Μεταγλωττιστές
(Εαρινό 2023)

Εισαγωγή στο Σχεδιασμό και στην Υλοποίηση των Γλωσσών Προγραμματισμού

How are Languages Implemented?

- Two major strategies:
  - Interpreters (older, less studied)
  - Compilers (newer, much more studied)

- Interpreters run programs "as is"
  - Little or no preprocessing

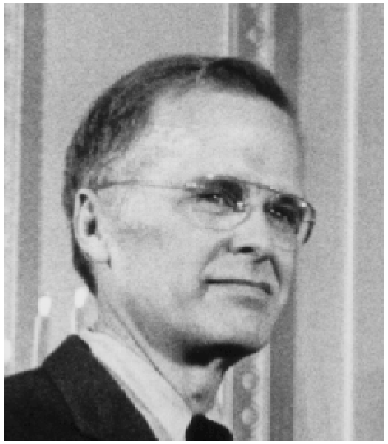
- Compilers do extensive preprocessing

Language Implementations

- Today, batch compilation systems dominate
  - gcc, clang, ...
- Some languages are primarily interpreted
  - Java bytecode compiler (javac)
  - Scripting languages (perl, python, javascript, ...)
- Some languages (e.g. Lisp) provide both
  - Interpreter for development
  - Compiler for production

(Short) History of High-Level Languages

- 1953 IBM develops the 701
- Till then, all programming is done in assembly
- Problem: Software costs exceeded hardware costs!
- John Backus: “Speedcoding”
  - An interpreter
  - Ran 10-20 times slower than hand-written assembly



FORTRAN I

- 1954 IBM develops the 704
- John Backus
  - Idea: translate high-level code to assembly
  - Many thought this impossible
    - Had already failed in other projects
- 1954-7 FORTRAN I project
- By 1958, >50% of all software is in FORTRAN
- Cut development time dramatically
  - (2 weeks → 2 hours)

FORTRAN I

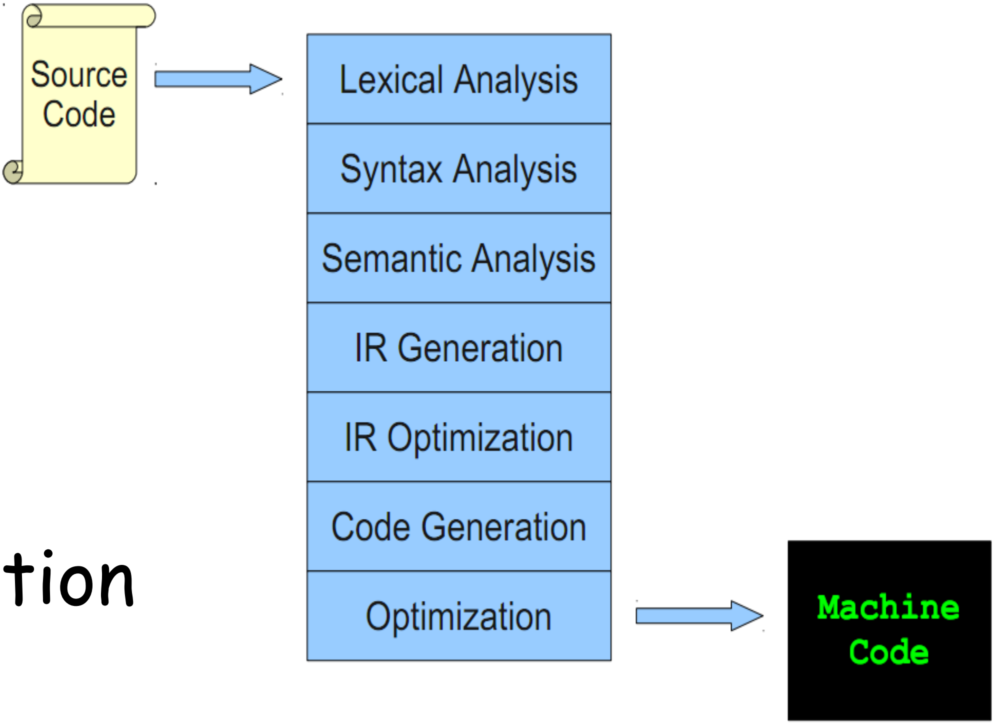
- The first compiler
  - Produced code almost as good as hand-written
  - Huge impact on computer science

- Led to an enormous body of theoretical work

- Modern compilers preserve the outlines of the FORTRAN I compiler

The Structure of a Compiler

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. IR Optimization
5. Code Generation
6. Low-level Optimization



The first 3 phases can be understood by analogy to how humans comprehend natural languages (e.g., English, Greek, etc.).

First Step: Lexical Analysis

- Recognize words
  - Smallest unit above letters

This is a sentence.

- Note the
  - Capital "T" (start of sentence symbol)
  - Blank " " (word separator)
  - Period "." (end of sentence symbol)

More Lexical Analysis

- Lexical analysis is not trivial. Consider:

  ist his ase nte nce

- Plus, programming languages are typically more cryptic than English:

```
*p->f ++ = -.12345e-5
```

And More Lexical Analysis

- Lexical analyzer divides program text into “words” or “tokens”

```
if (x == y) then z = 1; else z = 2;
```

- Units:

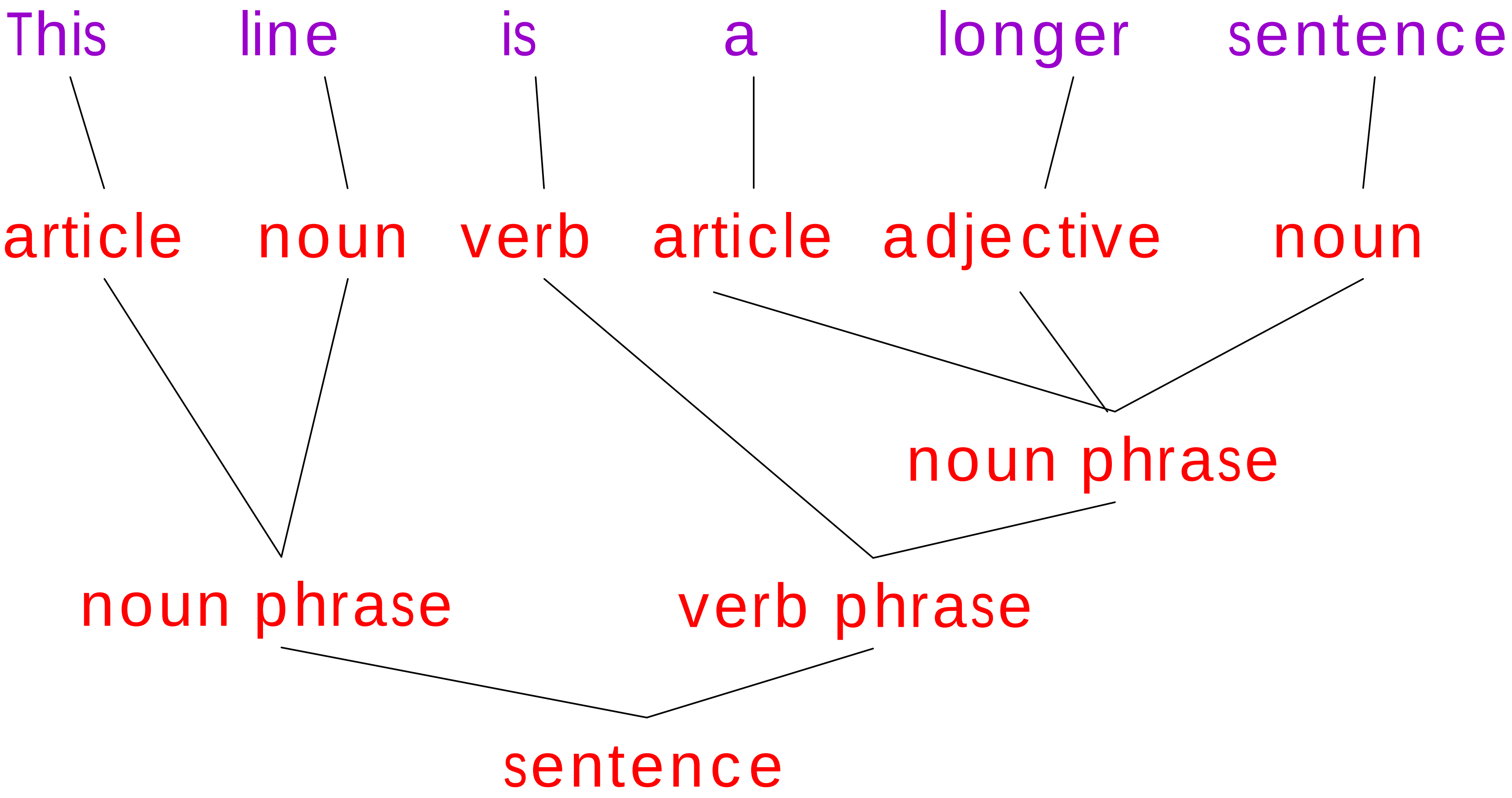
  if, (, x, ==, y,), then, z, =, 1, ;, else, z, =, 2, ;

Second Step: Syntax Analysis (Parsing)

- Once words are identified, the next step is to understand the sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence (1)



This line is a longer sentence

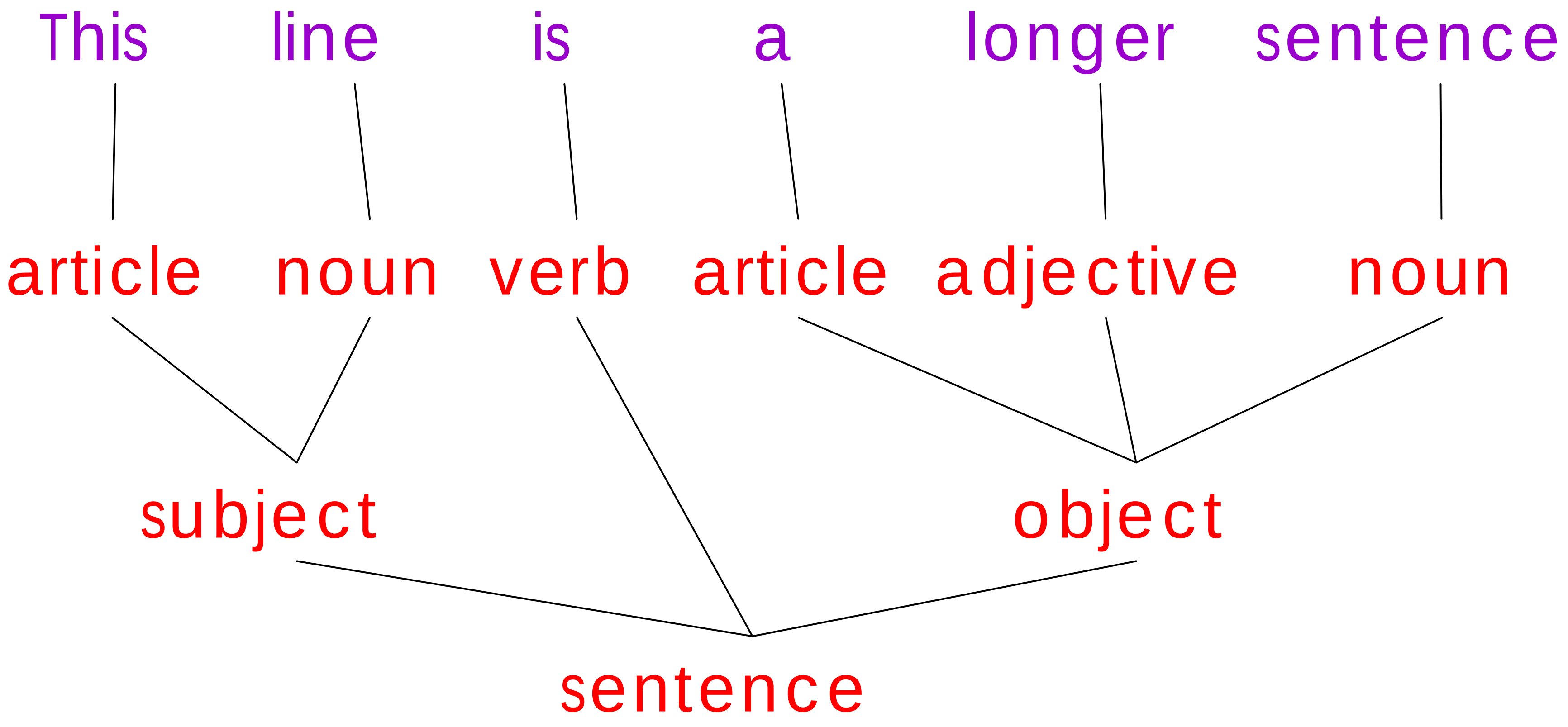
article noun verb article adjective noun

noun phrase

noun phrase verb phrase

sentence

Diagramming a Sentence (2)



This line is a longer sentence

article noun verb article adjective noun

subject object

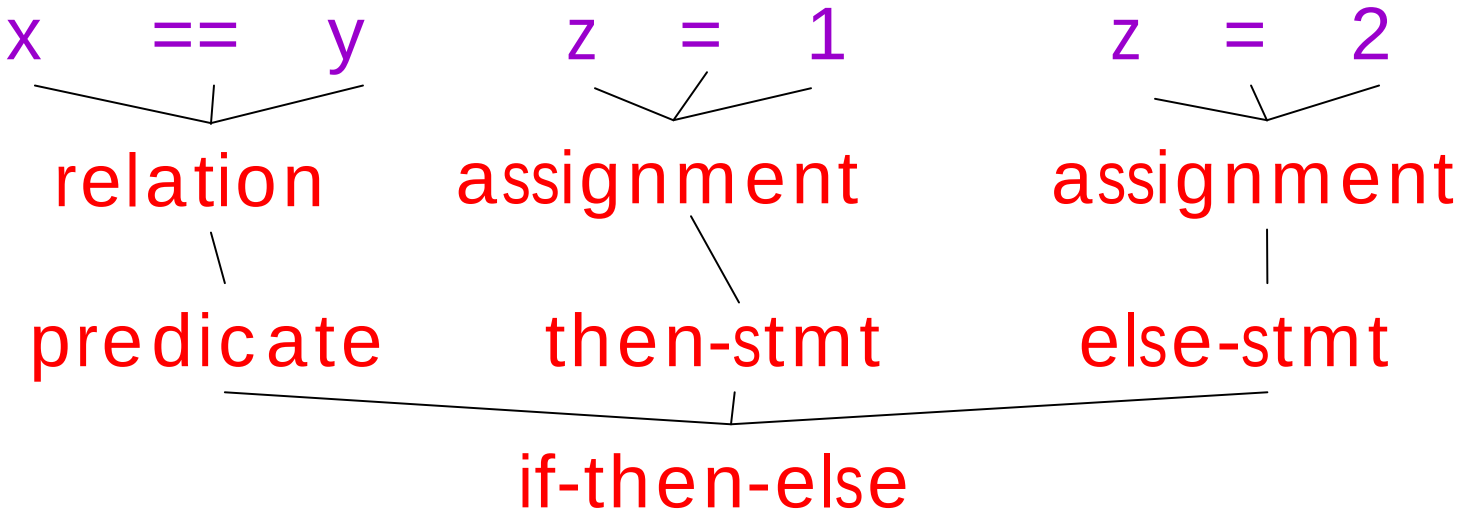
sentence

Parsing Programs

- Parsing program expressions is the same
- Consider:

```
    if (x == y) then z = 1; else z = 2;
```

- Diagrammed:



Third Step: Semantic Analysis

- Once the sentence structure is understood, we can try to understand its “meaning”
  - But meaning is too hard for compilers

- Most compilers perform limited analysis to catch inconsistencies

- Some optimizing compilers do more analysis to improve the performance of the program

Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does “his” refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his assignment at home.

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis in Programming Languages

- Programming languages define strict rules to avoid such ambiguities

- This C++ code prints 42; the inner definition is used

```
{
  int Jack = 17;
  {
    int Jack = 42;
    cout << Jack;
  }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings

- Example:

  Arnold left her homework at home.

- A "type mismatch" between her and Arnold; we know they are different people

  (Presumably Arnold is male...)

Optimization

- No strong counterpart in English, but akin to editing
- Automatically modify programs so that they
  - Run faster
    - avoid some source code redundancy
    - exploit the underlying hardware more effectively
  - Use less memory/cache/power
  - In general, conserve some resource more economically

Optimization Example

`X = Y * 0` is the same as `X = 0`

NO!

Valid for integers, but not for floating point numbers

Code Generation

- Produces assembly code (usually)

- A translation into another language
  - Analogous to human translation

Intermediate Languages

- Many compilers perform translations between successive intermediate forms
  - All but first and last are *intermediate languages* internal to the compiler
  - Typically there is one IL

- Intermediate languages generally ordered in descending level of abstraction
  - Highest is source
  - Lowest is assembly

Intermediate Languages (Cont.)

- IL's are useful because lower levels expose features hidden by higher levels
  - registers
  - memory/frame layout
  - etc.

- But lower levels obscure high-level meaning

Issues

- Compiling is almost this simple, but there are many pitfalls

- Example: How are erroneous programs handled?

- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design

Compilers Today

- The overall structure of almost every compiler adheres to our outline

- The proportions have changed since FORTRAN
  - Early:
    - lexical analysis, parsing most complex, expensive
  - Today:
    - lexical analysis and parsing are well-understood and cheap
    - semantic analysis and optimization dominate
    - focus on concurrency/parallelism and interactions with the memory model of the underlying platform
    - optimization for code size and energy consumption

Current Trends in Compilation

- Compilation for speed is less interesting. However, there are exceptions:
  - scientific programs
  - advanced processors (Digital Signal Processors, advanced speculative architectures, GPUs)

- Ideas from compilation used for improving code reliability:
  - memory safety
  - detecting data races
  - security properties
  - ...

Programming Language Economics

- Programming languages are designed to fill a void
  - enable a previously difficult/impossible application
  - orthogonal to language design quality (almost)

- Programming training is the dominant cost
  - Languages with a big user base are replaced rarely
  - Popular languages become ossified
  - But it is easy to start in a new niche...

Why So Many Programming Languages?

- Application domains have distinctive (and sometimes conflicting) needs
- Examples:
  - ***Scientific computing***: High performance
  - ***Business***: report generation
  - ***Artificial intelligence***: symbolic computation
  - ***Systems programming***: efficient low-level access
  - ***Web programming***: scripts that run everywhere
  - ***Multicores***: concurrency and parallelism
  - Other special purpose languages...

Topic: Language Design

- No universally accepted metrics for design

- “A good language is one people use”

- NO !
  - Is COBOL the best language?

- Good language design is hard

Language Evaluation Criteria

Characteristic	Criteria		
	Readability	Writeability	Reliability
Simplicity	YES	YES	YES
Data types	YES	YES	YES
Syntax design	YES	YES	YES
Abstraction		YES	YES
Expressivity		YES	YES
Type checking			YES
Exceptions			YES

History of Ideas: Abstraction

- Abstraction = detached from concrete details
- Necessary for building software systems
- Modes of abstraction:
  - Via languages/compilers
    - higher-level code; few machine dependencies
  - Via subroutines
    - abstract interface to behavior
  - Via modules
    - export interfaces which hide implementation
  - Via abstract data types
    - bundle data with its operations

History of Ideas: Types

- Originally, languages had only few types
  - FORTRAN: scalars, arrays
  - LISP: no static type distinctions
- Realization: types help
  - provide code documentation
  - allow the programmer to express abstraction
  - allow the compiler to check among many frequent errors and sometimes guarantee various forms of safety
- More recently:
  - experiments with various forms of parameterization
  - best developed in functional languages

History of Ideas: Reuse

- Exploits common patterns in software development
- Goal: mass produced software components
- Reuse is difficult
- Two popular approaches (combined in C++)
  - Type parameterization (List(Int) & List(Double))
  - Class and inheritance: C++ derived classes
- Inheritance allows:
  - specialization of existing abstractions
  - extension, modification and information hiding

Current Trends

- **Language design**
  - Many new special-purpose languages
  - Popular languages to stay

- **Compilers**
  - More needed and more complex
  - Driven by increasing gap between
    - new languages
    - new architectures
  - Venerable and healthy area

Why Study Compilers?

- Increase your knowledge of common programming constructs and their properties
- Improve your understanding of program execution
- Increase your ability to learn new languages

- Learn how languages are implemented
- Learn new (programming) techniques
- See many basic CS concepts at work

Εισαγωγή στη Λεκτική Ανάλυση

Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexical analyzers (lexers)
  - Regular expressions
  - Examples of regular expressions

Lexical Analysis

- What do we want to do? Example:

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

- The input is just a string of characters:

```
if (i == j)\nthen\n\tz = 0;\n\telse\n\t\tz = 1;
```

- Goal: Partition input string into substrings
  - where the substrings are tokens
  - and classify them according to their role

What's a Token?

- A syntactic category
  - In a natural language:

    noun, verb, adjective, …

  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …

Tokens

- Tokens correspond to sets of strings
  - these sets depend on the programming language

For example, our language could specify:

- Identifier: *strings of letters or digits, starting with a letter.*
- Integer: *a non-empty string of digits.*
- Keyword: "else" *or* "if" *or* "begin" *or …*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs.*

What are Tokens Used for?

- Classify program substrings according to role
- Output of lexical analysis is a stream of tokens . . .
- . . . which is input to the parser
- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser

- For our running example:

  `if (i == j)\nthen\n\tz = 0;\n\telse\n\t\tz = 1;`

- Useful tokens are:

  Integer, Keyword, Relation, Identifier, Whitespace, (,), =, ;

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token

- Recall our language's specification:
  - Identifier: *strings of letters or digits, starting with a letter.*
  - Integer: *a non-empty string of digits.*
  - Keyword: "else" *or* "if" *or* "begin" *or …*
  - Whitespace: *a non-empty sequence of blanks, newlines, and tabs.*

Lexical Analyzer: Implementation

An implementation must do two things:

1. Recognize substrings corresponding to tokens

2. Return the value or lexeme of the token
   - The lexeme is the substring

Example

- For our running example:

  if (i == j)\nthen\n\tz = 0;\n\telse\n\t\tz = 1;

- Token-lexeme groupings:
  - Identifier: i, j, z
  - Keyword: if, then, else
  - Relation: ==
  - Integer: 0, 1
  - (,), =, ; single character of the same token name

Why do Lexical Analysis?

- Simplify parsing
  - The lexer usually discards “uninteresting” tokens that don’t contribute to parsing
    - E.g. Whitespace, Comments
  - Converts data early
- Separate out logic to read source files
  - Potentially an issue on multiple platforms
  - Can optimize reading code independently of parser

True Crimes of Lexical Analysis

- Is it as easy as it sounds?

- Not quite!

- Look at some programming language history . . .

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant

- E.g., **`VAR1`** is the same as **`VA    R1`**

FORTRAN whitespace rule was motivated by inaccuracy of punch card operators

A terrible design! Example

- Consider
  - `DO 5 I = 1,25`
  - `DO 5 I = 1.25`

- The first is DO 5 I = 1 , 25 (iteration)
- The second is DO5I = 1.25 (assignment)

- Reading left-to-right, the lexical analyzer cannot tell if DO5I is a variable or a DO statement until after "," is reached

Lexical Analysis in FORTRAN. Lookahead.

Two important points:

1. The goal is to partition the string
   - This is implemented by reading left-to-right, recognizing one token at a time

2. "Lookahead" may be required to decide where one token ends and the next token begins
   - Even our simple example has lookahead issues

     `i` vs. `if`

     `=` vs. `==`

Another Great Moment in Scanning History

PL/1: Keywords can be used as identifiers:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

can be difficult to determine how to label lexemes

More Modern True Crimes in Scanning

Nested template declarations in C++

```
vector<vector<int>> myVector

vector < vector < int >> myVector

(vector < (vector < (int >> myVector)))
```

Review

- The goal of lexical analysis is to
  - Partition the input string into *lexemes* (the smallest program units that are individually meaningful)
  - Identify the token of each lexeme
- Left-to-right scan ⇒ lookahead sometimes required
- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is `if` two variables `i` and `f`?
    - Is `==` two equal signs `=` `=`?

Regular Languages

- There are several formalisms for specifying tokens

- *Regular languages* are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

Def. Let Σ be a set of characters.

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

(Σ is called the *alphabet* of Λ)

Examples of Languages

- Alphabet = set of English characters
- Language = set of English sentences
- Not every string of English characters is an English word

- Alphabet = set of ASCII characters
- Language = set of C programs
- Not every string of ASCII characters is a valid C token

Notation

- Languages are sets of strings

- Need some notation for specifying which sets of strings we want our language to contain

- The standard notation for regular languages is *regular expressions*

Atomic Regular Expressions

- Single character

$$'c' = \{"c"\}$$

- Epsilon

$$\varepsilon = \{""\}$$

Compound Regular Expressions

- Union

$$A + B = \{ s \mid s \in A \text{ or } s \in B \}$$

- Concatenation

$$AB = \{ ab \mid a \in A \text{ and } b \in B \}$$

- Iteration

$$A^* = \bigcup_{i \geq 0} A^i \quad \text{where} \quad A^i = A \text{..} i \text{ times ...} A$$

Regular Expressions

Def. The *regular expressions over* Σ are the smallest set of expressions including

ε

$'c'$ where $c \in \Sigma$

$A + B$ where A, B are rexp over Σ

AB " " "

A^* where A is a rexp over Σ

Syntax vs. Semantics

- To be careful, we should distinguish syntax and semantics (meaning) of regular expressions

$$
\begin{aligned}
L(\varepsilon) &= \{""\} \\
L('c') &= \{"c"\} \\
L(A+B) &= L(A) \cup L(B) \\
L(AB) &= \{ab \mid a \in L(A) \text{ and } b \in L(B)\} \\
L(A^*) &= \bigcup_{i \geq 0} L(A^i)
\end{aligned}
$$

Example: Keyword

Keyword: "else" *or* "if" *or* "begin" *or* …

'else' + 'if' + 'begin' + ⋯

Note: 'else' abbreviates 'e''l''s''e'

Example: Integers

Integer: *a non-empty string of digits*

$$\text{digit} = \text{'0'}+\text{'1'}+\text{'2'}+\text{'3'}+\text{'4'}+\text{'5'}+\text{'6'}+\text{'7'}+\text{'8'}+\text{'9'}$$

$$\text{integer} = \text{digit digit}^*$$

Abbreviation: $A^+ = AA^*$

Example: Identifier

Identifier: *strings of letters or digits, starting with a letter*

$$\text{letter} \quad = \quad \text{'A'} + \ldots + \text{'Z'} + \text{'a'} + \ldots + \text{'z'}$$

$$\text{identifier} \quad = \quad \text{letter}\ (\text{letter} + \text{digit})^{*}$$

Is $(\text{letter}^{*} + \text{digit}^{*})$ the same?

Example: Whitespace

Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

$$(\text{' '} + \text{'\textbackslash n'} + \text{'\textbackslash t'})^{+}$$

Example 1: Phone Numbers

- Regular expressions are all around you!
- Consider +30 210-772-2487

Σ	= digits ∪ {+,–}
country	= digit digit
city	= digit digit digit
univ	= digit digit digit
extension	= digit digit digit digit
phone_num	= ‘+’country‘ ’city‘–’univ‘–’extension

Example 2: Email Addresses

- Consider *kostis@cs.ntua.gr*

$$\Sigma = \text{letters} \cup \{.,@\}$$

$$\text{name} = \text{letter}^{+}$$

$$\text{address} = \text{name}\ \text{'@'}\ \text{name}\ \text{'.'}\ \text{name}\ \text{'.'}\ \text{name}$$

Summary

- Regular expressions describe many useful languages
- Regular languages are a language specification
  - We still need an implementation

- Next: *Given a string s and a regular expression R, is*

$$s \in L(R)?$$

- *A* yes/no answer is not enough!
- Instead: partition the input into tokens
- We will adapt regular expressions to this goal

Υλοποίηση της Λεκτικής Ανάλυσης

Outline

- Specifying lexical structure using regular expressions

- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)

- Implementation of regular expressions

  RegExp ⇒ NFA ⇒ DFA ⇒ Tables

Notation

- For convenience, we will use a variation (we will allow user-defined abbreviations) in regular expression notation

- Union: $A + B \equiv A \mid B$
- Option: $A + \varepsilon \equiv A?$
- Range: 'a'+'b'+…+'z' ≡ [a-z]
- Excluded range:

  complement of [a-z] ≡ [^a-z]

Regular Expressions ⇒ Lexical Specifications

1. Select a set of tokens
   - Integer, Keyword, Identifier, LeftPar, ...

2. Write a regular expression (pattern) for the lexemes of each token
   - Integer = digit+
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - LeftPar = '('
   - ...

Regular Expressions ⇒ Lexical Specifications

3. Construct R, a regular expression matching all lexemes for all tokens

$$R = \text{Integer} + \text{Keyword} + \text{Identifier} + \ldots$$
$$= R_1 + R_2 + R_3 + \ldots$$

Facts: If $s \in L(R)$ then s is a lexeme

- Furthermore $s \in L(R_j)$ for some "j"
- This "j" determines the token that is reported

Regular Expressions ⇒ Lexical Specifications

4. Let input be $x_1 \ldots x_n$
   - ($x_1 \ldots x_n$ are characters in the language alphabet)
   - For $1 \leq i \leq n$ check

     $x_1 \ldots x_i \in L(R)$?

5. It must be that $x_1 \ldots x_i \in L(R_j)$ for some i and j

   (if there is a choice, pick the smallest such j)

6. Report token j, remove $x_1 \ldots x_i$ from input and go to step 4

How to Handle Spaces and Comments?

1. We could create a token Whitespace

   Whitespace = (' ' + '\n' + '\t')$^+$

   - We could also add comments in there
   - An input "   \t\n   555   " is transformed into

     Whitespace Integer Whitespace

2. Lexical analyzer skips spaces (not always!)
   - Modify step 5 from before as follows:

     It must be that $x_k \dots x_i \in L(R_j)$ for some j such that $x_1 \dots x_{k-1} \in L(\text{Whitespace})$
   - Parser is not bothered with spaces

Ambiguities (1)

- There are ambiguities in the algorithm.

- How much input is used?
- What if

  $x_1 \ldots x_i \in L(R)$ and also $x_1 \ldots x_K \in L(R)$

- The "maximal munch" rule: Pick the longest possible substring that matches R

Ambiguities (2)

- Which token is used?
- What if

  $x_1 \ldots x_i \in L(R_j)$ and also $x_1 \ldots x_i \in L(R_k)$
- Rule: use rule listed first (j if $j < k$)

- Example:
  - R_2 = Keyword and R_3 = Identifier
  - "if" matches both
  - Treats "if" as a keyword not an identifier

Error Handling

- What if

  No rule matches a prefix of input?
- Problem: Can't just get stuck …
- Solution:
  - Write a rule matching all "bad" strings
  - Put it last
- Lexical analysis tools allow the writing of:

  $R = R_1 + \ldots + R_n + \text{Error}$
  - Token Error matches if nothing else matches

Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)

Regular Languages & Finite Automata

Basic formal language theory result:

> *Regular expressions and finite automata both define the class of regular languages.*

Thus, we are going to use:

- Regular expressions for specification
- Finite automata for implementation (automatic generation of lexical analyzers)

Finite Automata

A finite automaton is a *recognizer* for the strings of a regular language

A finite automaton consists of

- A finite input alphabet Σ
- A set of states S
- A start state n
- A set of accepting states $F \subseteq S$
- A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \rightarrow^{a} s_2$$

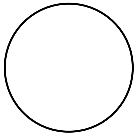
- Is read

  In state s_1 on input "a" go to state s_2

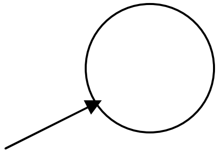
- If end of input
  - If in accepting state ⇒ accept
- Otherwise
  - If no transition is possible ⇒ reject

Finite Automata State Graphs

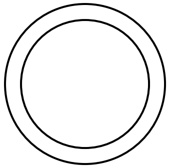
- A state



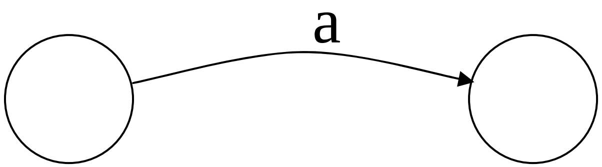
- The start state



- An accepting state

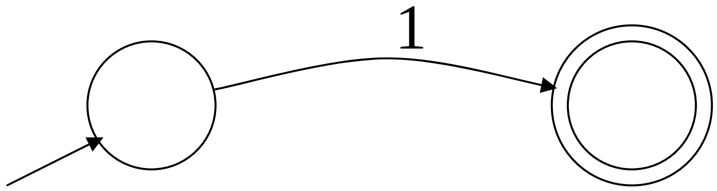


- A transition



A Simple Example

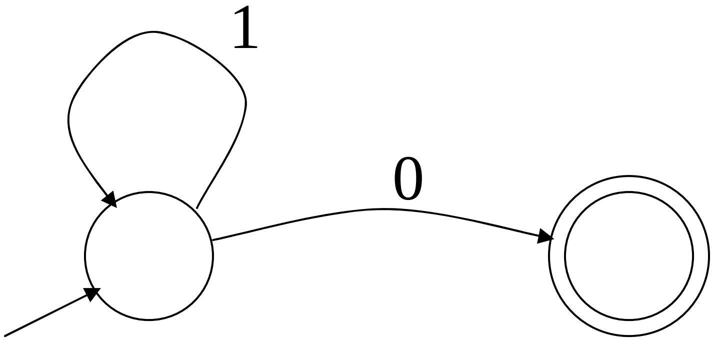
- A finite automaton that accepts only "1"



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

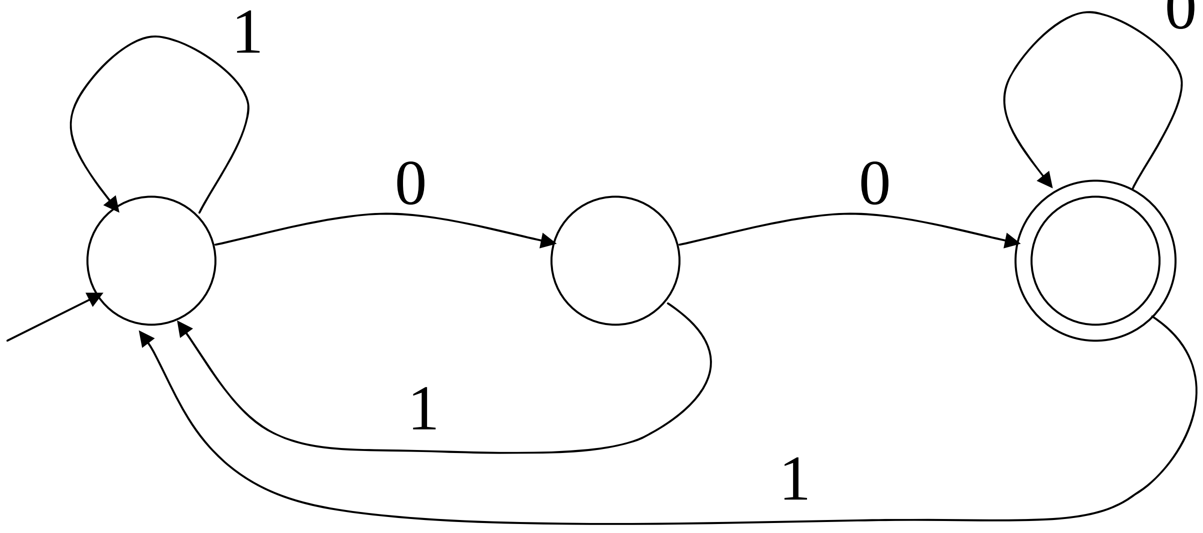
Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



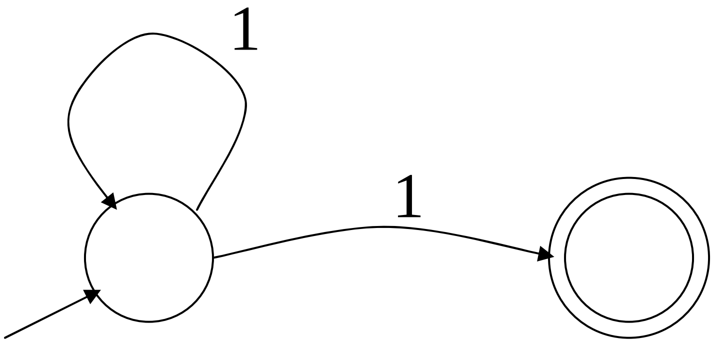
And Another Example

- Alphabet {0,1}
- What language does this recognize?



And Another Example

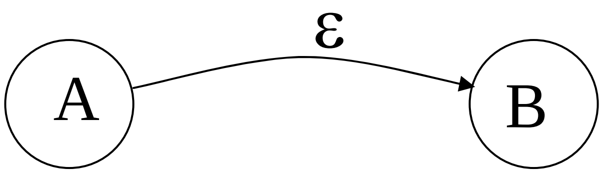
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input "11" the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Deterministic and Non-Deterministic Automata

- **Deterministic Finite Automata (DFA)**
  - One transition per input per state
  - No ε-moves
- **Non-deterministic Finite Automata (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- Finite automata have finite memory
  - Enough to only encode the current state

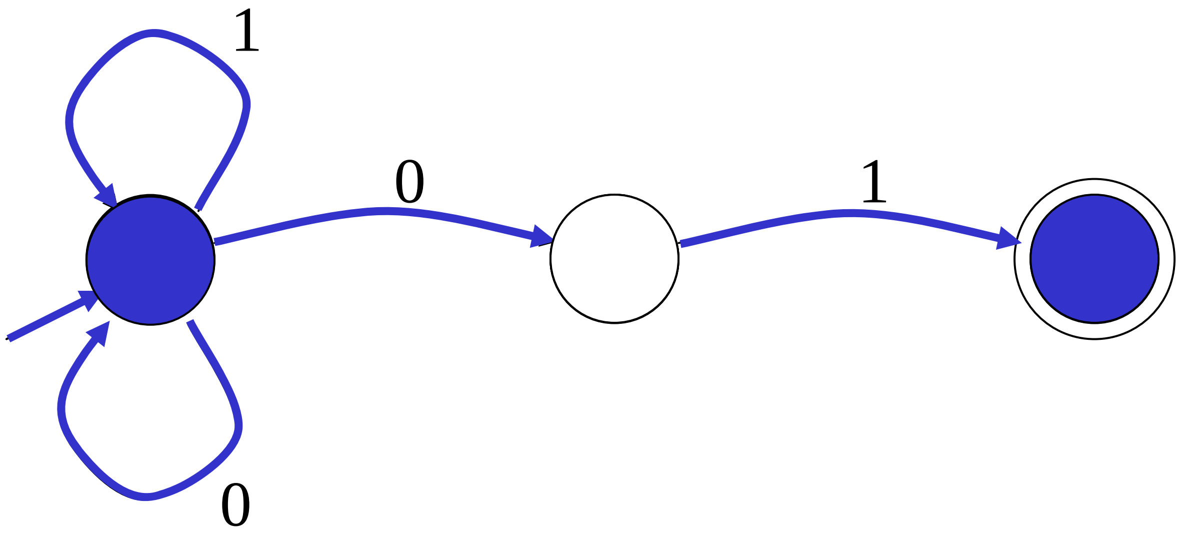
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts an input if it <u>can</u> get in a final state

NFA vs. DFA (1)

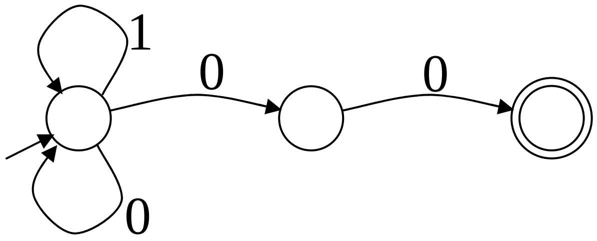
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider

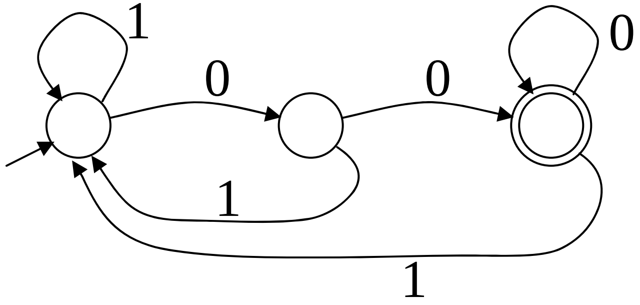
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



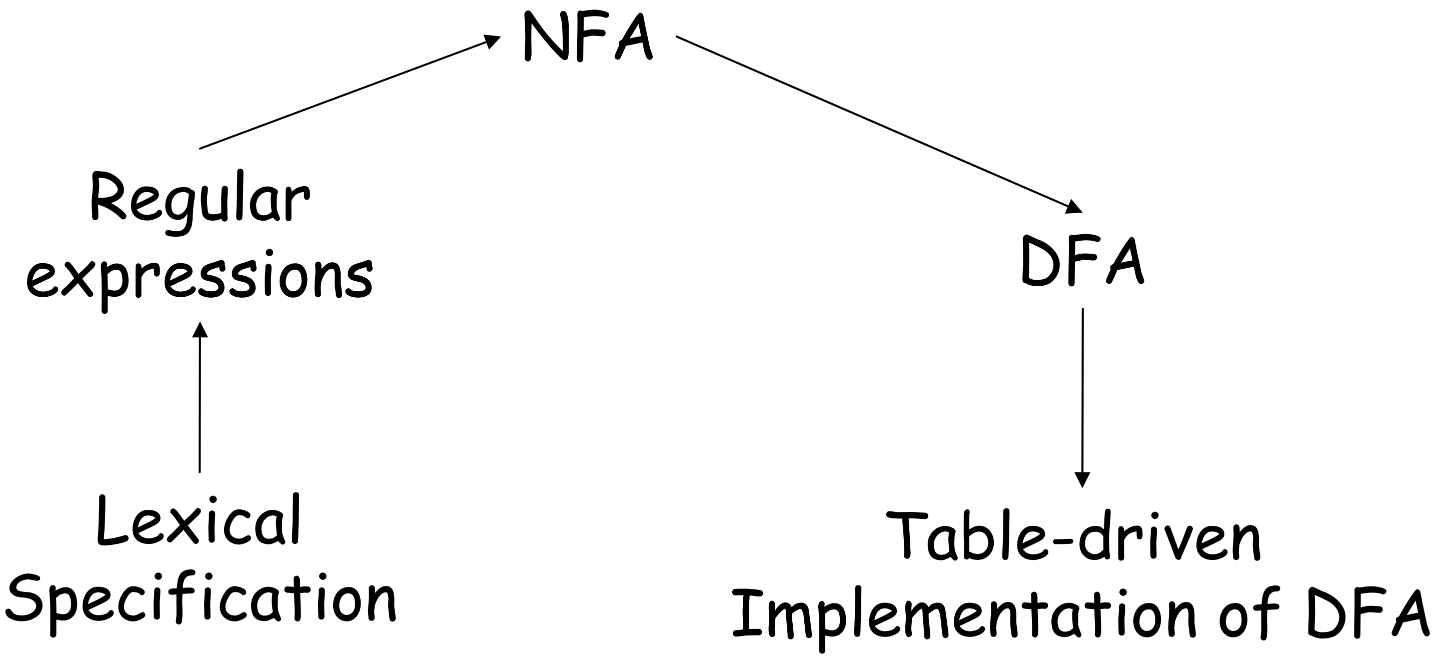
DFA



- DFA can be exponentially larger than NFA
  (contrary to what is shown in the above example)

Regular Expressions to Finite Automata

- High-level sketch



NFA

Regular expressions

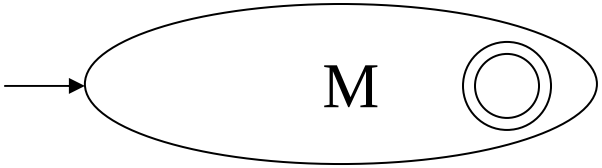
DFA

Lexical Specification

Table-driven Implementation of DFA

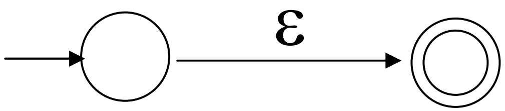
Regular Expressions to NFA (1)

- For each kind of reg. expr, define an NFA
  - Notation: NFA for regular expression M

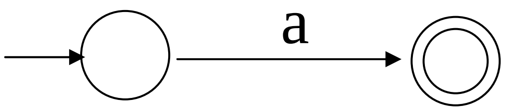


i.e. our automata have **one** start and **one** accepting state

- For ε

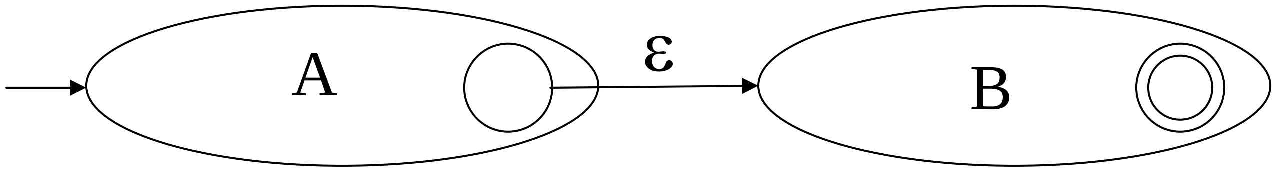


- For input a

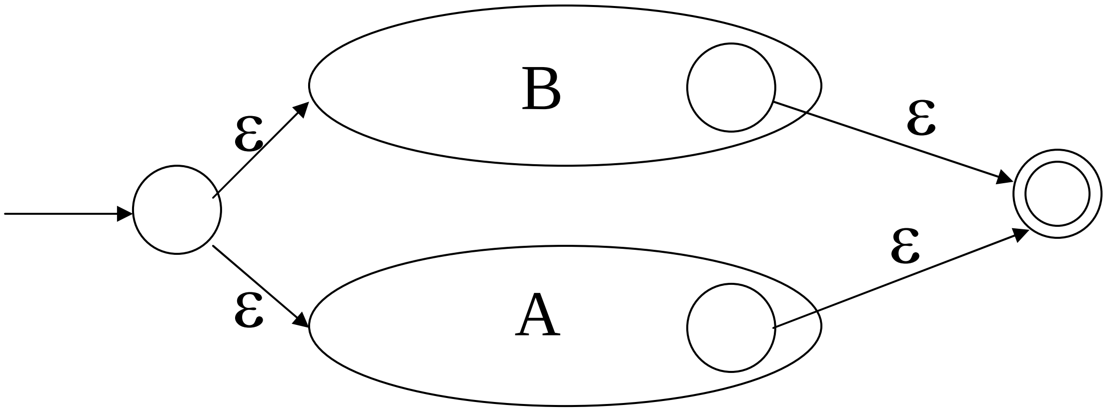


Regular Expressions to NFA (2)

- For AB

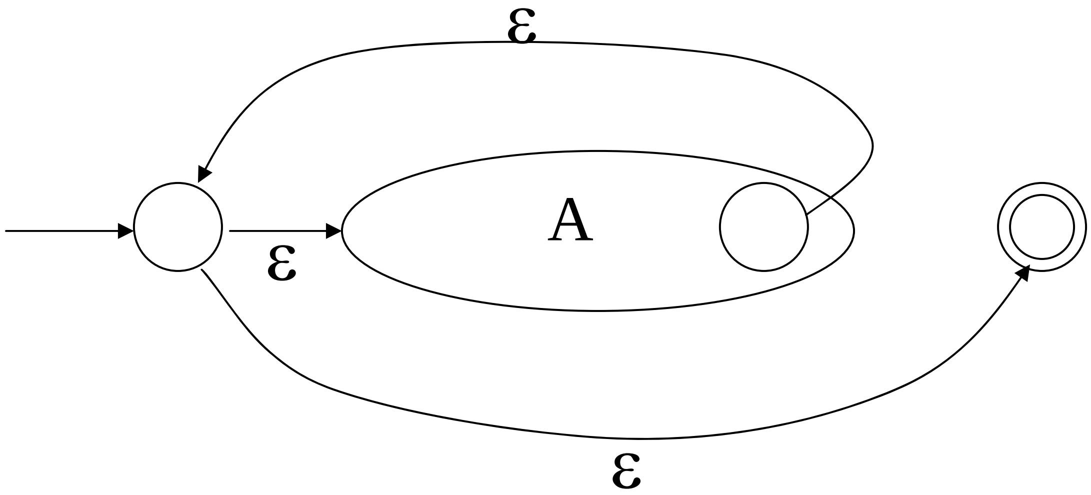


- For A + B



Regular Expressions to NFA (3)

- For *A**

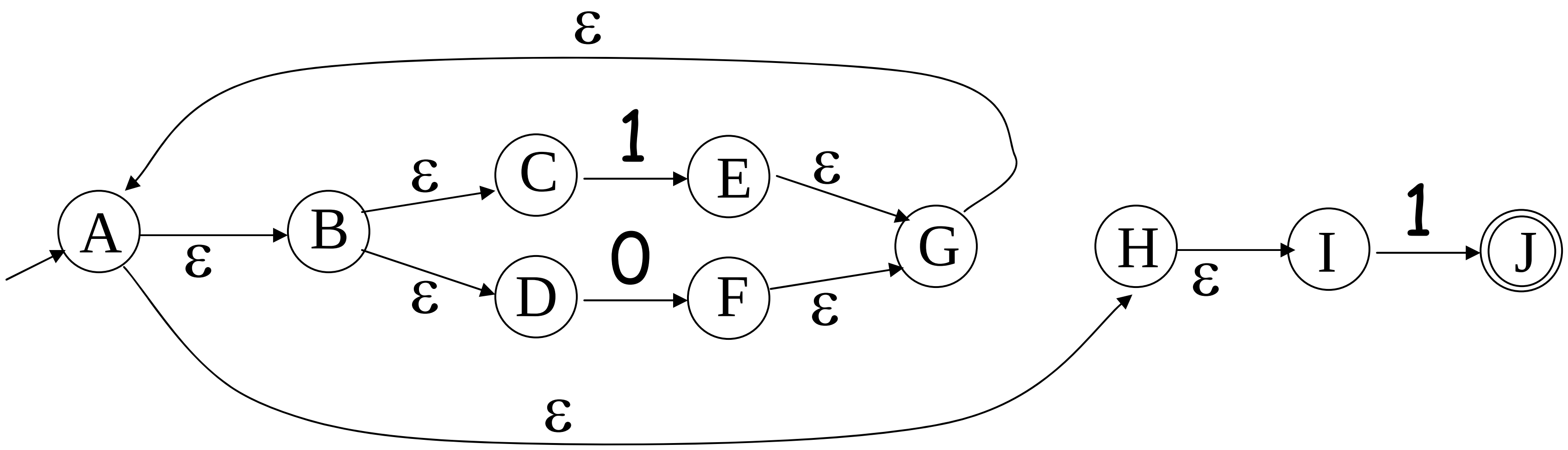


Example of Regular Expression → NFA conversion

- Consider the regular expression

  (1+0)*1

- The NFA is



NFA to DFA. The Trick

- Simulate the NFA
- Each state of DFA
  = a non-empty subset of states of the NFA
- Start state
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from <u>any</u> state in S after seeing the input a
    - considering ε-moves as well

NFA to DFA. Remark

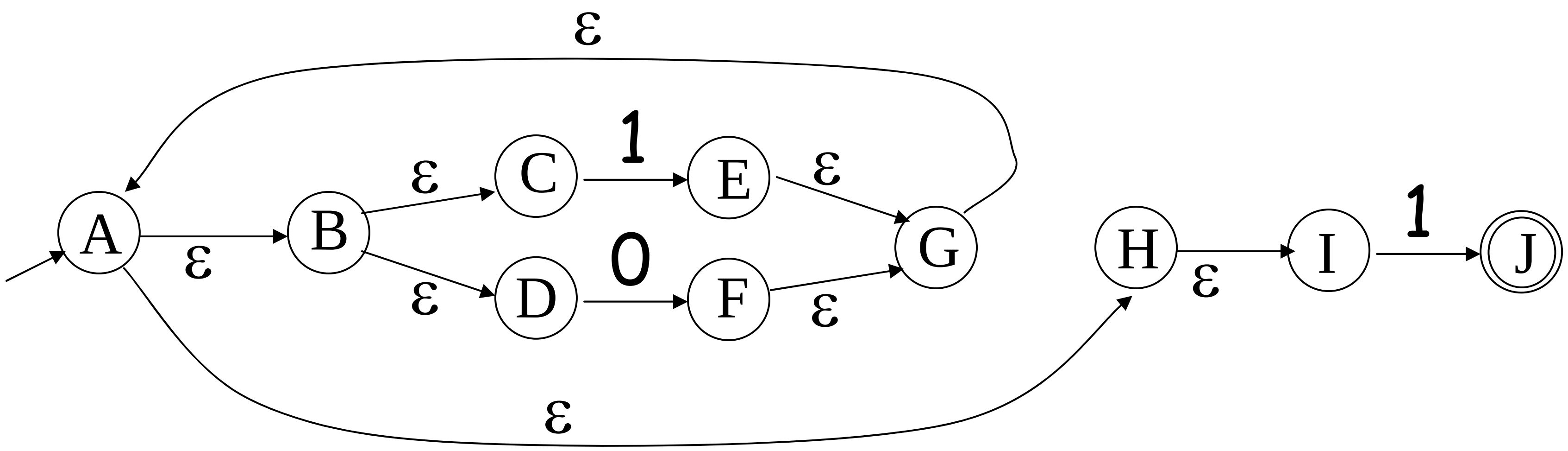
- An NFA may be in many states at any time

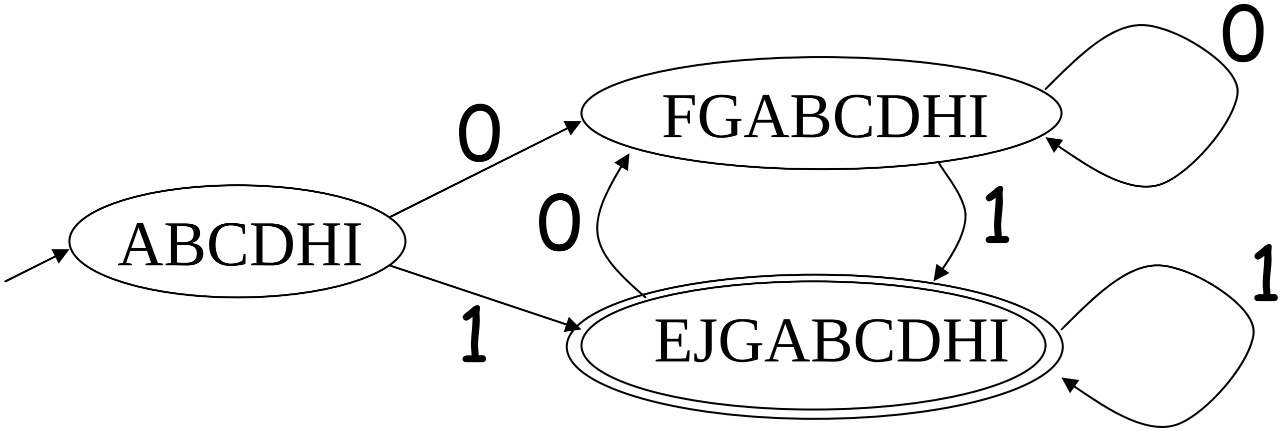
- How many different states ?

- If there are N states, the NFA must be in some subset of those N states

- How many subsets are there?
  - $2^N - 1$ = finitely many

NFA to DFA Example



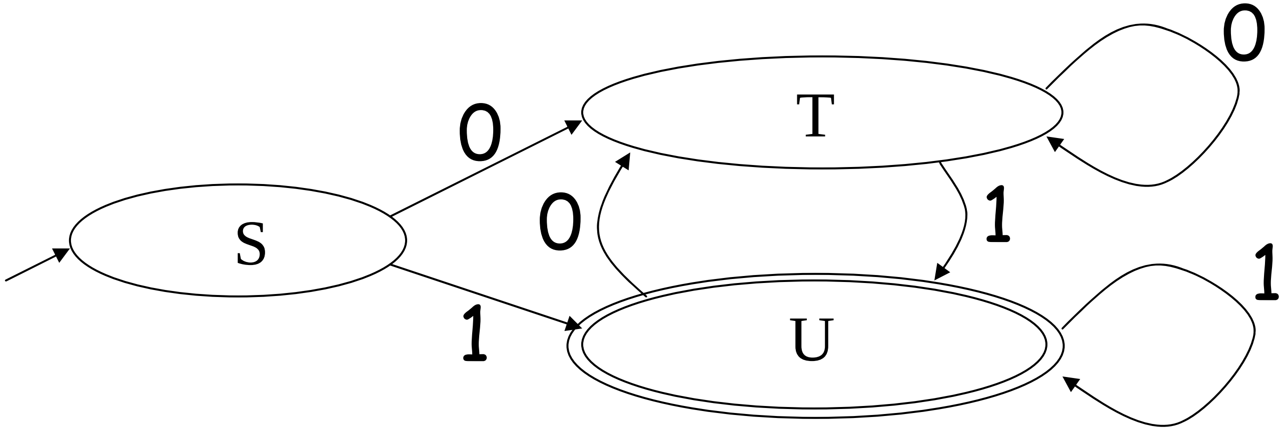


Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k

- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA → DFA conversion is at the heart of tools such as lex, ML-Lex, flex or jlex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

Theory vs. Practice

Two differences:

- DFAs *recognize* lexemes. A lexer must return a *type of acceptance* (token type) rather than simply an accept/reject indication.
- DFAs consume the complete string and accept or reject it. A lexer must *find* the end of the lexeme in the input stream and then find the *next* one, etc.

Introduction to Parsing, Ambiguity, and Syntax Errors

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFG's)
- Derivations
- Ambiguity
- Syntax errors

Languages and Automata

- Formal languages are very important in CS.
  - Especially in programming languages and compilers.

- Regular languages:
  - The weakest formal languages widely used.
  - Sufficient for many applications.

- We will also study context-free languages.

Limitations of Regular Languages

Intuition: A finite automaton that runs long enough must repeat states.

- A finite automaton *cannot remember* number of times it has visited a particular state ...
- ... because a finite automaton has finite memory.
  - Only enough to store in which state it is.
  - Cannot count, except up to a finite limit.
- Many languages are not regular.
- E.g., the language of balanced parentheses is not regular: $\{ (^n\ )^n \mid n \geq 0\}$

The Functionality of the Parser

- **Input:** sequence of tokens from lexer.

- **Output:** parse tree of the program.

Example

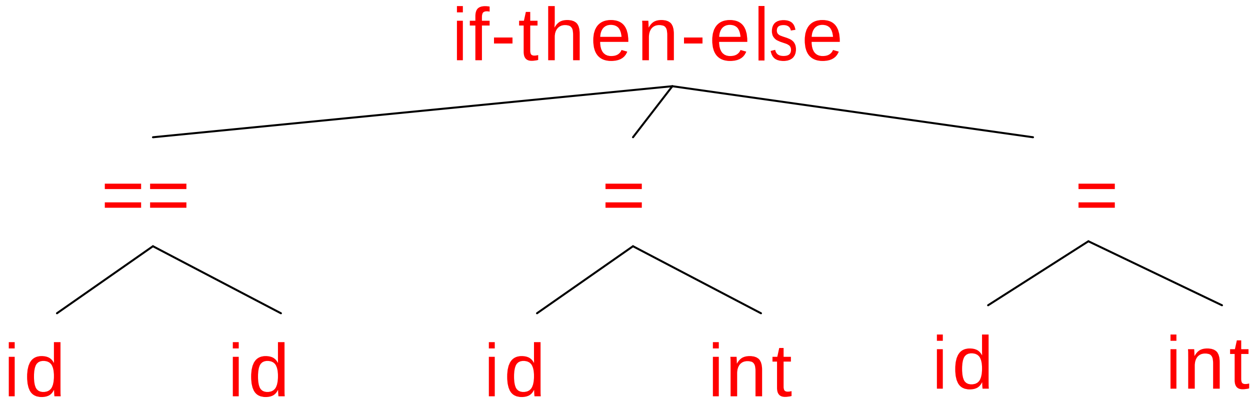
- If-then-else statement

