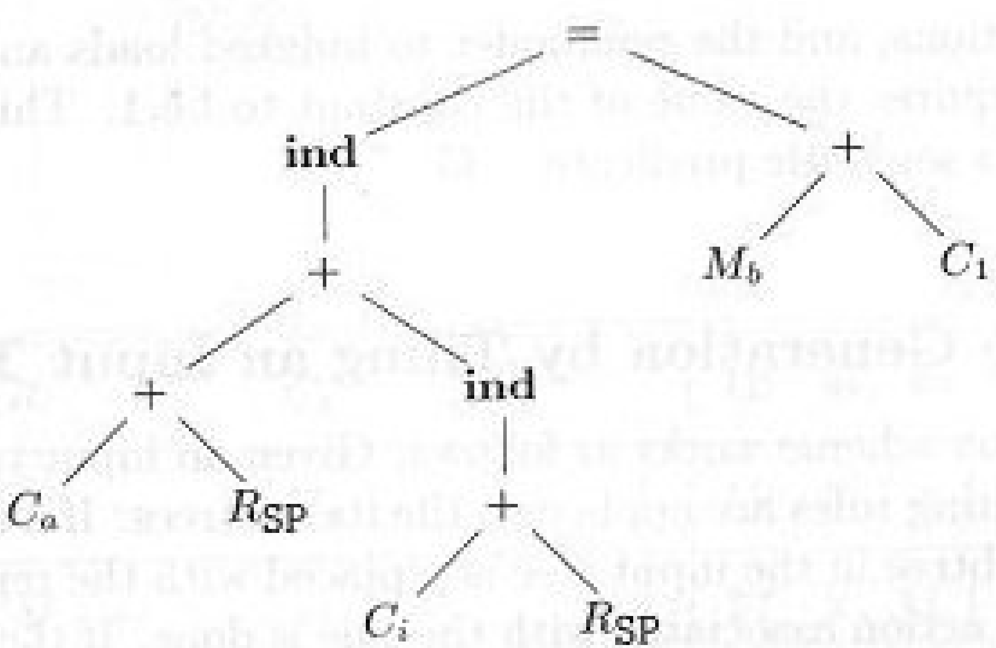
1)	$R_i \leftarrow C_a$	`{ LD Ri, #a }`
2)	$R_i \leftarrow M_x$	`{ LD Ri, x }`
3)	$M \leftarrow$ 	`{ ST x, Ri }`
4)	$M \leftarrow$ 	`{ ST *Ri, Rj }`
5)	$R_i \leftarrow$ 	`{ LD Ri, a(Rj) }`
6)	$R_i \leftarrow$ 	`{ ADD Ri, Ri, a(Rj) }`
7)	$R_i \leftarrow$ 	`{ ADD Ri, Ri, Rj }`
8)	$R_i \leftarrow$ 	`{ INC Ri }`

Instruction Selection Example

- Intermediate code for a[i] = b+1
- Rewrite tree repeatedly using rules corresponding to instruction specifications until you get to a single node tree.
- Result

```
LD   R0, #a
ADD  R0, R0, SP
ADD  R0, R0, i[SP]
LD   R1, b
INC  R1
ST   *R0, R1
```





1)	R_i	$\leftarrow C_a$	{ LD Ri, #a }
2)	R_i	$\leftarrow M_x$	{ LD Ri, x }
3)	M	$\leftarrow =(M_x, R_i)$	{ ST x, Ri }
4)	M	$\leftarrow =(\text{ind}(R_i), R_j)$	{ ST *Ri, Rj }
5)	R_i	$\leftarrow \text{ind}(+(C_a, R_j))$	{ LD Ri, a(Rj) }
6)	R_i	$\leftarrow +(R_i, \text{ind}(+(C_a, R_j)))$	{ ADD Ri, Ri, a(Rj) }
7)	R_i	$\leftarrow +(R_i, R_j)$	{ ADD Ri, Ri, Rj }
8)	R_i	$\leftarrow +(R_i, C_1)$	{ INC Ri }

Optimal Code Generation

- Some intermediate operations may not have equivalent instructions
  - e.g.,“add R0, R0, M” versus “ld R1, M; add R0, R0, R1”
- Multiple rules may match the same node
  - Cost of evaluation may hinge on which match is chosen
  - Example: “inc R0” versus “add R0, 1”
- The order of rewriting can change the cost
  - Mainly due to selection of registers, and based on which intermediate results remain in registers as opposed to being stored in memory.