```
if (x == y) then z = 1; else z = 2;
```

- Parser input (lexer output)

```
if (id == id) then id = int; else id = int;
```

- Possible parser output



Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

The Role of the Parser

- Not all sequences of tokens are programs ...
- Parser must distinguish between valid and invalid sequences of tokens.

- We need:
  - A language for describing valid sequences of tokens.
  - A method for distinguishing valid from invalid sequences of tokens.

Context-Free Grammars

- Many programming language constructs have a recursive structure.

- E.g. A STMT is of the form:

  if COND then STMT else STMT , or

  while COND do STMT , or

  …

- Context-free grammars (CFGs) are a natural notation for this recursive structure.

CFGs (Cont.)

A CFG consists of

- A set of *terminals* T
- A set of *non-terminals* N
- A *start symbol* S (a non-terminal)
- A set of *productions*

Assuming $X \in N$ the productions are of the form

$X \to \varepsilon$, or

$X \to Y_1 Y_2 \ldots Y_n$ where $Y_i \in N \cup T$

Notational Conventions

- In these lecture notes:
  - Non-terminals are written upper-case.
  - Terminals are written lower-case.
  - The start symbol is the left-hand side of the first production.

Example: A small fragment of our language:

```
STMT → if COND then STMT else STMT
     | while COND do STMT
     | id = int
```

One More Example

Grammar for simple arithmetic expressions:

$$
\begin{array}{lll}
E & \rightarrow & E * E \\
  & | & E + E \\
  & | & (E) \\
  & | & \mathrm{int}
\end{array}
$$

The Language of a CFG

Read productions as replacement rules:

$X \rightarrow Y_1 \dots Y_n$

Means X can be replaced by $Y_1 \dots Y_n$ (in this order).

$X \rightarrow \varepsilon$

Means X can be erased (replaced with empty string).

Key Idea

(1) Begin with a string consisting of the start symbol “S”

(2) Replace any non-terminal X in the string by the right-hand side of some production

$$X \to Y_1 \ldots Y_n$$

(3) Repeat (2) until there are no non-terminals in the string

The Language of a CFG (Cont.)

More formally, we write

$$X_1 \cdots X_i \cdots X_n \rightarrow X_1 \cdots X_{i-1} Y_1 \cdots Y_m X_{i+1} \cdots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \cdots Y_m$$

We write

$$X_1 \cdots X_n \overset{*}{\rightarrow} Y_1 \cdots Y_m$$

if

$$X_1 \cdots X_n \rightarrow \cdots \rightarrow \cdots \rightarrow Y_1 \cdots Y_m$$

in 0 or more steps.

The Language of a CFG (Cont.)

Let *G* be a context-free grammar with start symbol *S*. Then the language of *G* is:

$$\left\{ a_1 \ldots a_n \mid S \overset{*}{\rightarrow} a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \right\}$$

Terminals

- Terminals are called so because there are no rules for replacing them.
- Once generated, terminals are permanent.
- Terminals ought to be tokens of the language.

Examples

L(G) is the language of the CFG G

Strings of balanced parentheses $\{(^i\,)^i \mid i \geq 0\}$

Two equivalent ways of writing the grammar G:

$$\boxed{\begin{array}{lll} S & \rightarrow & (S) \\ S & \rightarrow & \varepsilon \end{array}} \quad \textit{or} \quad \boxed{\begin{array}{lll} S & \rightarrow & (S) \\ & | & \varepsilon \end{array}}$$

Example

A fragment of our example language (simplified):

$$
\begin{array}{lcl}
\text{STMT} & \rightarrow & \text{if COND then STMT} \\
 & \mid & \text{if COND then STMT else STMT} \\
 & \mid & \text{while COND do STMT} \\
 & \mid & \text{id = int} \\
\text{COND} & \rightarrow & \text{(id == id)} \\
 & \mid & \text{(id != id)}
\end{array}
$$

Example (Cont.)

Some elements of our example language:

```
id = int
if (id == id) then id = int else id = int
while (id != id) do id = int
while (id == id) do while (id != id) do id = int
if (id != id) then if (id == id) then id = int else id = int
```

Arithmetic Expressions Example

Grammar for simple arithmetic expressions:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$$

Some elements of the language:

int

(int)

(int) * int

int + int

int * int

int * (int)

Notes

The idea of a CFG is a big step.

But:

- Membership in a language is just "yes" or "no"; we also need the parse tree of the input.
- Must handle errors gracefully.
- Need an implementation of CFG's
  - e.g., yacc/bison/ML-yacc/...

Derivations and Parse Trees

A *derivation* is a sequence of productions:

$$S \rightarrow \cdots \rightarrow \cdots \rightarrow \cdots$$

A derivation can be drawn as a tree.

- Start symbol is the tree's root.
- For a production $X \rightarrow Y_1 \cdots Y_n$ add children $Y_1 \cdots Y_n$ to node X

Derivation Example

- Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$$

- String

int * int + int

Derivation Example (Cont.)

$E \rightarrow E{+}E \mid E{*}E \mid (E) \mid \mathrm{int}$

E

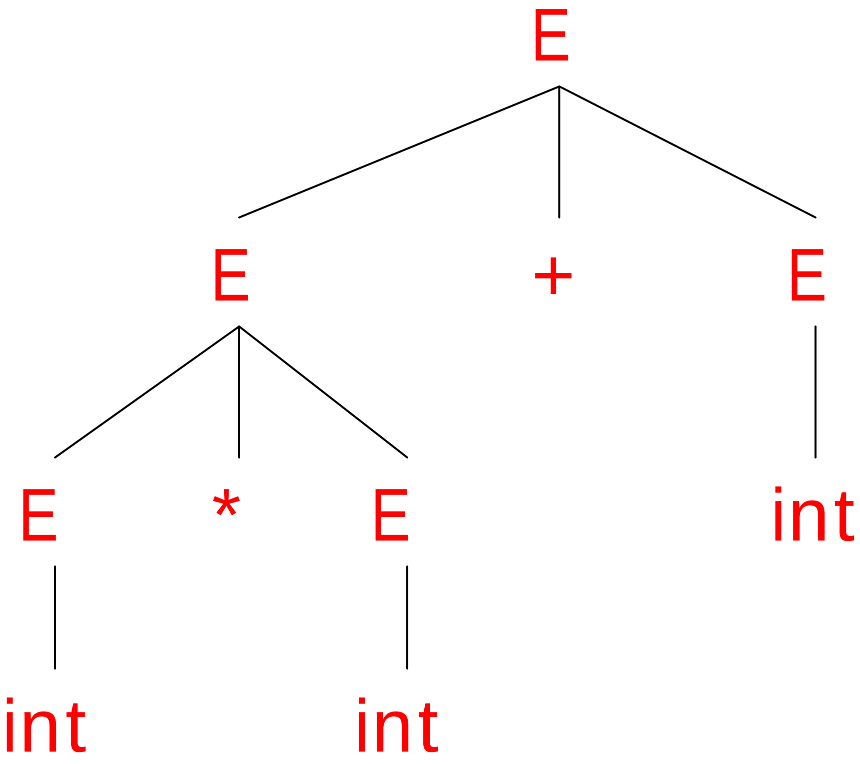
→ E + E

→ E * E + E

→ int * E + E

→ int * int + E

→ int * int + int



Derivation in Detail (1)

E → E+E | E*E | (E) | int

E

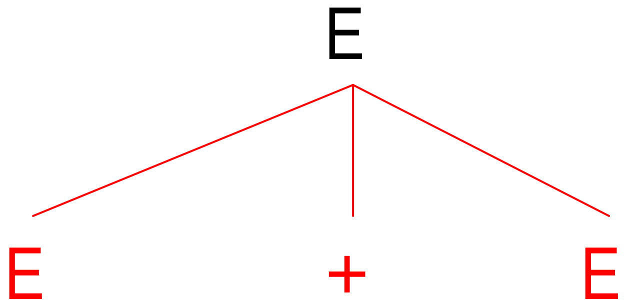
E

Derivation in Detail (2)

E → E+E | E*E | (E) | int

E

→ E + E



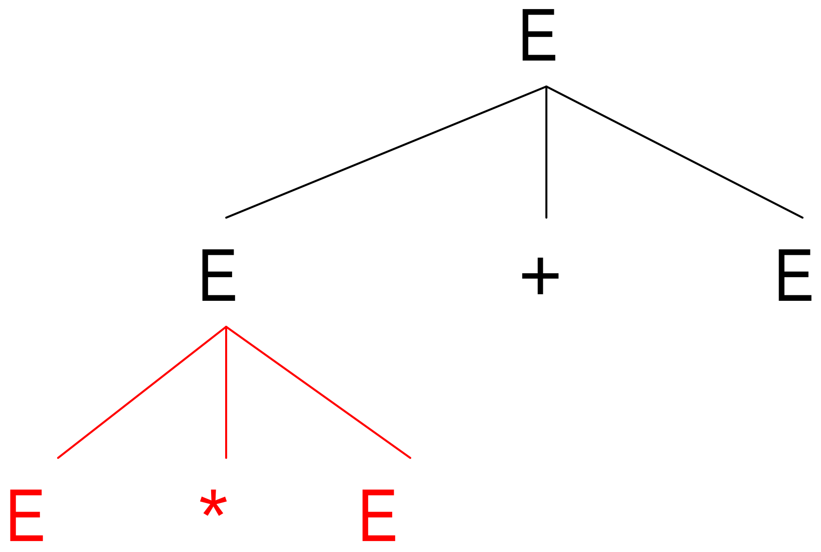
Derivation in Detail (3)

E → E+E | E*E | (E) | int

E

→ E + E

→ E * E + E



Derivation in Detail (4)

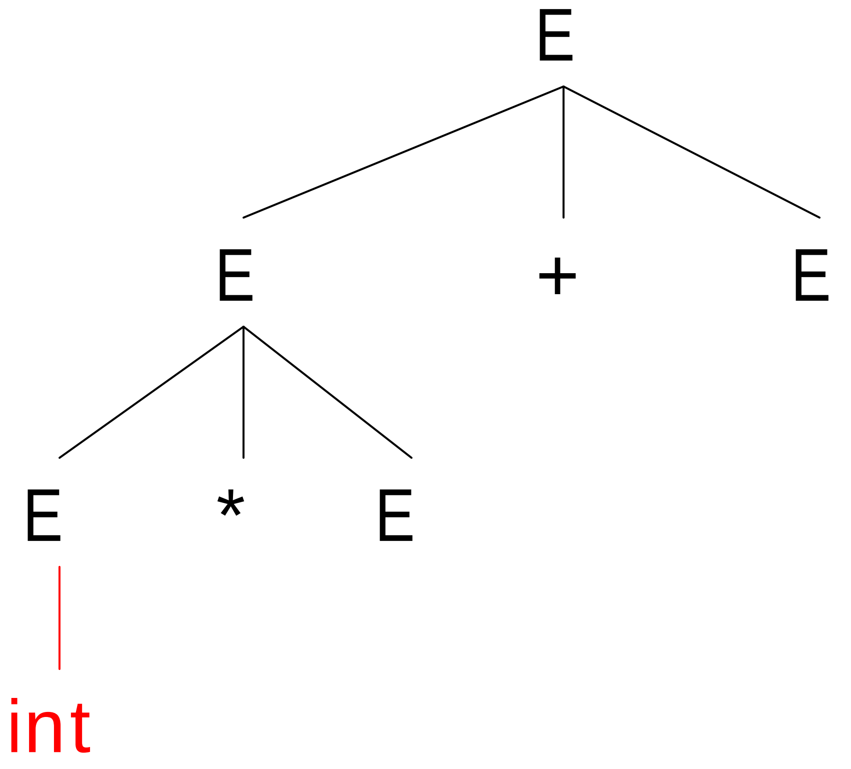
E → E+E | E*E | (E) | int

E

→ E + E

→ E * E + E

→ int * E + E



Derivation in Detail (5)

E → E+E | E*E | (E) | int

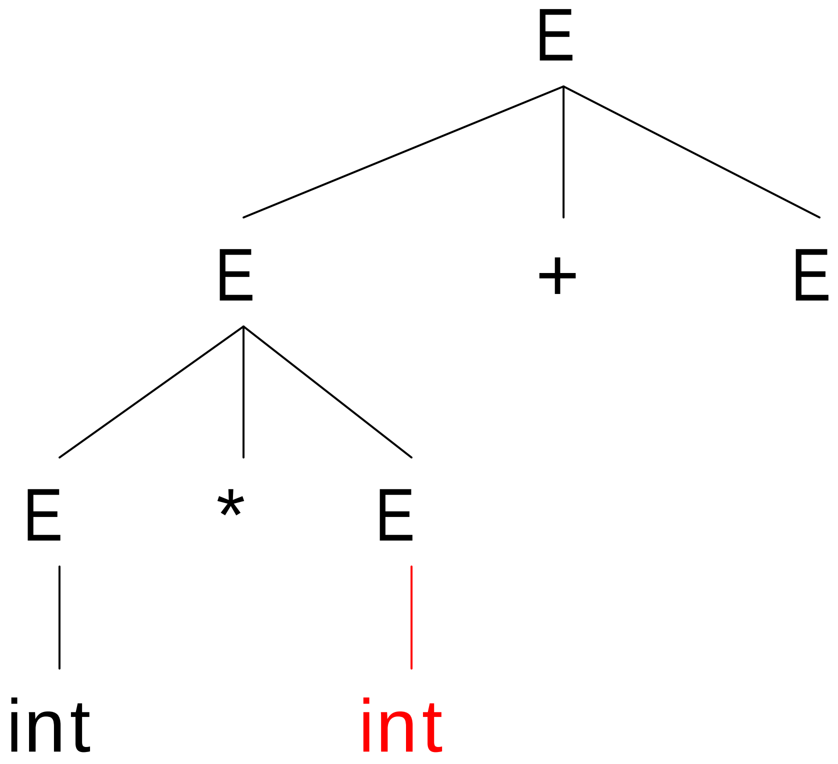
E

→ E + E

→ E * E + E

→ int * E + E

→ int * int + E



Derivation in Detail (6)

E → E+E | E*E | (E) | int

E

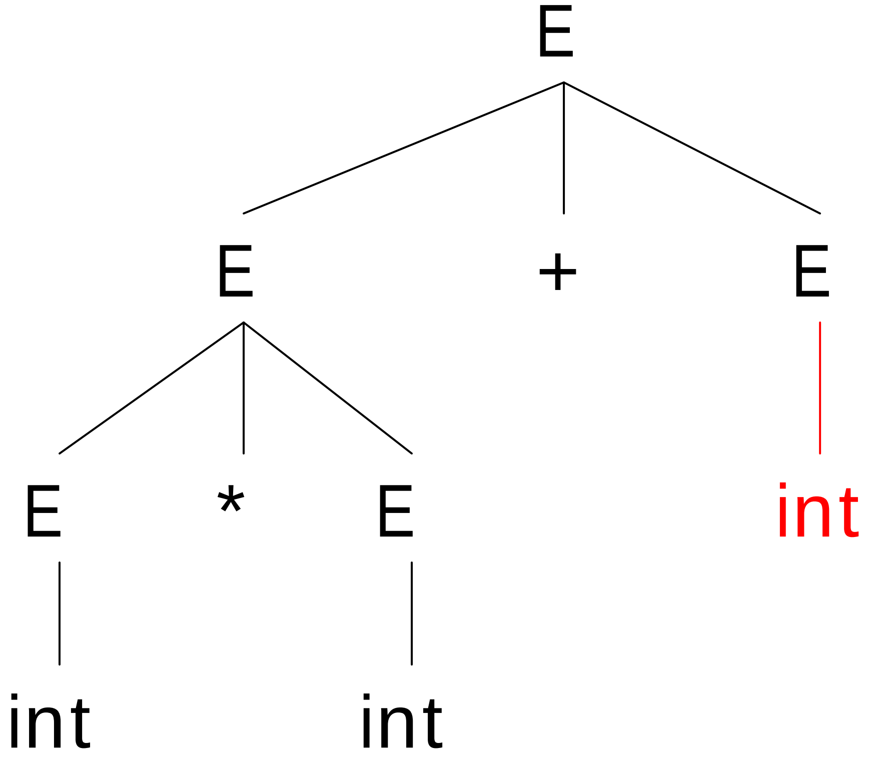
→ E + E

→ E * E + E

→ int * E + E

→ int * int + E

→ int * int + int



Notes on Derivations

- A parse tree has:
  - terminals at the leaves;
  - non-terminals at the interior nodes.
- An in-order traversal of the leaves is the original input.
- The parse tree shows the association of operations; the input string does not !

Left-most and Right-most Derivations

- What was shown before was a *left-most derivation*.
  - At each step, we replaced the left-most non-terminal.
- There is an equivalent notion of a *right-most derivation*.
  - Shown on the right.

E → E+E | E*E | (E) | int

$$
\begin{aligned}
& E \\
\rightarrow\ & E + E \\
\rightarrow\ & E + \text{int} \\
\rightarrow\ & E * E + \text{int} \\
\rightarrow\ & E * \text{int} + \text{int} \\
\rightarrow\ & \text{int} * \text{int} + \text{int}
\end{aligned}
$$

Right-most Derivation in Detail (1)

E → E+E | E*E | (E) | int

E

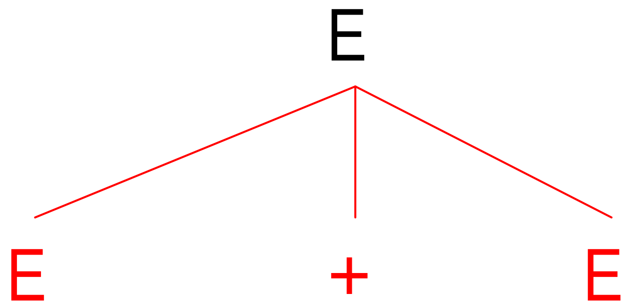
E

Right-most Derivation in Detail (2)

E → E+E | E*E | (E) | int

E

→ E + E



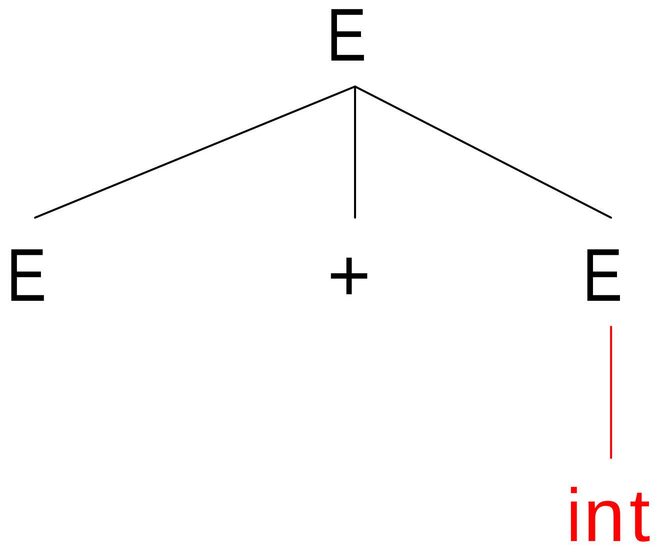
Right-most Derivation in Detail (3)

E → E+E | E*E | (E) | int

E

→ E + E

→ E + int



Right-most Derivation in Detail (4)

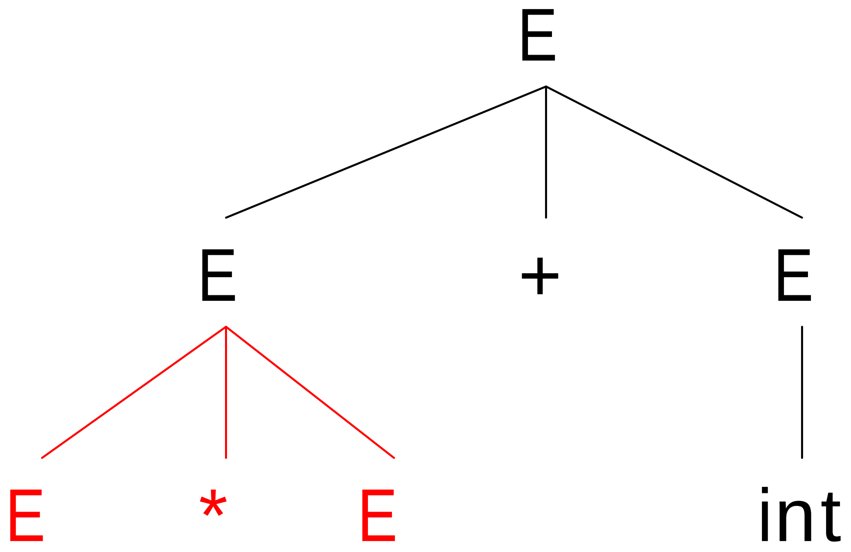
E → E+E | E*E | (E) | int

E

→ E + E

→ E + int

→ E * E + int



Right-most Derivation in Detail (5)

E → E+E | E*E | (E) | int

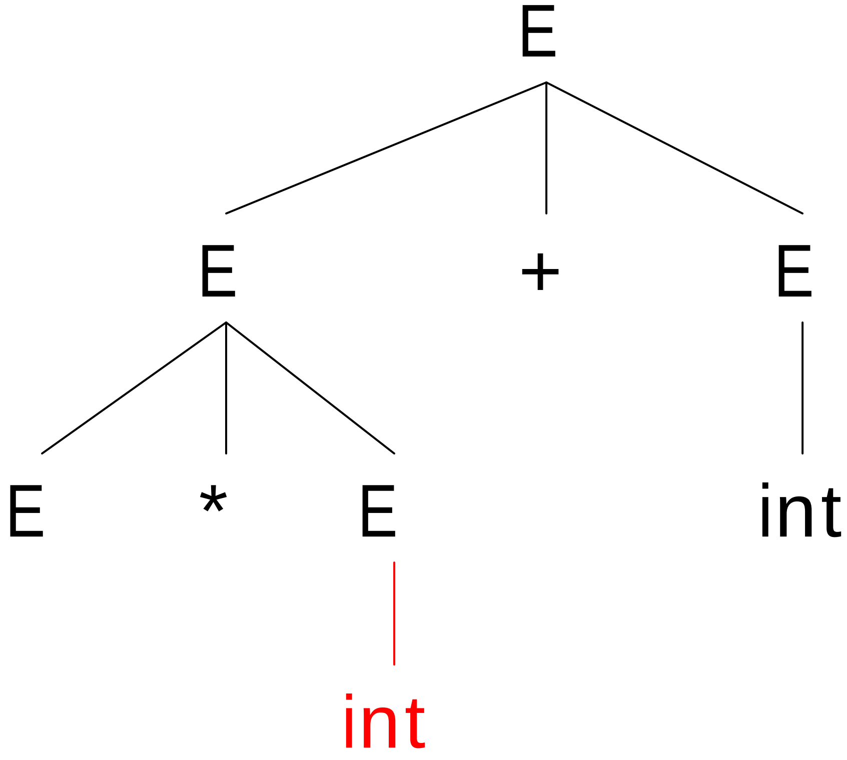
E

→ E + E

→ E + int

→ E * E + int

→ E * int + int



Right-most Derivation in Detail (6)

E → E+E | E*E | (E) | int

E

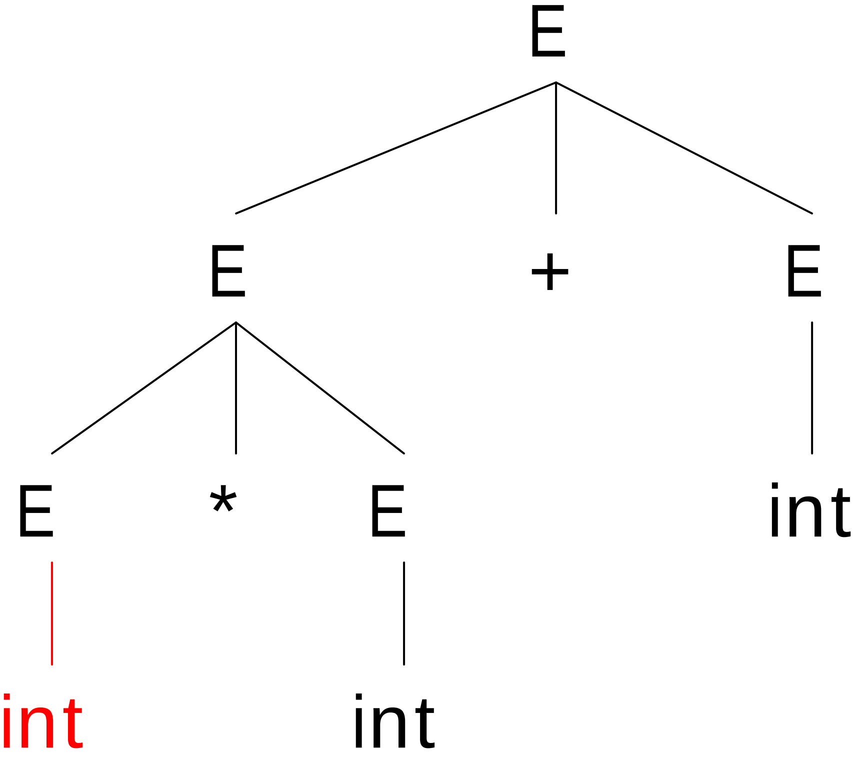
→ E + E

→ E + int

→ E * E + int

→ E * int + int

→ int * int + int



Derivations and Parse Trees

- Note that:
  - Right-most and left-most derivations have the same parse tree.
  - For each parse tree, there is a right-most and a left-most derivation.

- The difference *is just in the order* in which branches are added.

Summary of Derivations

- We are not just interested in whether

  $$s \in L(G)$$

  - We also need a parse tree for s.

- A derivation defines a parse tree.
  - But one parse tree may have many derivations.

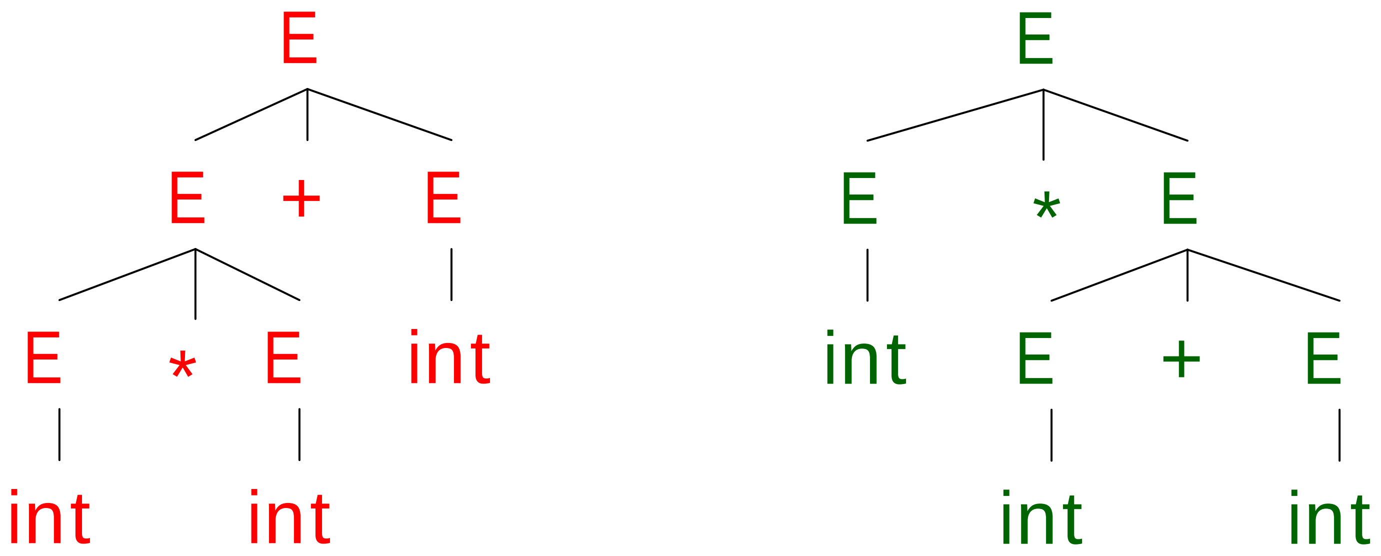
- Left-most and right-most derivations are important in parser implementation.

Ambiguity

- Grammar:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$$

- The string int * int + int has two parse trees



Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string.
  - Equivalently, if there is more than one right-most or left-most derivation for some string.
- Ambiguity is <u>bad</u> in programming languages.
  - Leaves meaning of some programs ill-defined.
- Ambiguity is <u>common</u> in programming languages:
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity.

- Most direct method is to rewrite the grammar unambiguously:

$$E \to T + E \mid T$$
$$T \to \text{int} * T \mid \text{int} \mid (E)$$

- This grammar enforces precedence of * over +.

Ambiguity: The Dangling Else

- Consider the following grammar:

$$
\begin{aligned}
S \rightarrow\ & \text{if } C \text{ then } S \\
\mid\ & \text{if } C \text{ then } S \text{ else } S \\
\mid\ & \text{OTHER}
\end{aligned}
$$

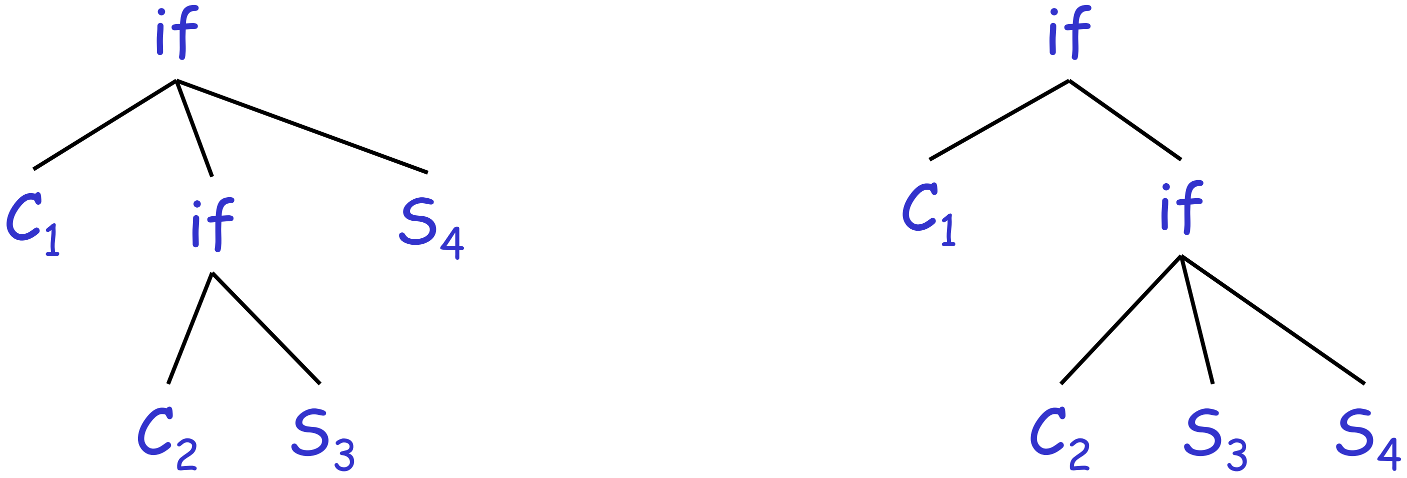
- This grammar is also ambiguous.

The Dangling Else: Example

- The expression

  if C_1 then if C_2 then S_3 else S_4

  has two parse trees.



- Typically we want the second form.

The Dangling Else: A Fix

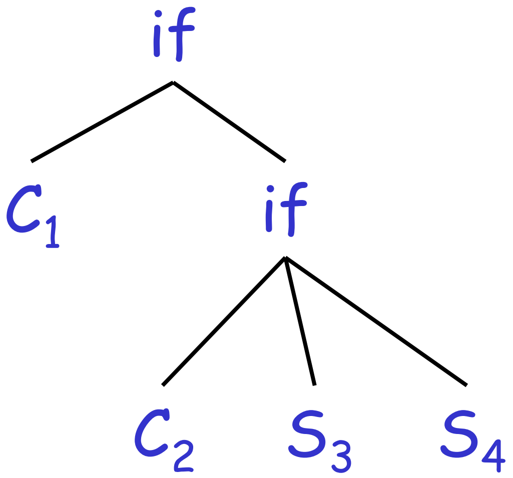
- else should match the closest unmatched then
- We can describe this in the grammar:

```
S → MIF                     /* all then are matched */
  | UIF                     /* some then are unmatched */
MIF → if C then MIF else MIF
    | OTHER
UIF → if C then S
    | if C then MIF else UIF
```

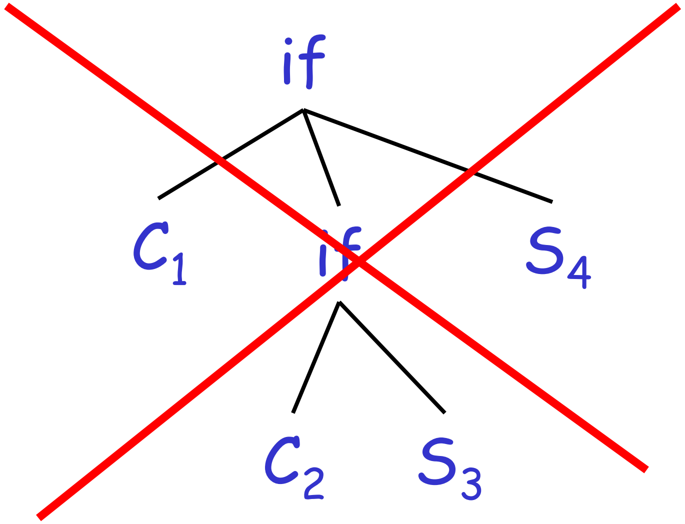
- Describes the same set of strings.

The Dangling Else: Example Revisited

The expression if C_1 then if C_2 then S_3 else S_4



A valid parse tree (for a UIF).



Not valid because the then expression is not a MIF.

Ambiguity

- No general techniques for handling ambiguity.
- In general, impossible to convert automatically an ambiguous grammar to an unambiguous one.
- Used with care, ambiguity can simplify the grammar.
  - Sometimes allows more natural definitions.
  - However, we need disambiguation mechanisms.

Precedence and Associativity Declarations

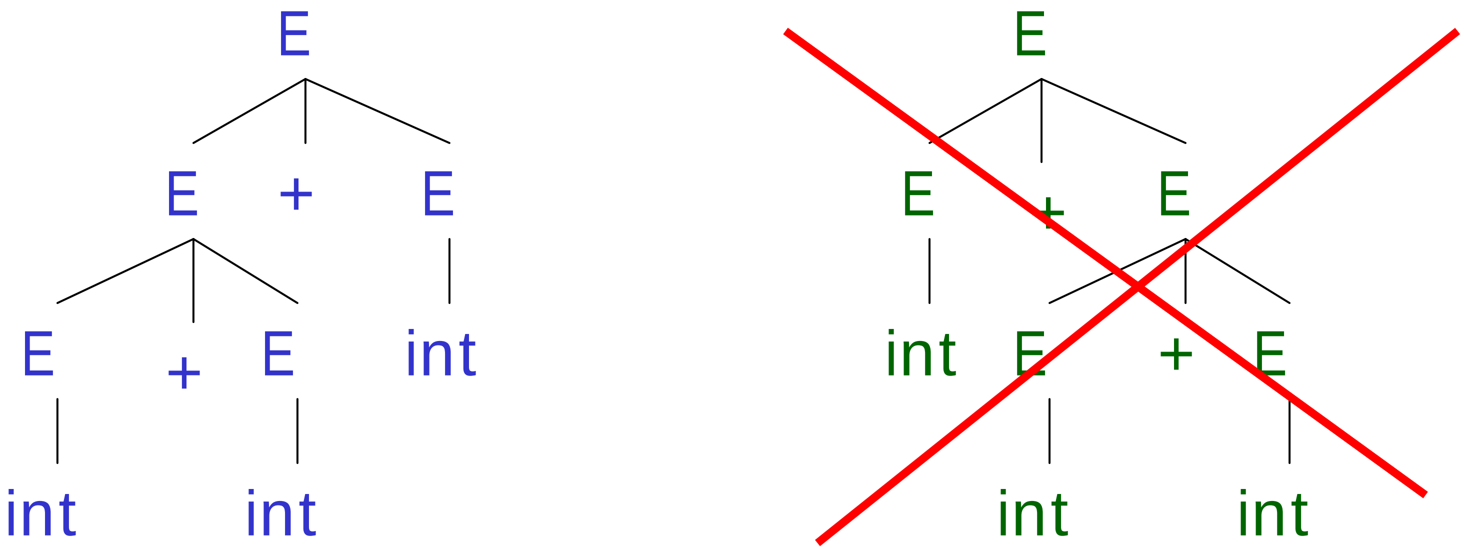
- Instead of rewriting the grammar:
  - use the more natural (ambiguous) grammar,
  - along with disambiguating declarations.

- Most tools allow <u>precedence and associativity declarations</u> to disambiguate grammars.

- Examples …

Associativity Declarations

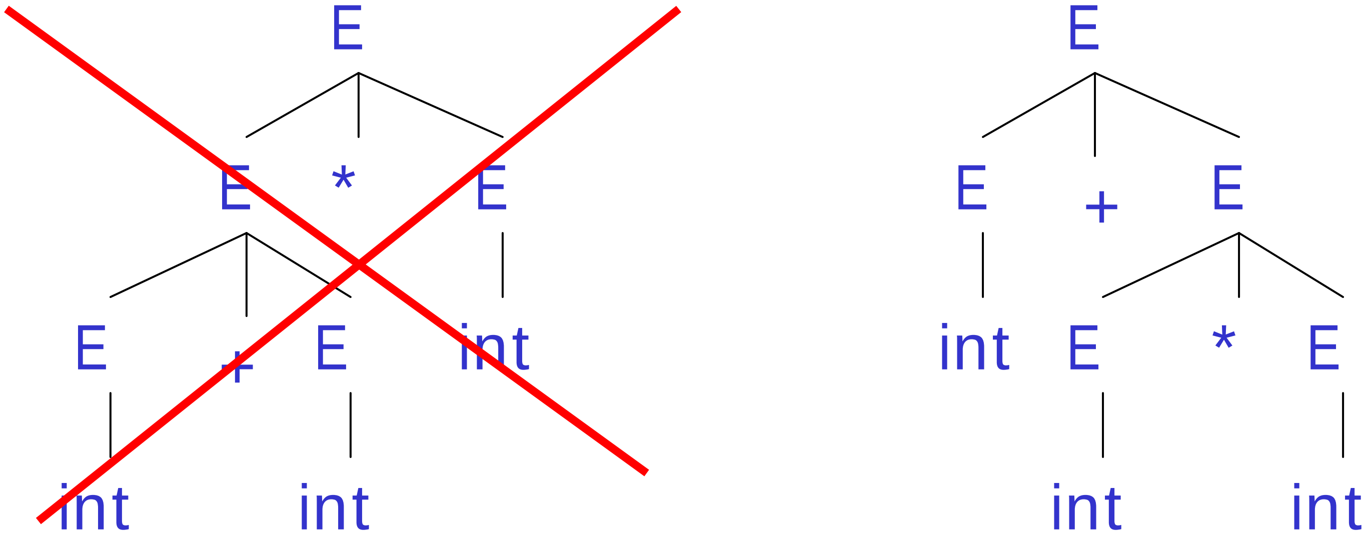
- Consider the grammar $E \to E + E \mid \text{int}$
- Ambiguous: two parse trees of int + int + int



- Left associativity declaration: `%left +`

Precedence Declarations

- Consider the grammar E → E + E | E * E | int and the string int + int * int



- Precedence declarations: **%left +**

  %left *

Error Handling

- Purpose of the compiler is:
  - To detect non-valid programs.
  - To translate the valid ones.
- Many kinds of possible errors (e.g. in C)

Error kind	Example	Detected by …
Lexical	… $ …	Lexer
Syntax	… x *% …	Parser
Semantic	… int x; y = x(3); …	Type checker
Correctness	your favorite program	Tester/User

Syntax Error Handling

- Error handler should:
  - Report errors accurately and clearly.
  - Recover from an error quickly.
  - Not slow down compilation of valid code.

- Good error handling is not easy to achieve.

Approaches to Syntax Error Recovery

- From simple to complex:
  - Panic mode.
  - Error productions.
  - Automatic local or global correction.
- Not all are supported by all parser generators.

Error Recovery: Panic Mode

- Simplest, most popular method.

- When an error is detected:
  - Discard tokens until one with a clear role is found.
  - Continue from there.

- Such tokens are called synchronizing tokens.
  - Typically the statement or expression terminators.

Syntax Error Recovery: Panic Mode (Cont.)

- Consider the erroneous expression:

  (1 + + 2) + 3

- Panic-mode recovery:
  - Skip ahead to next integer and then continue.

- (ML)-Yacc: use the special terminal **error** to describe how much input to skip:

  E → int | E + E | (E) | **error** int | (**error**)

Syntax Error Recovery: Error Productions

- Idea: specify some recovery rules in the grammar based on known common mistakes.
- Essentially promotes common errors to alternative syntax.
- Example:
  - Write 5 x instead of 5 * x
  - Add the production E → … | E E
- Disadvantage:
  - Complicates the grammar.

Syntax Error Recovery: Past and Present

- (Distant) Past
  - Slow recompilation cycle (even once a day!).
  - Goal: find as many errors in one cycle as possible.
  - Researchers could not let go of the topic.

- Present
  - Quick recompilation cycle.
  - Users tend to correct one error/cycle.
  - Complex error recovery is needed less.
  - Panic-mode seems enough.

Abstract Syntax Trees
&
Top-Down Parsing

Review of Parsing

- Given a language $L(G)$, a parser consumes a sequence of tokens s and produces a parse tree
- Issues:
  - How do we recognize that $s \in L(G)$?
  - A parse tree of s describes how $s \in L(G)$
  - Ambiguity: more than one parse tree (possible interpretation) for some string s
  - Error: no parse tree for some string s
  - How do we construct the parse tree?

Abstract Syntax Trees

- So far, a parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program
- Abstract syntax trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

Abstract Syntax Trees (Cont.)

- Consider the grammar

  E → int | (E) | E + E

- And the string

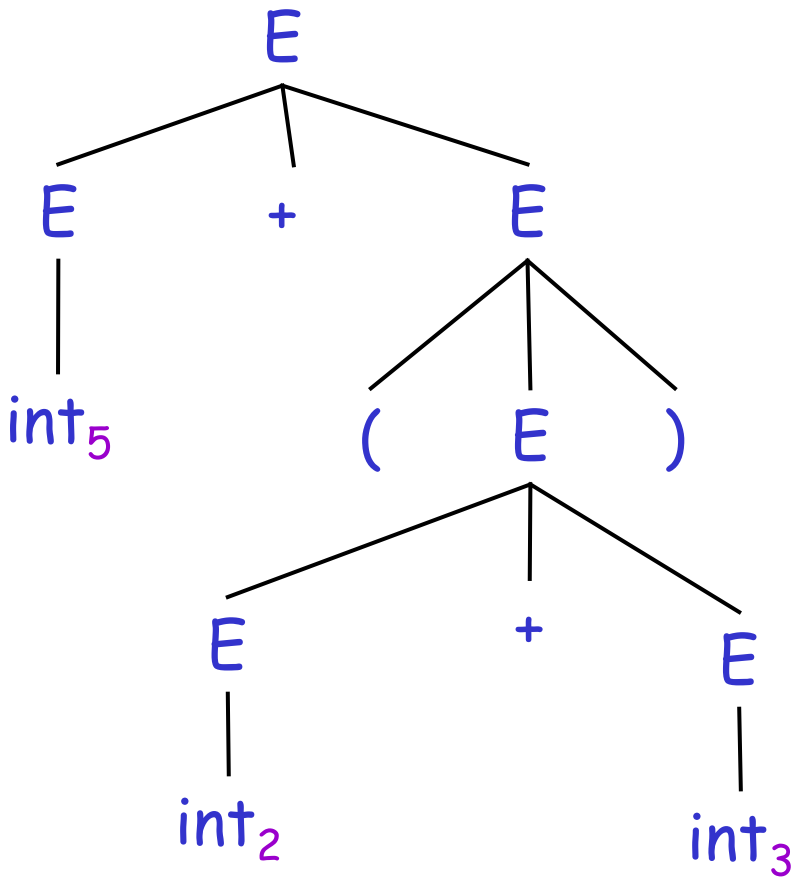
  5 + (2 + 3)

- After lexical analysis (a list of tokens)

  int_5 '+' '(' int_2 '+' int_3 ')'

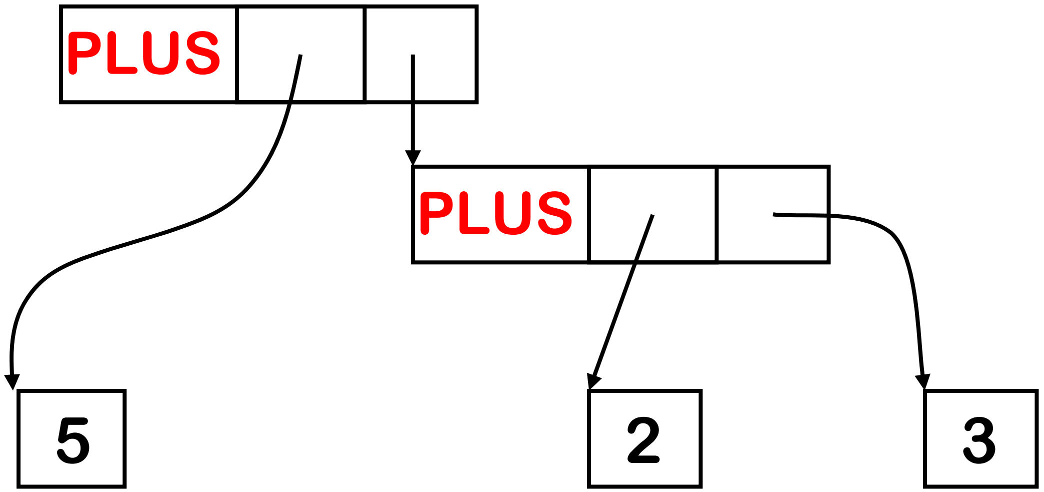
- During parsing we build a parse tree …

Example of Parse Tree



- Traces the operation of the parser
- Captures the nesting structure
- But too much information
  - Parentheses
  - Single-successor nodes

Example of Abstract Syntax Tree



- Also captures the nesting structure
- But <u>abstracts</u> from the concrete syntax

  ↦ more compact and easier to use
- An important data structure in a compiler

Semantic Actions

- This is what we will use to construct ASTs

- Each grammar symbol may have attributes
  - An attribute is a property of a programming language construct
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer
- Each production may have an action
  - Written as: $X \rightarrow Y_1 \ldots Y_n$ { action }
  - That can refer to or compute symbol attributes

Semantic Actions: An Example

- Consider the grammar

$$E \to \text{int} \mid E + E \mid (E)$$

- For each symbol X define an attribute X.val
  - For terminals, val is the associated lexeme
  - For non-terminals, val is the expression's value (which is computed from values of subexpressions)
- We annotate the grammar with actions:

$$\begin{array}{lll} E \to & \text{int} & \{ E.val = \text{int}.val \} \\ \mid & E_1 + E_2 & \{ E.val = E_1.val + E_2.val \} \\ \mid & (E_1) & \{ E.val = E_1.val \} \end{array}$$

Semantic Actions: An Example (Cont.)

- String: **5 + (2 + 3)**
- Tokens: int_5 '+' '(' int_2 '+' int_3 ')'

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.\text{val} = E_1.\text{val} + E_2.\text{val}$
$E_1 \rightarrow \text{int}_5$	$E_1.\text{val} = \text{int}_5.\text{val} = 5$
$E_2 \rightarrow (E_3)$	$E_2.\text{val} = E_3.\text{val}$
$E_3 \rightarrow E_4 + E_5$	$E_3.\text{val} = E_4.\text{val} + E_5.\text{val}$
$E_4 \rightarrow \text{int}_2$	$E_4.\text{val} = \text{int}_2.\text{val} = 2$
$E_5 \rightarrow \text{int}_3$	$E_5.\text{val} = \text{int}_3.\text{val} = 3$

Semantic Actions: Dependencies

Semantic actions specify a system of equations

- Order of executing the actions is not specified

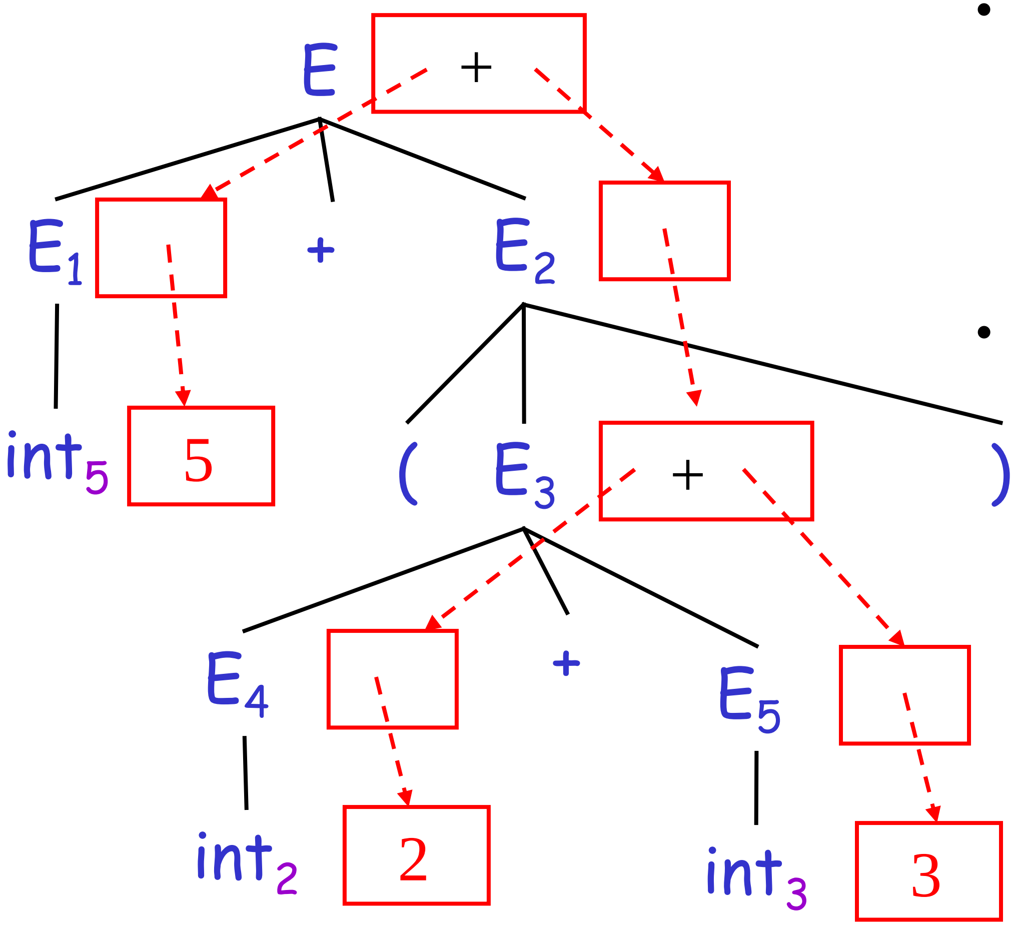
- Example:

  E_3.val = E_4.val + E_5.val

  - Must compute E_4.val and E_5.val before E_3.val
  - We say that E_3.val *depends on* E_4.val and E_5.val

- The parser must find the order of evaluation

Dependency Graph



- Each node labeled with a non-terminal E has one slot for its val attribute
- Note the dependencies

Evaluating Attributes

- An attribute must be computed after all its successors in the dependency graph have been computed
  - In the previous example attributes can be computed bottom-up
- Such an order exists when there are no cycles
  - Cyclically defined attributes are not legal

Semantic Actions: Notes (Cont.)

- Synthesized attributes
  - Calculated from attributes of descendents in the parse tree
  - E.val is a synthesized attribute
  - Can always be calculated in a bottom-up order

- Grammars with only synthesized attributes are called S-attributed grammars
  - Most frequent kinds of grammars

Inherited Attributes

- Another kind of attributes
- Calculated from attributes of the parent node(s) and/or siblings in the parse tree

- Example: a line calculator

A Line Calculator

- Each line contains an expression

  $E \rightarrow \text{int} \mid E + E$

- Each line is terminated with the = sign

  $L \rightarrow E = \mid + E =$

- In the second form, the value of evaluation of the previous line is used as starting value
- A program is a sequence of lines

  $P \rightarrow \varepsilon \mid P\,L$

Attributes for the Line Calculator

- Each E has a synthesized attribute val
  - Calculated as before
- Each L has a synthesized attribute val

```
L → E =        { L.val = E.val }
    | + E =    { L.val = E.val + L.prev }
```

- We need the value of the previous line
- We use an inherited attribute L.prev

Attributes for the Line Calculator (Cont.)

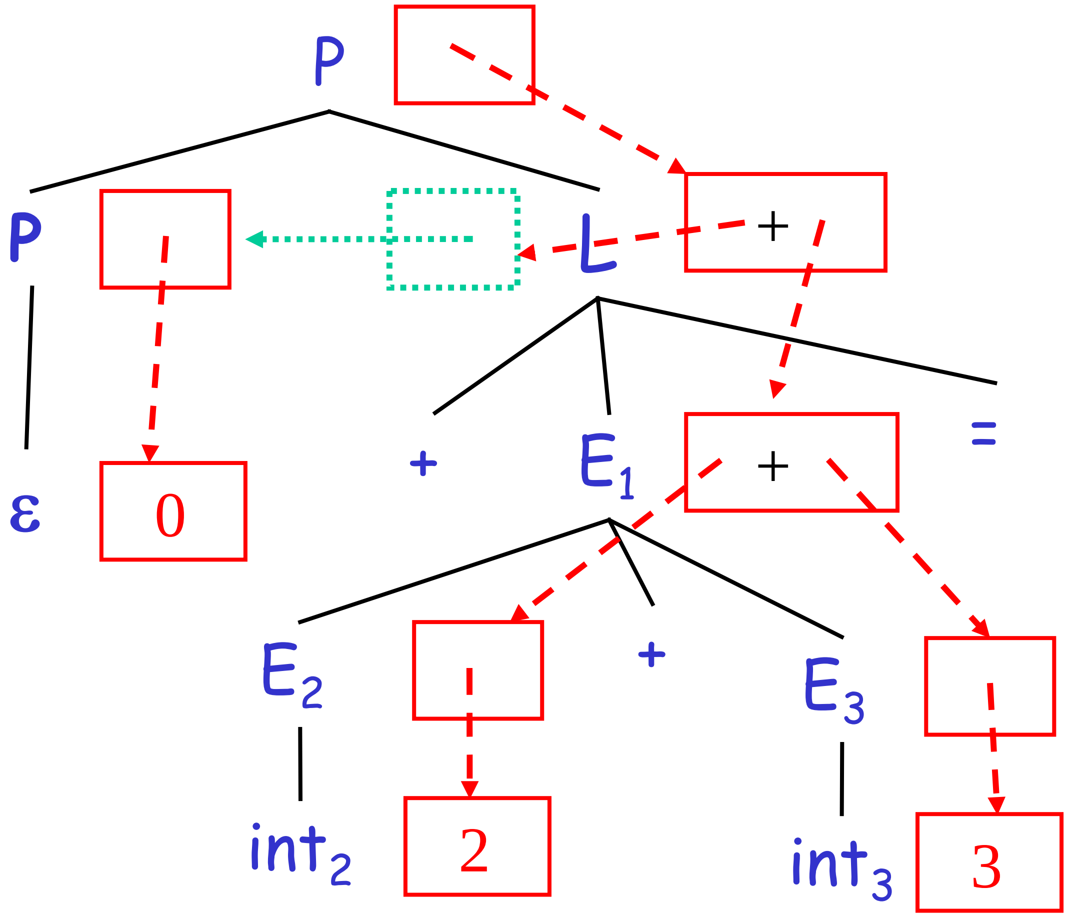
- Each P has a synthesized attribute val
  - The value of its last line

$$\begin{array}{lll} P \to \varepsilon & & \{\ P.val = 0\ \} \\ \quad \mid\ P_1\ L & & \{\ P.val = L.val; \\ & & \quad L.prev = P_1.val\ \} \end{array}$$

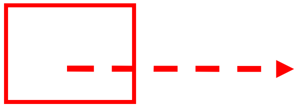
- Each L has an inherited attribute prev
  - L.prev is inherited from sibling P_1.val

- Example …

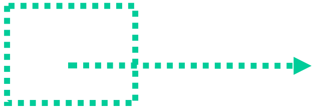
Example of Inherited Attributes



- val synthesized



- prev inherited



- All can be computed in depth-first order

Semantic Actions: Notes (Cont.)

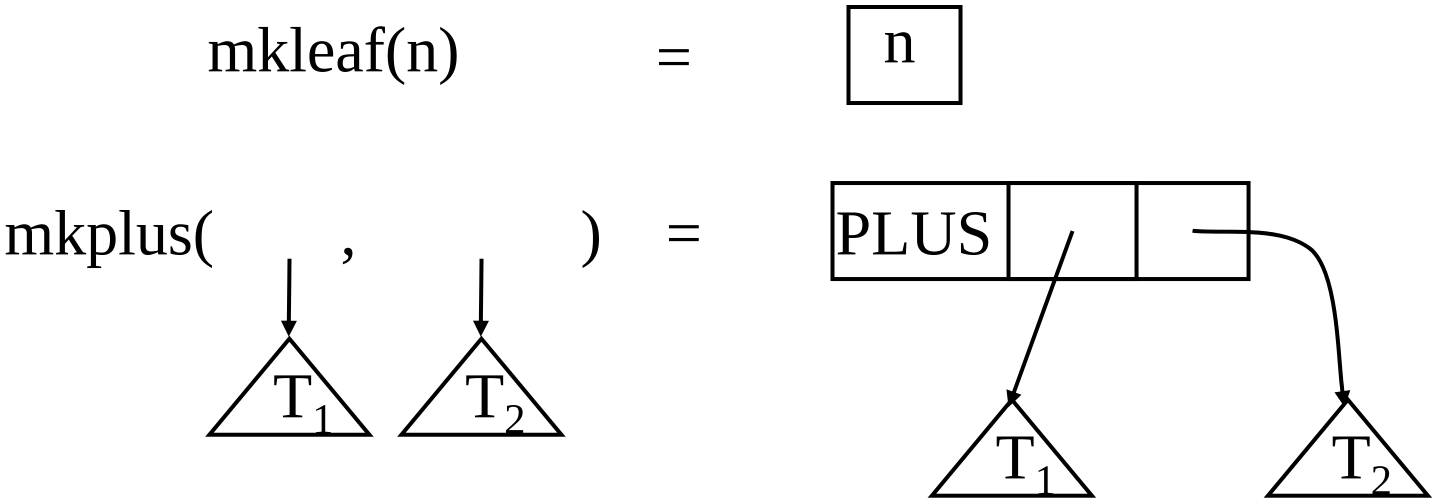
- Semantic actions can be used to build ASTs

- And many other things as well
  - Also used for type checking, code generation, …

- Process is called syntax-directed translation
  - Substantial generalization over CFGs

Constructing an AST

- We first define the AST data type
- Consider an abstract tree type with two constructors:



mkleaf(n) = n

mkplus(T_1 , T_2) = PLUS T_1 T_2

Constructing a Parse Tree

- We define a synthesized attribute ast
  - Values of ast values are ASTs
  - We assume that int.lexval is the value of the integer lexeme
  - Computed using semantic actions

$E \rightarrow$ int   { E.ast = mkleaf(int.lexval) }

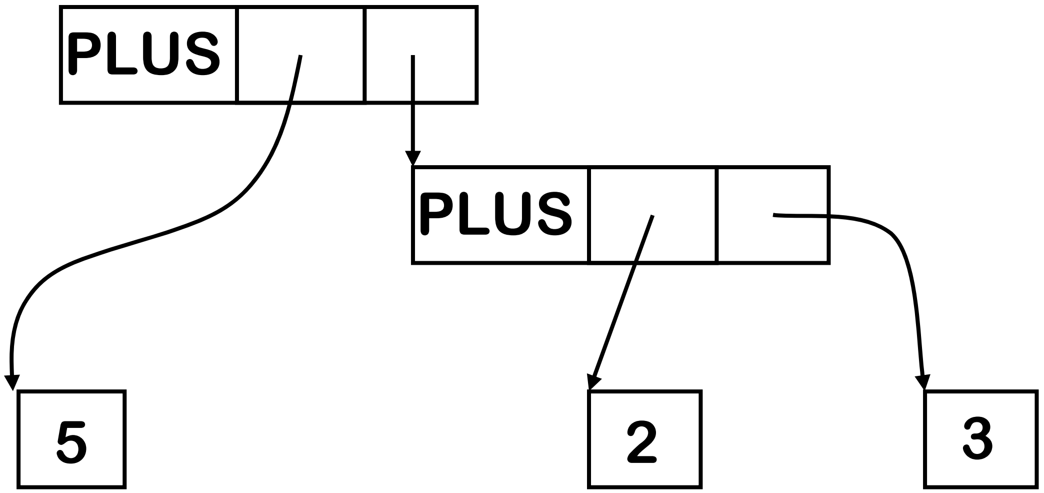
    | $E_1 + E_2$ { E.ast = mkplus(E_1.ast, E_2.ast) }

    | (E_1)   { E.ast = E_1.ast }

Parse Tree Example

- Consider the string int_5 '+' '(' int_2 '+' int_3 ')'
- A bottom-up evaluation of the ast attribute:

  E.ast = mkplus(mkleaf(5),
  mkplus(mkleaf(2), mkleaf(3))



Review of Abstract Syntax Trees

- We can specify language syntax using CFG.
- The parser answers whether $s \in L(G)$
- … and builds a parse tree
- … which it converts to an AST
- … and passes on to the rest of the compiler.

- In the next "parsing" lectures:
  - How do we answer $s \in L(G)$ and build a parse tree?
- After that: from AST to assembly language.

Second-Half of Lecture: Outline

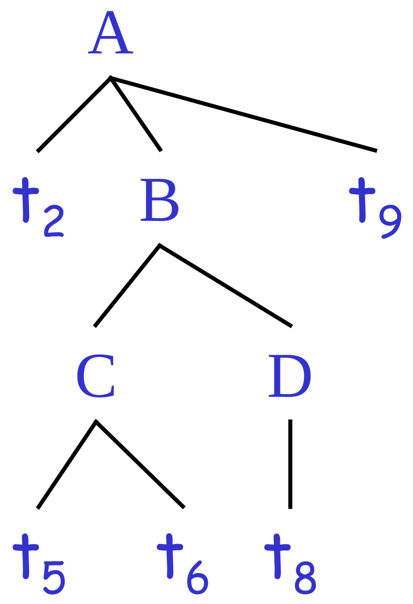
- Implementation of parsers
- Two approaches
  - Top-down
  - Bottom-up
- These slides: Top-Down
  - Easier to understand and program manually
- Next lectures: Bottom-Up
  - More powerful and used by most parser generators

Introduction to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:

  t_2 t_5 t_6 t_8 t_9

- The parse tree is constructed
  - From the top
  - From left to right



Recursive Descent Parsing: Example

- Consider the grammar

  $E \rightarrow T + E \mid T$

  $T \rightarrow (E) \mid \text{int} \mid \text{int} * T$

- Token stream is: $\text{int}_5 * \text{int}_2$
- Start with top-level non-terminal E

- Try the rules for E in order

Recursive Descent Parsing: Example

Token stream: int_5 * int_2

- Try $E_0 \to T_1 + E_2$
- Then try a rule for $T_1 \to (E_3)$
  - But (does not match input token int_5 ; we backtrack.
- Try $T_1 \to int$. Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \to int * T_2$
  - This will match and will consume the two tokens.
    - Try $T_2 \to int$ (matches) but + after T_1 will be unmatched.
    - Try $T_2 \to int * T_3$ but * does not match with end-of-input.
- We have exhausted all the choices for T_1
  - Backtrack to choice for E_0

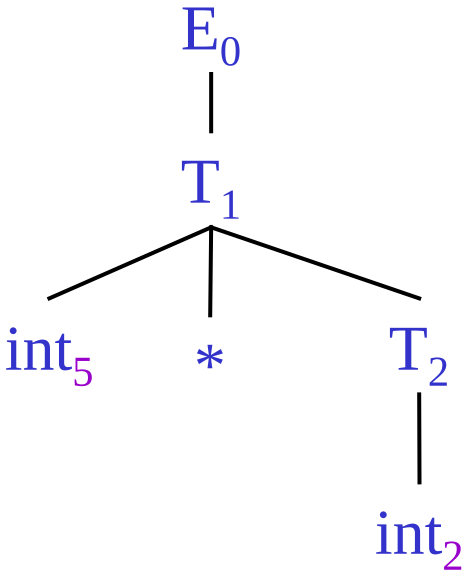
$E \to T + E \mid T$

$T \to (E) \mid int \mid int * T$

Recursive Descent Parsing: Example

Token stream: int_5 * int_2

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \rightarrow int_5 * T_2$ and $T_2 \rightarrow int_2$
  - With the following parse tree



$E \rightarrow T + E \mid T$
$T \rightarrow (E) \mid int \mid int * T$

Recursive Descent Parsing: Notes

- Easy to implement by hand

- Somewhat inefficient (due to backtracking)

- But does not always work …

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S\ a$

```
bool S1() { return S() && term(a);  }
bool S() { return S1(); }
```

- S() will get into an infinite loop

- We call a grammar <u>left-recursive</u> if it has a non-terminal S

$$S \rightarrow^{+} S\alpha \quad \text{for some } \alpha$$

- Recursive descent does not work in such cases – it goes into an infinite loop.

Elimination of Left Recursion

- Consider the left-recursive grammar:

$$S \rightarrow S\ \alpha \mid \beta$$

- Generates all strings starting with a β and followed by any number of α's.

- The grammar can be rewritten using right recursion:

$$S \rightarrow \beta\ S'$$

$$S' \rightarrow \alpha\ S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

$$S \rightarrow S\ \alpha_1 \mid \dots \mid S\ \alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1, \dots, \beta_m$ and continue with several instances of $\alpha_1, \dots, \alpha_n$
- Rewrite as

$$S \rightarrow \beta_1\ S' \mid \dots \mid \beta_m\ S'$$
$$S' \rightarrow \alpha_1\ S' \mid \dots \mid \alpha_n\ S' \mid \varepsilon$$

General Left Recursion

- The grammar

$$S \rightarrow A\ \alpha \mid \delta$$

$$A \rightarrow S\ \beta$$

is also left-recursive because

$$S \rightarrow^{+} S\ \beta\ \alpha$$

- This left-recursion can also be eliminated

[See a Compilers book for a general algorithm]

Summary of Recursive Descent

- Simple and general parsing strategy.
  - Left-recursion must be eliminated first
  - … but that can be done automatically.
- Unpopular because of backtracking.
  - Thought to be too inefficient.

- In practice, backtracking is eliminated by restricting the grammar.

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
- In practice, LL(1) is used

LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of productions
- LL(1) means that for each non-terminal and token there is only one production that could lead to success
- Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

Predictive Parsing and Left Factoring

- Recall the grammar for arithmetic expressions

  $$E \to T + E \mid T$$
  $$T \to (E) \mid \text{int} \mid \text{int} * T$$

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict

- A grammar must be left-factored before it is used for predictive parsing

Left-Factoring Example

- Recall the grammar

$$E \rightarrow T + E \mid T$$
$$T \rightarrow (E) \mid \text{int} \mid \text{int} * T$$

- Factor out common prefixes of productions:

$$E \rightarrow T\, X$$
$$X \rightarrow + E \mid \varepsilon$$
$$T \rightarrow (E) \mid \text{int}\, Y$$
$$Y \rightarrow \varepsilon \mid * T$$

- This grammar is equivalent to the original one.

LL(1) Parsing Table Example

- Left-factored grammar

$E \to T\ X$ $\qquad\qquad$ $X \to +\ E \mid \varepsilon$

$T \to (\ E\ ) \mid \text{int}\ Y$ $\qquad\qquad$ $Y \to *\ T \mid \varepsilon$

- The LL(1) parsing table ($ is the end marker)

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T X$ "
  - This production can generate an int in the first place
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only in a derivation in which $Y \rightarrow \varepsilon$

LL(1) Parsing Tables: Errors

- Blank entries indicate error situations
  - Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal X
  - We look at the next token a
  - And choose the production shown at [X,a]
- We use a stack to keep track of pending non-terminals.
- We reject when we encounter an error state.
- We accept when we encounter end-of-input.

LL(1) Parsing Algorithm

```
initialize stack ← <S $> and next
repeat
  case stack of
    <X, rest> : if T[X,*next] == $Y_1…Y_n$
                then stack ← <$Y_1…Y_n$ rest>;
                else error();
    <t, rest> : if t == *next++
                then stack ← <rest>;
                else error();
until stack == <>
```

LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

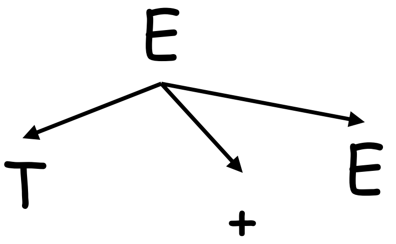
Introduction to Bottom-Up Parsing

Outline

- Review LL parsing
- Shift-reduce parsing
- The LR parsing algorithm
- Using LR parsing tables

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves.
  - Always expand the leftmost non-terminal.

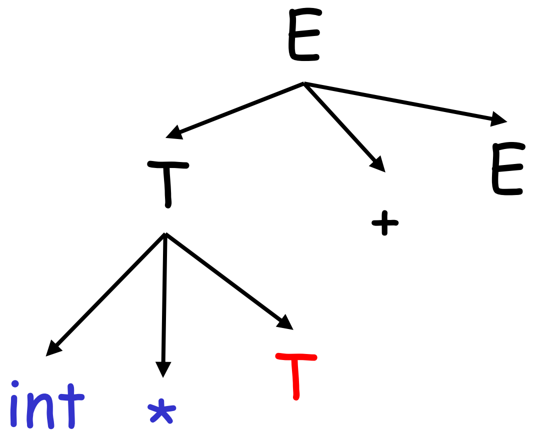


int * int + int

E → T + E | T
T → (E) | int | int * T

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves.
  - Always expand the leftmost non-terminal.



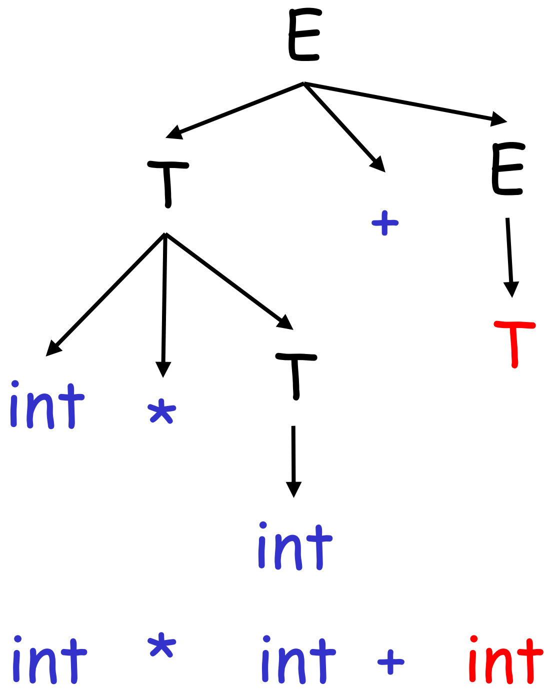
- The leaves at any point form a string βAγ, where
  - β contains only terminals.
  - The input string is βbδ.
  - The prefix β matches.
  - The next token is b.

int * int + int

E → T + E | T
T → (E) | int | int * T

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves.
  - Always expand the leftmost non-terminal.

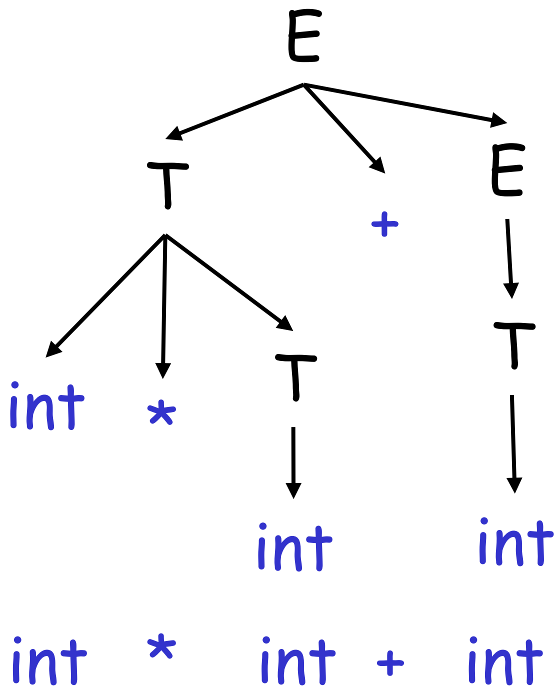


- The leaves at any point form a string $\beta A\gamma$, where
  - β contains only terminals.
  - The input string is $\beta b\delta$.
  - The prefix β matches.
  - The next token is b.

E → T + E | T
T → (E) | int | int * T

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$, where
  - β contains only terminals.
  - The input string is $\beta b \delta$.
  - The prefix β matches.
  - The next token is b.

E → T + E | T
T → (E) | int | int * T

Predictive Parsing: Review

- A predictive parser is described by a table.
  - For each non-terminal A and for each token b we specify a production $A \to \alpha$.
  - When trying to expand A we use $A \to \alpha$ if b is the token that follows next.

- Once we have the table:
  - The parsing algorithm is simple and fast.
  - No backtracking is necessary.

Constructing Predictive Parsing Tables

Consider the state $S \rightarrow^* \beta A\gamma$

- With b the next token
- Trying to match $\beta b\delta$

There are two possibilities:

1. Token b belongs to an expansion of A
   - Any $A \rightarrow \alpha$ can be used if b can start a string derived from α
   - We say that $b \in \text{First}(\alpha)$

Or…

Constructing Predictive Parsing Tables (Cont.)

2. Token b does not belong to an expansion of *A*
   - The expansion of *A* is empty and b belongs to an expansion of γ
   - Means that b can appear after *A* in a derivation of the form $S \rightarrow^* \beta A b \omega$
   - We say that $b \in$ Follow(*A*) in this case

   - What productions can we use in this case?
     - Any $A \rightarrow \alpha$ can be used if α can expand to ε
     - We say that $\varepsilon \in$ First(*A*) in this case

Computing First Sets

Definition

$$\text{First}(X) = \{\, b \mid X \rightarrow^{*} b\alpha \,\} \cup \{\, \varepsilon \mid X \rightarrow^{*} \varepsilon \,\}$$

Algorithm sketch

1. $\text{First}(b) = \{\, b \,\}$
2. $\varepsilon \in \text{First}(X)$ if $X \rightarrow \varepsilon$ is a production
3. $\varepsilon \in \text{First}(X)$ if $X \rightarrow A_1 \ldots A_n$
   and $\varepsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$
4. $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \rightarrow A_1 \ldots A_n\ \alpha$
   and $\varepsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$

First Sets: Example

- Recall the grammar

  $E \to T\,X$ $\qquad$ $X \to +\,E \mid \varepsilon$

  $T \to (\,E\,) \mid \text{int}\ Y$ $\qquad$ $Y \to *\,T \mid \varepsilon$

- First sets

  First(( ) = { (}

  First( )) = {) }

  First(int) = { int }

  First(+) = { + }

  First(*) = { * }

  First(T) = { int, (}

  First(E) = { int, (}

  First(X) = { +, ε }

  First(Y) = { *, ε }

Computing Follow Sets

- Definition

$$\mathrm{Follow}(X) = \{ b \mid S \to^* \beta\, X\, b\, \delta \}$$

- Intuition
  - If $X \to A\ B$ then $\mathrm{First}(B) \subseteq \mathrm{Follow}(A)$ and $\mathrm{Follow}(X) \subseteq \mathrm{Follow}(B)$
  - Also if $B \to^* \varepsilon$ then $\mathrm{Follow}(X) \subseteq \mathrm{Follow}(A)$
  - If S is the start symbol then $\$ \in \mathrm{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets: Example

First(T) = { int, (}
First(E) = { int, (}
First(X) = { +, ε }
First(Y) = { *, ε }

- Recall the grammar

  E → T X   X → + E | ε
  T → (E) | int Y   Y → * T | ε

- Follow sets

  Follow(+) = { int, (}
  Follow(( ) = { int, (}
  Follow(X) = {), $ }
  Follow( )) = { +,) , $ }
  Follow(int) = { *, +,) , $ }

  Follow(*) = { int, (}
  Follow(E) = {), $ }
  Follow(T) = { +,) , $ }
  Follow(Y) = { +,) , $ }

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G.

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $b \in \text{First}(\alpha)$ do
    $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $b \in \text{Follow}(A)$ do
    $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    $T[A, \$] = \alpha$

Constructing LL(1) Tables: Example

- Recall the grammar

  $E \to T\,X \qquad\qquad X \to +\,E \mid \varepsilon$

  $T \to (\,E\,) \mid \text{int}\;Y \qquad Y \to *\,T \mid \varepsilon$

- Where in the line of Y do we put $Y \to * \; T$?
  - In the lines of First(*T) = { * }

- Where in the line of Y do we put $Y \to \varepsilon$?
  - In the lines of Follow(Y) = { $, +,) }

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1). This happens:
  - if G is ambiguous;
  - if G is left recursive;
  - if G is not left-factored;
  - and in other cases as well.

- For some grammars there is a simple parsing strategy: *Predictive parsing.*
- Most programming language grammars are not LL(1).
- Thus, we need more powerful parsing strategies.

Bottom Up Parsing

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing – and just as efficient.
  - Builds on ideas in top-down parsing.
  - Preferred method in practice.

- Also called LR parsing
  - L means that tokens are read left-to-right.
  - R means that it constructs a rightmost derivation.

An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars.

- Consider the following grammar:

  E → E + (E) | int

  - Why is this not LL(1)?

- Consider the string: int + (int) + (int)

The Idea

- LR parsing *reduces* a string to the start symbol by inverting productions.