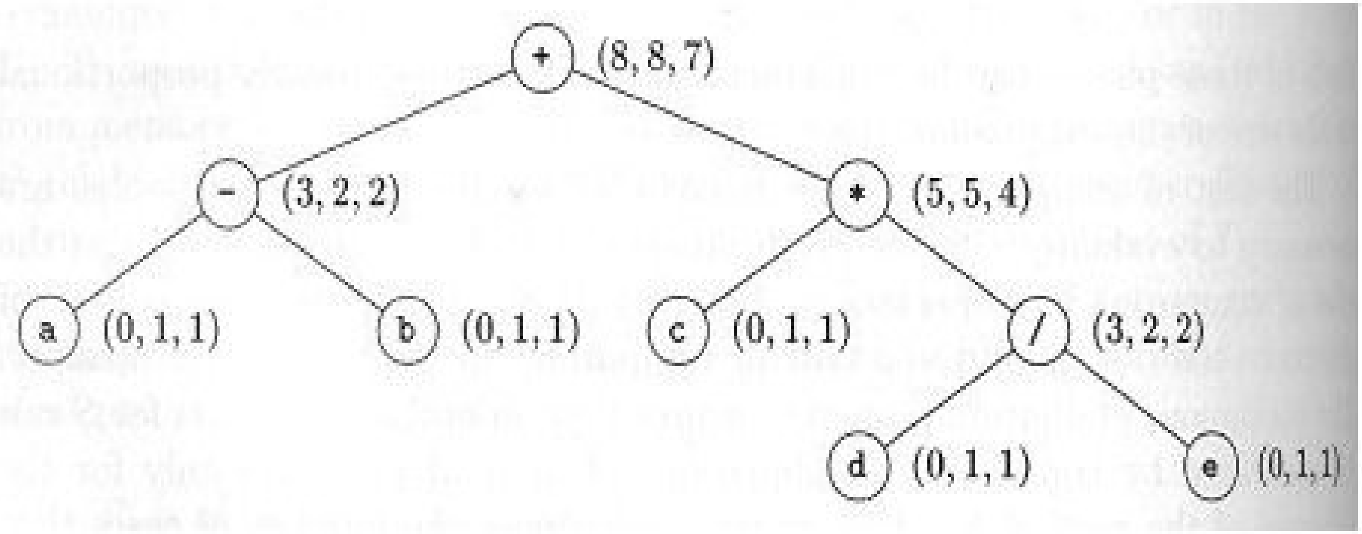
Optimal Code Generation

- But, dynamic programming algorithms for optimal code generation exist under reasonable assumptions
  - Optimal code for E1 op E2 will contain optimal code for evaluating E1 and optimal code for evaluating E2
  - Dynamic programming algorithm tries to construct the optimal code bottom-up: from E1 and E2's optimal codes, build optimal code for E1 op E2
  - Dynamic programming algorithm iterates over
    - number of registers used for operand evaluation
    - order of evaluation of operand (when permissible)

Dynamic Programming Algorithm

- For each node n in tree, compute $C[n][i]$ which represents the minimum cost for evaluating the subtree rooted at n using at most i registers, for 0 <= i <= k (# of registers in the target architecture)
- The operands for evaluating the operation at n may differ, depending on the matching instruction
- While evaluating operands of n, we may use:
  - All i registers for evaluating each operand, but this requires evaluation results to be stored in memory in order to free up registers for evaluating other operands
  - Use less than i registers so that operands can be retained in registers
  - We prefer an order of evaluation that minimizes the number of registers that need to be saved to memory
- For the root node r, pick how many registers to use (may be k)
- Generate instructions based on the choices at each node that result in the least cost for $C[r][k]$

Illustration of Dynamic Programming Algorithm




```
LD  Ri, Mj        // Ri = Mj
op  Ri, Ri, Rj    // Ri = Ri op Rj
op  Ri, Ri, Mj    // Ri = Ri op Mj
LD  Ri, Rj        // Ri = Rj
ST  Mi, Rj        // Mi = Rj
```

Target Instructions

```
LD  R0, c         // R0 = c
LD  R1, d         // R1 = d
DIV R1, R1, e     // R1 = R1 / e
MUL R0, R0, R1    // R0 = R0 * R1
LD  R1, a         // R1 = a
SUB R1, R1, b     // R1 = R1 - b
ADD R1, R1, R0    // R1 = R1 + R0
```

Optimal Code