```
str w input string of terminals
repeat
    – Identify $\beta$ in str such that $A \to \beta$ is a production
      (i.e., str = $\alpha\ \beta\ \gamma$)
    – Replace $\beta$ by $A$ in str (i.e., str w = $\alpha\ A\ \gamma$)
until str = S  (the start symbol)
  OR all possibilities are exhausted
```

A Bottom-up Parse in Detail (1)

E → E + (E) | int

int + (int) + (int)

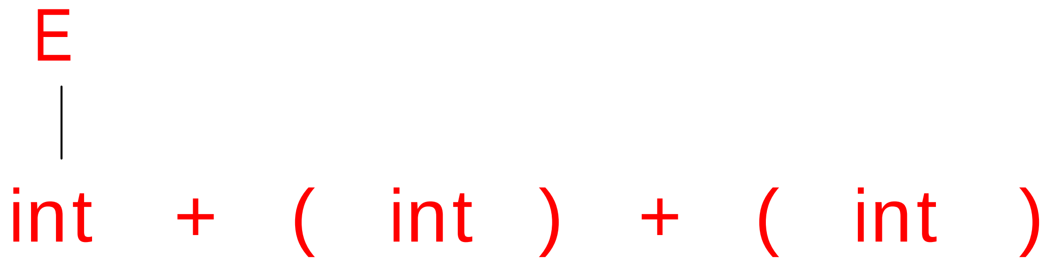
int + (int) + (int)

A Bottom-up Parse in Detail (2)

E → E + (E) | int

int + (int) + (int)

E + (int) + (int)



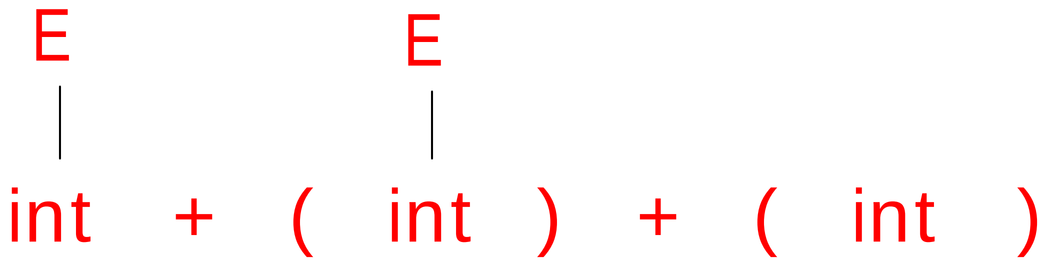
A Bottom-up Parse in Detail (3)

E → E + (E) | int

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)



A Bottom-up Parse in Detail (4)

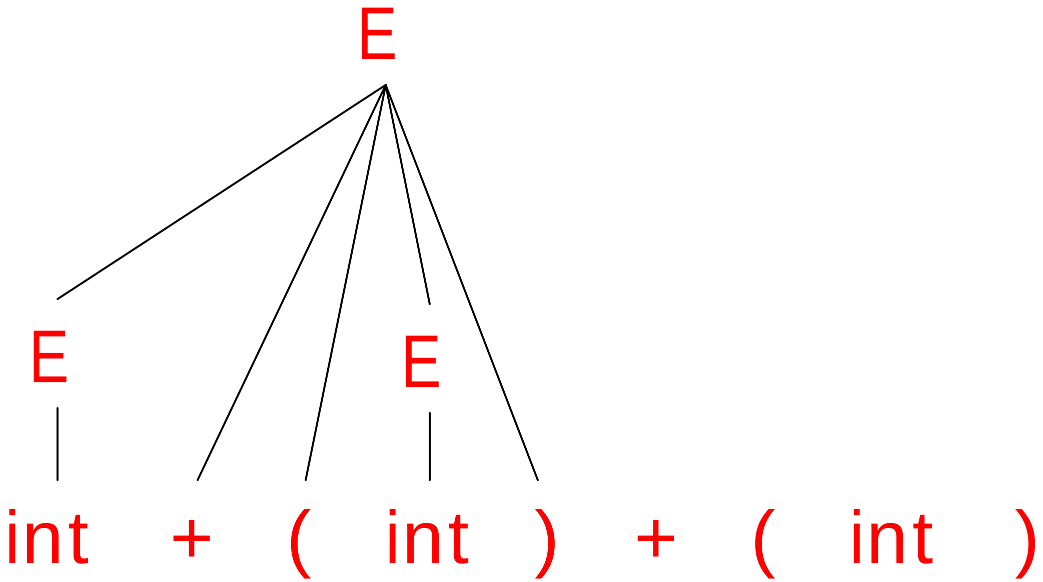
E → E + (E) | int

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

E → E + (E) | int

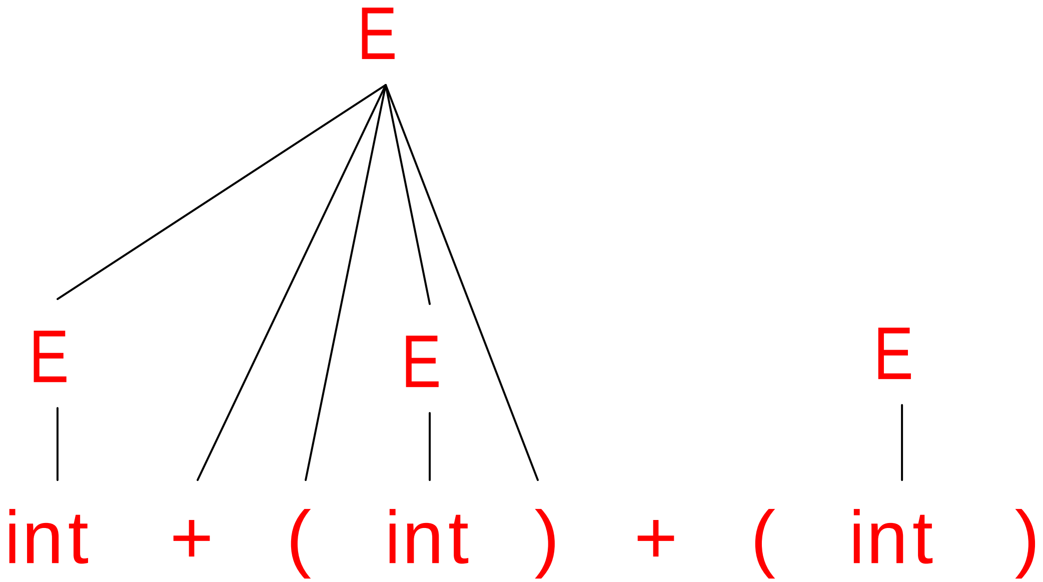
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)

E + (E)



A Bottom-up Parse in Detail (6)

E → E + (E) | int

int + (int) + (int)

E + (int) + (int)

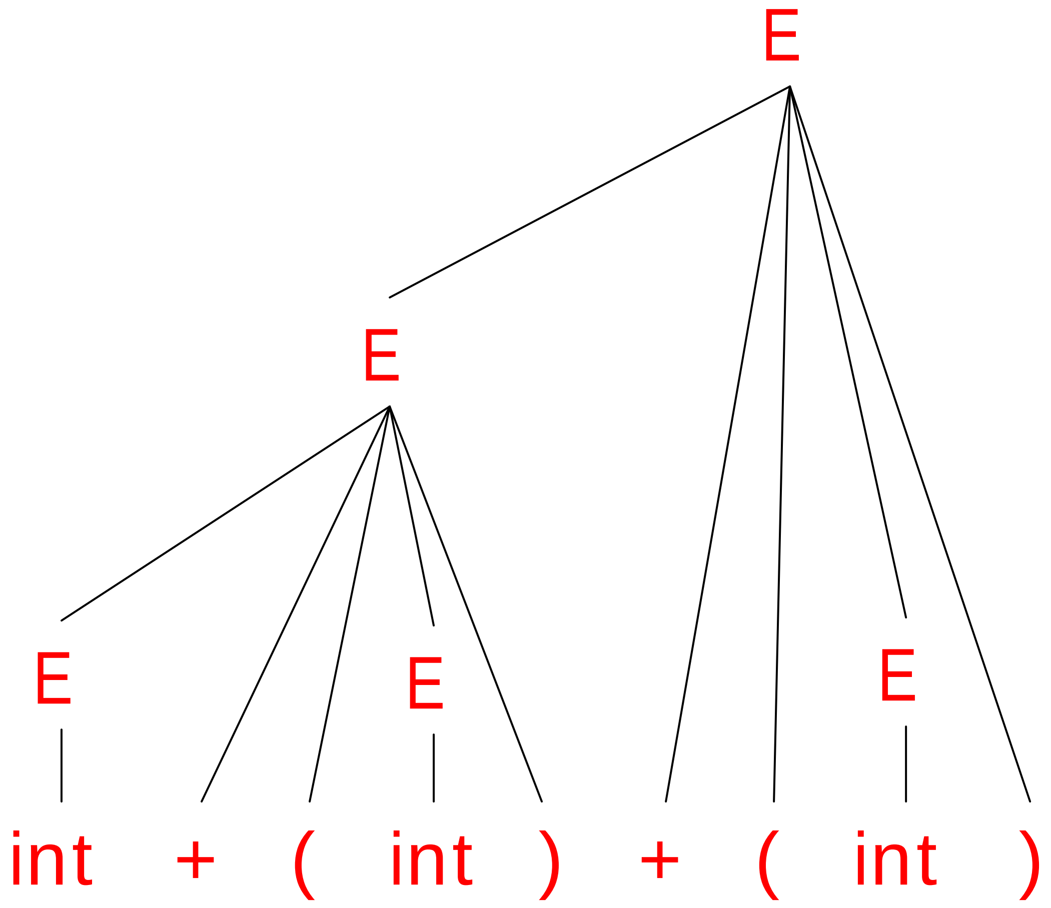
E + (E) + (int)

E + (int)

E + (E)

E

A rightmost derivation in reverse



Important Fact #1 about Bottom-up Parsing

An LR parser traces a rightmost derivation in reverse.

Where Do Reductions Happen

Fact #1 has an interesting consequence:

- Let $\alpha\beta\gamma$ be a string of non-terminals and terminals of a bottom-up parse.
- Assume the next reduction is by using $A \rightarrow \beta$.
- Then γ is a string of terminals.

Why?

Because $\alpha A \gamma \rightarrow \alpha\beta\gamma$ is a step in a right-most derivation.

Notation

- Idea: Split string into two substrings:
  - Right substring is as yet unexamined by parsing (a string of terminals).
  - Left substring has terminals and non-terminals.

- The dividing point is marked by a I
  - The I is not part of the string.

- Initially, all input is unexamined: $\mid x_1 x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

Shift: Move | one place to the right.

- Shifts a terminal to the left string.

E + (| int) ⇒ E + (int |)

In general:

ABC | xyz ⇒ ABCx | yz

Reduce

Reduce: Apply an inverse production at the right end of the left string.

If $E \rightarrow E + (E)$ is a production, then:

$$E + (\underline{E + (E)} \mid) \Rightarrow E + (\underline{E} \mid)$$

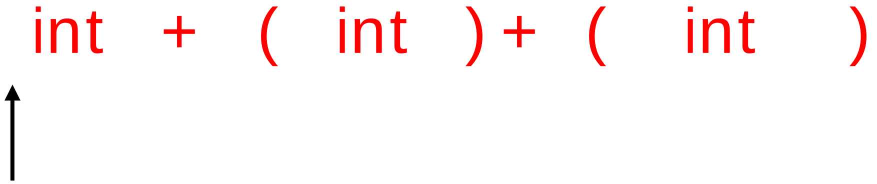
In general, given $A \rightarrow xy$, then:

$$Cbxy \mid ijk \Rightarrow CbA \mid ijk$$

Shift-Reduce Example

E → E + (E) | int

| int + (int) + (int)$ shift

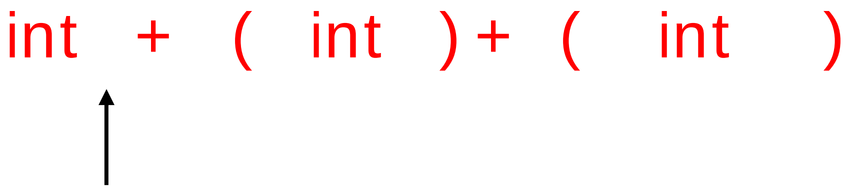

int + (int) + (int)

Shift-Reduce Example

E → E + (E) | int

I int + (int) + (int)$ shift

int I + (int) + (int)$ reduce E → int

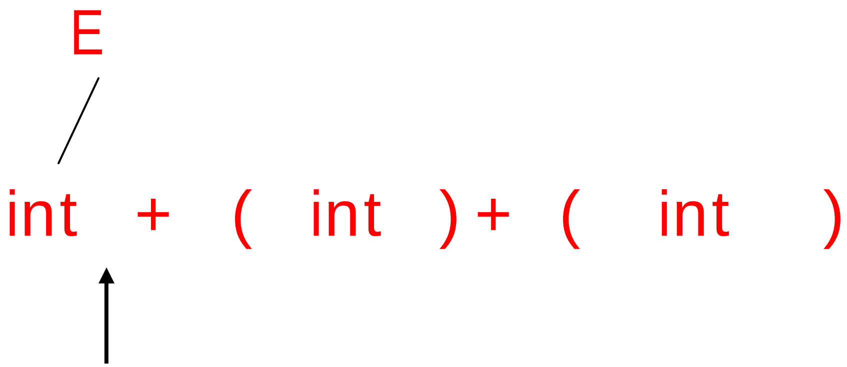


int + (int) + (int)

Shift-Reduce Example

E → E + (E) | int

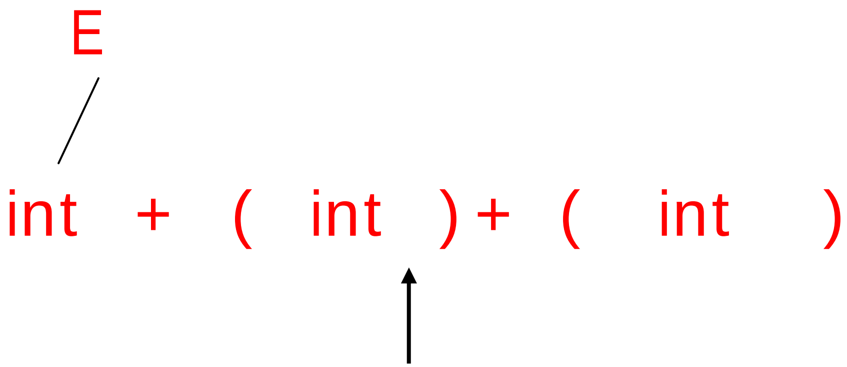
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times



Shift-Reduce Example

E → E + (E) | int

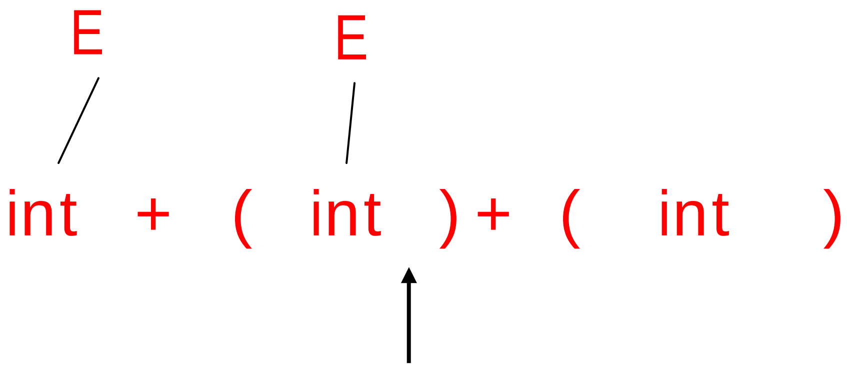
I int + (int) + (int)$	shift
int I + (int) + (int)$	reduce E → int
E I + (int) + (int)$	shift 3 times
E + (int I) + (int)$	reduce E → int



Shift-Reduce Example

E → E + (E) | int

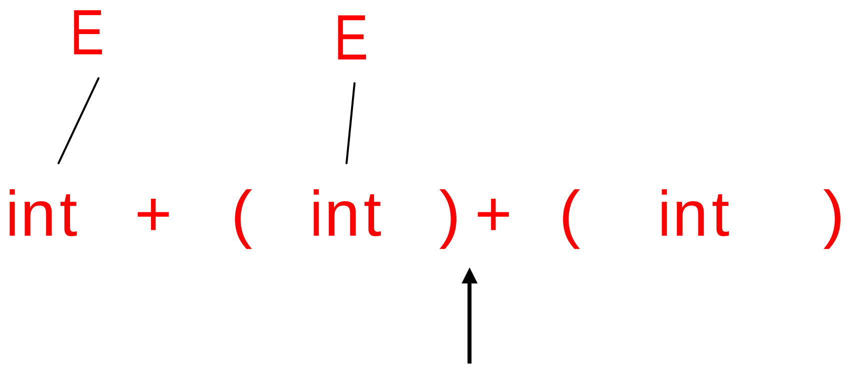
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift



Shift-Reduce Example

E → E + (E) | int

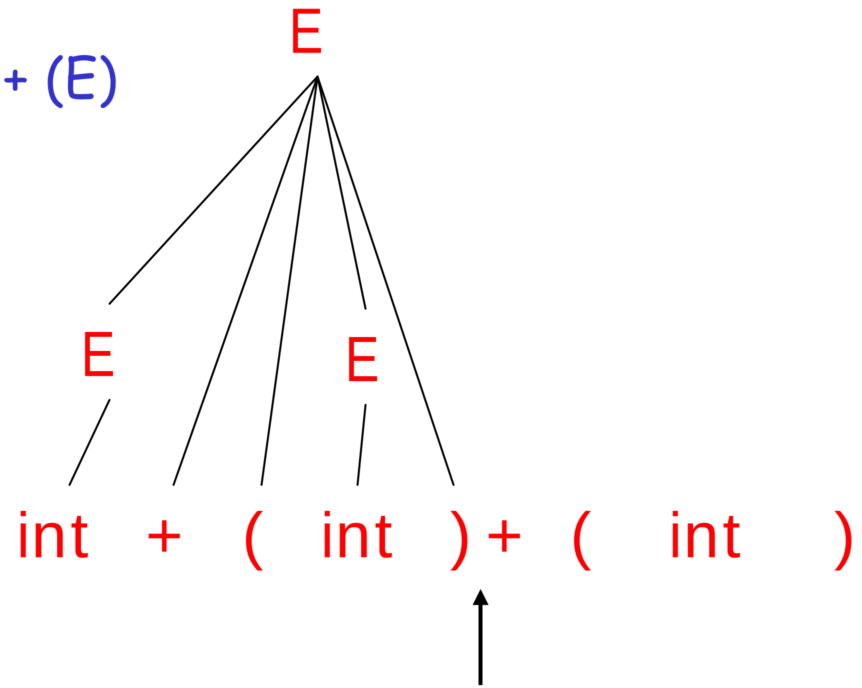
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	reduce E → E + (E)



Shift-Reduce Example

E → E + (E) | int

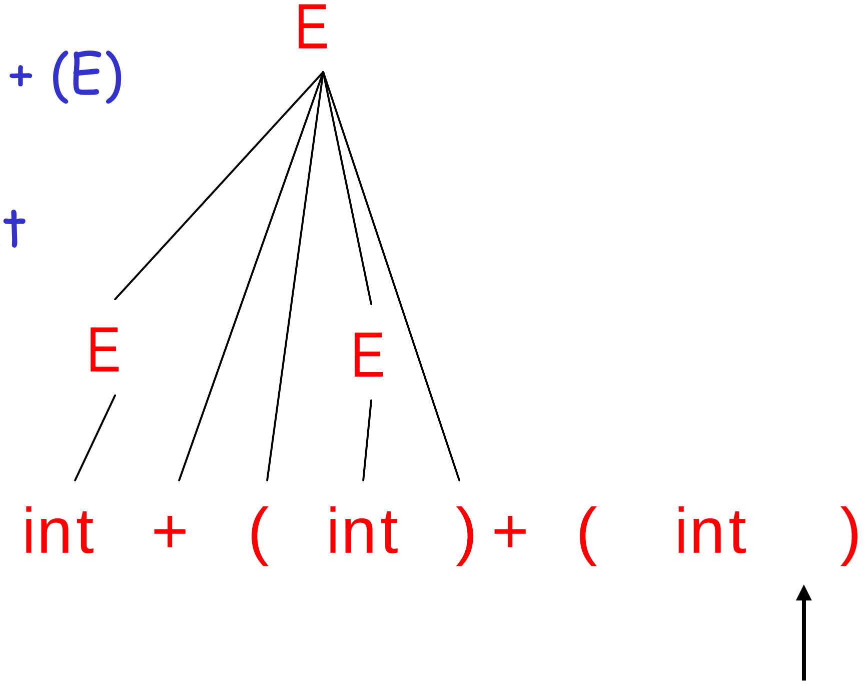
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	reduce E → E + (E)
E | + (int)$	shift 3 times



Shift-Reduce Example

E → E + (E) | int

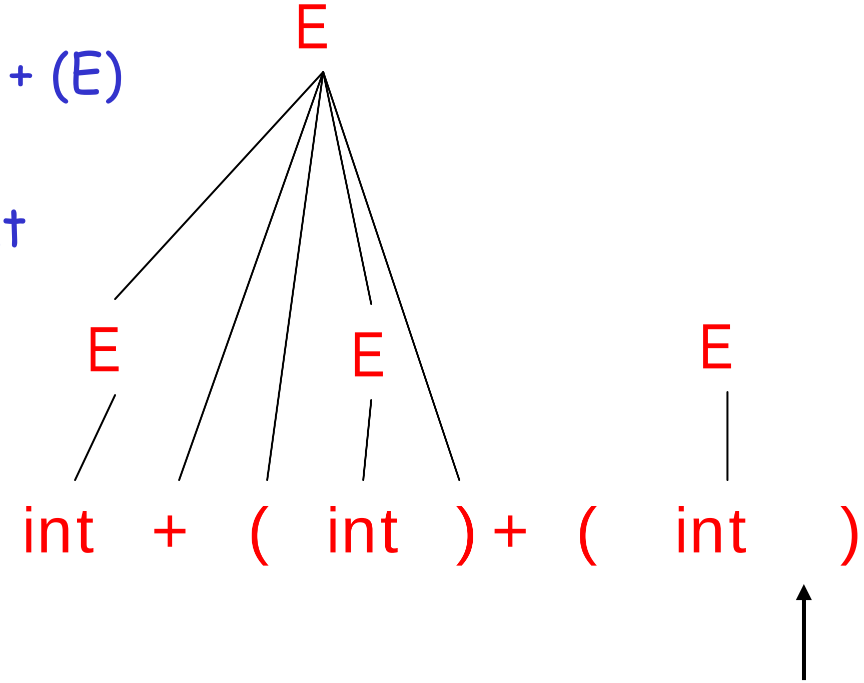
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	reduce E → E + (E)
E | + (int)$	shift 3 times
E + (int |)$	reduce E → int



Shift-Reduce Example

E → E + (E) \| int

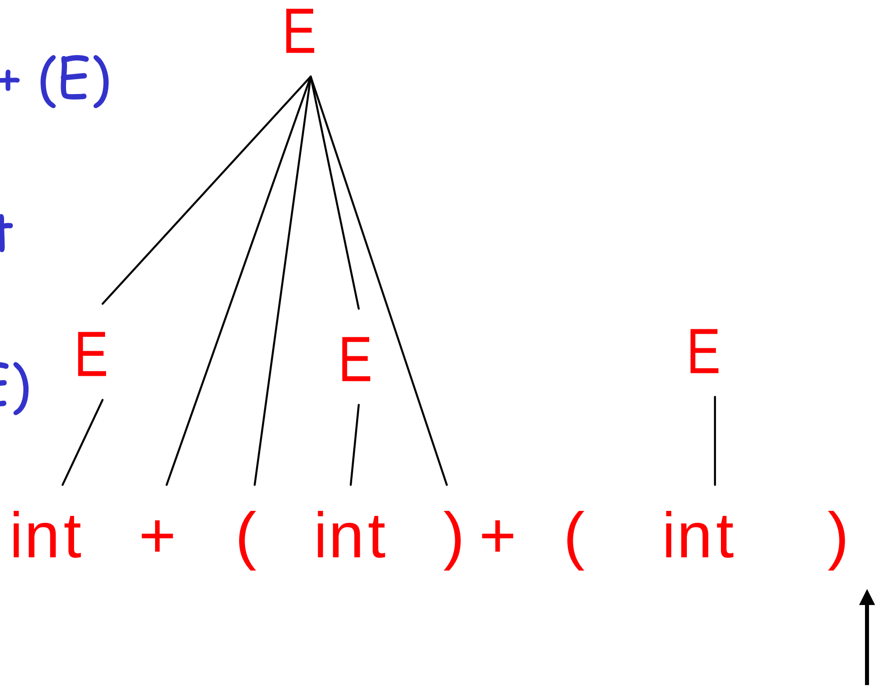
| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	reduce E → E + (E)
E | + (int)$	shift 3 times
E + (int |)$	reduce E → int
E + (E |)$	shift



Shift-Reduce Example

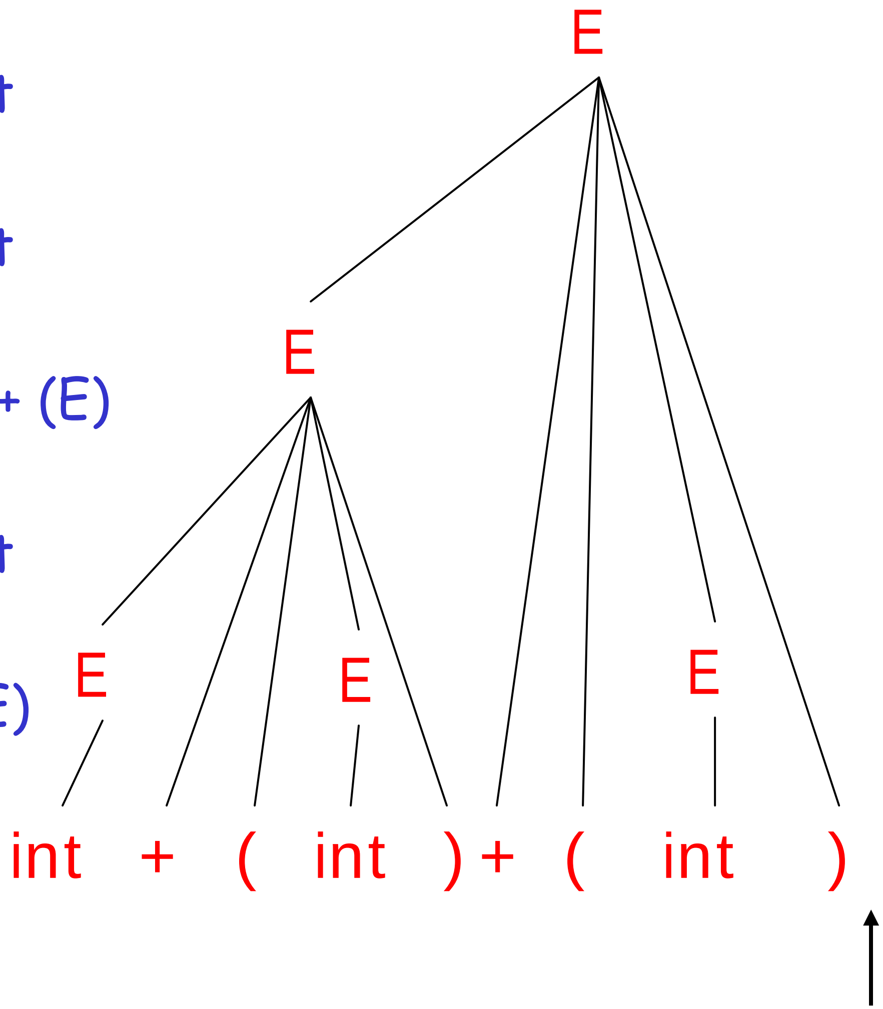
E → E + (E) | int

| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	reduce E → E + (E)
E | + (int)$	shift 3 times
E + (int |)$	reduce E → int
E + (E |)$	shift
E + (E) | $	reduce E → E + (E)



Shift-Reduce Example

| int + (int) + (int)$	shift
int | + (int) + (int)$	reduce E → int
E | + (int) + (int)$	shift 3 times
E + (int |) + (int)$	reduce E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	reduce E → E + (E)
E | + (int)$	shift 3 times
E + (int |)$	reduce E → int
E + (E |)$	shift
E + (E) | $	reduce E → E + (E)
E | $	accept



The Stack

- Left string can be implemented by a stack.
  - Top of the stack is the |

- Shift pushes a terminal on the stack.

- Reduce pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS).

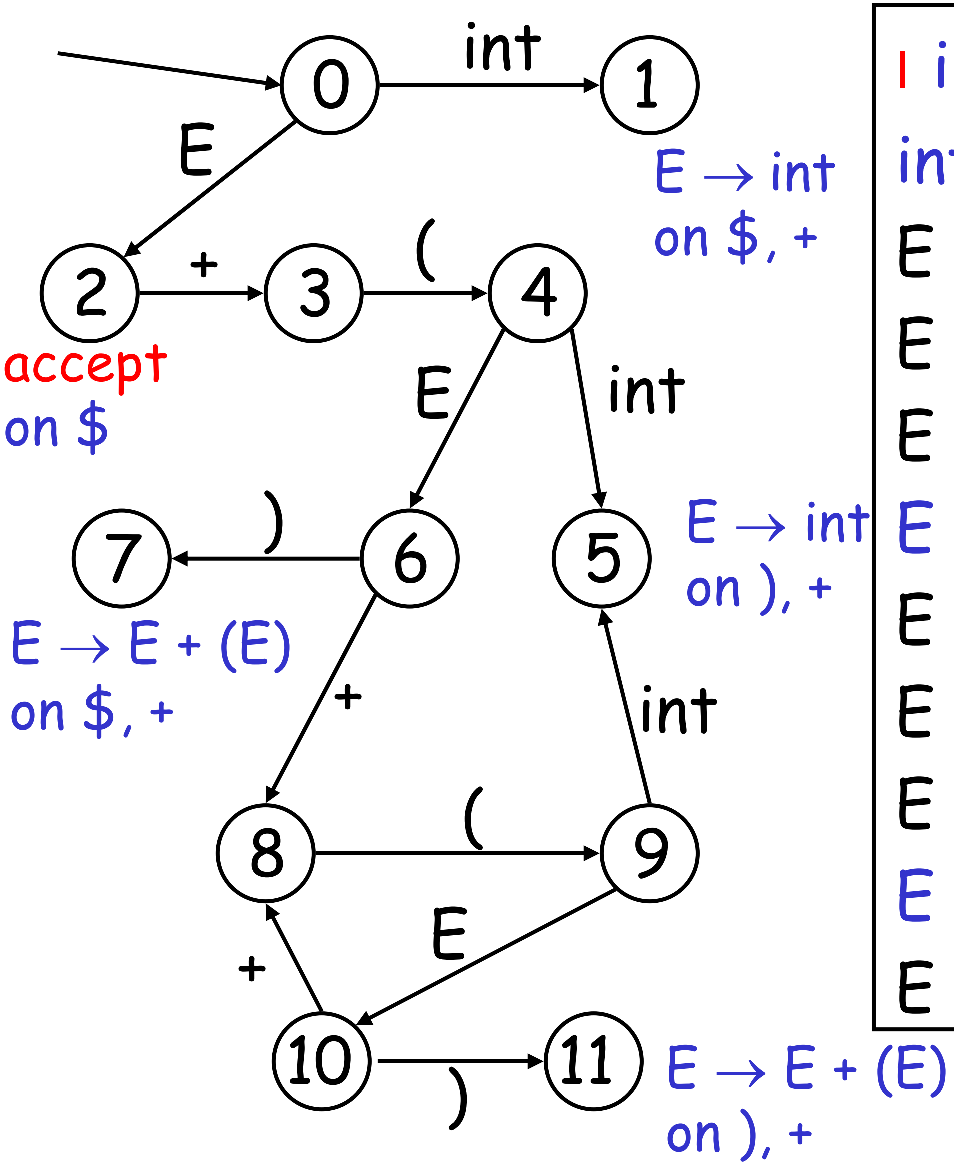
Key Question: To Shift or to Reduce?

<u>Idea</u>: use a finite automaton (DFA) to decide when to shift or reduce.

- The input is the stack.
- The language consists of terminals and non-terminals.

- We run the DFA on the stack and examine the resulting state X and the token tok after |
  - If X has a transition labeled tok then <u>shift</u>
  - If X is labeled with "*A* → β on tok" then <u>reduce</u>

LR(1) Parsing: An Example



| int + (int) + (int)$	shift
int | + (int) + (int)$	E → int
E | + (int) + (int)$	shift(x3)
E + (int |) + (int)$	E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	E → E+(E)
E | + (int)$	shift (x3)
E + (int |)$	E → int
E + (E |)$	shift
E + (E) | $	E → E+(E)
E | $	accept

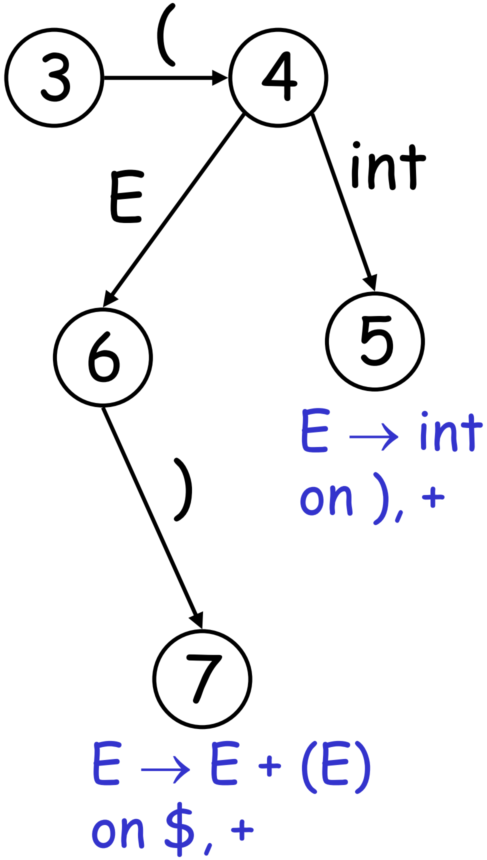
Representing the DFA

- Parsers represent the DFA as a 2D table.

  (Recall table-driven lexical analysis)
- Lines correspond to DFA states.
- Columns correspond to terminals and non-terminals.
- Typically columns are split into:
  - Those for terminals: action table.
    - action = shift or reduce
  - Those for non-terminals: goto table.

Representing the DFA: Example

The table for a fragment of our DFA:



	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E \to \text{int}}$		$r_{E \to \text{int}}$		
6		s8		s7		
7		$r_{E \to E+(E)}$			$r_{E \to E+(E)}$	
…						

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack.
  - This is wasteful, since most of the work is repeated.

- Remember for each stack element on which state it brings the DFA.

- LR parser maintains a stack

$$\langle \text{sym}_1, \text{state}_1 \rangle \; . \; . \; . \; \langle \text{sym}_n, \text{state}_n \rangle$$

state_k is the final state of the DFA on $\text{sym}_1 \ldots \text{sym}_k$.

The LR Parsing Algorithm

```
let I = w$ be initial input
let j = 0
let DFA state 0 be the start state
let stack = ⟨ dummy, 0 ⟩
   repeat
      case action[top_state(stack), I[j]] of
         shift k: push ⟨ I[j++], k ⟩
         reduce $X \to A$:
            pop $|A|$ pairs,
            push ⟨$X$, Goto[top_state(stack),$X$]⟩
         accept: halt normally
         error: halt and report error
```

LR Parsers

- Can be used to parse more grammars than LL.
- Most programming languages grammars are LR.
- LR parsers can be described as a simple table.
- There are tools for building the table.

Next Lecture: How is the DFA constructed?

LR Parsing
LALR Parser Generators

Outline

- Review of bottom-up parsing
- Computing the parsing DFA
- Using parser generators

Bottom-up Parsing (Review)

- A bottom-up parser rewrites the input string to the start symbol.
- The state of the parser is described as:

$$\alpha \mid \gamma$$

  - α is a stack of terminals and non-terminals;
  - γ is the string of terminals not yet examined.

- Initially: $\mid x_1x_2 \ldots x_n$

The Shift and Reduce Actions (Review)

Recall the CFG: $E \rightarrow E + (E) \mid \text{int}$

A bottom-up parser uses two kinds of actions:

- Shift pushes a terminal from input on the stack

$$E + (\mid \text{int}) \Rightarrow E + (\text{int} \mid)$$

- Reduce pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

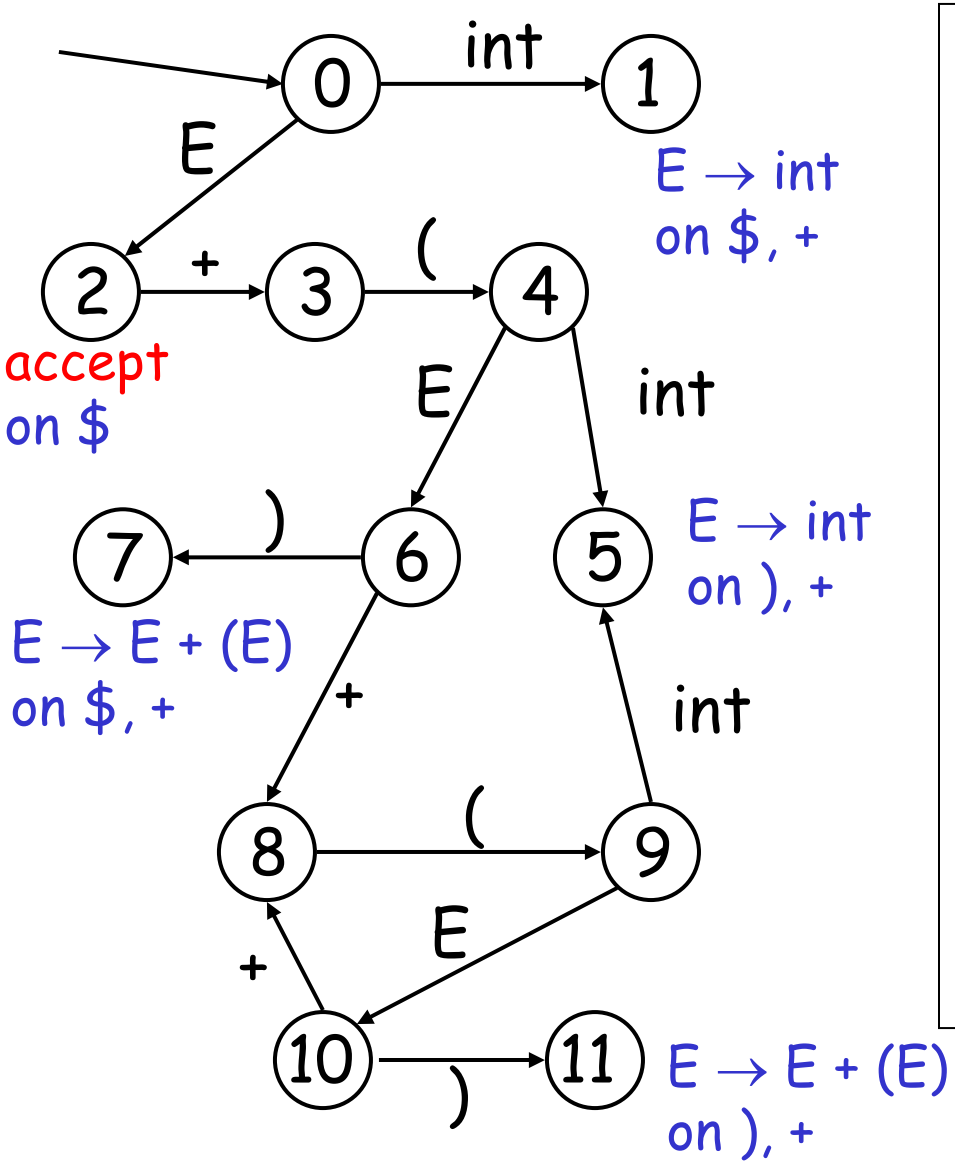
$$E + (\underline{E + (E)} \mid) \Rightarrow E + (\underline{E} \mid)$$

Key Issue: When to Shift or Reduce?

- Idea: use a deterministic finite automaton (DFA) to decide when to shift or reduce
  - The input is the stack
  - The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state X and the token tok after |
  - If X has a transition labeled tok then shift
  - If X is labeled with "$A \to \beta$ on tok" then reduce

LR(1) Parsing: An Example



| int + (int) + (int)$	shift
int | + (int) + (int)$	E → int
E | + (int) + (int)$	shift (x3)
E + (int |) + (int)$	E → int
E + (E |) + (int)$	shift
E + (E) | + (int)$	E → E+(E)
E | + (int)$	shift (x3)
E + (int |)$	E → int
E + (E |)$	shift
E + (E) | $	E → E+(E)
E | $	accept

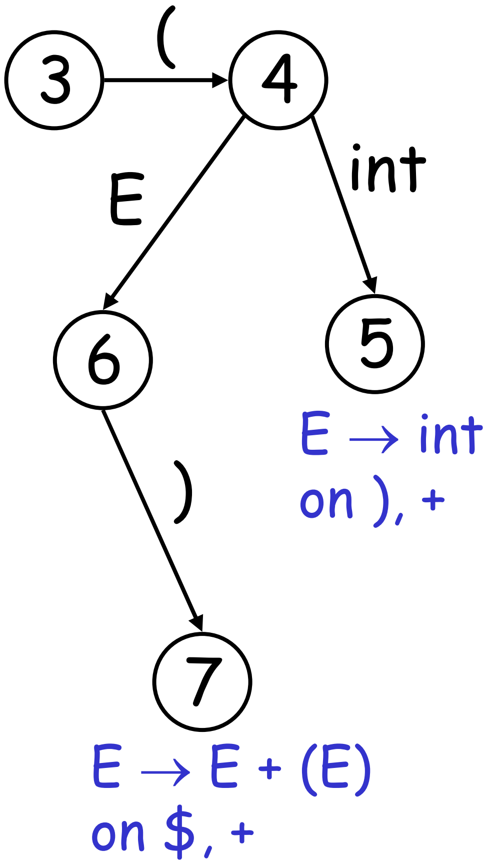
Representing the DFA

- Parsers represent the DFA as a 2D table.

  (Recall table-driven lexical analysis.)
- Lines correspond to DFA states.
- Columns correspond to terminals and non-terminals.
- Typically columns are split into:
  - Those for terminals: the action table.
  - Those for non-terminals: the goto table.

Representing the DFA: Example

The table for a fragment of our DFA:



	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E \to \text{int}}$		$r_{E \to \text{int}}$		
6		s8		s7		
7		$r_{E \to E+(E)}$			$r_{E \to E+(E)}$	
…						

sk is shift and goto state k
$r_{X \to \alpha}$ is reduce
gk is goto state k

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated

- To avoid this, we remember for each stack element on which state it brings the DFA.

- LR parser maintains a stack

$\langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle$

state_k is the final state of the DFA on $\text{sym}_1 \ldots \text{sym}_k$

The LR Parsing Algorithm

```
let I = w$ be initial input
let j = 0
let DFA state 0 be the start state
let stack = ⟨ dummy, 0 ⟩
  repeat
    case action[top_state(stack), I[j]] of
      shift k: push ⟨ I[j++], k ⟩
      reduce $X \to A$:
        pop $|A|$ pairs,
        push ⟨ $X$, goto[top_state(stack), $X$] ⟩
      accept: halt normally
      error: halt and report error
```

Key Issue: How is the DFA Constructed?

- The stack describes the context of the parse:
  - What non-terminal we are looking for.
  - What production RHS we are looking for.
  - What we have seen so far from the RHS.

- Each DFA state describes several such contexts.

  E.g., when we are looking for non-terminal E, we might be looking either for an int or an E + (E) RHS.

LR(0) Items

- An <u>LR(0) item</u> is a production with a “|” somewhere on the RHS.

- The LR(0) items for $T \rightarrow (E)$ are

  $T \rightarrow \mid (E)$

  $T \rightarrow (\mid E)$

  $T \rightarrow (E \mid)$

  $T \rightarrow (E) \mid$

- The only LR(0) item for $X \rightarrow \varepsilon$ is $X \rightarrow \mid$

LR(0) Items: Intuition

- An item $[X \rightarrow \alpha \mid \beta]$ says that the parser:
  - is looking for an X
  - has an α on top of the stack
  - expects to find a string derived from β next in the input.

- Notes:
  - $[X \rightarrow \alpha \mid a\beta]$ means that a should follow.
    - Then we can shift it and still have a viable prefix.
  - $[X \rightarrow \alpha \mid]$ means that we could reduce X.
    - But this is not always a good idea !

LR(1) Items

- An LR(1) item is a pair:

$$X \to \alpha \,\vert\, \beta,\ a$$

  - $X \to \alpha\beta$ is a production.
  - a is a terminal (the lookahead terminal).
  - LR(1) means 1 lookahead terminal.
- $[X \to \alpha \,\vert\, \beta, a]$ describes a context of the parser.
  - We are trying to find an X followed by an a.
  - We have (at least) α already on top of the stack.
  - Thus we need to see next a prefix derived from βa.

Note

- The symbol | was used before to separate the stack from the rest of input:

  $\alpha \mid \gamma$, where α is the stack and γ is the remaining string of terminals.

- In items, | is used to mark a prefix of a production RHS:

$$X \rightarrow \alpha \mid \beta,\ a$$

  - Here β might contain non-terminals as well.

- In either case, the stack is on the left of |

Convention

- We add to our grammar a fresh new start symbol S and a production $S \rightarrow E$
  - Where E is the old start symbol.

- The initial parsing context contains:

$$S \rightarrow \mid E \;,\; \$$$

  - Trying to find an S as a string derived from E$
  - The stack is empty.

LR(1) Items (Cont.)

- In context containing

$$E \rightarrow E + \mid (E) \quad , +$$

  - If (follows then we can perform a shift to context containing

$$E \rightarrow E + (\mid E) \quad , +$$

- In context containing

$$E \rightarrow E + (E) \mid \quad , +$$

  - We can perform a reduction with $E \rightarrow E + (E)$
  - But only if a + follows

LR(1) Items (Cont.)

- Consider the item

$$E \rightarrow E + (\mid E) \ , +$$

- We expect a string derived from $E\ )\ +$
- Our example has two productions for E

$$E \rightarrow \text{int} \quad \text{and} \quad E \rightarrow E + (E)$$

- We describe this by extending the context with two more items:

$$E \rightarrow \mid \text{int} \qquad ,)$$

$$E \rightarrow \mid E + (E) \ ,)$$

The Closure Operation

- The operation of extending the context with items is called the closure operation.

```
Closure(Items) =
  repeat
    for each [$X \to \alpha \mid Y\beta$, $a$] in Items
      for each production $Y \to \gamma$
        for each $b$ in First($\beta a$)
          add [$Y \to \mid \gamma$, $b$] to Items
  until Items is unchanged
```

Constructing the Parsing DFA (1)

$E \to E + (E) \mid \text{int}$

- Construct the start context:

  Closure({$S \to | E, \$$})

  $S \to | E \quad , \$$
  $E \to | E\text{+}(E), \$$
  $E \to | \text{int} \quad , \$$
  $E \to | E\text{+}(E), +$
  $E \to | \text{int} \quad , +$

- We abbreviate as:

  $S \to | E \quad , \$$
  $E \to | E\text{+}(E) \quad , \$/+$
  $E \to | \text{int} \quad , \$/+$

Constructing the Parsing DFA (2)

- A DFA state is a closed set of LR(1) items.

- The start state contains $[S \rightarrow \mid E\ , \$]$.

- A state that contains $[X \rightarrow \alpha \mid, b]$ is labeled with "reduce with $X \rightarrow \alpha$ on b".

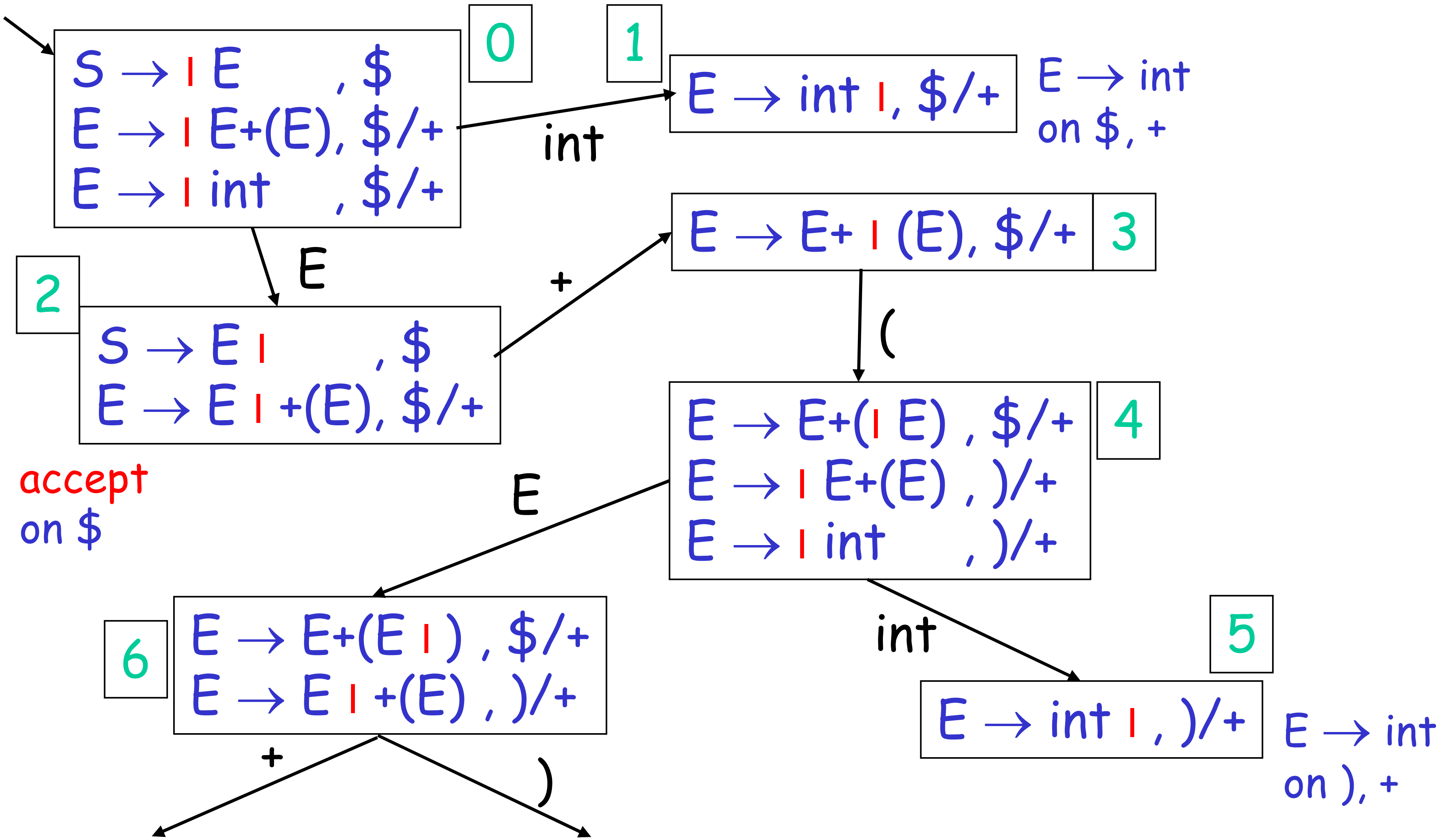
- And now the transitions …

The DFA Transitions

- A state "State" that contains $[X \rightarrow \alpha \mid y\beta, b]$ has a transition labeled y to a state that contains the items "**Transition**(State, y)"
  - y can be a terminal or a non-terminal

```
Transition(State, y)
  Items = $\varnothing$
  for each $[X \rightarrow \alpha \mid y\beta, b]$ in State
      add $[X \rightarrow \alpha y \mid \beta, b]$ to Items
  return Closure(Items)
```

Constructing the Parsing DFA: Example



0

S → | E , $

E → | E+(E), $/+

E → | int , $/+

1

E → int |, $/+

E → int on $, +

2

S → E | , $

E → E | +(E), $/+

accept on $

3

E → E+ | (E), $/+

4

E → E+(| E) , $/+

E → | E+(E) ,)/+

E → | int ,)/+

5

E → int | ,)/+

E → int on), +

6

E → E+(E |) , $/+

E → E | +(E) ,)/+

and so on…

LR Parsing Tables: Notes

- Parsing tables (i.e., the DFA) can be constructed automatically for a CFG.

- But we still need to understand the construction to work with parser generators.
  - E.g., they report errors in terms of sets of items.

- What kind of errors can we expect?

Shift/Reduce Conflicts

- If a DFA state contains both

  $[X \rightarrow \alpha \mid a\beta, b]$ and $[Y \rightarrow \gamma \mid, a]$

- Then on input "a" we could either
  - Shift into state $[X \rightarrow \alpha a \mid \beta, b]$, or
  - Reduce with $Y \rightarrow \gamma$

- This is called a *shift-reduce conflict*

Shift/Reduce Conflicts

- Typically due to ambiguities in the grammar.
- Classic example: the dangling else

  S → if E then S | if E then S else S | OTHER

- We will have a DFA state containing:

  [S → if E then S I,   else]

  [S → if E then S I else S,   x]

- If else follows then we can shift or reduce.
- Default (yacc, ML-yacc, bison, etc.) is to shift.
  - Default behavior is as needed in this case.

More Shift/Reduce Conflicts

- Consider the ambiguous grammar:

$$E \rightarrow E + E \;\mid\; E * E \;\mid\; \text{int}$$

- We will have the states containing:

$$\begin{array}{lcl}
[E \rightarrow E * \,|\, E,\ +] & & [E \rightarrow E * E \,|,\ +] \\
[E \rightarrow \,|\, E + E,\ +] & \Rightarrow^{E} & [E \rightarrow E \,|\, + E,\ +] \\
\quad\ldots & & \quad\ldots
\end{array}$$

- Again we have a shift/reduce on input +
  - We need to reduce (* binds more tightly than +)
  - Recall solution: declare the precedence of * and +

More Shift/Reduce Conflicts

- In yacc declare precedence and associativity:

```
%left +
%left *
```

- Precedence of a rule = that of its last terminal.

  See yacc manual for ways to override this default.

- Resolve shift/reduce conflict with a shift if:
  - no precedence declared for either rule or terminal;
  - input terminal has higher precedence than the rule;
  - the precedences are the same and right associative.

Using Precedence to Solve S/R Conflicts

- Back to our example:

$[E \to E * | E,\ +]$ $\qquad$ $[E \to E * E\, |,\ +]$

$[E \to | E + E,\ +]$ $\overset{E}{\Rightarrow}$ $[E \to E\, | + E,\ +]$

… $\qquad$ …

- We will choose reduce because precedence of rule $E \to E * E$ is higher than of terminal $+$.

Using Precedence to Solve S/R Conflicts

- Same grammar as before:

$$E \to E + E \ \mid \ E * E \ \mid \ \text{int}$$

- We will also have the states:

$$[E \to E + \mid E,\ +] \qquad\qquad [E \to E + E \mid,\ +]$$

$$[E \to \mid E + E,\ +] \quad \Rightarrow^{E} \quad [E \to E \mid + E,\ +]$$

$$\ldots \qquad\qquad\qquad\qquad \ldots$$

- Now we also have a shift/reduce on input +
  - We will choose reduce because $E \to E + E$ and + have the same precedence and + is left-associative.

Using Precedence to Solve S/R Conflicts

- Back to our dangling else example:

  [S → if E then S •, else]

  [S → if E then S • else S, x]

- Can eliminate conflict by declaring else having higher precedence than then.
- But this starts to look like "hacking the tables".
- Best to avoid overuse of precedence declarations or we will end with unexpected parse trees.

Precedence Declarations Revisited

The term "precedence declaration" is misleading!

These declarations do not define precedence; instead, they define conflict resolutions.

- I.e., they instruct shift-reduce parsers to resolve conflicts in certain ways.
- These two are not quite the same!

Reduce/Reduce Conflicts

- If a DFA state contains both

  $[X \to \alpha \,|, a]$ and $[Y \to \beta \,|, a]$

  - Then on input "a" we do not know which production to reduce.

- This is called a *reduce/reduce conflict*

Reduce/Reduce Conflicts

- Usually due to gross ambiguity in the grammar.
- Ex. A grammar for a sequence of identifiers:

$$S \rightarrow \varepsilon \mid \text{id} \mid \text{id } S$$

- There are two parse trees for the string id

$$S \rightarrow \text{id}$$

$$S \rightarrow \text{id } S \rightarrow \text{id}$$

- How does this confuse the parser?

More on Reduce/Reduce Conflicts

- Consider the states

$$\begin{array}{lcl}
 & & [S \to \text{id} \mid, \quad \$] \\
[S' \to \mid S, \quad \$] & & [S \to \text{id} \mid S, \quad \$] \\
[S \to \mid, \quad \$] & \Rightarrow^{\text{id}} & [S \to \mid, \quad \$] \\
[S \to \mid \text{id}, \quad \$] & & [S \to \mid \text{id}, \quad \$] \\
[S \to \mid \text{id}\ S, \quad \$] & & [S \to \mid \text{id}\ S, \quad \$]
\end{array}$$

- Reduce/reduce conflict on input $\$$

$$S' \to S \to \text{id}$$

$$S' \to S \to \text{id}\ S \to \text{id}$$

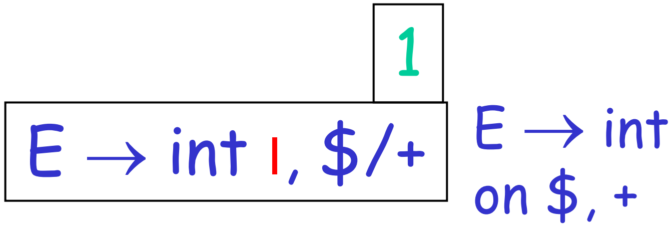
- Better to rewrite the grammar as: $S \to \varepsilon \mid \text{id}\ S$

Using Parser Generators

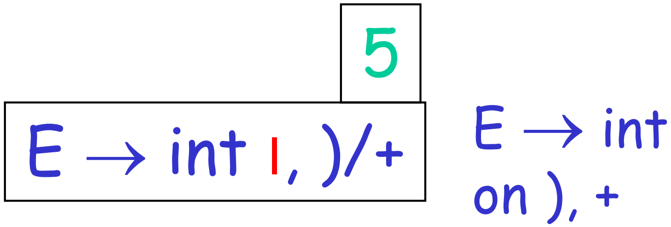
- Parser generators automatically construct the parsing DFA given a CFG.
    - Use precedence declarations and default conventions to resolve conflicts.
    - The parser algorithm is the same for all grammars (and is provided as a library function).
- But most parser generators do not construct the DFA as described before.
    - Because the LR(1) parsing DFA has 1000s of states even for a simple language.

LR(1) Parsing Tables are Big

- But many states are similar, e.g.

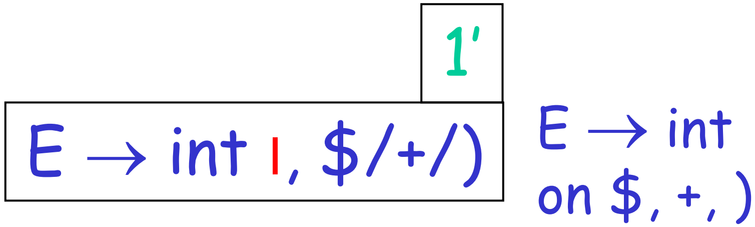


and



- Idea: merge the DFA states whose items differ only in the lookahead tokens
  - We say that such states have the same core

- We obtain



The Core of a Set of LR Items

Definition: The core of a set of LR items is the set of first components

- Without the lookahead terminals

- Example: the core of

$$\{[X \to \alpha \mid \beta, b], [Y \to \gamma \mid \delta, d]\}$$

is

$$\{X \to \alpha \mid \beta,\ Y \to \gamma \mid \delta\}$$

LALR States

- Consider for example the LR(1) states

$$\{[X \to \alpha \mid, a], [Y \to \beta \mid, c]\}$$
$$\{[X \to \alpha \mid, b], [Y \to \beta \mid, d]\}$$

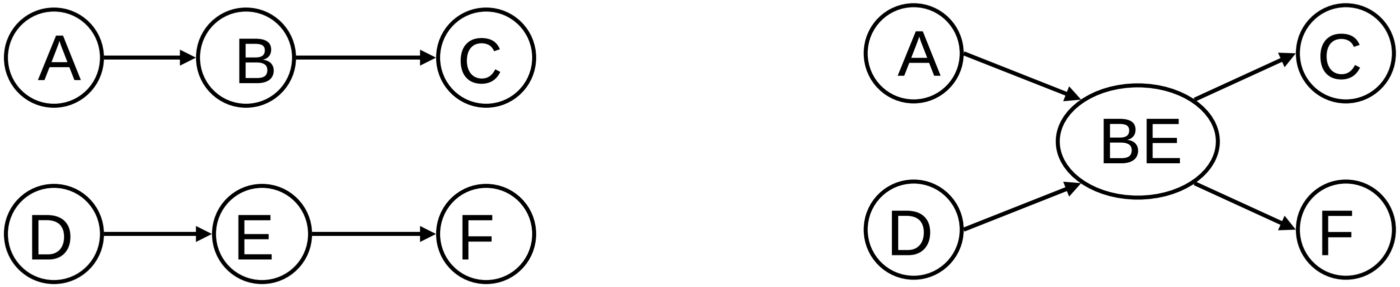
- They have the same core and can be merged
- The merged state contains:

$$\{[X \to \alpha \mid, a/b], [Y \to \beta \mid, c/d]\}$$

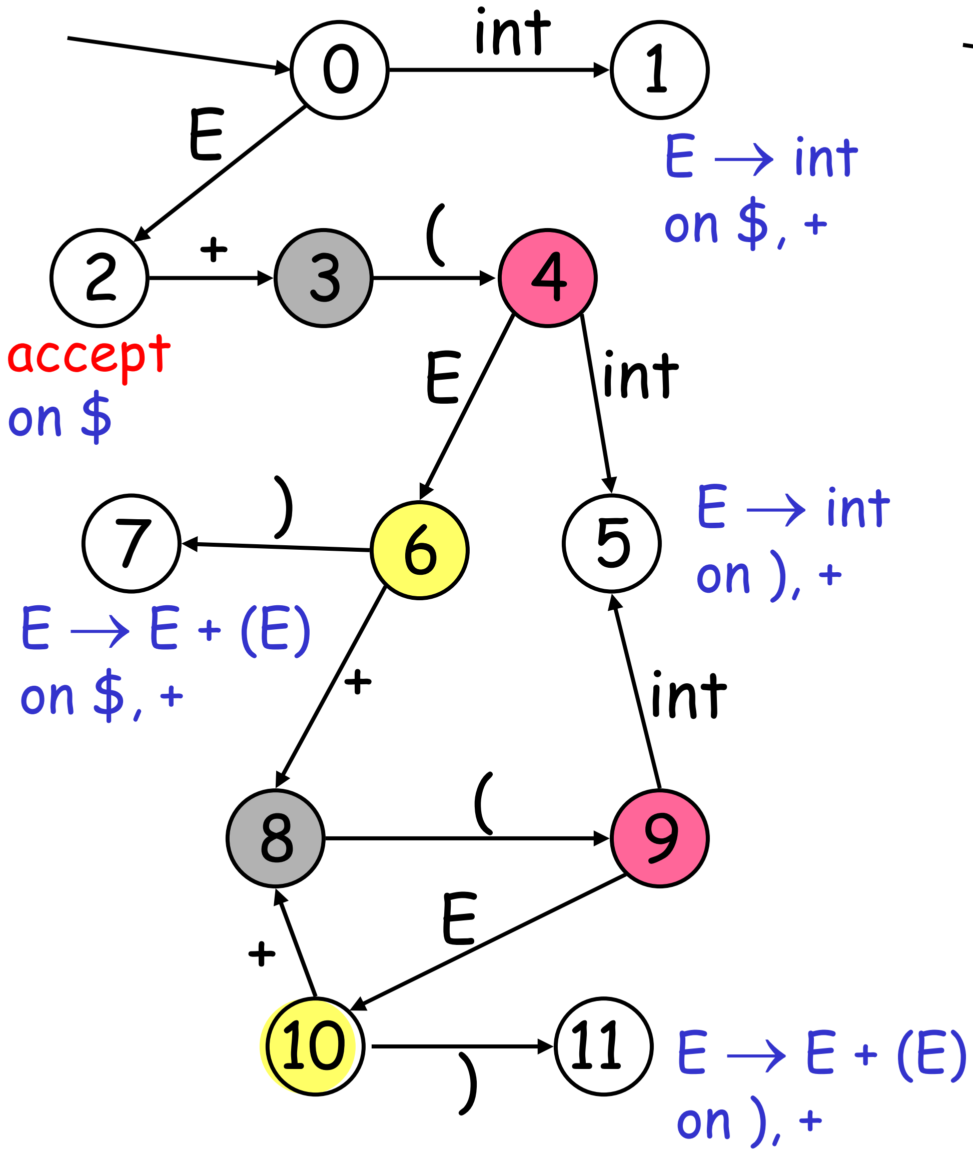
- These are called LALR(1) states
  - Stands for LookAhead LR
  - Typically 10 times fewer LALR(1) states than LR(1)

A LALR(1) DFA

- Repeat until all states have distinct core
  - Choose two distinct states with same core
  - Merge the states by creating a new one with the union of all the items
  - Point edges from predecessors to new state
  - New state points to all the previous successors



Conversion LR(1) to LALR(1): Example.



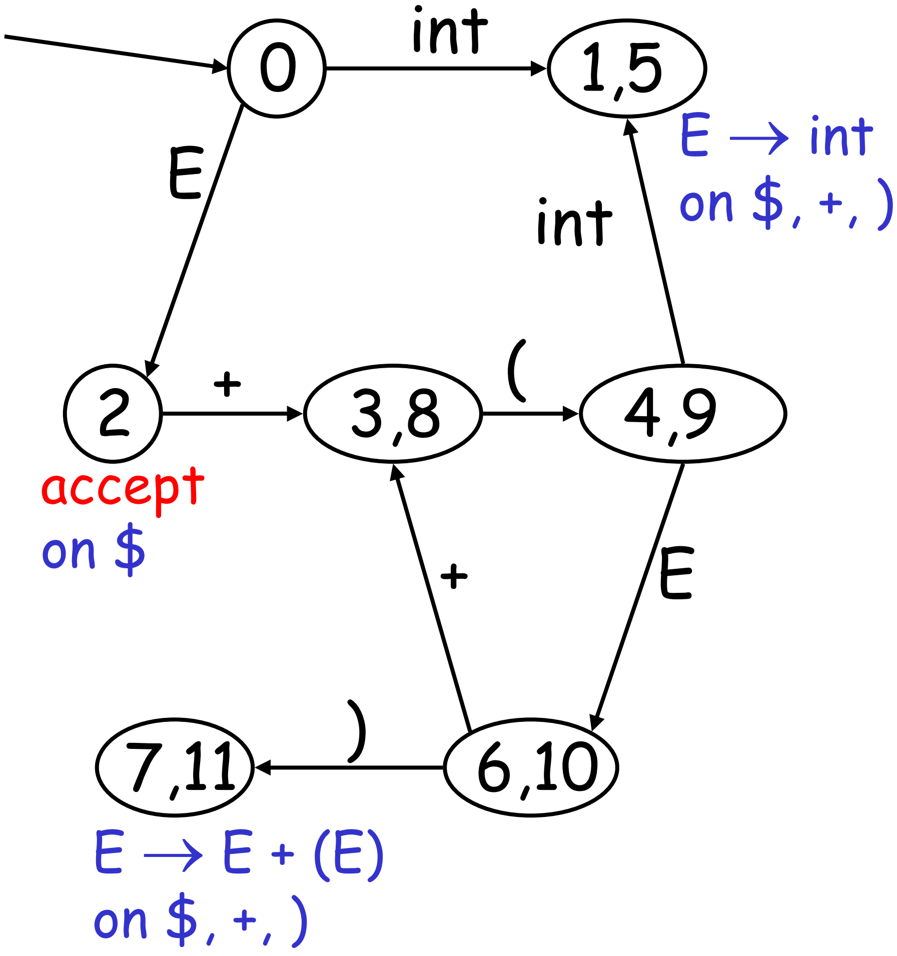
E → int
on $, +

accept
on $

E → int
on), +

E → E + (E)
on $, +

E → E + (E)
on), +



E → int
on $, +,)

accept
on $

E → E + (E)
on $, +,)

The LALR Parser Can Have Conflicts

- Consider for example the LR(1) states:

$$\{[X \to \alpha \mid, a], [Y \to \beta \mid, b]\}$$

$$\{[X \to \alpha \mid, b], [Y \to \beta \mid, a]\}$$

- And the merged LALR(1) state:

$$\{[X \to \alpha \mid, a/b], [Y \to \beta \mid, a/b]\}$$

- Has a new reduce/reduce conflict!

- In practice, such cases are rare.

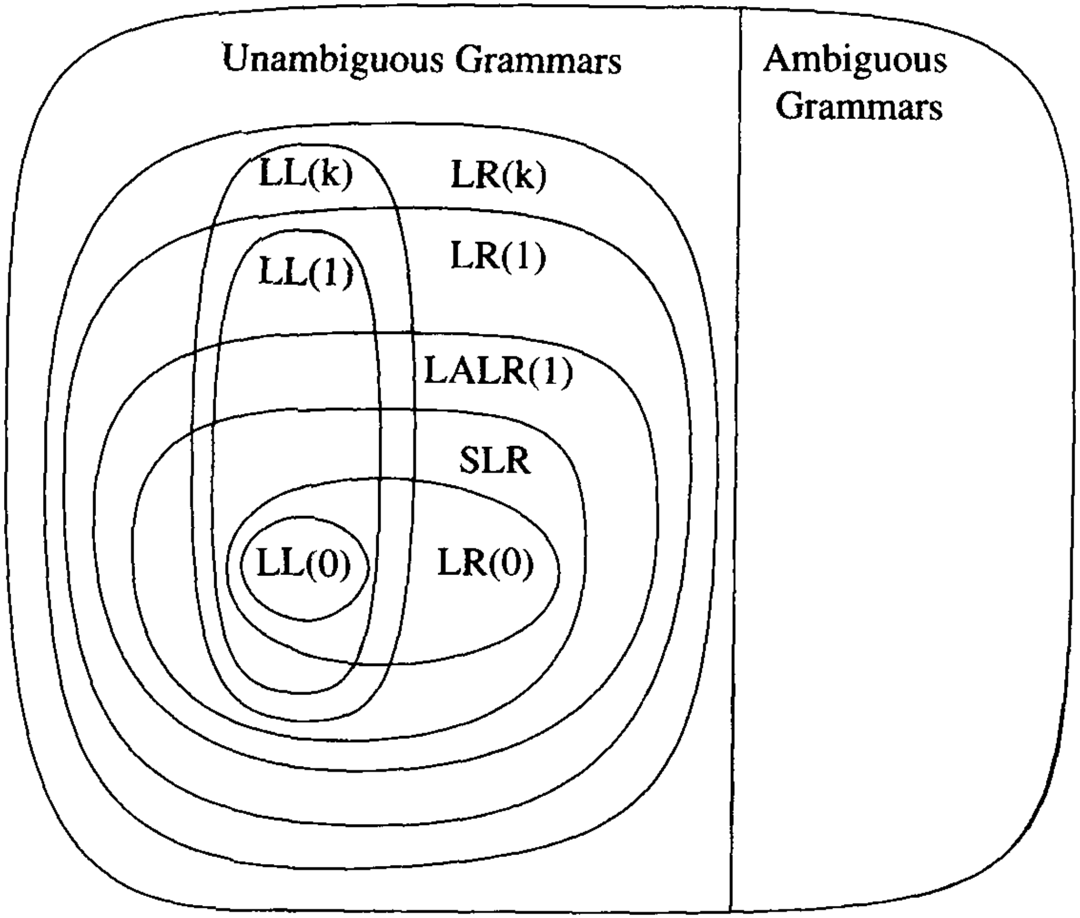
LALR vs. LR Parsing: Things to keep in mind

- LALR languages are not natural.
  - They are an "efficiency hack" on LR languages.

- Any reasonable programming language has a LALR(1) grammar.

- LALR(1) parsing has become a standard for programming languages and parser generators.

A Hierarchy of Grammar Classes



From Andrew Appel,
"Modern Compiler
Implementation in ML"

Semantic Analysis

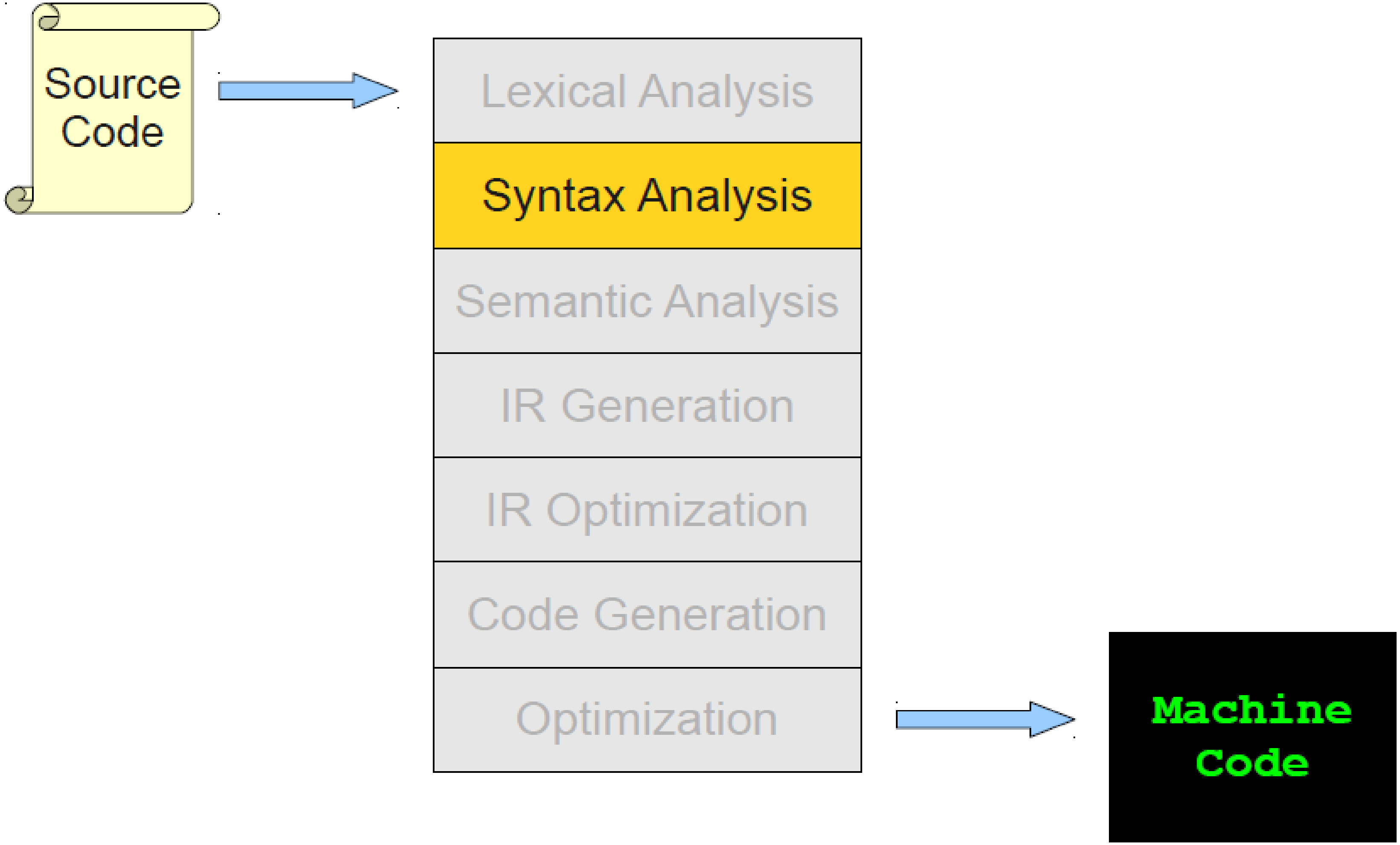
Outline

- The role of semantic analysis in a compiler
  - A laundry list of tasks

- **Scope**
  - Static vs. Dynamic scoping
  - Implementation: symbol tables

- **Types**
  - Static analyses that detect type errors
  - Statically vs. Dynamically typed languages

Where we are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

The Compiler Front-End

Lexical analysis: program is *lexically* well-formed

- Tokens are legal
  - e.g. identifiers have valid names, no stray characters, etc.
- Detects inputs with illegal tokens

Parsing: program is *syntactically* well-formed

- Declarations have correct structure, expressions are syntactically valid, etc.
- Detects inputs with ill-formed syntax

Semantic analysis:

- Last "front end" compilation phase
- Catches all remaining errors

Beyond Syntax Errors

- What's wrong with this C code? (Note: it parses correctly)

- Undeclared identifier
- Multiply declared identifier
- Index out of bounds
- Wrong number or types of arguments to function call
- Incompatible types for operation
- A **break** statement outside switch/loop
- A **goto** a non-existing label

```
foo(int a, char * s){...}

int bar() {
  int f[3];
  int i, j, k;
  char q, *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 42;
  j = m + k;
  printf("%s,%s.\n",p,q);
  goto label42;
}
```

Program Checking

- Why do we care?
  - To report mistakes to the programmer early
  - To avoid bugs: **`f[6]`** will cause a run-time failure
  - To help programmers verify intent

- How do these check help the compiler?
  - To allocate the right amount of space for variables
  - To select the right machine instructions
  - To properly implement control structures

Why Have a Separate Semantic Analysis?

Parsing cannot catch some errors

Some language constructs are not context-free

- Example: Identifier declaration and use
- An abstract version of the problem is:

$$L = \{ wcw \mid w \in (a + b)^* \}$$

- The 1st w represents the identifier's declaration; the 2nd w represents a use of the identifier
- This language is not context-free

What Does Semantic Analysis Do?

Performs checks beyond syntax of many kinds ...

Examples:

1. All used identifiers are declared (i.e. scoping)
2. Identifiers declared only once
3. Types (e.g. operators are used with right operands)
4. Procedures and functions defined only once
5. Procedures and functions used with the right number and type of arguments
6. Control-flow checks

And many others . . .

The requirements depend on the language

What's Wrong?

Example 1

```
let string y ← "abc" in y + 42
```

Example 2

```
let integer y in x + 42
```

Semantic Processing: Syntax-Directed Translation

Basic idea: Associate information with language constructs by attaching *attributes* to the grammar symbols that represent these constructs

- Values for attributes are computed using semantic rules associated with grammar productions
- An attribute can represent anything (reasonable) that we choose; e.g. a string, number, type, etc.
- A parse tree showing the values of attributes at each node is called an *annotated parse tree*

Attributes of an Identifier

name: character string (obtained from scanner)

scope: program region in which identifier is valid

type:

- integer
- array:
  - number of dimensions
  - upper and lower bounds for each dimension
  - type of elements
- function:
  - number and type of parameters (in order)
  - type of returned value
  - size of stack frame

Scope

- The scope of an identifier (a binding of a name to the entity it names) is the textual part of the program in which the binding is active

- Scope matches identifier declarations with uses
  - Important static analysis step in most languages

Scope (Cont.)

- The *scope* of an identifier is the portion of a program in which that identifier is accessible

- The same identifier may refer to different things in different parts of the program
  - Different scopes for same name don't overlap

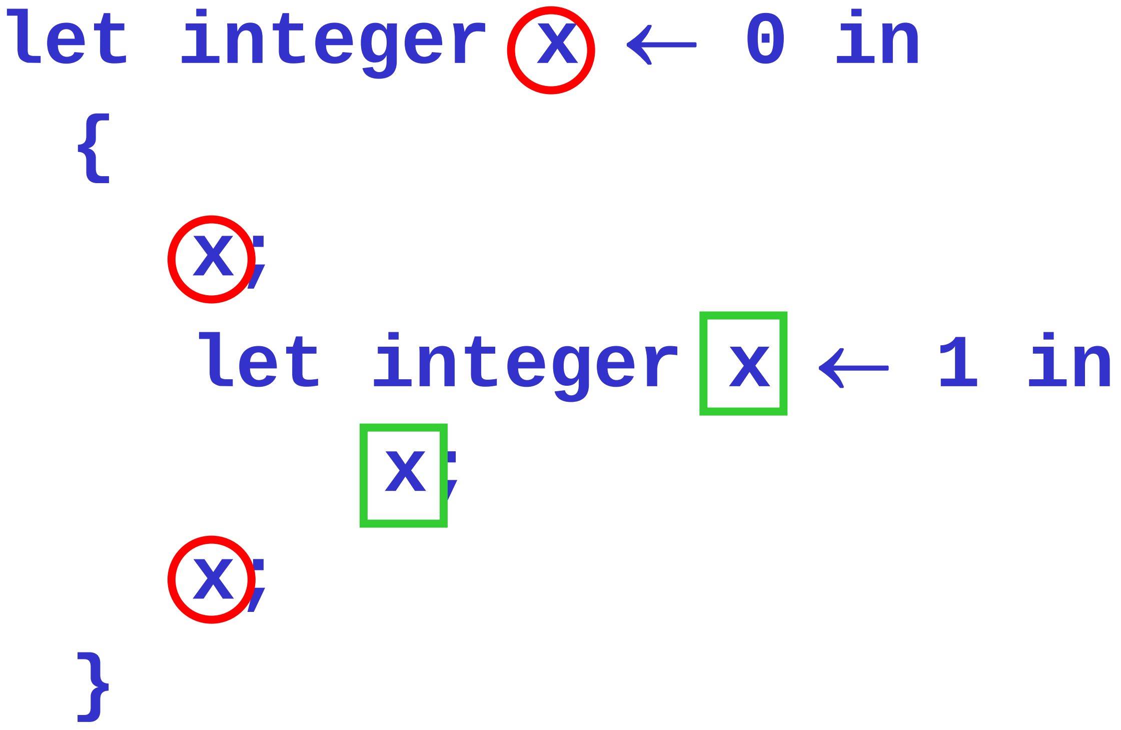
- An identifier may have restricted scope

Static vs. Dynamic Scope

- Most languages have *static* (lexical) scope
  - Scope depends only on the physical structure of program text, not its run-time behavior
  - The determination of scope is made by the compiler
  - C, Java, ML have static scope; so do most languages

- A few languages are *dynamically* scoped
  - Lisp, SNOBOL, Perl
  - Lisp has changed to mostly static scoping
  - Scope depends on execution of the program

Static Scoping Example



```
let integer x ← 0 in
  {
     x;
      let integer x ← 1 in
           x;
     x;
  }
```

Uses of x refer to closest enclosing definition

Dynamic Scope

- A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program

Example

```
g(y) = let integer a ← 42 in f(3);
f(x) = a;
```

- When invoking `g(54)` the result will be `42`

Static vs. Dynamic Scope

```
program scopes (input, output);
var a: integer;
procedure first;
  begin
    a := 1;
  end;
procedure second;
  var a: integer;
  begin
    first;
  end;
begin
  a := 2;  second;  write(a);
end.
```

With static scope rules, it prints 1

With dynamic scope rules, it prints 2

Dynamic Scope (Cont.)

- With dynamic scope, bindings cannot always be resolved by examining the program because they are dependent on calling sequences
- Dynamic scope rules are usually encountered in interpreted languages
- Also, usually these languages do not normally have static type checking:
  - type determination is not always possible when dynamic rules are in effect

Scope of Identifiers

- In most programming languages identifier bindings are introduced by
  - Function declarations (introduce function names)
  - Procedure definitions (introduce procedure names)
  - Identifier declarations (introduce identifiers)
  - Formal parameters (introduce identifiers)

Scope of Identifiers (Cont.)

- Not all kinds of identifiers follow the most-closely nested scope rule

- For example, function declarations
  - often cannot be nested
  - are *globally visible* throughout the program

- With globally visible function names, a function can be used before it is defined

Example: Use Before Definition

```
foo (integer x)
{
  integer y
  y ← bar(x)
  ...
}
bar (integer i): integer
{
  ...
}
```

Other Kinds of Scope

- In most O-O languages, method and attribute names have more sophisticated (static) scope rules

- A method need not be defined in the class in which it is used, but in some parent class

- Methods may also be redefined (overridden)

Implementing the Most-Closely Nested Rule

- Much of semantic analysis can be expressed as a recursive descent of an AST
  - Process an AST node *n*
  - Process the children of *n*
  - Finish processing the AST node *n*

- When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined

Implementing Most-Closely Nesting (Cont.)

- Example:
  - the scope of variable declarations is one subtree

```
let integer x ← 42 in E
```

  - `x` can be used in subtree `E`

Symbol Tables

Purpose: To hold information about identifiers that is computed at some point and looked up at later times during compilation

Examples:

- type of a variable
- entry point for a function

Operations: insert, lookup, delete

Common implementations:

linked lists, search trees, hash tables

Symbol Tables

- Assuming static scope, consider again:

```
let integer x ← 42 in E
```

- Idea:
  - Before processing **E**, add definition of **x** to current definitions, overriding any other definition of **x**
  - After processing **E**, remove definition of **x** and, if needed, restore old definition of **x**

- A *symbol table* is a data structure that tracks the current bindings of identifiers

A Simple Symbol Table Implementation

- Structure is a stack

- Operations

  add_symbol(x) push x and associated info, such as x's type, on the stack

  find_symbol(x) search stack, starting from top, for x. Return first x found or NULL if none found

  remove_symbol() pop the stack

- Why does this work?

Limitations

- The simple symbol table works for variable declarations
  - Symbols added one at a time
  - Declarations are perfectly nested

- Doesn't work for

```
foo(x: integer, x: float);
```

- Other problems?

A Fancier Symbol Table

enter_scope()	start/push a new nested scope
find_symbol(x)	finds current x (or null)
add_symbol(x)	add a symbol x to the table
check_scope(x)	true if x defined in current scope
exit_scope()	exits/pops the current scope

Function/Procedure Definitions

- Function/class names can be used prior to their definition
- We can't check this property
  - using a symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all function/class names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two

Types

- What is a type?
  - This is a subject of some debate
  - The notion varies from language to language

- Consensus
  - A type is a set of values and
  - A set of operations on those values

- Type errors arise when operations are performed on values that do not support that operation

Why Do We Need Type Systems?

Consider the assembly language fragment

```
addi $r1, $r2, $r3
```

What are the types of `$r1, $r2, $r3`?

Types and Operations

- Certain operations are legal only for values of some types
  - It doesn't make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types

- The goal of type checking is to ensure that operations are used with the correct types
    - Enforces intended interpretation of values, because nothing else will!

- Type systems provide a concise formalization of the semantic checking rules

What Can Types do For Us?

- Allow for a more efficient compilation of programs
  - Allocate right amount of space for variables
    - Use fewer bits when possible
  - Select the right machine operations

- Detect statically certain kinds of errors
  - Memory errors
    - Reading from an invalid pointer, etc.
  - Violation of abstraction boundaries
  - Security and access rights violations

Type Checking Overview

Three kinds of languages:

- ***Statically typed***: All or almost all checking of types is done as part of compilation
  - C, C++, ML, Haskell, Java, C#, ...

- ***Dynamically typed***: Almost all checking of types is done as part of program execution
  - Scheme, Prolog, Erlang, Python, Ruby, PHP, Perl, ...

- ***Untyped***: No type checking (machine code)

The Type Wars

- Competing views on static vs. dynamic typing

- Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks

- Dynamic typing proponents say:
  - Static type systems are restrictive
  - Rapid prototyping easier in a dynamic type system

The Type Wars (Cont.)

- In practice, most code is written in statically typed languages with an “escape” mechanism
  - Unsafe casts in C, Java

- It is debatable whether this compromise represents the best or worst of both worlds

Type Checking

Outline

- General properties of type systems
- Types in programming languages
- Notation for type rules
  - Logical rules of inference
- Common type rules

Static Checking

- Refers to the compile-time checking of programs in order to ensure that the semantic conditions of the language are being followed

Examples of static checks include:

- Type checks
- Flow-of-control checks
- Uniqueness checks
- Name-related checks

Static Checking (Cont.)

Flow-of-control checks: statements that cause flow of control to leave a construct must have some place where control can be transferred;
e.g., **break** statements in C

Uniqueness checks: a language may dictate that in some contexts, an entity can be defined exactly once;
e.g., identifier declarations, labels, values in **case** expressions

Name-related checks: Sometimes the same name must appear two or more times;
e.g., in Ada a loop or block can have a name that must then appear both at the beginning and at the end

Types and Type Checking

- A *type* is a set of values together with a set of operations that can be performed on them
- The purpose of *type checking* is to verify that operations performed on a value are in fact permissible
- The type of an identifier is typically available from declarations, but we may have to keep track of the type of intermediate expressions

Type Expressions and Type Constructors

A language usually provides a set of *base types* that it supports together with ways to construct other types using *type constructors*

Through *type expressions* we are able to represent types that are defined in a program

Type Expressions

- A base type is a type expression
- A type name (e.g., a record name) is a type expression
- A type constructor applied to type expressions is a type expression. E.g.,
  - arrays: If T is a type expression and I is a range of integers, then array(I,T) is a type expression
  - records: If T1, …, Tn are type expressions and f1, …, fn are field names, then record((f1,T1),…,(fn,Tn)) is a type expression
  - pointers: If T is a type expression, then pointer(T) is a type expression
  - functions: If T1, …, Tn, and T are type expressions, then so is (T1,…,Tn) →T

Notions of Type Equivalence

Name equivalence: In many languages, e.g. Pascal, types can be given names. Name equivalence views each distinct name as a distinct type. So, two type expressions are name equivalent if and only if they are identical.

Structural equivalence: Two expressions are structurally equivalent if and only if they have the same structure; i.e., if they are formed by applying the same constructor to structurally equivalent type expressions.

Example of Type Equivalence

In the Pascal fragment

```
type nextptr = ^node;
     prevptr = ^node;
var  p : nextptr;
     q : prevptr;
```

p is not name equivalent to q,
but p and q are structurally equivalent.

Static Type Systems & their Expressiveness

- A static type system enables a compiler to detect many common programming errors
- The cost is that some correct programs are disallowed
    - Some argue for dynamic type checking instead
    - Others argue for more expressive static type checking
    - But more expressive type systems are also more complex

Compile-time Representation of Types

- Need to represent type expressions in a way that is both easy to construct and easy to check

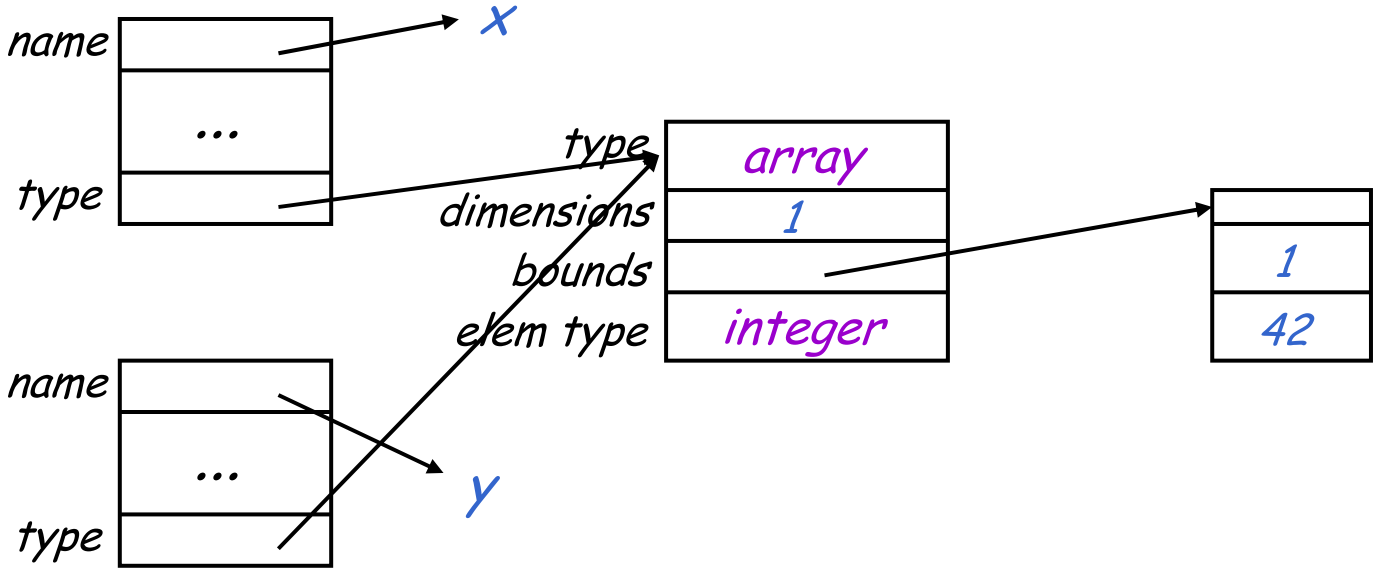
Approach 1: Type Graphs

- Basic types can have predefined "internal values", e.g., small integer values
- Named types can be represented using a pointer into a hash table
- Composite type expressions: the node for f(T1,…,Tn) contains a value representing the type constructor f, and pointers to the nodes for the expressions T1,…,Tn

Compile-time Representation of Types (Cont.)

Example:

```
var x, y : array[1..42] of integer;
```



Compile-Time Representation of Types

Approach 2: Type Encodings

Basic types use a predefined encoding of the low-order bits

BASIC TYPE	ENCODING
boolean	0000
char	0001
integer	0010

The encoding of a type expression op(T) is obtained by concatenating the bits encoding op to the left of the encoding of T. E.g.:

TYPE EXPRESSION	ENCODING
char	00 00 00 0001
array(char)	00 00 01 0001
ptr(array(char))	00 10 01 0001
ptr(ptr(array(char)))	10 10 01 0001

Compile-Time Representation of Types: Notes

- Type encodings are simple and efficient
- On the other hand, named types and type constructors that take more than one type expression as argument are hard to represent as encodings. Also, recursive types cannot be represented directly.

- Recursive types (e.g. lists, trees) are not a problem for type graphs: the graph simply contains a cycle

Types in an Example Programming Language

- Let's assume that types are:
  - integers & floats (base types)
  - arrays of a base type
  - booleans (used in conditional expressions)

- The user declares types for all identifiers

- The compiler infers types for expressions
  - Infers a type for *every* expression

Type Checking and Type Inference

Type Checking is the process of verifying fully typed programs

Type Inference is the process of filling in missing type information

- The two are different, but are often used interchangeably

Rules of Inference

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions (for the lexer)
  - Context-free grammars (for the parser)

- The appropriate formalism for type checking is logical rules of inference

Why Rules of Inference?

- Inference rules have the form

  If Hypothesis is true, then Conclusion is true

- Type checking computes via reasoning

  If E_1 and E_2 have certain types, then E_3 has a certain type

- Rules of inference are a compact notation for "If-Then" statements

From English to an Inference Rule

- The notation is easy to read (with practice)

- Start with a simplified system and gradually add features

- Building blocks:
  - Symbol $\wedge$ is “and”
  - Symbol $\Rightarrow$ is “if-then”
  - $x{:}T$ is “x has type T”

From English to an Inference Rule (2)

If e_1 has type int and e_2 has type int,
then $e_1 + e_2$ has type int

$(e_1$ has type int $\wedge$ e_2 has type int$) \Rightarrow$
$e_1 + e_2$ has type int

$(e_1\text{: int} \wedge e_2\text{: int}) \Rightarrow e_1 + e_2\text{: int}$

From English to an Inference Rule (3)

The statement

$$(e_1\text{: int} \wedge e_2\text{: int}) \Rightarrow e_1 + e_2\text{: int}$$

is a special case of

$$\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n \Rightarrow \text{Conclusion}$$

This is an inference rule

Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\vdash \text{Hypothesis}_1 \quad \ldots \quad \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

- Type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means "it is provable that . . ."

Two Rules

$$\frac{i \text{ is an integer}}{\vdash i : \text{int}} \quad [\text{Int}]$$

$$\frac{\vdash e_1 : \text{int} \qquad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}} \quad [\text{Add}]$$

Two Rules (Cont.)

- These rules give templates describing how to type integers and + expressions

- By filling in the templates, we can produce complete typings for expressions

Example: 1 + 2

$$\dfrac{\dfrac{1 \text{ is an integer}}{\vdash 1 : \text{int}} \qquad \dfrac{2 \text{ is an integer}}{\vdash 2 : \text{int}}}{\vdash 1 + 2 : \text{int}}$$

Soundness

- A type system is *sound* if
  - Whenever $\vdash e : T$
  - Then e evaluates to a value of type T

- We only want sound rules
  - But some sound rules are better than others:

$$\frac{\text{i is an integer}}{\vdash i : \text{number}}$$

  - This rule loses some information

Type Checking Proofs

- Type checking proves facts e: T
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each kind of AST node
- In the type rule used for a node e:
  - Hypotheses are the proofs of types of e's subexpressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

Rules for Constants

$$\frac{}{\vdash \text{true} : \text{bool}} \;[\text{Bool}] \qquad \frac{}{\vdash \text{false} : \text{bool}} \;[\text{Bool}]$$

$$\frac{f \text{ is a floating point number}}{\vdash f : \text{float}} \;[\text{Float}]$$

Two More Rules

$$\frac{\vdash e : \text{bool}}{\vdash \text{not } e : \text{bool}} \quad \text{[Not]}$$

$$\frac{\vdash e_1 : \text{bool} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ do } e_2 : T} \quad \text{[While]}$$

A Problem

- What is the type of a variable reference?

$$\frac{x \text{ is an identifier}}{\vdash x : ?} \quad [\text{Var}]$$

- The local, structural rule does not carry enough information to give x a type

A Solution

- Put more information in the rules!

- A *type environment* gives types for *free* variables
  - A type environment is a function from Identifiers to Types
  - A variable is free in an expression if it is not defined within the expression

Type Environments

Let E be a function from Identifiers to Types

The sentence $E \vdash e : T$

is read:

> Under the assumption that variables have the types given by E, it is provable that the expression e has the type T

Modified Rules

The type environment is added to the earlier rules:

$$\frac{i \text{ is an integer}}{E \vdash i : \text{int}} \quad [\text{Int}]$$

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}} \quad [\text{Add}]$$

New Rules

And we can now write a rule for variables:

$$\frac{E(x) = T}{E \vdash x : T} \quad [\text{Var}]$$

Type Checking of Expressions

Production	*Semantic Rules*

E → id	{ if (declared(id.name)) then
	E.type := lookup(id.name).type
	else E.type := error(); }

E → int	{ E.type := integer; }

E → E1 + E2	{ if (E1.type == integer AND
	E2.type == integer) then
	E.type := integer;
	else E.type := error(); }

Type Checking of Expressions (Cont.)

May have automatic *type coercion*, e.g.

E1.type	E2.type	E.type
integer	integer	integer
integer	float	float
float	integer	float
float	float	float

Type Checking of Statements: Assignment

Semantic Rules:

S → Lval := Rval {check_types(Lval.type,Rval.type)}

Note that in general Lval can be a variable or it may be a more complicated expression, e.g., a dereferenced pointer, an array element, a record field, etc.

Type checking involves ensuring that:

- Lval is a type that can be assigned to, e.g. it is not a function or a procedure
- the types of Lval and Rval are "compatible", i.e, that the language rules provide for coercion of the type of Rval to the type of Lval

Type Checking of Statements: Loops, Conditionals

Semantic Rules:

Loop → while E do S {check_types(E.type,**bool**)}

Cond → if E then S1 else S2
{check_types(E.type,**bool**)}

Run-time Environments

Status

- We have so far covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next come the back-end phases
  - Code generation
  - Optimization
  - Register allocation
  - Instruction scheduling
- We will examine code generation first . . .

Run-time Environments

- Before discussing code generation, we need to understand what we are trying to generate

- There are a number of standard techniques for structuring executable code that are widely used

Outline

- Management of run-time resources

- Correspondence between static (compile-time) and dynamic (run-time) structures

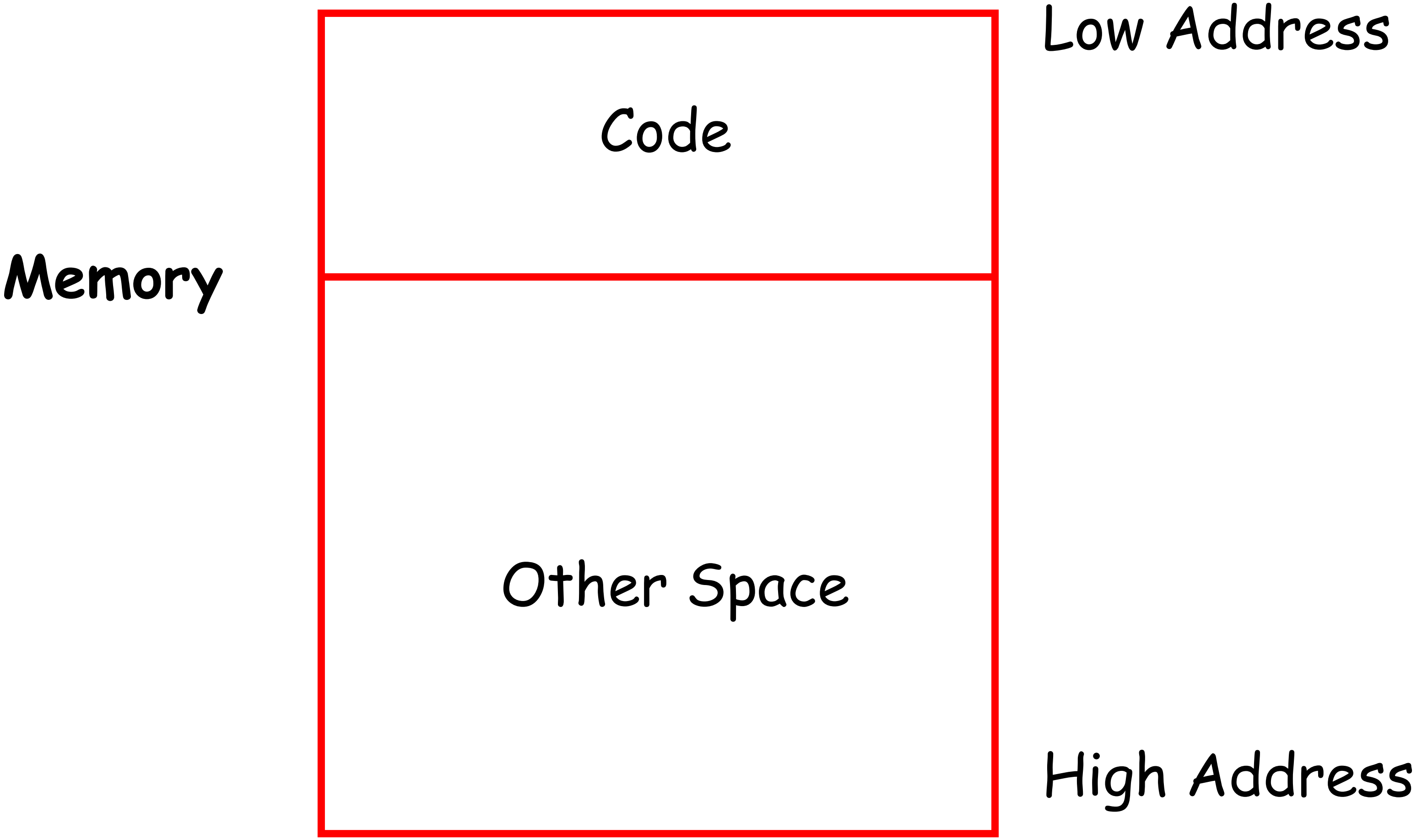
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system (OS)

- When a program is invoked:
  - The OS allocates space for the program
  - The program code is loaded into part of this space
  - The OS jumps to the entry point of the program (i.e., to the beginning of the "`main`" function)

Memory Layout



Low Address

Code

Memory

Other Space

High Address

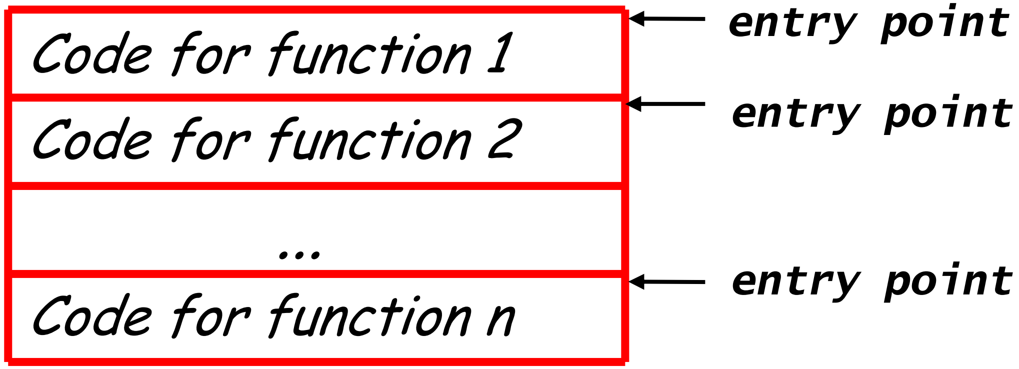
Notes

- By tradition, pictures of run-time memory organization have:
  - Low addresses at the top
  - High addresses at the bottom
  - Lines delimiting areas for different kinds of data

- These pictures are simplifications
  - E.g., not all memory need be contiguous

Organization of Code Space

- Usually, code is generated one function at a time. The code area thus is of the form:



Code for function 1

Code for function 2

...

Code for function n

- Careful layout of code within a function can improve i-cache utilization and give better performance
- Careful attention in the order in which functions are processed can also improve i-cache utilization

What is Other Space?

- Holds all data needed for the program's execution
- Other Space = Data Space
- Compiler is responsible for:
  - Generating code
  - Orchestrating the use of the data area

Code Generation Goals

- Two goals:
  - Correctness
  - Efficiency

- Most complications in code generation come from trying to be efficient as well as correct

Assumptions about Flow of Control

(1) Execution is sequential; at each step, control is at some specific program point and moves from one point to another in a well-defined order

(2) When a procedure is called, control eventually returns to the point immediately after the place where the call was made

Do these assumptions always hold?

Language Issues that affect the Compiler

- Can procedures be recursive?
- What happens to the values of the locals on return from a procedure?
- Can a procedure refer to non-local variables?
- How are parameters to a procedure passed?
- Can procedures be passed as parameters?
- Can procedures be returned as results?
- Can storage be allocated dynamically under program control?
- Must storage be deallocated explicitly?

Activations

- An invocation of procedure P is an *activation* of P

- The *lifetime* of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures that P calls

Lifetimes of Variables

- The *lifetime* of a variable **x** is the portion of execution in which **x** is defined

- Note that:
  - Lifetime is a dynamic (run-time) concept
  - Scope is (usually) a static concept

Activation Trees

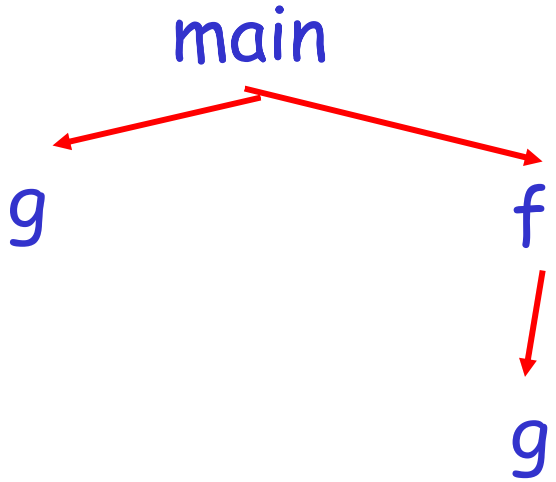
- Assumption (2) requires that when P calls Q, then Q returns before P does

- Lifetimes of procedure activations are thus either disjoint or properly nested

- Activation lifetimes can be depicted as a tree

Example 1

```
g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
```



Example 2

```
g(): int { return 42; }
f(x:int): int {
   if x = 0 then return g();
   else return f(x - 1);
}
main() { f(3); }
```

What is the activation tree for this example?

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input

Since activations are properly nested, a *(control) stack* can track currently active procedures

- push info about an activation at the procedure entry
- pop the info when the activation ends; i.e., at the return from the call

Example

```
g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
```

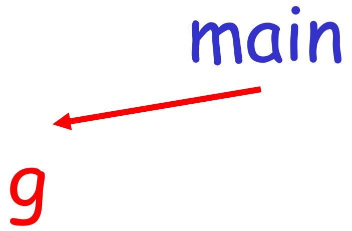
main

Stack

main

Example

```
g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
```



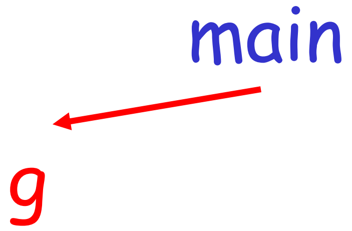
Stack

main

g

Example

```
g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
```


main

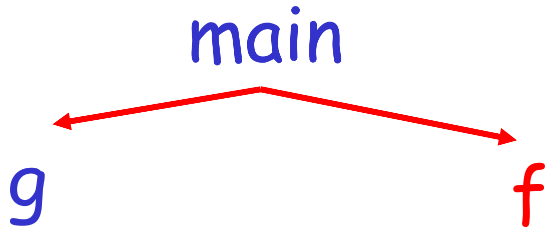
g

Stack

main

Example

```
g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
```



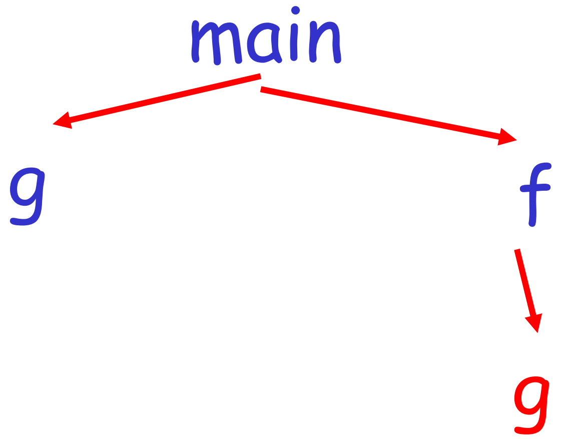
Stack

main

f

Example

```
g(): int { return 42; }
f(): int { return g(); }
main() { g(); f(); }
```



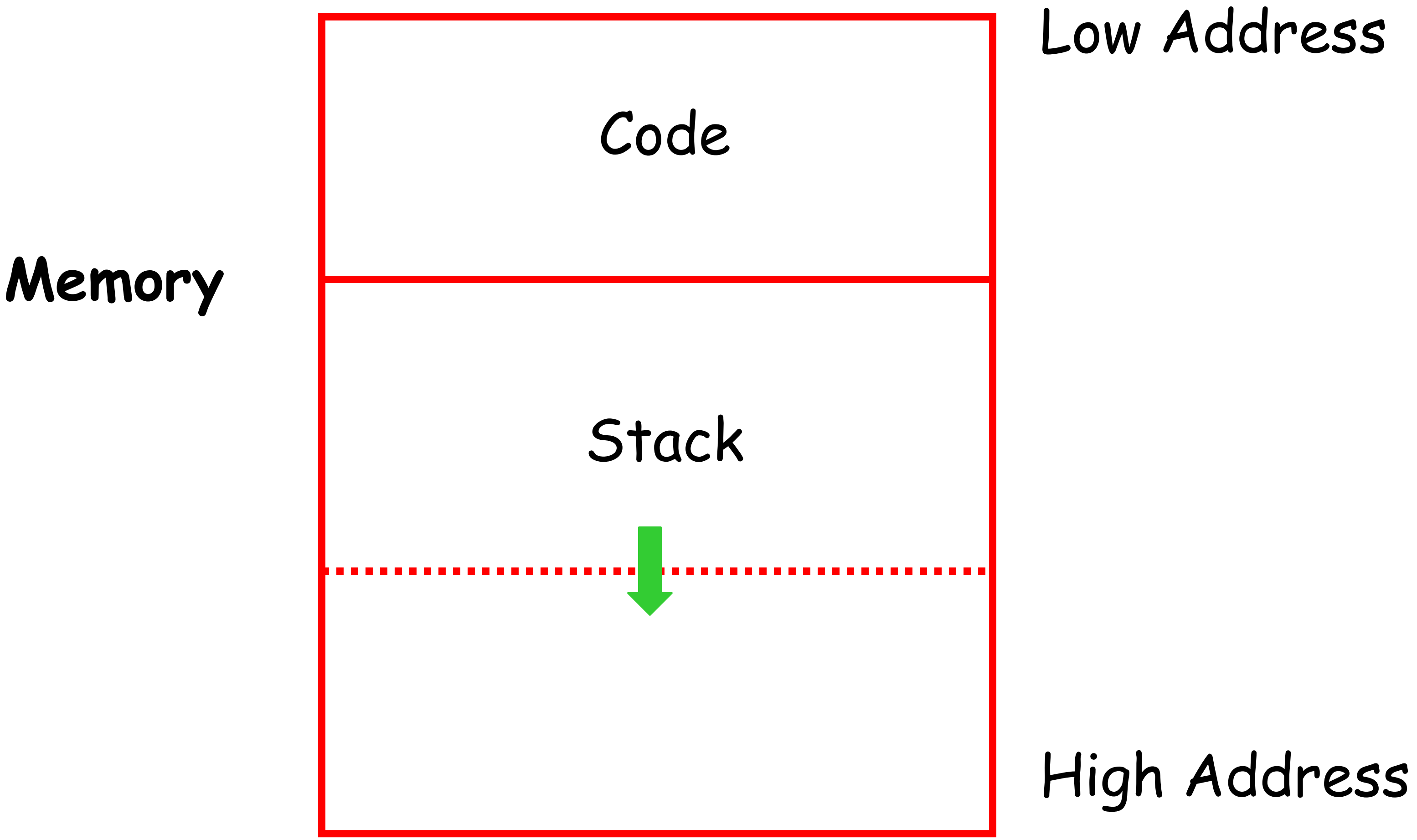
Stack

main

f

g

Revised Memory Layout



Activation Records

- The information needed to manage a single procedure activation is called an *activation record* (AR) or a *stack frame*

- If a procedure F calls G, then G's activation record contains a mix of info about F and G

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes. G's AR contains information needed to resume execution of F.

- G's AR may also contain:
  - G's return value (needed by F)
  - Actual parameters to G (supplied by F)
  - Space for G's local variables

The Contents of a Typical AR for *G*

- Space for *G*'s return value
- Actual parameters
- (optional) Pointer to the previous activation record
  - The *control link;* points to the AR of caller of *G*
- (optional) *Access link* for access to non-local names
  - Points to the AR of the function that statically contains *G*
- Machine status prior to calling *G*
  - Return address, values of registers & program counter
  - Local variables
- Other temporary values used during evaluation

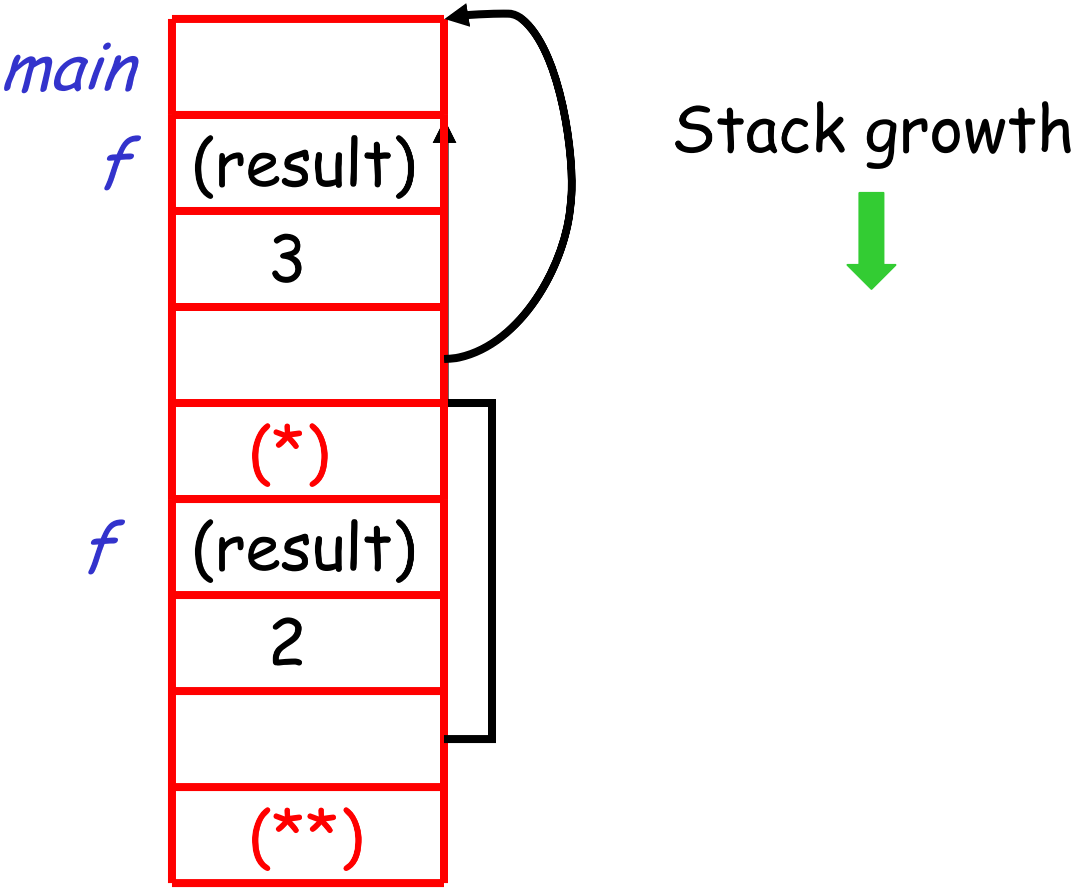
Example 2, Revisited

```
g(): int { return 42; }
f(x:int): int {
   if x = 0 then return g();
   else return f(x - 1);(**)
}
main() { f(3);(*) }
```

AR for f:

result
argument
control link
return address

Stack After Two Calls to *f*



Notes

- `main()` has no argument or local variables and returns no result; its AR is uninteresting
- (*) and (**) are return addresses (continuation points) of the invocations of `f()`
  - The return address is where execution resumes after a procedure call finishes
- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

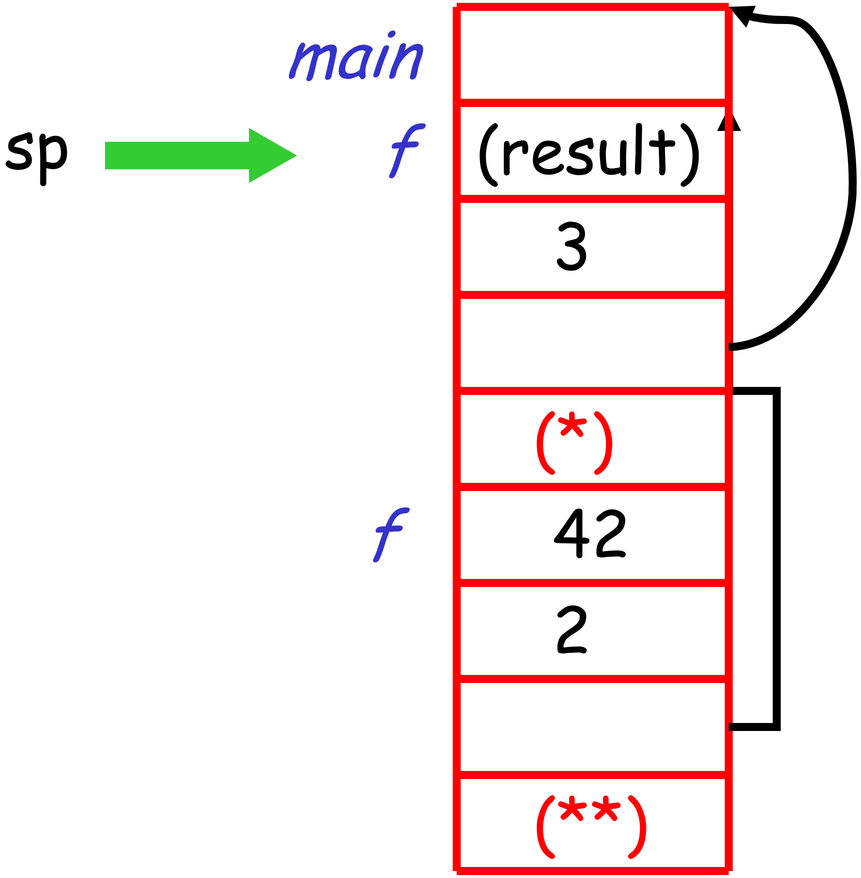
The Main Point

- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record (as displacements from **sp**)

Thus, the AR layout and the code generator must be designed together!

Example 2, continued

The picture shows the state after the call to the 2nd invocation of **f()** returns



Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame

- There is nothing magical about this run-time organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation

Discussion (Cont.)

- Real compilers hold as much of the frame as possible in registers
  - Especially the function result and (some of) the arguments

Storage Allocation Strategies for Activation Records (1)

Static Allocation (Fortran 77)

- Storage for all data objects is laid out at compile time
- Can be used only if size of data objects and constraints on their position in memory can be resolved at compile time ⇒ no dynamic structures
- Recursive procedures are restricted, since all activations of a procedure must share the same locations for local names

Storage Allocation Strategies for Activation Records (2)

Stack Allocation (Pascal, C)

- Storage organized as a stack
- Activation record pushed when activation begins and popped when it ends
- Cannot be used if the values of local names must be retained when the evaluation ends or if the called invocation outlives the caller

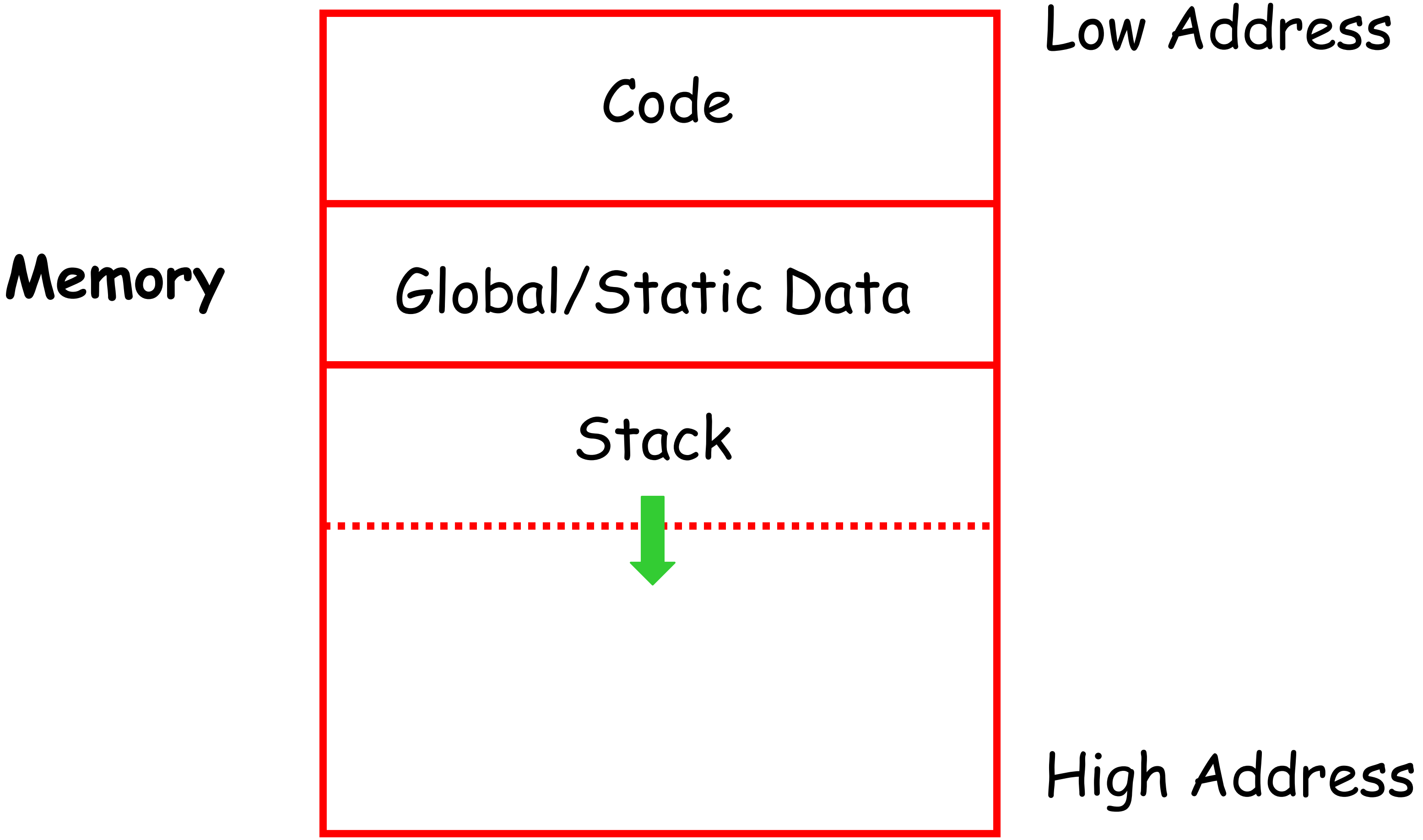
Heap Allocation (Lisp, ML)

- Activation records may be allocated and deallocated in any order
- Some form of garbage collection is needed to reclaim free space

Global Variables

- All references to a global variable point to the same object
  - Can't store a global in an activation record
- Globals are assigned a fixed address once
  - Variables with fixed address are "statically allocated"
- Depending on the language, there may be other statically allocated values
  - e.g., static variables in C

Memory Layout with Static Data



Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR

```
foo() { new bar; }
```

The **bar** value must survive deallocation of **foo**'s AR

- Languages with dynamically allocated data use a *heap* to store dynamic data

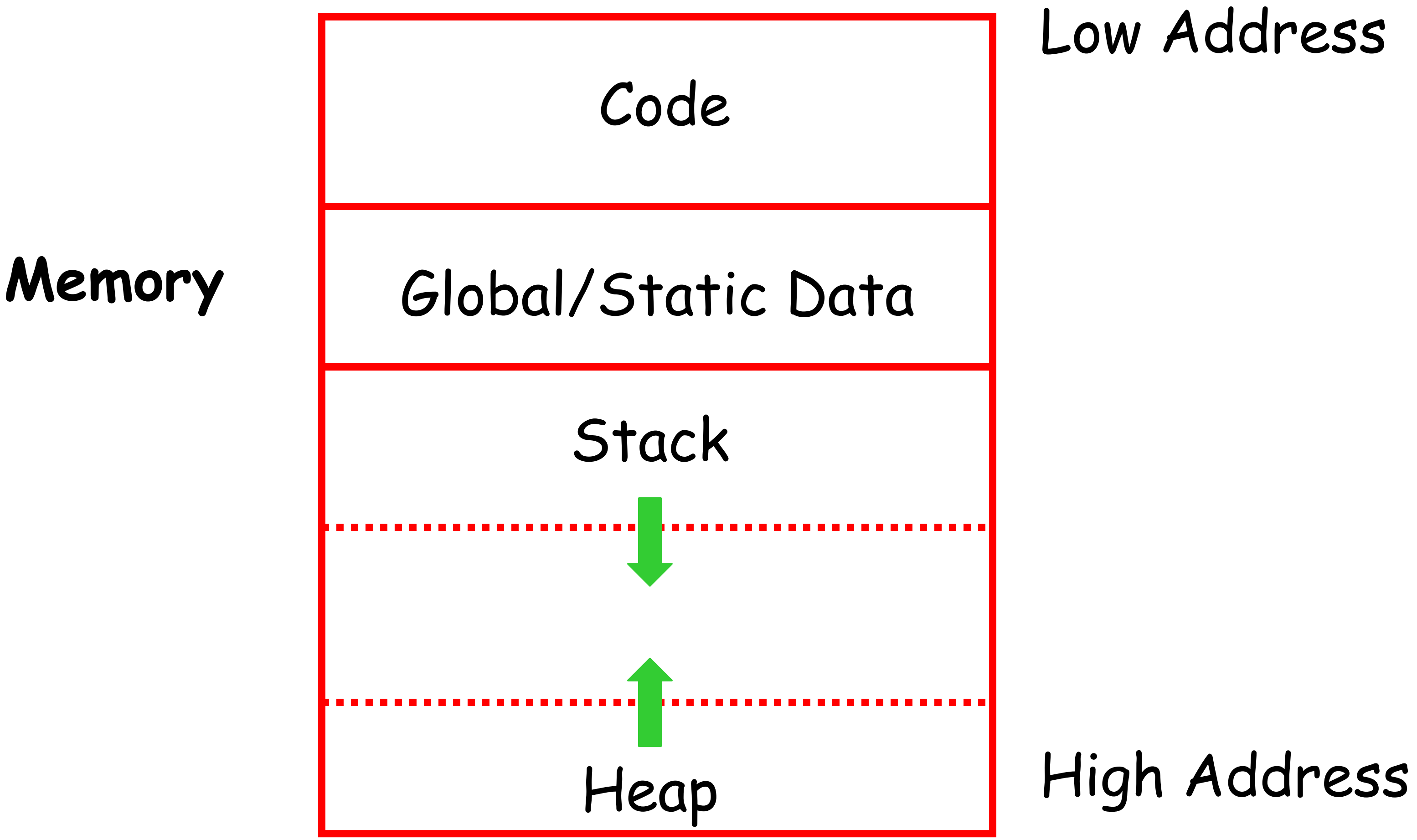
Review of Runtime Organization

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually has fixed size, contains locals
- The heap contains all other data
  - In C, heap is managed explicitly by **`malloc()`** and **`free()`**
  - In Java, heap is managed by **`new()`** and garbage collection
  - In ML, both allocation and deallocation in the heap is managed implicitly

Notes

- Both the heap and the stack grow
- Must take care so that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let them grow towards each other

Memory Layout with Heap



Data Layout

- Low-level details of computer architecture are important in laying out data for correct code and maximum performance

- Chief among these concerns is *alignment* of data

Alignment

- Most modern machines are 32 or 64 bit
  - 8 bits in a byte
  - 4 or 8 bytes in a word
  - Machines are either byte or word addressable
- Data is *word-aligned* if it begins at a word boundary

Most machines have some alignment restrictions

(Or performance penalties for poor alignment)

Alignment (Cont.)

Example: An ASCII string:

`"Hello"`

Takes 5 characters (without the terminating \0)

- To word-align next datum on a 32-bit machine, add 3 “padding” characters to the string
- The padding is not part of the string, it’s just unused memory

Code Generation

Main Idea of (First Half of) Today's Lecture

We can emit stack-machine-style code for expressions via recursion.

(We will use MIPS assembly as our target language.)

Lecture Outline

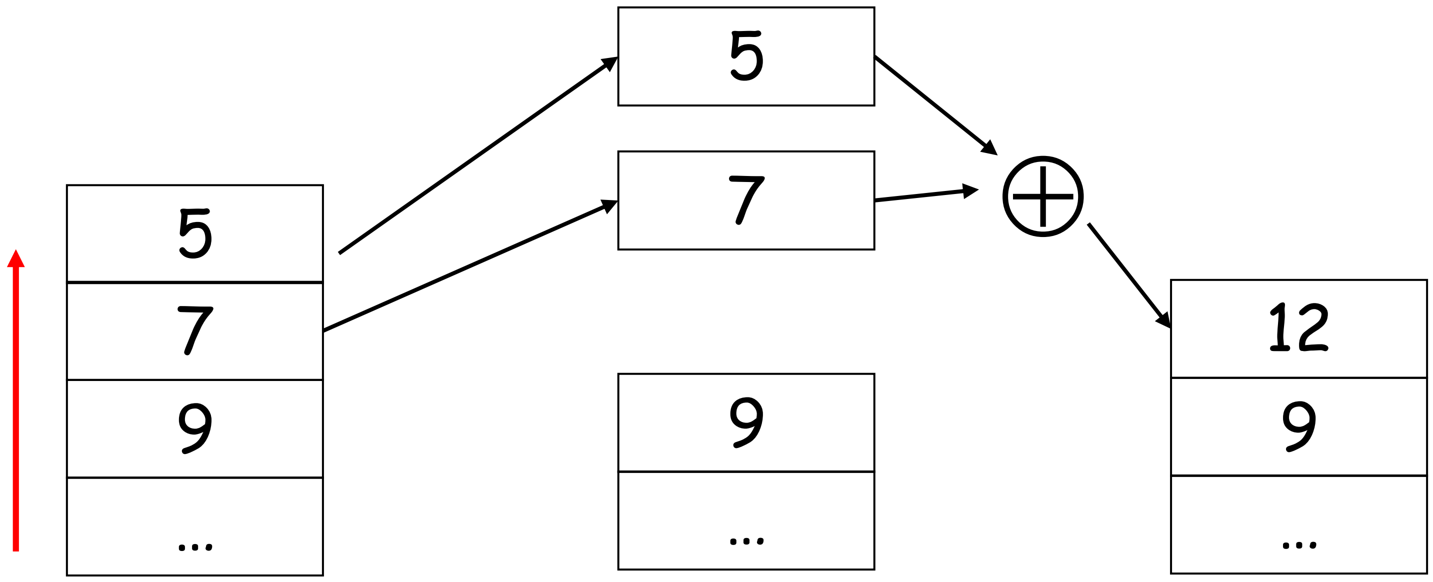
- What are stack machines?
- The MIPS assembly language.
- A simple source language ("**Mini Bar**").
- A stack machine implementation of the simple language.
- Pushing and popping activation records.
- Placing temporaries in the activation record.

Stack Machines

- A simple evaluation model.
- No variables or registers.
- A stack of values for intermediate results.
- Each instruction:
  - Takes its operands from the top of the stack.
  - Removes those operands from the stack.
  - Computes the required operation on them.
  - Pushes the result onto the stack.

Example of Stack Machine Operation

The addition operation on a stack machine:



pop add push

Example of a Stack Machine Program

- Consider another machine with two instructions
  - push i - place the integer i on top of the stack.
  - add - pop topmost two elements, add them and put the result back onto the stack.
- A program to compute 7 + 5:

```
push 7
push 5
add
```

Why Use a Stack Machine?

- Each operation takes operands from the same place and puts results in the same place.
- This means a uniform compilation scheme.
- Therefore, a simpler compiler.

Why Use a Stack Machine?

- Location of the operands is implicit.
  - Always on the top of the stack.
- No need to specify operands explicitly.
- No need to specify the location of the result.
- Instruction is "add" as opposed to "add r_1, r_2" (or "add r_d r_{i1} r_{i2}").

  ⇒ Smaller encoding of instructions.

  ⇒ More compact programs.
- This is one of the reasons why Java Bytecode uses a stack evaluation model.

Optimizing the Stack Machine

- The add instruction does 3 memory operations:
  - Two reads and one write to the stack.
  - The top of the stack is frequently accessed.
- Idea: keep the top of the stack in a dedicated register (called the "accumulator").
  - Register accesses are faster (why?)
- The "add" instruction is now:

  acc ← acc + top_of_stack

  which performs only one memory operation!

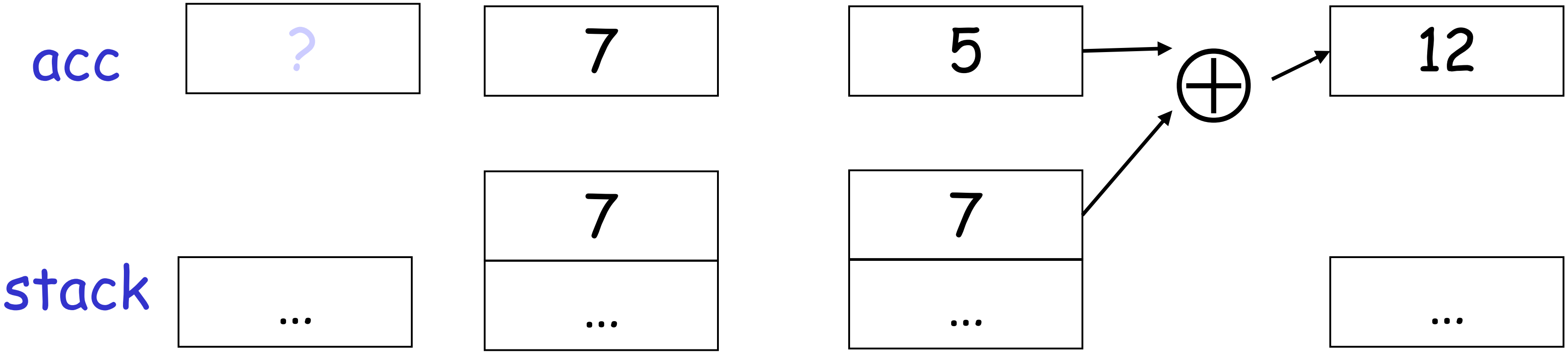
Stack Machine with Accumulator

Invariants

- The result of computing an expression is always placed in the accumulator.
- For an operation $op(e_1,\ldots,e_n)$ compute each e_i and then push the accumulator (= the result of evaluating e_i) onto the stack.
- After the operation pop n-1 values.
- After computing an expression the stack is as before.

Stack Machine with Accumulator: Example

Compute 7 + 5 using an accumulator:



acc ← 7
push acc

acc ← 5

acc ← acc + top_of_stack
pop

A Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
	?	<init>
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

Notes

- It is very important that the stack is preserved across the evaluation of a subexpression.
  - Stack before the evaluation of 7 + 5 is 3, <init>
  - Stack after the evaluation of 7 + 5 is 3, <init>
  - The first operand is on top of the stack.

From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator.

- We want to run the resulting code on the MIPS processor (or simulator).

- We simulate the stack machine instructions using MIPS instructions and registers.

Simulating a Stack Machine on the MIPS…

- The accumulator is kept in MIPS register $a0.
- The stack is kept in memory.
- The stack grows towards lower addresses.

  (Standard convention on the MIPS architecture.)
- The address of the next location on the stack is kept in MIPS register $sp.
  - Guess: what does "sp" stand for?
  - The top of the stack is at address $sp + 4.

MIPS Assembly

MIPS architecture

- Prototypical Reduced Instruction Set Computer (RISC) architecture.
- Arithmetic operations use registers for operands and results.
- Must use load and store instructions to fetch operands and store results in memory.
- 32 general purpose registers (32 bits each).
  - We will use $sp, $a0 and $t1 (a temporary register).

A Sample of MIPS Instructions

- lw reg_1 offset(reg_2) “load word“
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg_1 offset(reg_2) “store word“
  - Store 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1 reg_2 imm “add immediate“
  - $reg_1 \leftarrow reg_2 + imm$
  - “u” means overflow is not checked
- li reg imm “load immediate“
  - reg ← imm

MIPS Assembly: Example

- The stack-machine code for 7 + 5 in MIPS:

```
acc ← 7                         li $a0 7
push acc                        sw $a0 0($sp)
                                addiu $sp $sp -4
acc ← 5                         li $a0 5
acc ← acc + top_of_stack        lw $t1 4($sp)
                                add $a0 $a0 $t1
pop                             addiu $sp $sp 4
```

- We now generalize this to a simple language…

A Small Language

- A language with only integers and integer operations ("**Mini Bar**").

$$
\begin{aligned}
P &\to F\ P \mid F \\
F &\to \text{id(ARGS) begin } E \text{ end} \\
\text{ARGS} &\to \text{id, ARGS} \mid \text{id} \\
E &\to \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \\
&\quad \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id(ES)} \\
\text{ES} &\to E\text{, ES} \mid E
\end{aligned}
$$

A Small Language (Cont.)

- The first function definition f is the "main" routine.
- Running the program on input i means computing f(i).
- Program for computing the Fibonacci numbers:

```
fib(x)
begin
   if x = 1 then 0 else
   if x = 2 then 1 else fib(x - 1) + fib(x – 2)
end
```

Code Generation Strategy

- For each expression e we generate MIPS code that:
  - Computes the value of e in $a0
  - Preserves $sp and the contents of the stack

- We define a code generation function cgen(e) whose result is the code generated for e
  - cgen(e) will be recursive

Code Generation for Constants

- The code to evaluate an integer constant simply copies it into the accumulator:

cgen(int) = li $a0 int

- Note that this also preserves the stack, as required.

Code Generation for Addition

```
cgen($e_1$ + $e_2$) =
      cgen($e_1$)              ; $a0 ← value of $e_1$
      sw $a0 0($sp)            ; push that value
      addiu $sp $sp -4         ; onto the stack
      cgen($e_2$)              ; $a0 ← value of $e_2$
      lw $t1 4($sp)            ; grab value of $e_1$
      add $a0 $t1 $a0          ; do the addition
      addiu $sp $sp 4          ; pop the stack
```

Possible optimization:

Put the result of e_1 directly in register $t1?

Code Generation for Addition: Wrong Attempt!

Optimization: Put the result of e_1 directly in $t1?

```
cgen(e₁ + e₂) =
        cgen(e₁)            ; $a0 ← value of e₁
        move $t1 $a0        ; save that value in $t1
        cgen(e₂)            ; $a0 ← value of e₂
                            ; may clobber $t1
        add $a0 $t1 $a0     ; perform the addition
```

Try to generate code for : 3 + (7 + 5)

move reg_1 reg_2 is a MIPS pseudo-instruction (alias for add reg_1 reg_2 $zero)

Code Generation Notes

- The code for $e_1 + e_2$ is a template with "holes" for code for evaluating e_1 and e_2.
- Stack machine code generation is recursive.
- Code for $e_1 + e_2$ consists of code for e_1 and e_2 glued together.
- Code generation can be written as a recursive-descent of the AST.
  - At least for (arithmetic) expressions.

Code Generation for Subtraction and Constants

New instruction: sub reg_1 reg_2 reg_3

Implements $\text{reg}_1 \leftarrow \text{reg}_2 - \text{reg}_3$

cgen(e_1 - e_2) =

```
cgen(e1)              ; $a0 ← value of e1
sw $a0 0($sp)         ; push that value
addiu $sp $sp -4      ; onto the stack
cgen(e2)              ; $a0 ← value of e2
lw $t1 4($sp)         ; grab value of e1
sub $a0 $t1 $a0       ; do the subtraction
addiu $sp $sp 4       ; pop the stack
```

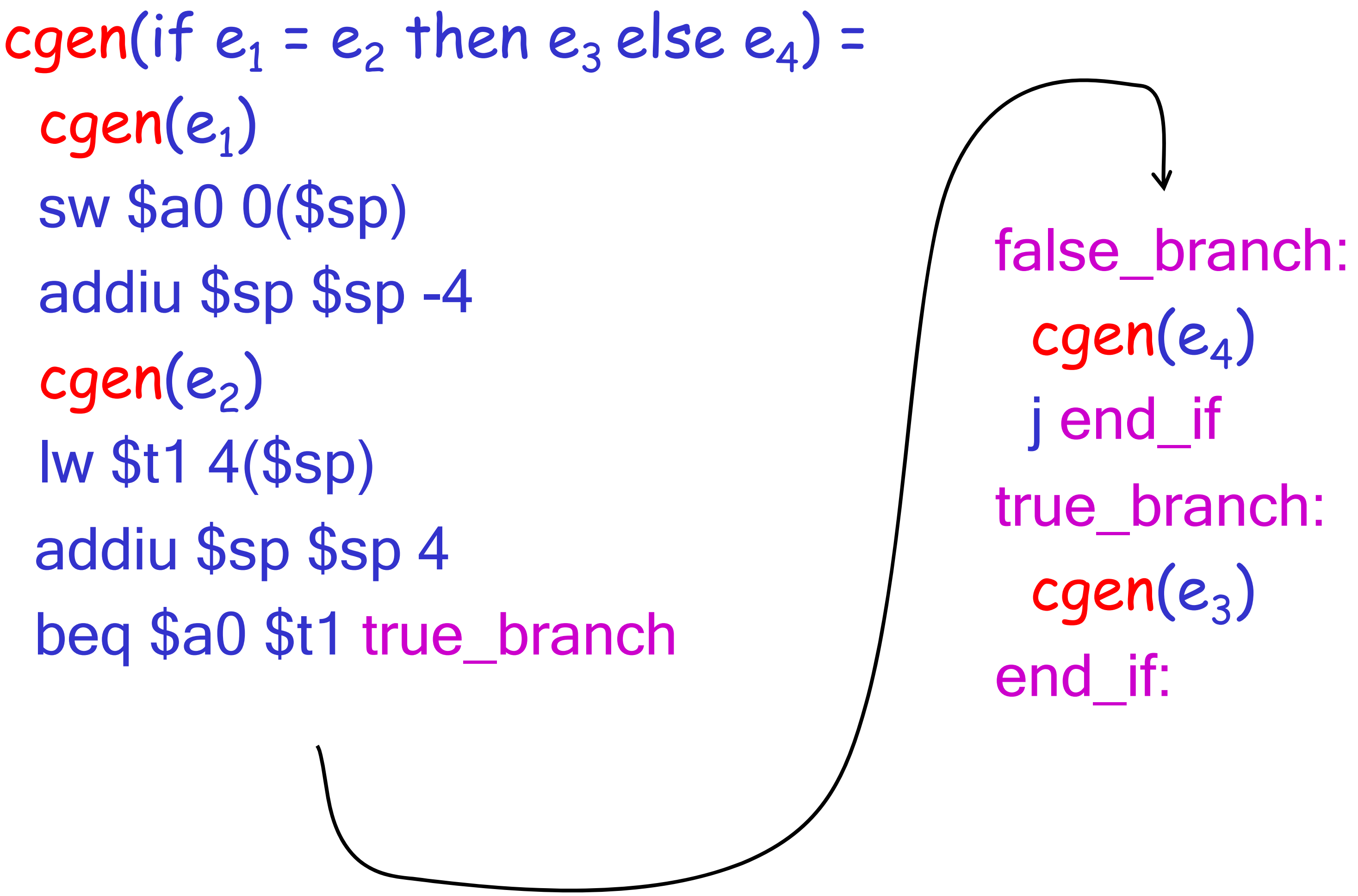
Code Generation for Conditional

We need control flow instructions.

- New MIPS instruction: beq reg_1 reg_2 label
  - Branch to label if $\text{reg}_1 = \text{reg}_2$

- New MIPS instruction: j label
  - Unconditional jump to label

Code Generation for If (Cont.)



```
cgen(if $e_1$ = $e_2$ then $e_3$ else $e_4$) =
  cgen($e_1$)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen($e_2$)
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch

false_branch:
  cgen($e_4$)
  j end_if
true_branch:
  cgen($e_3$)
end_if:
```

Meet The Activation Record

- Code for function calls and function definitions depends on the layout of the activation record (or "AR").
- A very simple AR suffices for this language:
  - The result is always in the accumulator.
    - No need to store the result in the AR.
  - The activation record holds actual parameters.
    - For $f(x_1,\ldots,x_n)$ push the arguments $x_n,\ldots,x_1$ onto the stack.
    - These are the only variables in this language.

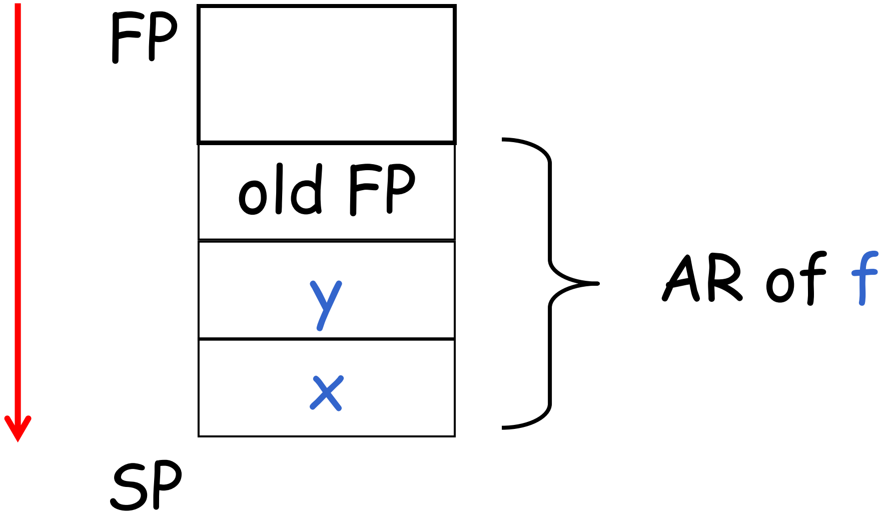
Meet The Activation Record (Cont.)

- The stack discipline guarantees that on function exit, $sp is the same as it was before the args got pushed (i.e., before function call).
- We need the return address.
- It's also handy to have a pointer to the current activation.
  - This pointer lives in register $fp (frame pointer).
  - Reason for frame pointer will become clear shortly (at least I hope!).

Layout of the Activation Record

Summary: For this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices.

Picture: Consider a call to f(x,y), the AR will be:



Code Generation for Function Call

- The calling sequence is the sequence of instructions (of both *caller* and *callee*) to set up a function invocation.
- New instruction: jal label
  - Jump to label, save address of next instruction in special register $ra.
  - On other architectures the return address is stored on the stack by the "call" instruction.

Code Generation for Function Call (Cont.)

```
cgen(f(e1,…,en)) =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen(en)
    sw $a0 0($sp)
    addiu $sp $sp -4
    …
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry
```

- The caller saves the value of the frame pointer.
- Then it pushes the actual parameters in reverse order.
- The caller's jal puts the return address in register $ra.
- The AR so far is 4*n+4 bytes long.

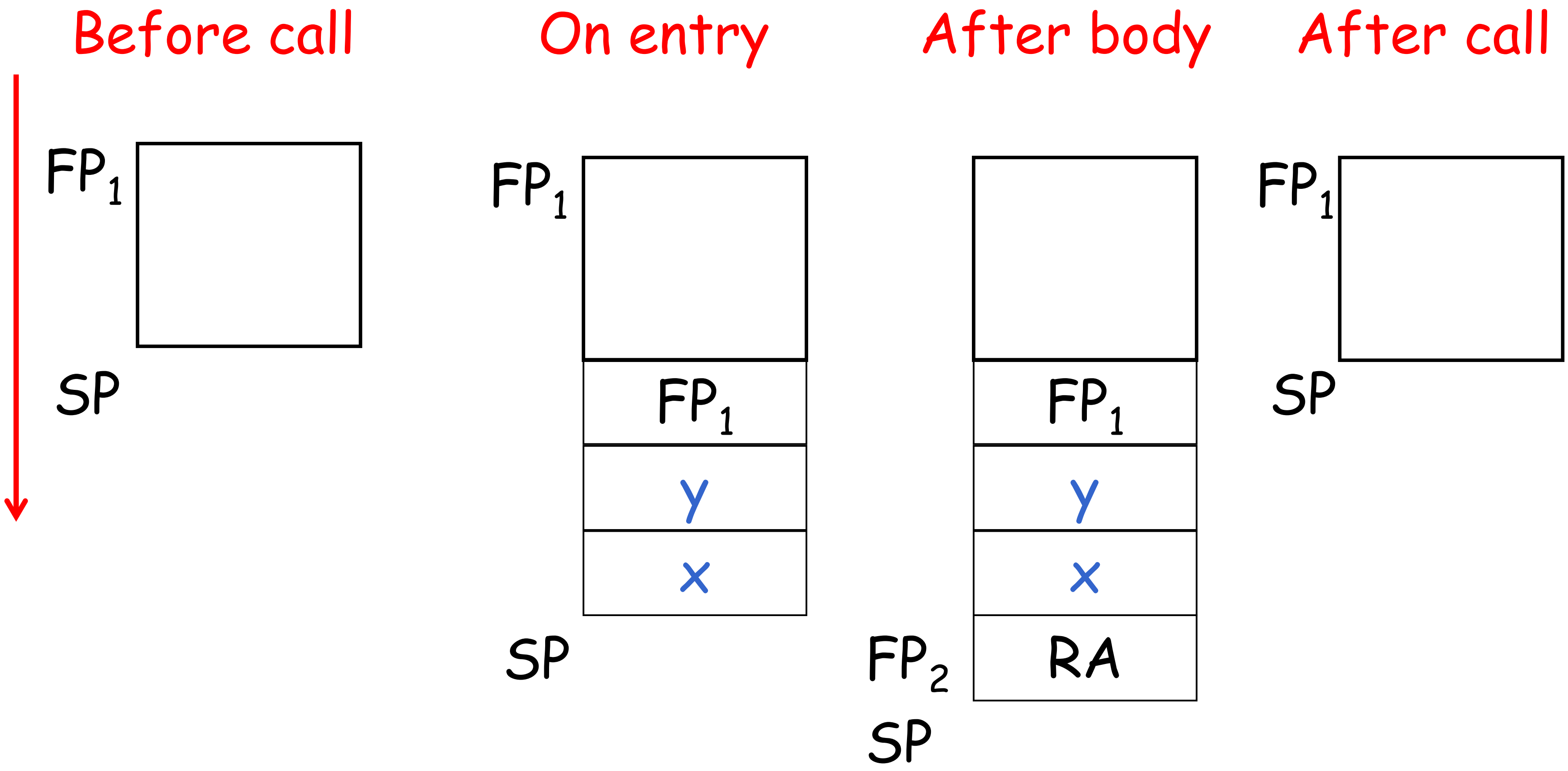
Code Generation for Function Definition

- New MIPS instruction: jr reg
  - Jump to address in register reg

```
cgen(f(x_1,…,x_n) begin e end) =
 f_entry:
   move $fp $sp
   sw $ra 0($sp)
   addiu $sp $sp -4
   cgen(e)
   lw $ra 4($sp)
   addiu $sp $sp frame_size
   lw $fp 0($sp)
   jr $ra
```

- Note: The frame pointer points to the top, not bottom of the frame.
- Callee saves old return address, evaluates its body, pops the return address, pops the arguments, and then restores $fp.
- frame_size = 4*n + 8

Calling Sequence: Example for f(x,y)



Code Generation for Variables/Parameters

- Variable references are the last construct.
- The “variables” of a function are just its parameters.
  - They are all in the AR.
  - Pushed there by the caller.

- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp.

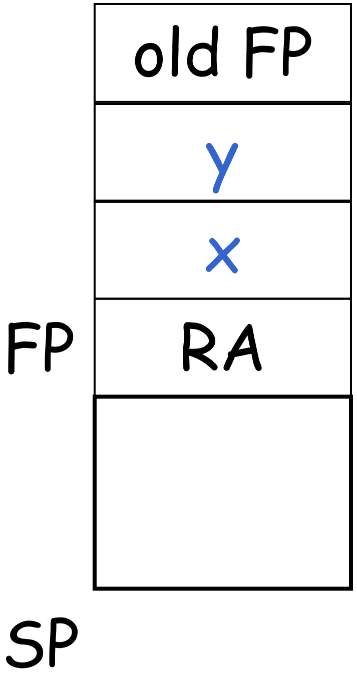
Code Generation for Variables/Parameters

- Solution: use the frame pointer!
  - Always points to the return address on the stack.
  - Since it does not move, it can be used to find the variables.
- Let x_i be the ith (i = 1,…,n) formal parameter of the function for which code is generated.

cgen(x_i) = lw $a0 offset($fp)   (offset = 4*i)

Code Generation for Variables/Parameters

- Example: For a function f(x,y) begin e end the activation and frame pointer are set up as follows (when evaluating e):



- x is at $fp + 4
- y is at $fp + 8

Activation Record & Code Generation Summary

- The activation record must be designed together with the code generator.

- Code generation can be done by recursive traversal of the AST.

Discussion

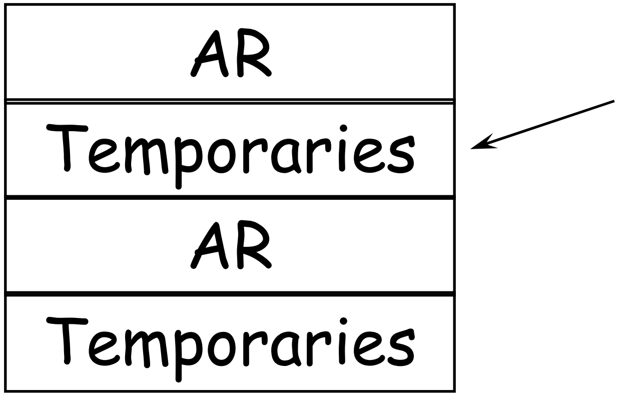
- Production compilers do different things.
  - Emphasis is on keeping values (esp. current stack frame) in registers.
  - Intermediate results are laid out in the AR, not pushed and popped from the stack.
  - As a result, code generation is often performed in synergy with register allocation.

Next slides: code generation for temporaries.

An Optimization:
Temporaries in the Activation Record

Review

- The stack machine has activation records and intermediate results interleaved on the stack
- The code generator must assign a location in the AR for each temporary



These get put here when we evaluate compound expressions like $e_1 + e_2$ (need to store e_1 while evaluating e_2)

Review (Cont.)

- Advantage: Simple code generation.
- Disadvantage: Slow code.
  - Storing/loading temporaries requires a store/load and $sp adjustment.

```
cgen($e_1$ + $e_2$) = cgen($e_1$)           ; eval $e_1$
                  sw $a0 0($sp)         ; save its value
                  addiu $sp $sp -4      ; adjust $sp (!)
                  cgen($e_2$)           ; eval $e_2$
                  lw $t1 4($sp)         ; get $e_1$
                  add $a0 $t1 $a0       ; $a0 = $e_1$ + $e_2$
                  addiu $sp $sp 4       ; adjust $sp (!)
```

An Optimization

- Idea: Predict how $sp will move at run time.
  - Do this prediction at compile time.
  - Move $sp to its limit, at the beginning.

- The code generator must *statically* assign a location in the AR for each temporary.

Improved Code

Old method

```
$cgen(e_1 + e_2)$ =
    $cgen(e_1)$
    sw $a0 0($sp)
    addiu $sp $sp -4
    $cgen(e_2)$
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

New idea

```
$cgen(e_1 + e_2)$ =
    $cgen(e_1)$
    sw $a0 ?($fp)

    $cgen(e_2)$
    lw $t1 ?($fp)
    add $a0 $t1 $a0
```

statically allocate

Example

```
add(w,x,y,z)
begin
    x + (y + (z + (w + 42)))
end
```

- What intermediate values are placed on the stack?

- How many slots are needed in the AR to hold these values?

How Many Stack Slots?

- Let $NS(e)$ = # of slots needed to evaluate e.
  - *Includes* slots for arguments to functions.

- E.g: $NS(e_1 + e_2)$
  - Needs at least as many slots as $NS(e_1)$.
  - Needs at least one slot to hold e_1, plus as many slots as $NS(e_2)$, i.e. $1 + NS(e_2)$.

- Space used for temporaries in e_1 can be reused for temporaries in e_2.

The Equations for the "Mini Bar" Language

$NS(e_1 + e_2) = \quad \max(NS(e_1), 1 + NS(e_2))$

$NS(e_1 - e_2) = \quad \max(NS(e_1), 1 + NS(e_2))$

$NS(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$

$\quad \max(NS(e_1), 1 + NS(e_2), NS(e_3), NS(e_4))$

$NS(f(e_1,\ldots,e_n)) =$

$\quad \max(NS(e_1), 1 + NS(e_2), 2 + NS(e_3), \ldots , (n-1) + NS(e_n), n)$

$NS(\text{int}) = \quad 0$

$NS(\text{id}) = \quad 0$

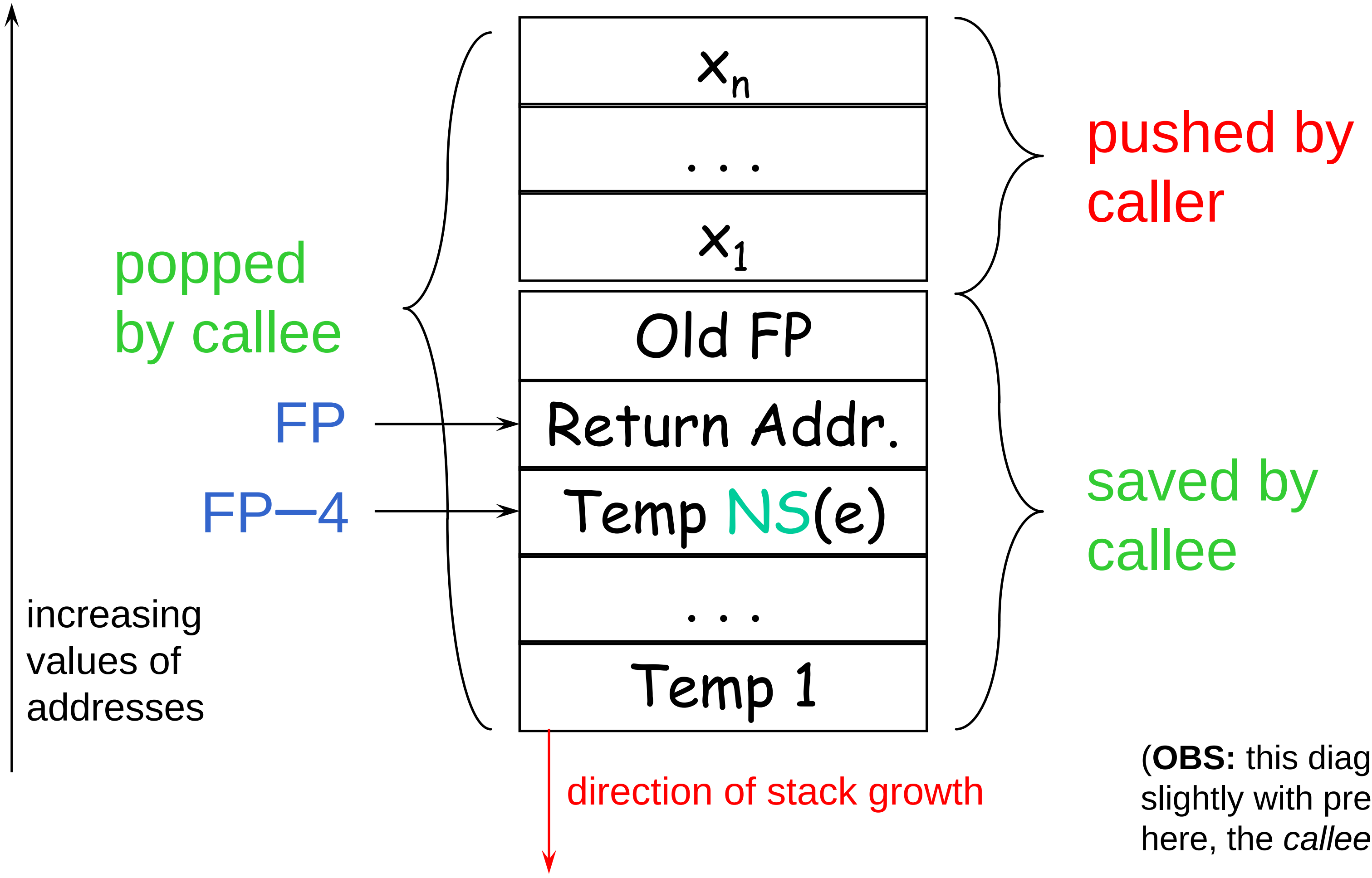
Rule for $f(e_1, \ldots , e_n)$: Each time we evaluate an argument, we put it on the stack.

The Revised Activation Record

- For a function definition $f(x_1,\ldots,x_n)$ begin e end the AR has 2 + NS(e) elements
  - Return address
  - Frame pointer
  - NS(e) locations for intermediate results

- Note that f's arguments are now considered to be part of its *caller's* AR.

Picture: Activation Record



(**OBS:** this diagram disagrees slightly with previous lecture: here, the *callee* saves FP)

Revised Code Generation

- Code generation must know how many slots are in use at each point.

- Add a new argument to code generation: the position of the *next available* slot.

Improved Code

Old method

$cgen(e_1 + e_2) =$

$cgen(e_1)$

```
sw $a0 0($sp)
addiu $sp $sp -4
```

$cgen(e_2)$

```
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

New method

$cgen(e_1 + e_2, \underline{ns}) =$

$cgen(e_1, \underline{ns})$

```
sw $a0 ns($fp)
```

$cgen(e_2, \underline{ns+4})$

```
lw $t1 ns($fp)
add $a0 $t1 $a0
```

compile-time prediction

static allocation

Notes

- The slots for temporary values are still used like a stack, but we predict usage at compile time.
  - This saves us from doing that work at run time.
  - Allocate all needed slots at start of a function.

Intermediate Code & Local Optimizations

Lecture Outline

- What is “Intermediate code” ?
- Why do we need it?
- How to generate it?
- How to use it?
- Local optimization

Code Generation Summary

- We have so far discussed:
  - Runtime organization.
  - Simple stack machine code generation.
  - Improvements to stack machine code generation.
- Our compiler goes directly from the abstract syntax tree (AST) to assembly language...
  - ... and does not perform optimizations.

Most real compilers use intermediate languages.

Why Intermediate Languages?

ISSUE: Reduce code complexity

- Multiple front-ends
  - gcc can handle C, C++, Java, Fortran, Ada, ...
  - each front-end translates source to the same generic language (called GENERIC).
- Multiple back-ends
  - gcc can generate machine code for various target architectures: x86, x86_64, SPARC, ARM, …
- **One Icode to bridge them!**
  - Do most optimization on intermediate representation before emitting machine code.

Why Intermediate Languages?

ISSUE: When to perform optimizations

- On abstract syntax trees
  - Pro: Machine independent
  - Con: Too high level
- On assembly language
  - Pro: Exposes most optimization opportunities
  - Con: Machine dependent
  - Con: Must re-implement optimizations when re-targeting
- On an intermediate language
  - Pro: Exposes optimization opportunities
  - Pro: Machine independent

Kinds of Intermediate Languages

High-level intermediate representations:

- closer to the source language (structs, arrays)
- easy to generate from the input program
- code optimizations may not be straightforward

Low-level intermediate representations:

- closer to target machine: *GCC's* RTL, 3-address code
- easy to generate code from
- generation from input program may require effort

"Mid"-level intermediate representations:

- programming language and target independent
- Java bytecode, Microsoft CIL, LLVM IR, ...

Intermediate Code Languages: Design Issues

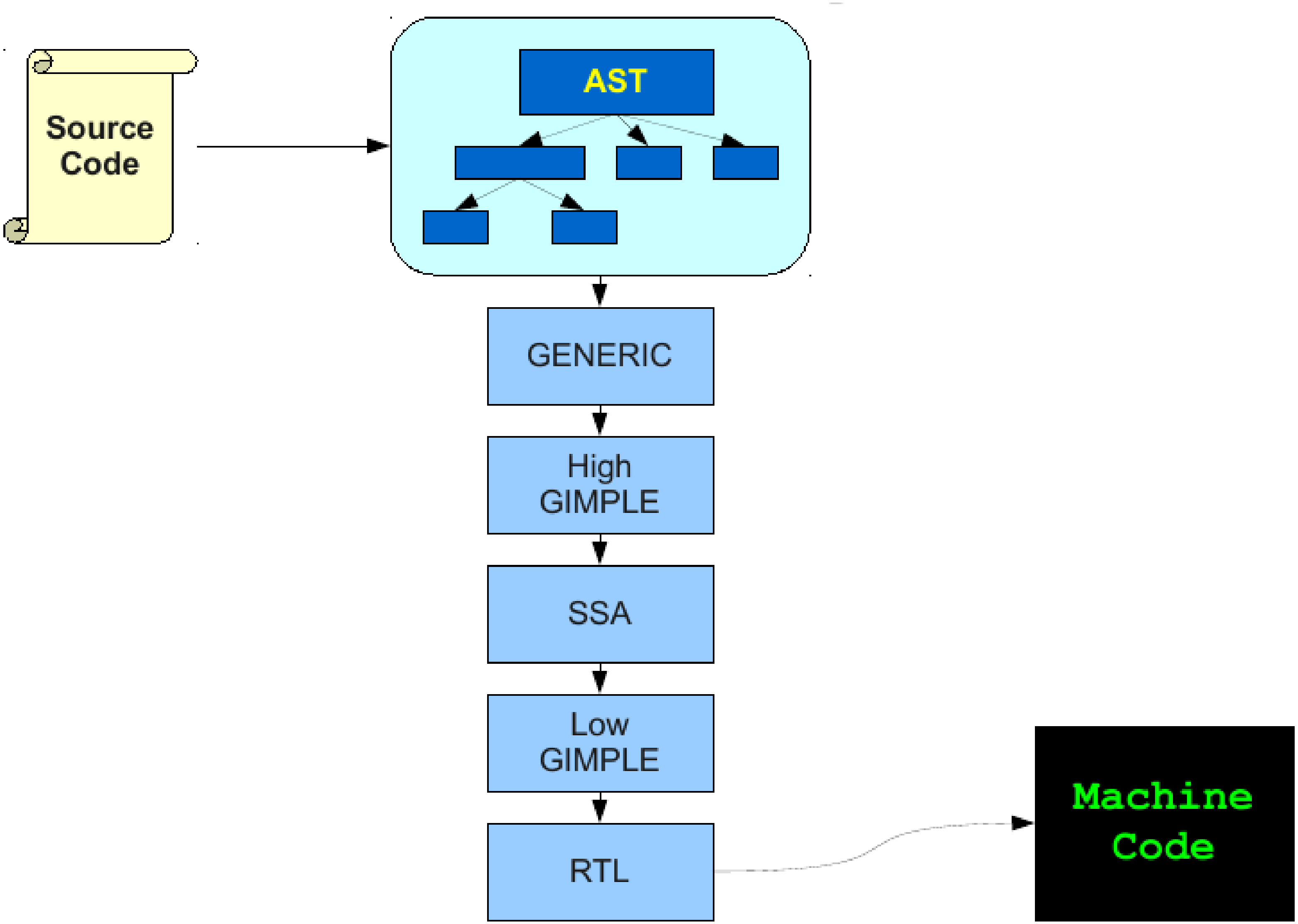
- Designing a good ICode language is not trivial.
- The set of operators in ICode must be rich enough to allow the implementation of source language operations.
- ICode operations that are closely tied to a particular machine or architecture, make retargeting harder.
- A small set of operations
  - may lead to long instruction sequences for some source language constructs,
  - but on the other hand makes retargeting easier.

Intermediate Languages

- Each compiler uses its own intermediate language.

- Nowadays, usually an intermediate language is a high-level assembly language.
  - Uses register names, but has an unlimited number.
  - Uses control structures like assembly language.
  - Uses opcodes but some are higher level.
    - E.g., push translates to several assembly instructions.
    - Most opcodes correspond directly to assembly opcodes.

Architecture of gcc



Three-Address Intermediate Code

- Each instruction is of the form:

$$x := y \text{ op } z$$

  - y and z can only be temporaries or constants.
  - Just like assembly.
- Common form of intermediate code.
- The expression $x + y * z$ gets translated as:

$$t_1 := y * z$$
$$t_2 := x + t_1$$

  - Temporary names are made up for internal nodes.
  - Each sub-expression has a “home”.

Generating Intermediate Code

- Similar to assembly code generation.
- Major difference:
  - Use any number of IL temporaries to hold intermediate results.

Example: `if (x + 2 > 3 * (y - 1) + 42) then z := 0;`

```
    $t_1$ := x + 2
    $t_2$ := y - 1
    $t_3$ := 3 * $t_2$
    $t_4$ := $t_3$ + 42
    if $t_1$ =< $t_4$ goto L
    z := 0
L:
```

Generating Intermediate Code (Cont.)

igen(e, t) : a function that generates code to compute the value of e in temporary t

- Example:

  igen($e_1 + e_2$, t) =

  igen(e_1, t_1) (t_1 is a fresh register)

  igen(e_2, t_2) (t_2 is a fresh register)

  $t := t_1 + t_2$

- Unlimited number of temporaries

  ⇒ simple code generation

From ICode to Machine Code

This is almost a macro expansion process.

ICode	MIPS assembly code
x := A[i]	load i into *r1* **la** *r2, A* **add** *r2, r2, r1* **lw** *r2*, (*r2*) **sw** *r2*, x
x := y + z	load y into *r1* load z into *r2* **add** *r3, r1, r2* **sw** *r3*, x
if x >= y goto L	load x into *r1* load y into *r2* **bge** *r1, r2*, L

Basic Blocks

- A *basic block* is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction).

- Idea:
  - Cannot jump into a basic block (except at beginning).
  - Cannot jump out of a basic block (except at end).
  - Each instruction in a basic block is executed after all the preceding instructions have been executed.

Basic Block Example

Consider the basic block

```
L:                               (1)
   t := 2 * x                    (2)
   w := t + x                    (3)
   if w > 0 goto L'              (4)
```

- No way for (3) to be executed without (2) having been executed right before.
  - We can change (3) to w := 3 * x ?
  - Can we eliminate (2) as well ?

Identifying Basic Blocks

- Determine the set of *leaders*, i.e., the first instruction of each basic block:
  - The first instruction of a function is a leader.
  - Any instruction that is a target of a branch is a leader.
  - Any instruction immediately following a (conditional or unconditional) branch is a leader.
- For each leader, its basic block consists of itself and all instructions up to, but not including, the next leader (or end of function).

Control-Flow Graphs

A *control-flow graph* is a directed graph with

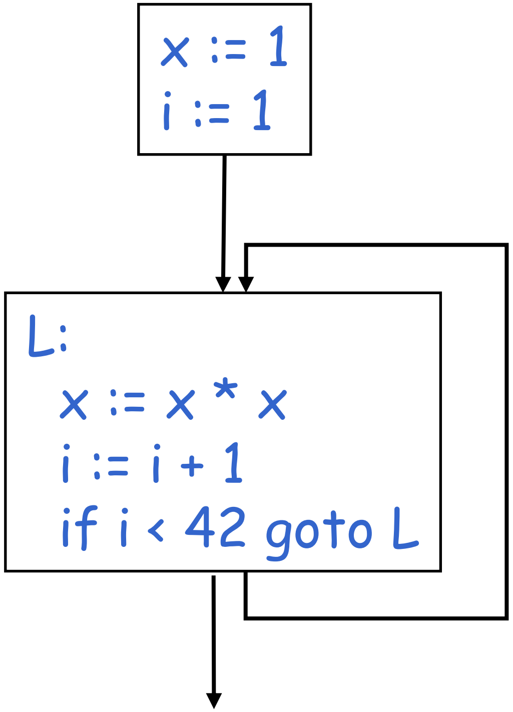
- Basic blocks as nodes.
- An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B.

  E.g., the last instruction in A is goto L_B .

  E.g., the execution can fall-through from block A to block B.

Frequently abbreviated as CFGs.

Control-Flow Graphs: Example



```
x := 1
i := 1
```

```
L:
  x := x * x
  i := i + 1
  if i < 42 goto L
```

- The body of a function (or method or procedure) can be represented as a control-flow graph.
- There is one initial node.
- All "return" nodes are terminal.

Constructing the Control Flow Graph

- First identify the basic blocks of the function.
- There is a directed edge between block B_1 to block B_2 if
  - there is a (conditional or unconditional) jump from the last instruction of B_1 to the first instruction of B_2 or
  - B_2 immediately follows B_1 in the textual order of the program, and B_1 does not end in an unconditional jump.

Optimization Overview

- Compiler "optimizations" seek to improve a program's utilization of some resource:
  - Execution time (most often).
  - Code size.
  - Network messages sent.
  - (Battery) power used, etc.

- Optimization should not alter what the program computes:
  - The return value must be the same.
  - Any observable behavior must be the same.

    (This typically also includes termination behavior.)

A Classification of Optimizations

For languages like C, there are three granularities of optimizations:

(1) Local optimizations

- Apply to a basic block in isolation.

(2) Global optimizations

- Apply to a control-flow graph (function body) in isolation.

(3) Inter-procedural optimizations

- Apply across function/procedure boundaries.

Most compilers do (1), many do (2), and very few do (3).

Note: there are also link-time optimizations.

Cost of Optimizations

- In practice, a conscious decision is made **not** to implement the fanciest optimizations.
- Why?
  - Some optimizations are hard to implement.
  - Some optimizations are costly in terms of compilation time.
  - Some optimizations are hard to get completely right.
  - The fancy optimizations are often hard, costly, and difficult to get completely correct.
- Goal: maximum improvement with minimum cost.

Local Optimizations

- The simplest form of optimizations.
- No need to analyze the whole procedure body.
  - Just the basic block in question.

- Example: algebraic simplification.

Algebraic Simplification

- Some statements can be deleted:

```
x := x + 0
x := x * 1
```

- Some statements can be simplified:

```
a := x * 0       ⇒     a := 0
b := y ** 2      ⇒     b := y * y
c := x * 8       ⇒     c := x << 3
d := x * 15      ⇒     t := x << 4; d := t - x
```

(on some machines << is faster than *; but not on all!)

Constant Folding

- Operations on constants can be computed at compile time.
- In general, if there is a statement

  x := y op z

  - where y and z are constants
  - then y op z can be computed at compile time.

- Example: x := 20 + 22 ⇒ x := 42
- Example: if 42 < 17 goto L can be deleted.

Flow of Control Optimizations

- Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph.
  - Basic blocks that are not the target of any jump or "fall through" from a conditional.
  - Such basic blocks can be eliminated.

- Why/how would such basic blocks occur?

- Removing unreachable code makes the program smaller.
  - And sometimes also faster.
    - Due to memory cache effects (increased spatial locality).

Single Assignment Form

- Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment.
- Basic blocks of intermediate code can be rewritten to be in *single assignment* form.

```
x := z + y                     b := z + y
a := x            ⇒            a := b
x := 2 * x                     x := 2 * b
```

(b is a fresh temporary.)

- More complicated in general, due to control flow (e.g., loops).
  - *Static single assignment (SSA)* form.

Common Subexpression Elimination

- Assume:
  - A basic block is in single assignment form.
  - A definition x := is the first use of x in a block.
- All assignments with same RHS compute the same value.

- Example:

```
x := y * z                  x := y * z
…               ⇒           …
w := y * z                  w := x
```

(Due to the block being in single assignment form, the values of x, y and z do not change in the … code.)

Copy Propagation

- If w := x appears in a block, all subsequent uses of w can be replaced with uses of x.

- Example:

```
b := z + y                  b := z + y
a := b          ⇒           a := b
x := 2 * a                  x := 2 * b
```

- This does not make the program smaller or faster but might enable other optimizations:
  - Constant folding.
  - Dead code elimination.

Constant Propagation and Constant Folding

- Example:

```
a := 5                 a := 5
x := 2 * a      ⇒      x := 10
y := x + 6             y := 16
t := x * y             t := 160
```

Dead Code Elimination

If

w := RHS appears in a basic block, and

w does not appear anywhere else in the program

Then

the statement w := RHS is dead and can be eliminated.

- <u>Dead</u> = does not contribute to the program's result.

Example: (a is not used anywhere else)

```
x := z + y          x := z + y          x := z + y
a := x         ⇒    a := x         ⇒    b := 2 * x
b := 2 * a          b := 2 * x
```

Applying Local Optimizations

- Each local optimization does very little by itself.
- However, typically optimizations interact.
  - Performing one optimization enables another.
- Optimizing compilers repeatedly perform optimizations until no improvement is possible.
  - The optimizer can also be stopped at any time to limit the compilation time.

An Example

Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Assume that only f and g are used in the rest of program.

An Example

Algebraic simplification:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

Algebraic simplification:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

An Example

Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

An Example

Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

Copy and constant propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

Dead code elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

Dead code elimination:

```
a := x * x




f := a + a
g := 6 * f
```

This is the final form.

Peephole Optimizations on Assembly Code

- The optimizations presented before work on intermediate code.
  - They are target independent.
  - But they can be applied on assembly language also.

Peephole optimization is an effective technique for improving assembly code.

- The "peephole" is a short sequence of (usually contiguous) instructions.
- The optimizer replaces the sequence with another equivalent (but faster) one.

Implementing Peephole Optimizations

- Write peephole optimizations as replacement rules:

  $i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m$

  where the RHS is the improved version of the LHS.
- Example:

  move $a $b, move $b $a → move $a $b

  - Works if move $b $a is not the target of a jump.
- Another example:

  addiu $a $a i, addiu $a $a j → addiu $a $a i+j

Peephole Optimizations

- Redundant instruction elimination, e.g.:

```
    . . .
      goto L
L:
    . . .
```

⇒

```
    . . .
L:

    . . .
```

- Flow of control optimizations, e.g.:

```
    . . .
      goto L1
    . . .
L1: goto L2
    . . .
```

⇒

```
    . . .
      goto L2
    . . .
L1: goto L2
    . . .
```

Peephole Optimizations (Cont.)

- Many (but not all) of the basic block optimizations can be cast as peephole optimizations.
  - Example: addiu $a $b 0 → move $a $b
  - Example: move $a $a →
  - These two together eliminate addiu $a $a 0.

- Just like for local optimizations, peephole optimizations need to be applied repeatedly to achieve maximum effect.

Concluding Remarks

- Multiple front-ends, multiple back-ends via intermediate codes.

- Intermediate code is the right representation for many optimizations.

- Many simple optimizations can still be applied on assembly language.

- Next time: global optimizations.

Global Optimization

Lecture Outline

- Global flow analysis

- Global constant propagation

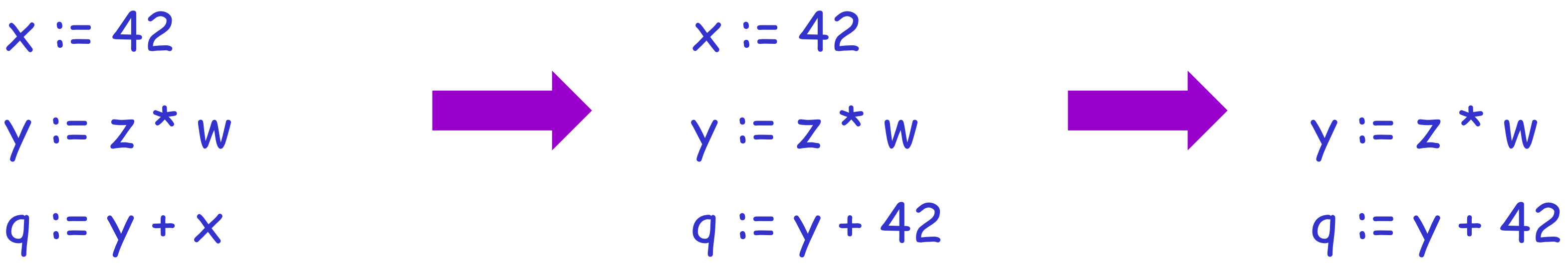
- Global dead code elimination

- Liveness analysis

Local Optimization

Recall the simple basic-block optimizations:

- Constant propagation.
- Dead code elimination.



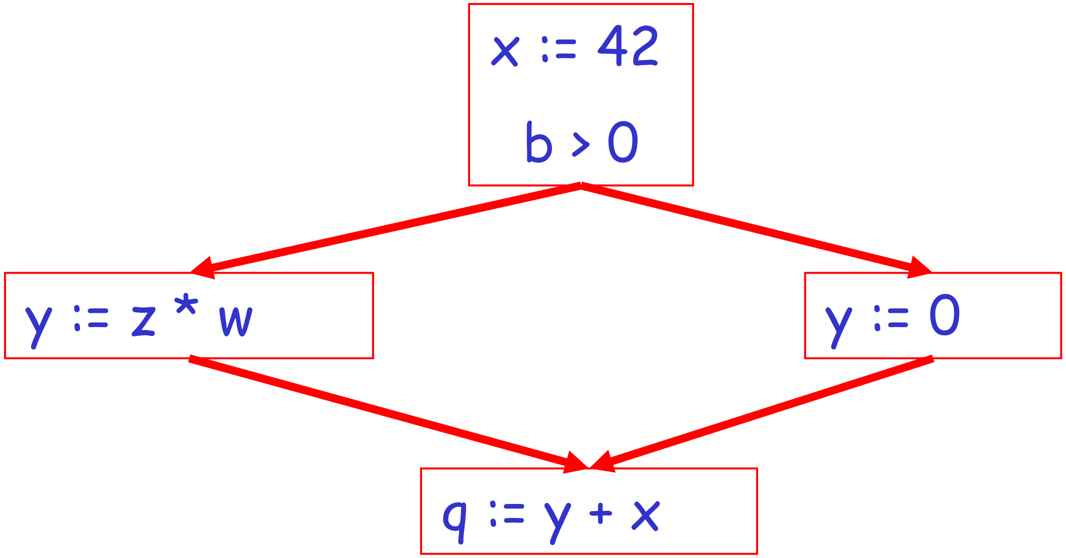
```
x := 42
y := z * w
q := y + x
```

```
x := 42
y := z * w
q := y + 42
```

```
y := z * w
q := y + 42
```

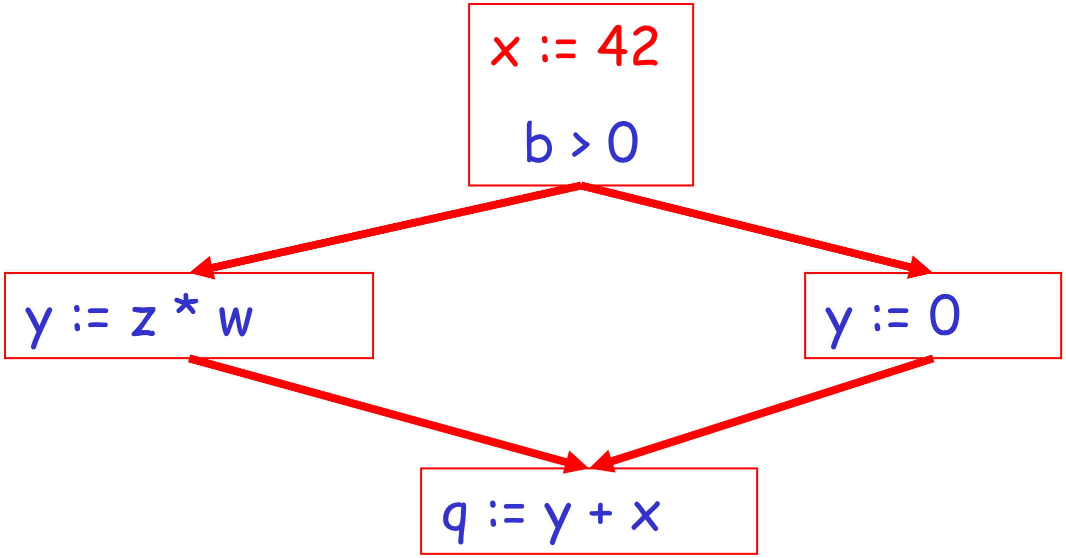
Global Optimization

These optimizations can be extended to an entire control-flow graph.



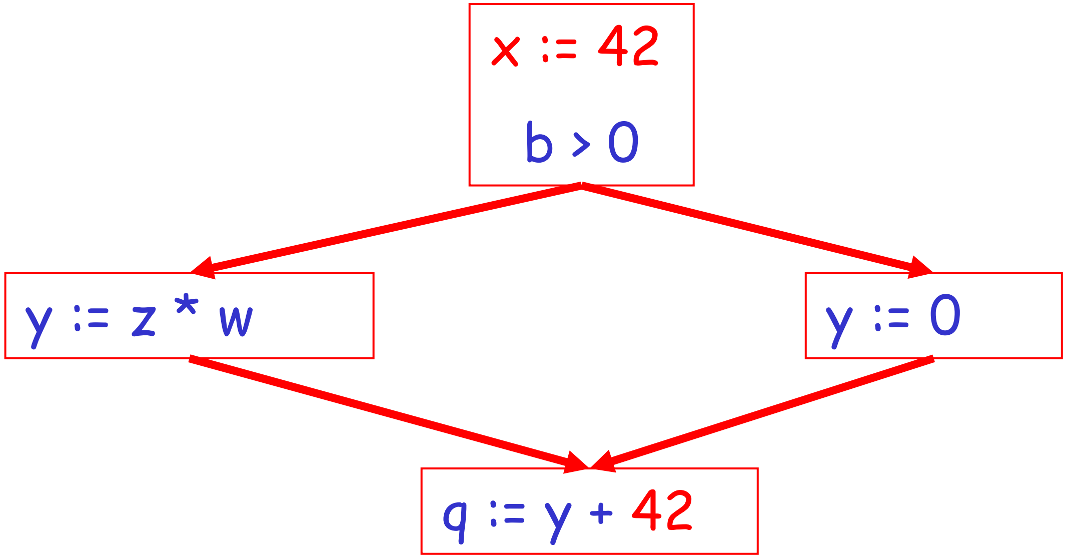
Global Optimization

These optimizations can be extended to an entire control-flow graph.



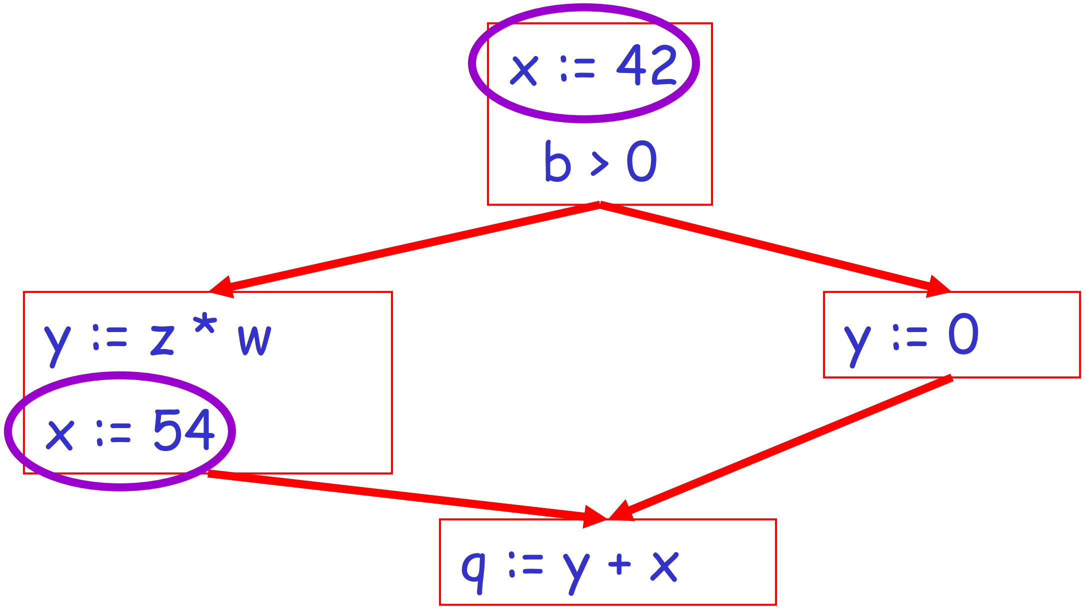
Global Optimization

These optimizations can be extended to an entire control-flow graph.



Correctness

- How do we know whether it is OK to globally propagate constants?
- There are situations where it is incorrect:

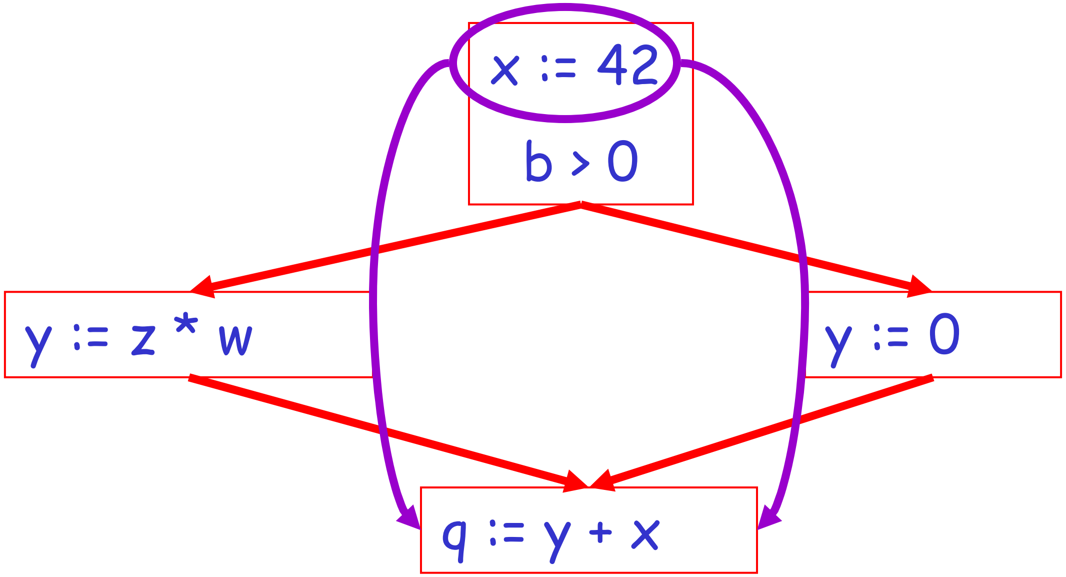


Correctness (Cont.)

To replace a use of x by a constant k we must know that the following property ** holds:

On every path to the use of x,
the last assignment to x is x := k **

Example 1 Revisited



x := 42

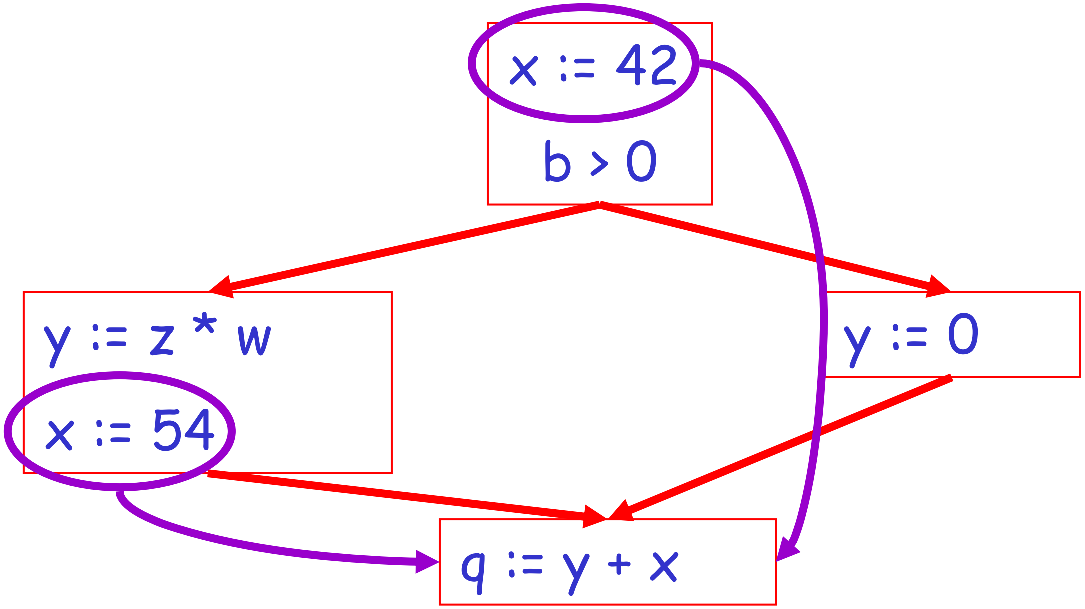
b > 0

y := z * w

y := 0

q := y + x

Example 2 Revisited



x := 42

b > 0

y := z * w

x := 54

y := 0

q := y + x

Discussion

- The correctness condition is not trivial to check.

- "All paths" includes paths around loops and through branches of conditionals.

- Checking the condition requires *global analysis.*
  - An analysis that determines how data flows over the entire control-flow graph of a function/method.

Global Analysis

Global optimization tasks share several traits:

- The optimization depends on knowing a property P at a particular point in program execution.
- Proving P at any point requires knowledge of the entire function body.
- Property P is typically undecidable !
- It is OK to be conservative: If the optimization requires P to be true, then we want an analysis which tells us:
  - that P is definitely true, or
  - that we don't know whether P is true.
- It is always safe to say "don't know".

  (But goal is to try to say "don't know" as rarely as possible.)

Global Analysis (Cont.)

- *Global dataflow analysis* is a standard technique for solving problems with these characteristics.
- Global constant propagation is one example of an optimization that requires global dataflow analysis.

Global Constant Propagation

- *On every path to the use of x,*
  the last assignment to x is x := k **

- Global constant propagation can be performed at any point where property ** holds.

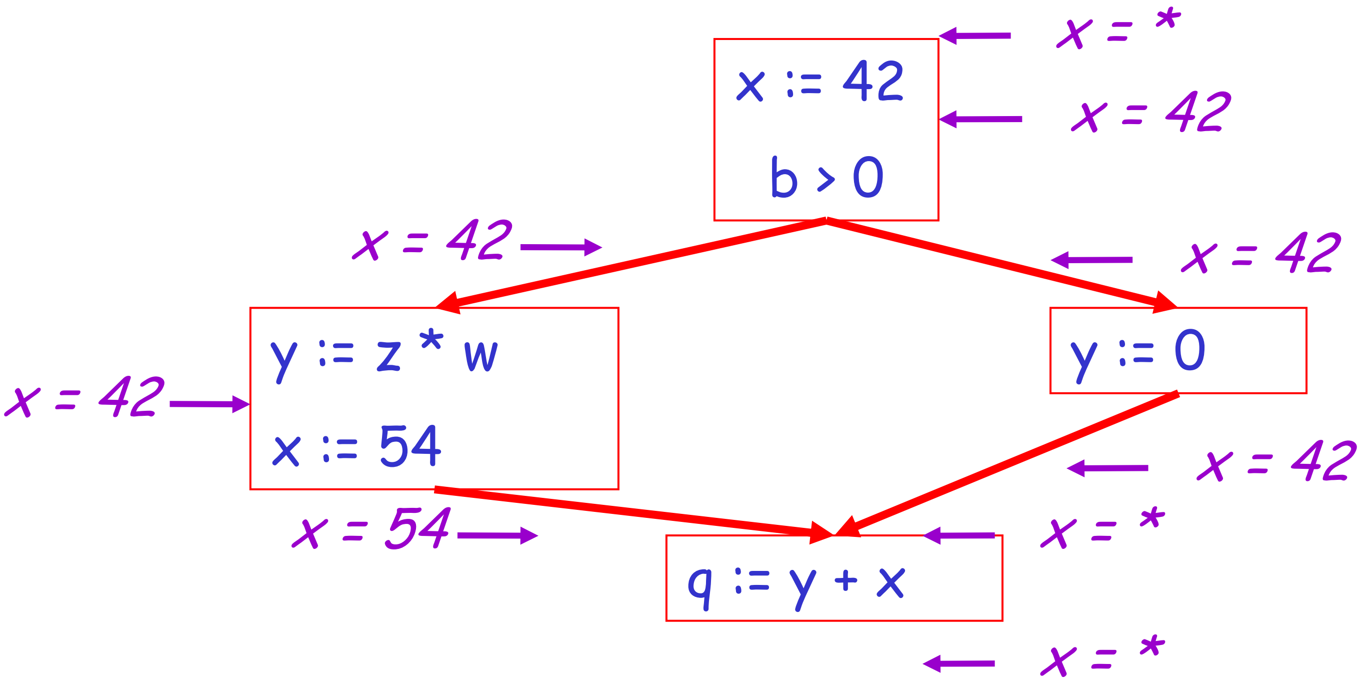
- Consider the case of computing ** for a single variable x at all program points.

Global Constant Propagation (Cont.)

- To make the problem precise, we associate one of the following values with x at every program point:

value	*interpretation*
#	This statement never executes
c_i	x = constant c_i
*	Don't know whether x is a constant

Example



Using the Information

- Given global constant information, it is easy to perform the optimization.
  - Simply inspect the x = ? associated with a statement using x.
  - If x is a constant k at that point replace that use of x by k.

- But how do we compute the properties x = ?

The Analysis Idea

The analysis of a (complicated) program can be expressed as a combination of simple rules relating the change in information between adjacent statements.

Explanation

- The idea is to "push" or "transfer" information from one statement to the next.

- For each statement s, we compute information about the value of x immediately before and after s

$$C_{in}(x,s) = \text{value of } x \text{ before } s$$

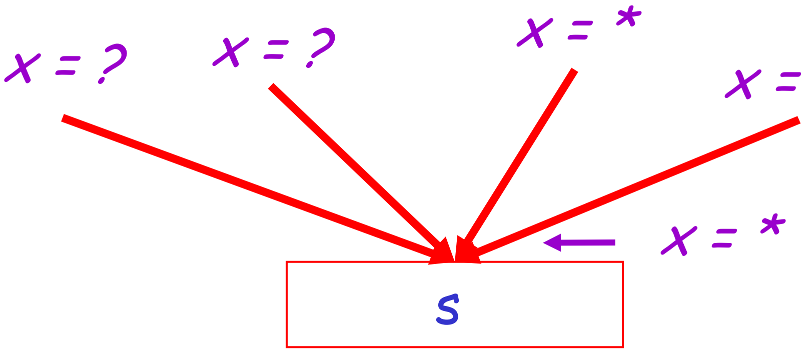
$$C_{out}(x,s) = \text{value of } x \text{ after } s$$

Transfer Functions

- Define a transfer function that transfers information from one statement to another.

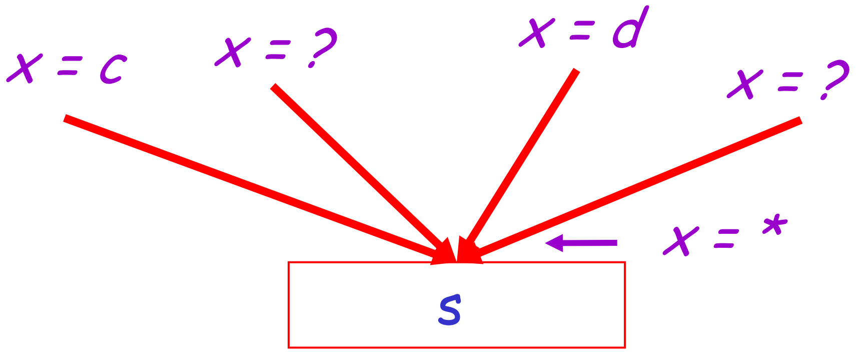
- In the following rules, let statement s have as immediate predecessors statements $p_1,\ldots,p_n$.

Rule 1



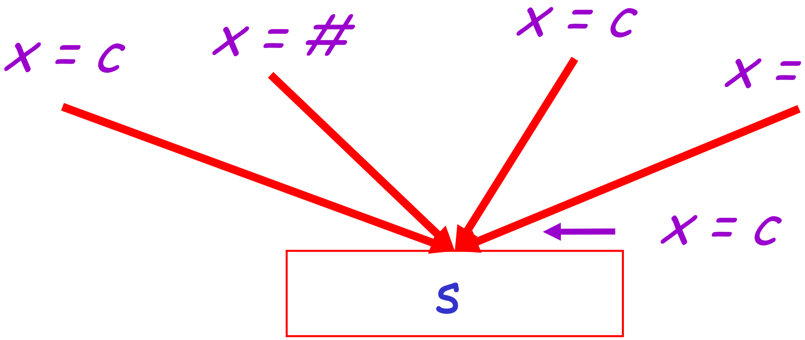
If $C_{out}(x, p_i) = *$ for any i, then $C_{in}(x, s) = *$

Rule 2



If $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$ then $C_{in}(x, s) = *$

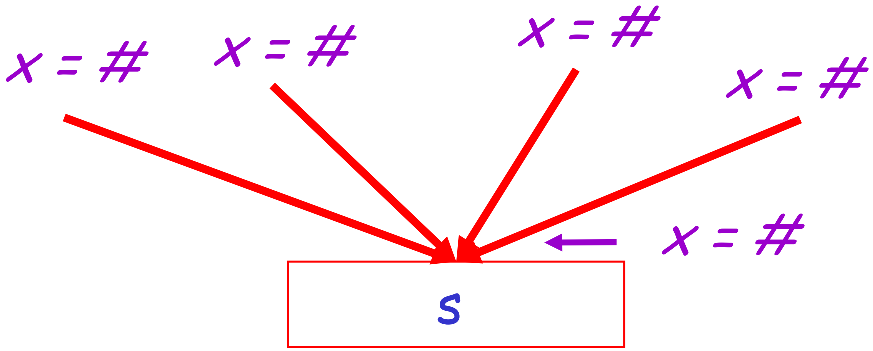
Rule 3



If $C_{out}(x, p_i) = c$ or # for all i,

then $C_{in}(x, s) = c$

Rule 4



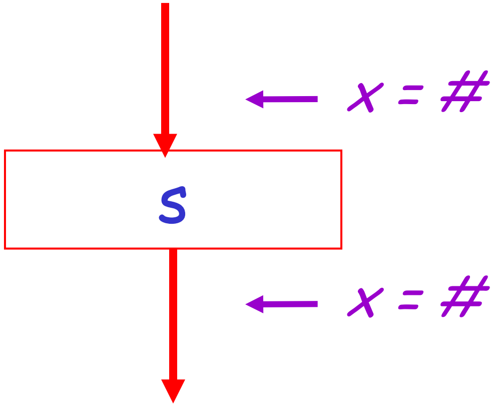
If $C_{out}(x, p_i) = \#$ for all i,
then $C_{in}(x, s) = \#$

The Other Half

- Rules 1-4 relate the *out* of one statement to the *in* of the successor statement.
  - I.e., they propagate information forward across CFG edges.

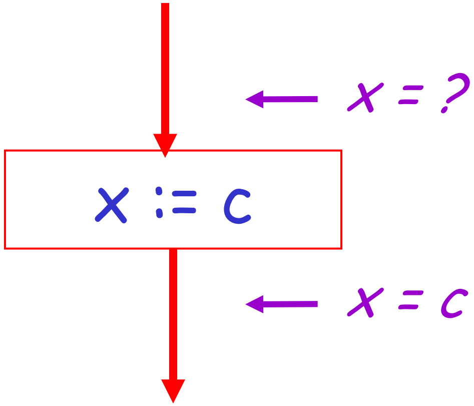
- We also need rules relating the *in* of a statement to the *out* of the same statement.
  - To propagate information across statements.

Rule 5



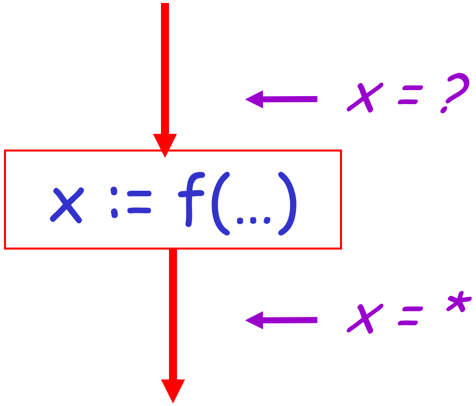
$$C_{out}(x, s) = \# \text{ if } C_{in}(x, s) = \#$$

Rule 6



$C_{out}(x, x := c) = c$ if c is a constant

Rule 7

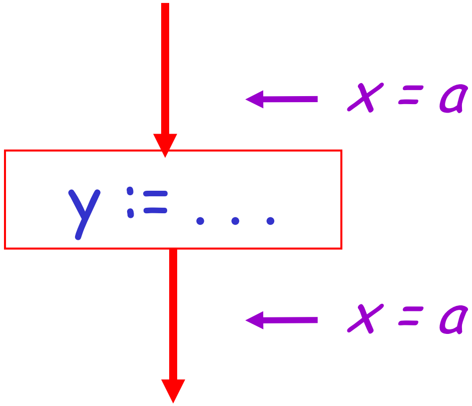


where f is a function other than the one being analyzed

$$C_{out}(x, x := f(...)) = *$$

This rule says that we do not perform inter-procedural analysis (i.e., we do not look at what other functions do).

Rule 8



$$C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots) \text{ if } x \neq y$$

An Algorithm

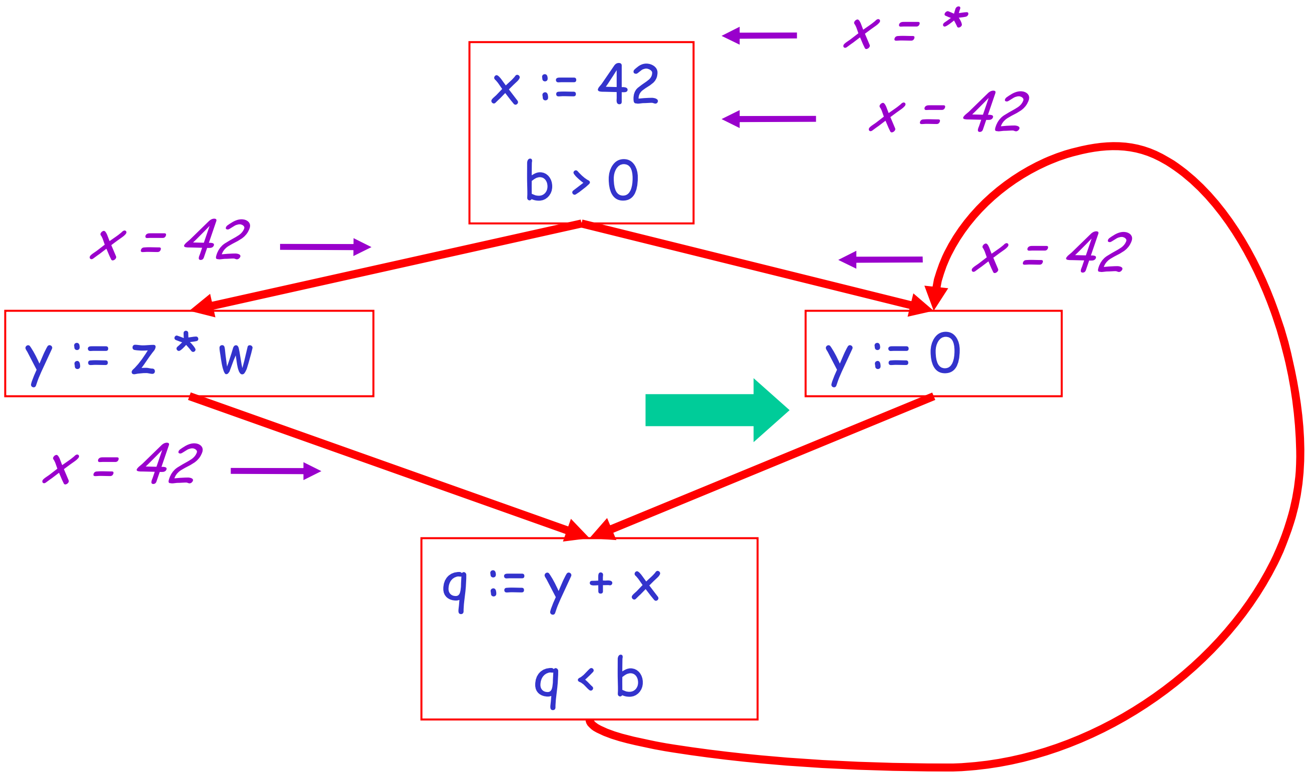
1. For every entry s to the function, set $C_{in}(x, s) = *$

2. Set $C_{in}(x, s) = C_{out}(x, s) = \#$ everywhere else

3. Repeat until all points satisfy 1-8:
    - pick an s not satisfying 1-8 and
    - update using the appropriate rule

The Value

To understand why we need #, look at a loop



Discussion

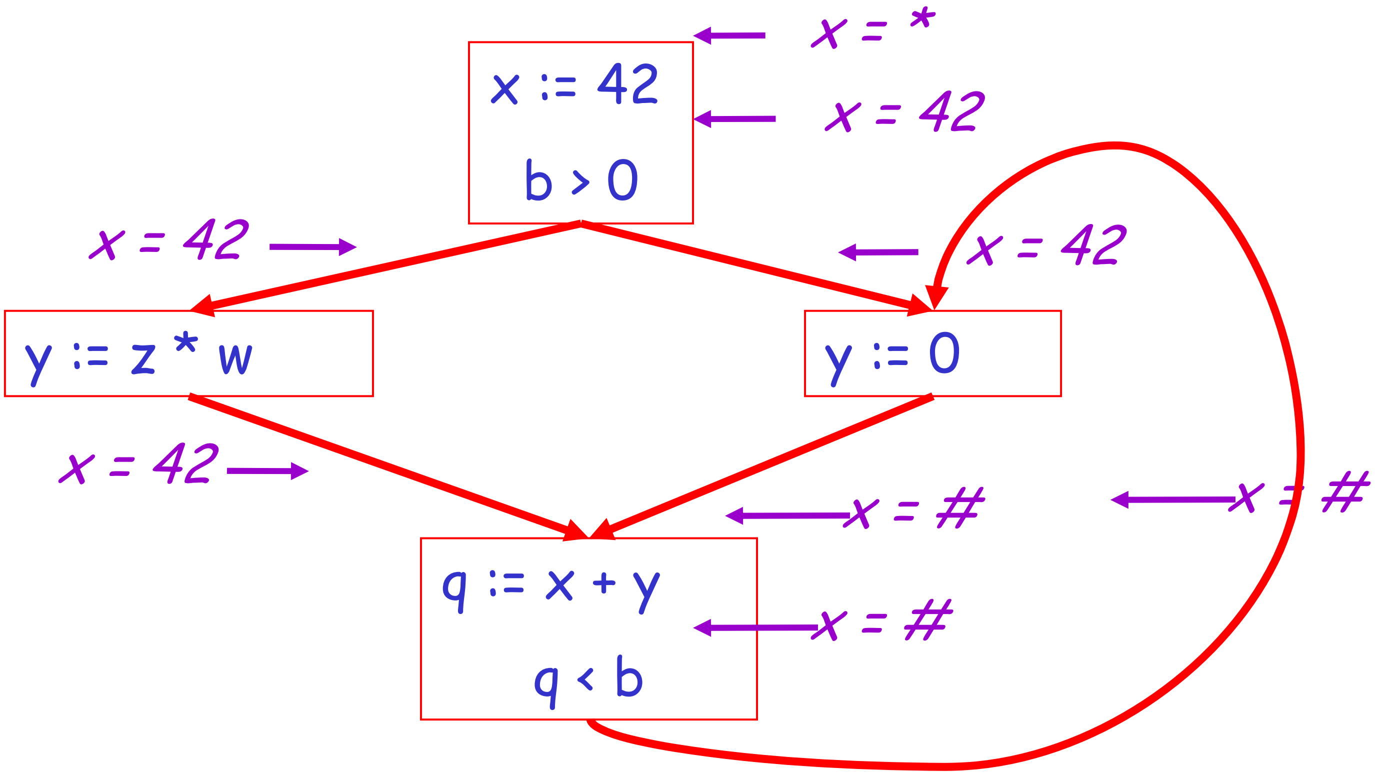
- Consider the statement y := 0
- To compute whether x is constant at this point, we need to know whether x is constant at its two predecessors
  - x := 42
  - q := y + x

- But information for q := y + x depends on its predecessors, including y := 0 !

The Value # (Cont.)

- Because of cycles, all points must have values at all times.
- Intuitively, assigning some initial value allows the analysis to break cycles.
- The initial value # means "So far as we know, control never reaches this point".

Example



x := 42

b > 0

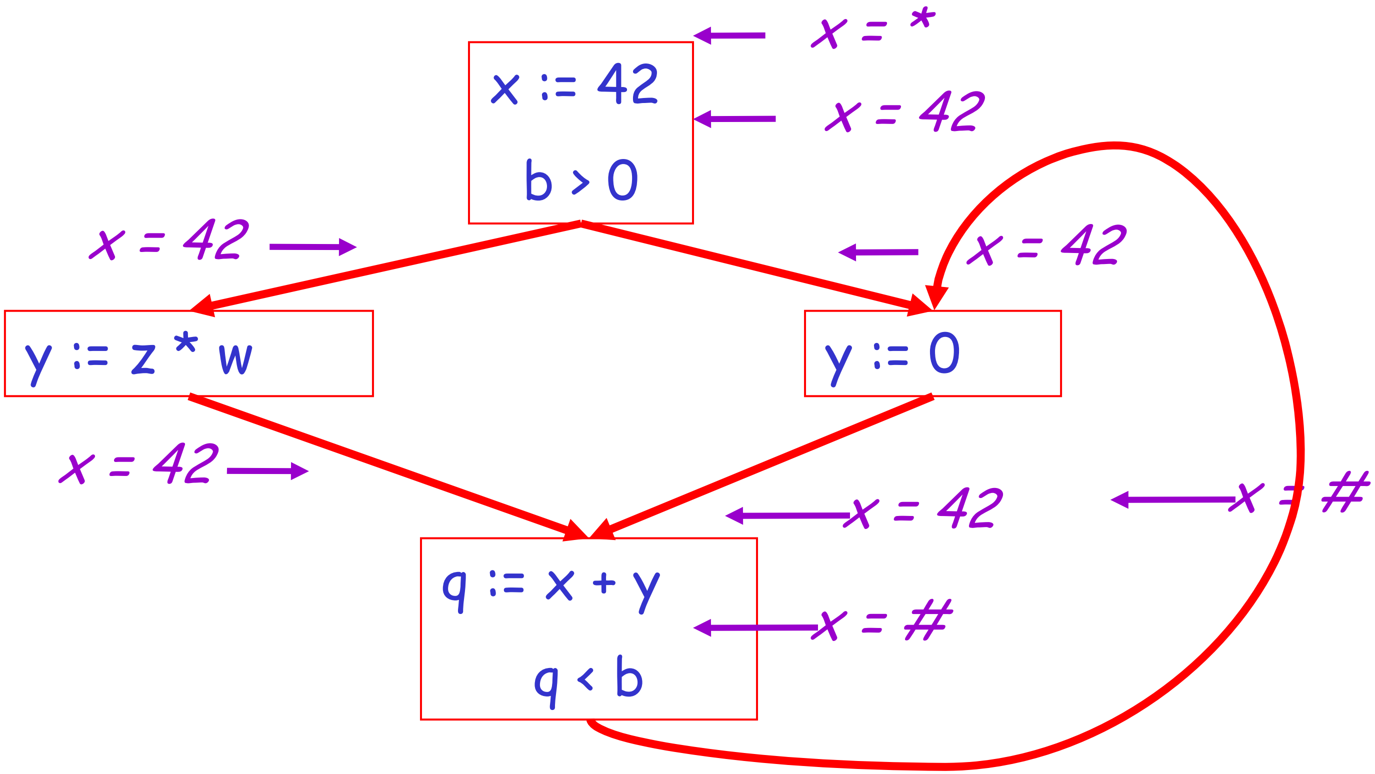
y := z * w

y := 0

q := x + y

q < b

Example



x := 42

b > 0

y := z * w

y := 0

q := x + y

q < b

x = *

x = 42

x = 42

x = 42

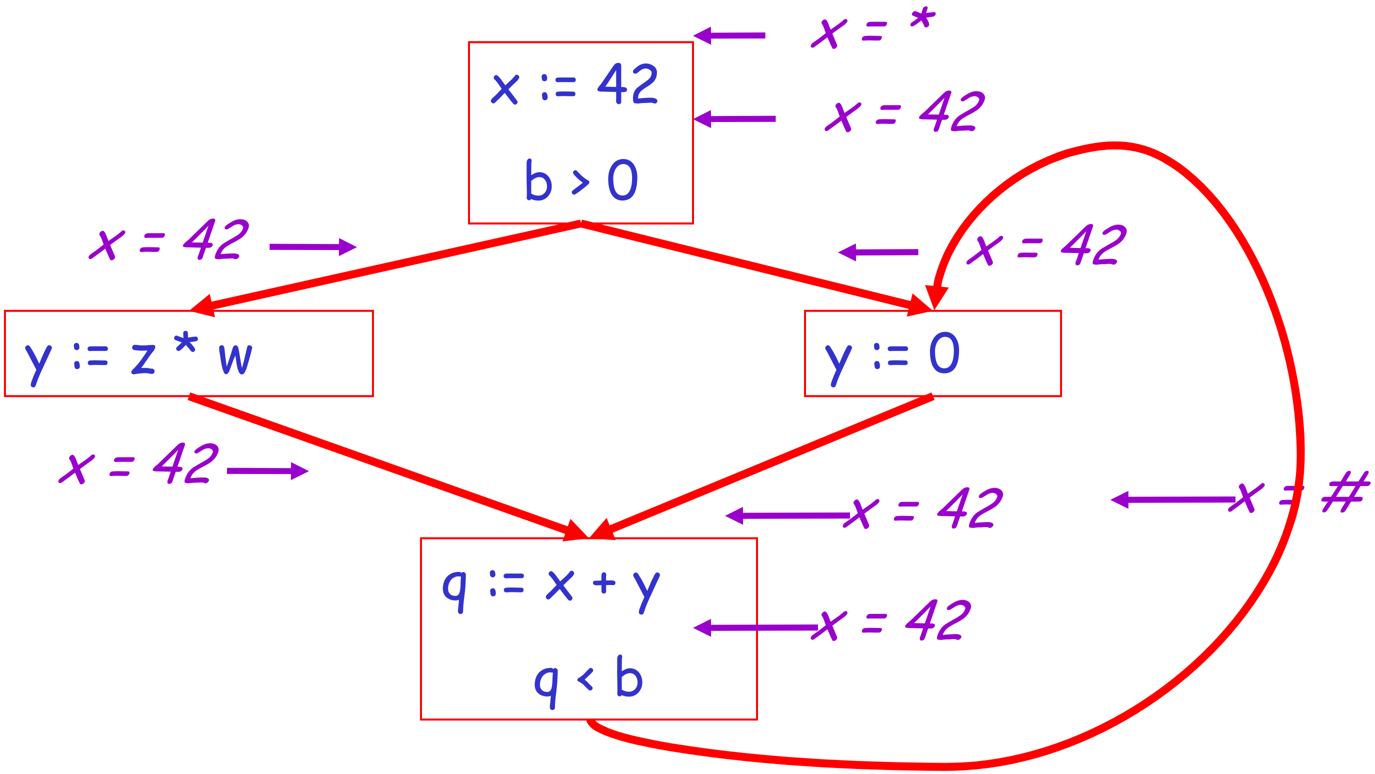
x = 42

x = 42

x = #

x = #

Example



x = *

x := 42

x = 42

b > 0

x = 42

x = 42

y := z * w

y := 0

x = 42

x = 42

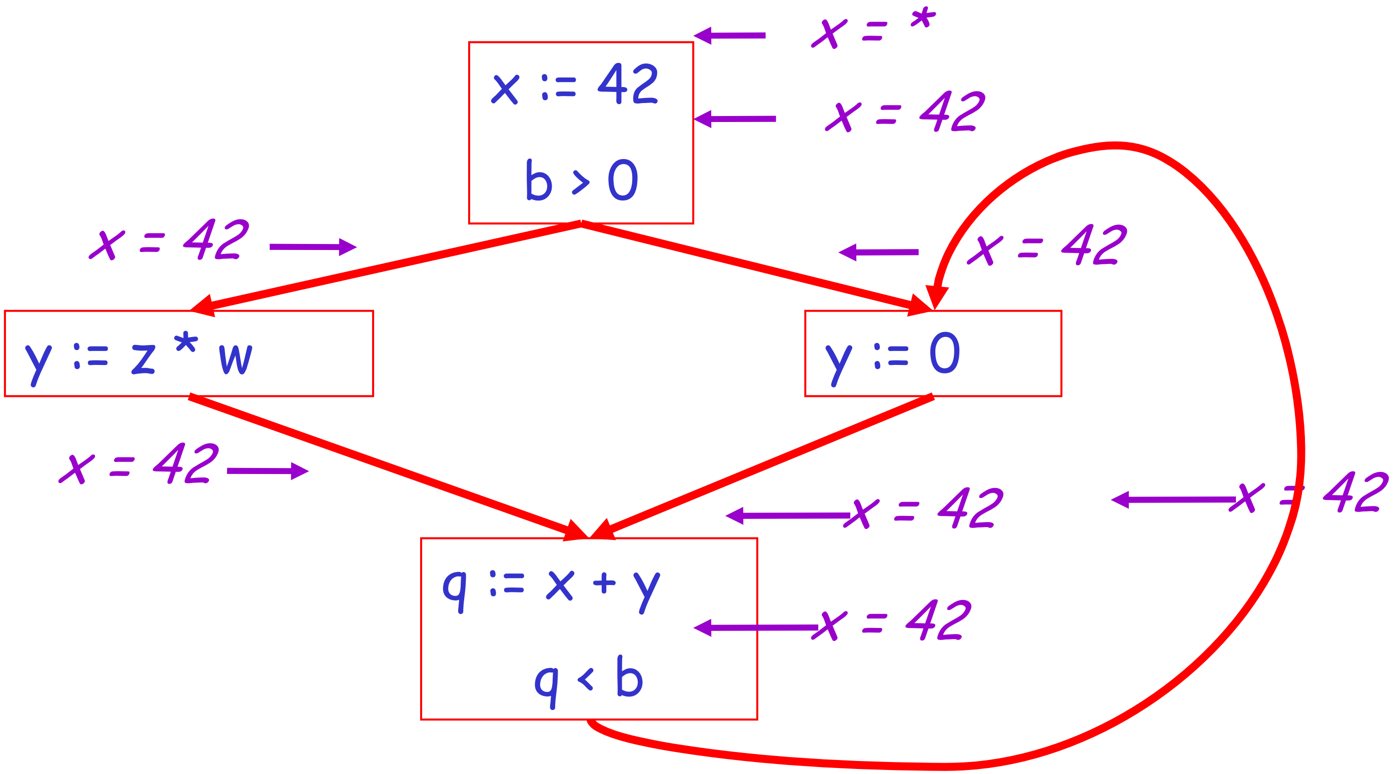
x = #

q := x + y

x = 42

q < b

Example



x := 42

b > 0

y := z * w

y := 0

q := x + y

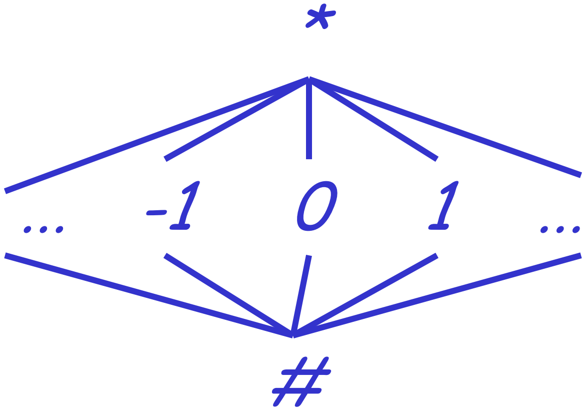
q < b

Orderings

- We can simplify the presentation of the analysis by ordering the values

$$\# < c_i < *$$

- Drawing a picture with "lower" values drawn lower, we get



Orderings (Cont.)

- * is the greatest value, # is the least.
  - All constants are in between and incomparable.

- Let lub be the least-upper bound in this ordering.

- Rules 1-4 can be written using lub:

  $C_{in}(x, s)$ = lub { $C_{out}(x, p)$ | p is a predecessor of s }

Termination

- Simply saying "repeat until nothing changes" does not guarantee that eventually we reach a point where nothing changes.

- The use of lub explains why the algorithm terminates.
  - Values start as # and only *increase*.
  - # can change to a constant, and a constant to *
  - Thus, C_(x, s) can change at most twice.

Termination (Cont.)

Thus, the algorithm is linear in program size.

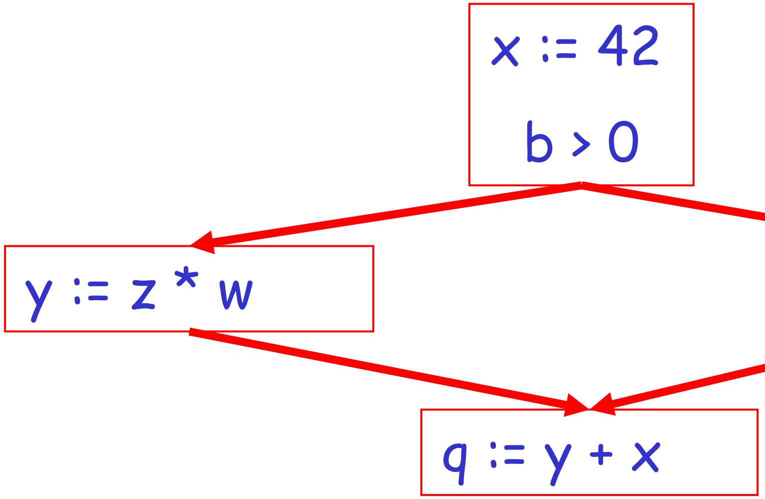
Number of steps = // worst case

  Number of C_(….) values computed * 2 =

    Number of program statements * 4

Liveness Analysis

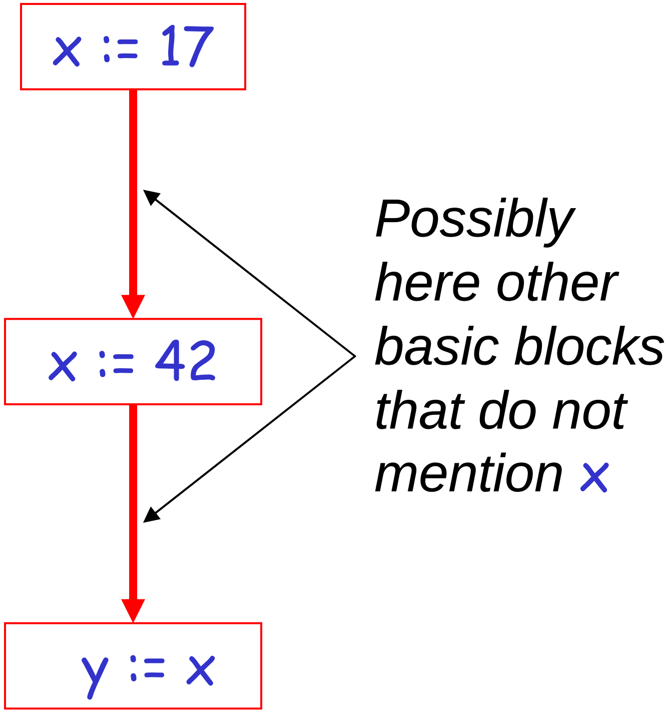
Once constants have been globally propagated, we would like to eliminate dead code.



After constant propagation, x := 42 is dead (assuming x is not used elsewhere).

Live and Dead Variables

- The first value of x is *dead* (never used).
- The second value of x is *live* (may be used).
- Liveness is an important concept for the compiler.



Liveness

A variable x is live at statement s if

- There exists a statement s′ that uses x
- There is a path from s to s′
- That path has no intervening assignment to x

Global Dead Code Elimination

- A statement x := … is dead code if x is dead after the assignment.

- Dead statements can be deleted from the program.

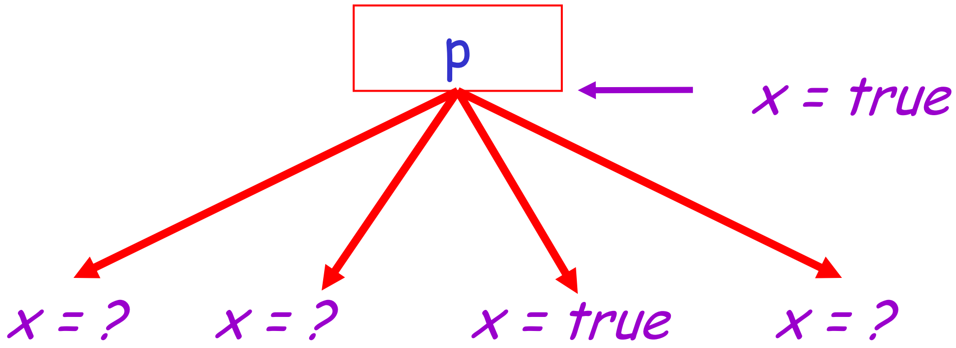
- But we need liveness information first . . .

Computing Liveness

- We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation.

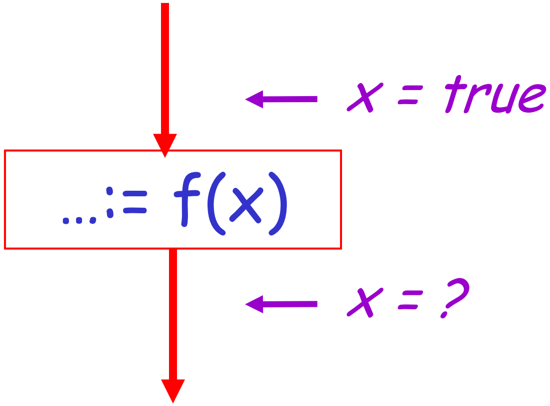
- Liveness is simpler than constant propagation, since it is a boolean property (true or false).

Liveness Rule 1



$$L_{out}(x, p) = \vee \{ L_{in}(x, s) \mid s \text{ a successor of } p \}$$

Liveness Rule 2



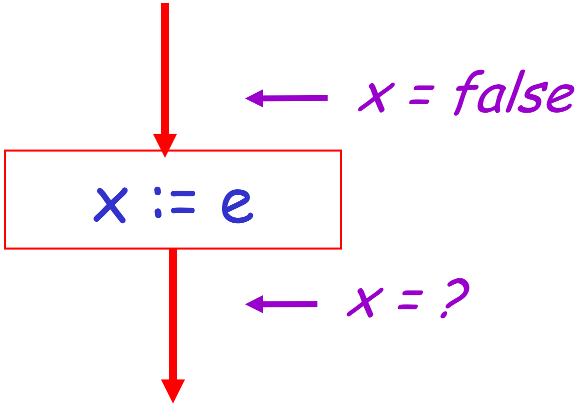
x = true

...:= f(x)

x = ?

$L_{in}(x, s)$ = true if s refers to x on the RHS

Liveness Rule 3



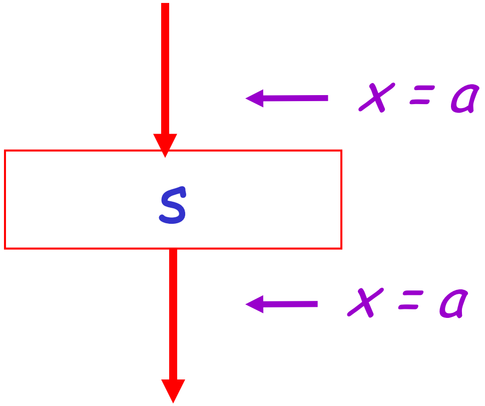
x = false

x := e

x = ?

$L_{in}(x, x := e) = false$ if e does not refer to x

Liveness Rule 4



$L_{in}(x, s) = L_{out}(x, s)$ if s does not refer to x

Algorithm

1. Let all L_(…) = false initially

2. Repeat until all statements s satisfy rules 1-4
   - pick an s where one of 1-4 does not hold and
   - update using the appropriate rule

Termination

- Each L_(...) value can change from false to true, but not the other way around.
- Each L_(...) value can change only once, so termination is guaranteed.
- Once the analysis information is computed, it is simple to eliminate dead code.

Forward vs. Backward Analysis

We have seen two kinds of analysis:

- An analysis that enables constant propagation:
  - This is a *forwards* analysis: information is pushed from inputs to outputs.
- An analysis that calculates variable liveness:
  - This is a *backwards* analysis: information is pushed from outputs back towards inputs.

Global Flow Analyses

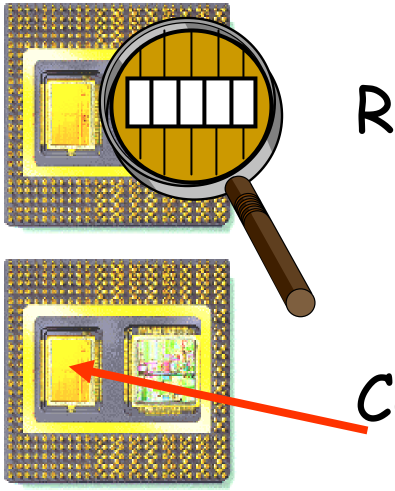
- There are many other global flow analyses.
- Most can be classified as either forward or backward.
- Most also follow the methodology of local rules relating information between adjacent program points.

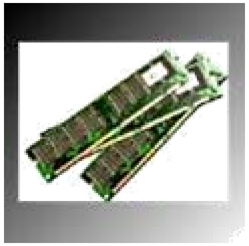
Global Register Allocation

Lecture Outline

- Memory Hierarchy Management
- Register Allocation via Graph Coloring
  - Register interference graph
  - Graph coloring heuristics
  - Spilling
- Cache Management

The Memory Hierarchy (circa 2004)





Registers	1 cycle	256-8000 bytes
Cache	3 cycles	256k-16M
Main memory	20-100 cycles	512M-64G
Disk	0.5-5M cycles	10G-1T

Managing the Memory Hierarchy

- Programs are written as if there are only two kinds of memory: main memory and disk.

- Programmer is responsible for moving data from disk to memory (e.g., file I/O).
- Hardware is responsible for moving data between memory and caches.
- Compiler is responsible for moving data between memory and registers.

Some Trends (circa 2004)

- Power usage limits
  - Size and speed of registers/caches.
  - Speed of processors.
    - Improves faster than memory speed (and disk speed).
    - The cost of a cache miss is growing.
    - The widening gap between processors and memory is bridged with more levels of caches.

- It is very important to:
  - Manage registers properly.
  - Manage caches properly.
- Compilers are good at managing registers.

The Register Allocation Problem

- Recall that intermediate code uses as many temporaries as necessary.
  - Typical intermediate code uses too many temporaries.
  - This simplifies code generation and optimization.
  - But complicates final translation to assembly.

- The register allocation problem:
  - Rewrite the intermediate code to use at most as many temporaries as there are machine registers.
  - Method: Assign multiple temporaries to a register.
    - But without changing the program behavior.

History

- Register allocation is as old as intermediate code.
  - Register allocation was used in the original FORTRAN compiler in the '50s.
  - Very crude algorithms were used back then.
- A breakthrough was not achieved until 1980.
  - Register allocation scheme based on graph coloring.
  - Relatively simple, global, and works well in practice.

An Example

- Consider the program

```
a := c + d
e := a + b
f := e - 1
```

with the assumption that a and e die after use.

- Temporary a can be "reused" after "a + b".
- Same with temporary e after "e - 1".

- Can allocate a, e, and f all to one register (r_1):

$r_1 := r_2 + r_3$

$r_1 := r_1 + r_4$

$r_1 := r_1 - 1$

Basic Register Allocation Idea

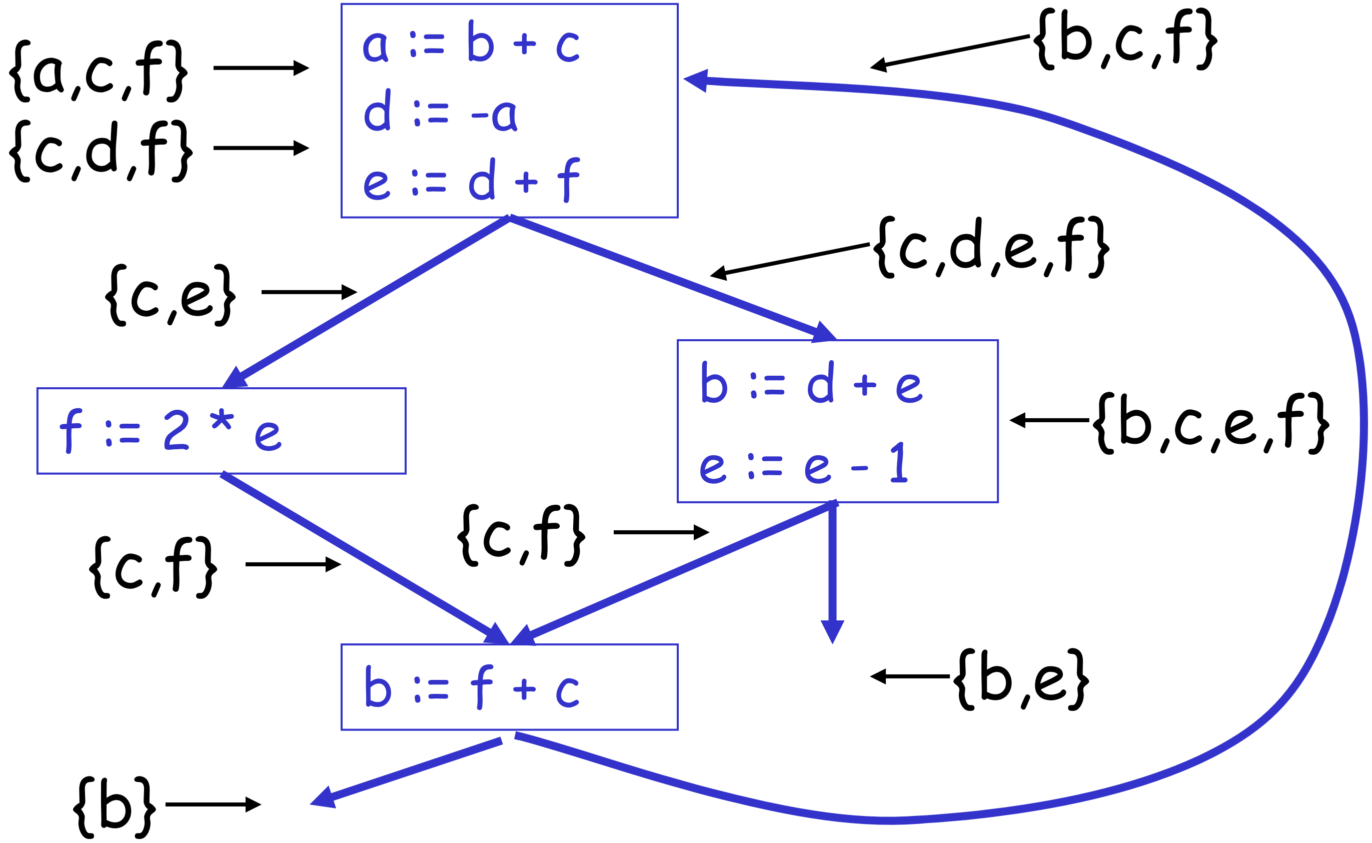
- The value in a dead temporary is not needed for the rest of the computation.
  - A dead temporary can be reused.

- Basic rule:

Temporaries t_1 and t_2 can share the same register if <u>at all points in the program at most one</u> of t_1 or t_2 is live !

Algorithm: Part I

Compute live variables for each program point:

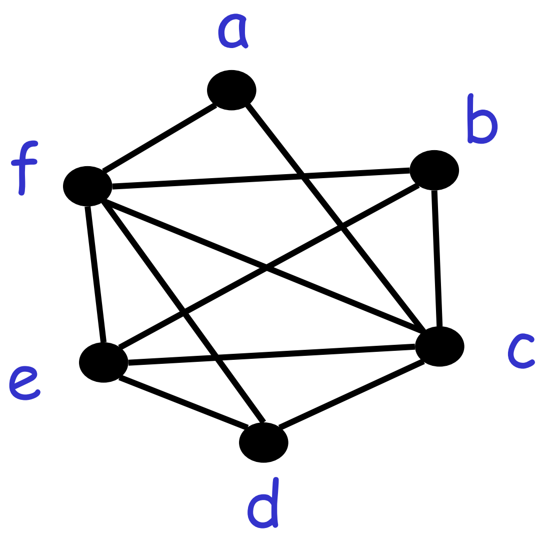


The Register Interference Graph

- Two temporaries that are live simultaneously cannot be allocated in the same register.
- We construct an undirected graph with:
  - a node for each temporary, and
  - an edge between t_1 and t_2 if they are live simultaneously at some point in the program.
- This is the register interference graph (RIG).
  - Two temporaries can be allocated to the same register if there is no edge connecting them.

Register Interference Graph: Example

- For our example:



- E.g., b and c cannot be in the same register.
- E.g., b and d can be in the same register.

Register Interference Graph: Properties

- It extracts exactly the information needed to characterize legal register assignments.

- It gives a global (i.e., over the entire flow graph) picture of the register requirements.

- After RIG construction, the register allocation algorithm is architecture independent.

Graph Coloring: Definitions

- A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors.

- A graph is k-colorable if it has a coloring with k colors.

Register Allocation Through Graph Coloring

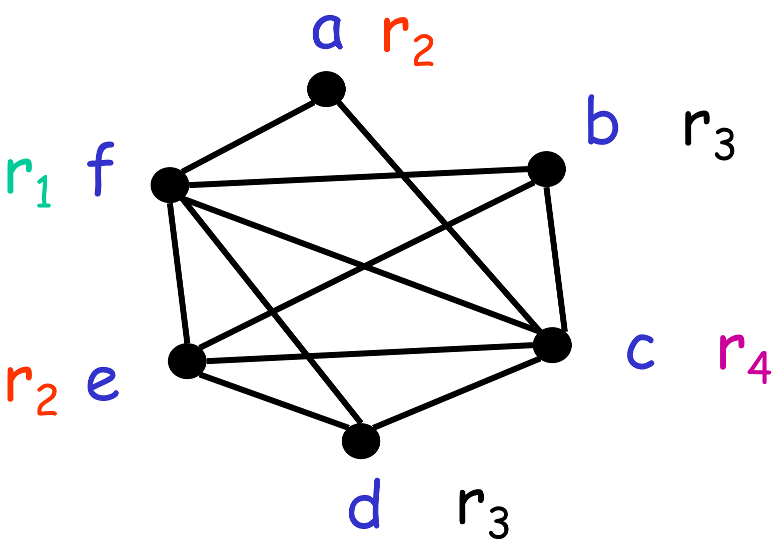
- Assume a regular architecture.
- In our problem, colors = registers.
  - We need to assign colors (registers) to graph nodes (temporaries).

- Let k = number of machine registers.

- If the RIG is k-colorable then there is a register assignment that uses no more than k registers.

Graph Coloring: Example

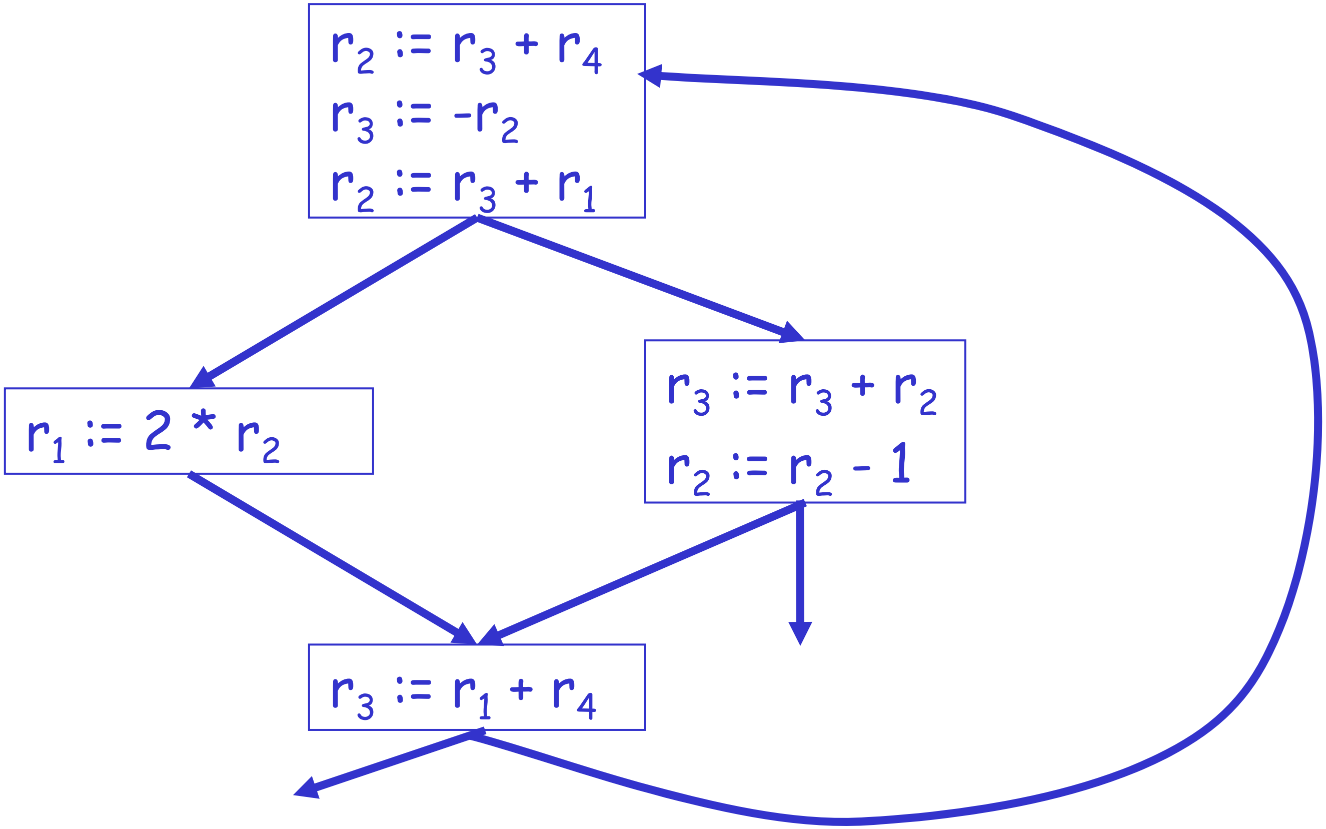
- Consider the example RIG



- There is no coloring with less than 4 colors.
- There are various 4-colorings of this graph.
  (One of them is shown in the figure.)

Graph Coloring: Example

- Under this coloring, the code becomes:



$r_2 := r_3 + r_4$
$r_3 := -r_2$
$r_2 := r_3 + r_1$

$r_1 := 2 * r_2$

$r_3 := r_3 + r_2$
$r_2 := r_2 - 1$

$r_3 := r_1 + r_4$

Computing Graph Colorings

- The remaining problem is how to compute a coloring for the interference graph.
- But:

  (1) Computationally this problem is NP-hard.

  - No efficient algorithms are known.

  (2) A coloring might not even exist for a given number of registers.

- The solution to (1) is to use heuristics.
- We will consider the other problem later.

Graph Coloring Heuristic

- Observation:
  - Pick a node t with fewer than k neighbors in RIG.
  - Eliminate t and its edges from RIG.
  - If the resulting graph has a k-coloring then so does the original graph.

- Why:
  - Let $c_1, \ldots, c_n$ be the colors assigned to the neighbors of t in the reduced graph.
  - Since $n < k$ we can pick some color for t that is different from those of its neighbors.

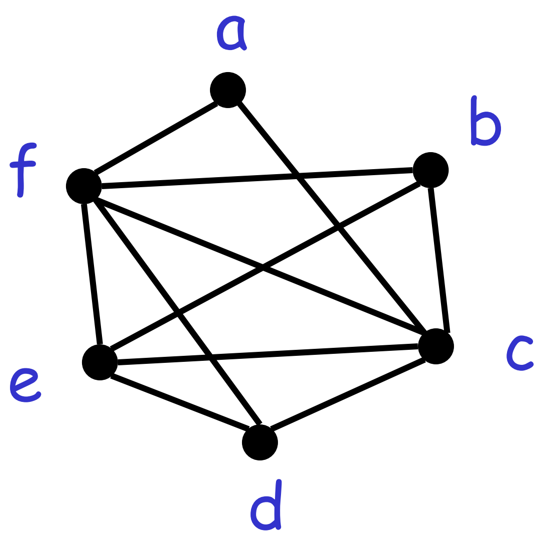
Graph Coloring Simplification Heuristic

- The following works well in practice:
  - Pick a node t with fewer than k neighbors.
  - Put t on a stack and remove it from the RIG.
  - Repeat until the graph has one node.

- Then start assigning colors to nodes on the stack (starting with the last node added).
  - At each step pick a color different from those assigned to already colored neighbors.

Graph Coloring Example (1)

- Start with the RIG and with k = 4:

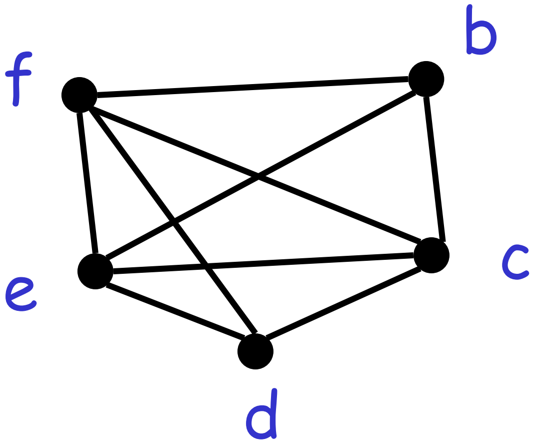


Stack: []

- Remove a

Graph Coloring Example (2)

- Start with the RIG and with k = 4:

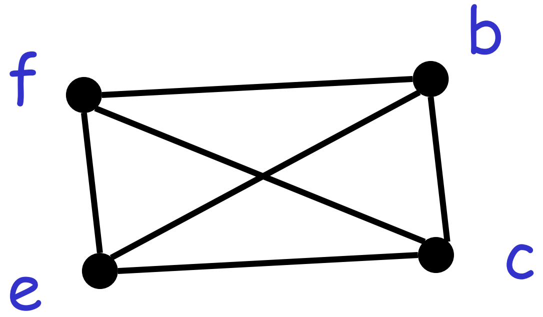


Stack: [a]

- Remove d

Graph Coloring Example (3)

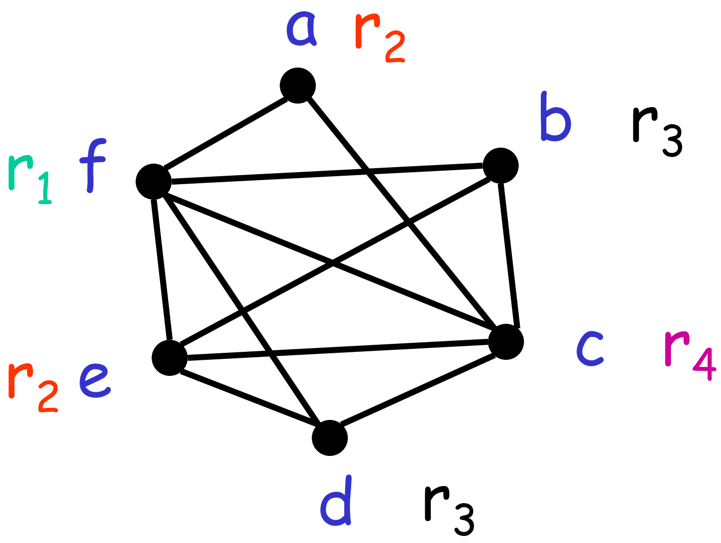
- Now all nodes have fewer than 4 neighbors and can be removed in e.g. the order: c, b, e, f



Stack: [d, a]

Graph Coloring Example (4)

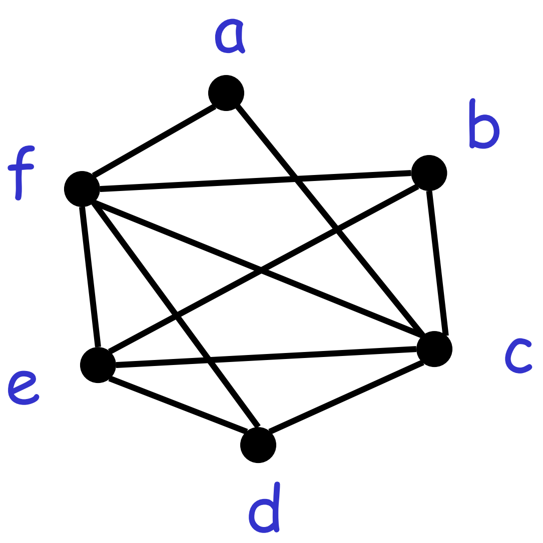
- Start assigning colors to: [f, e, b, c, d, a]



What if the Heuristic Fails?

- What if during simplification we get to a state where all nodes have k or more neighbors ?

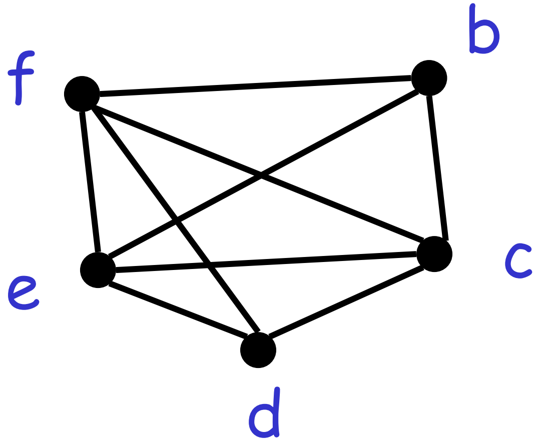
- Example: try to find a 3-coloring of the RIG:



What if the Heuristic Fails?

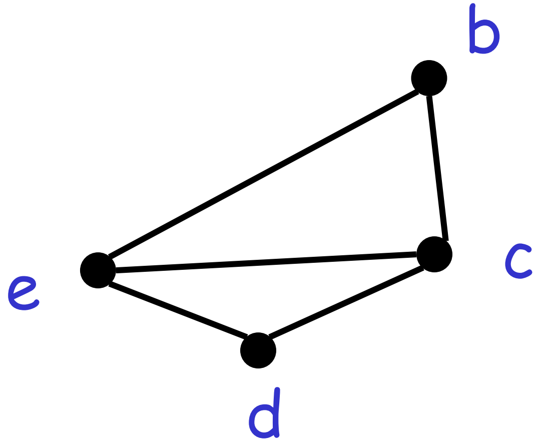
- Remove a and get stuck (as shown below).

- Pick a node as a possible candidate for spilling.
  - A spilled temporary "lives" is memory.
  - Assume that f is picked as a candidate.



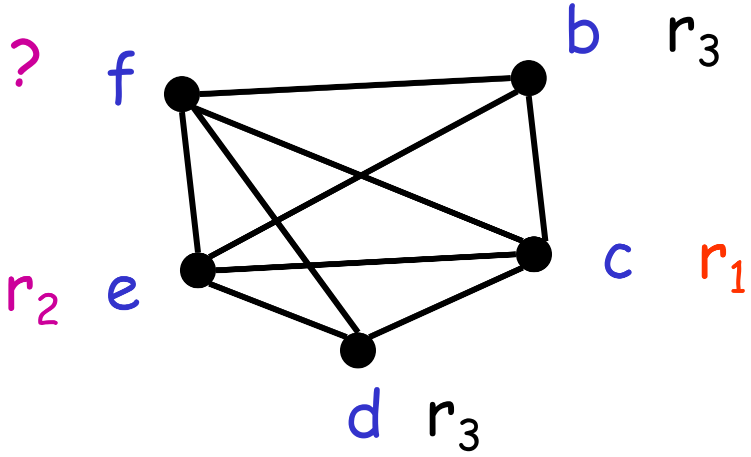
What if the Heuristic Fails?

- Remove f and continue the simplification.
  - Simplification now succeeds: b, d, e, c



What if the Heuristic Fails?

- On the assignment phase we get to the point when we have to assign a color to f.
- We hope that among the 4 neighbors of f we used less than 3 colors ⇒ optimistic coloring.



Spilling

- Since optimistic coloring failed, we must spill temporary f (actual spill).
- We must allocate a memory location as the "home" of f.
  - Typically this is in the current stack frame.
  - Call this address fa.
- Before each operation that uses f, insert

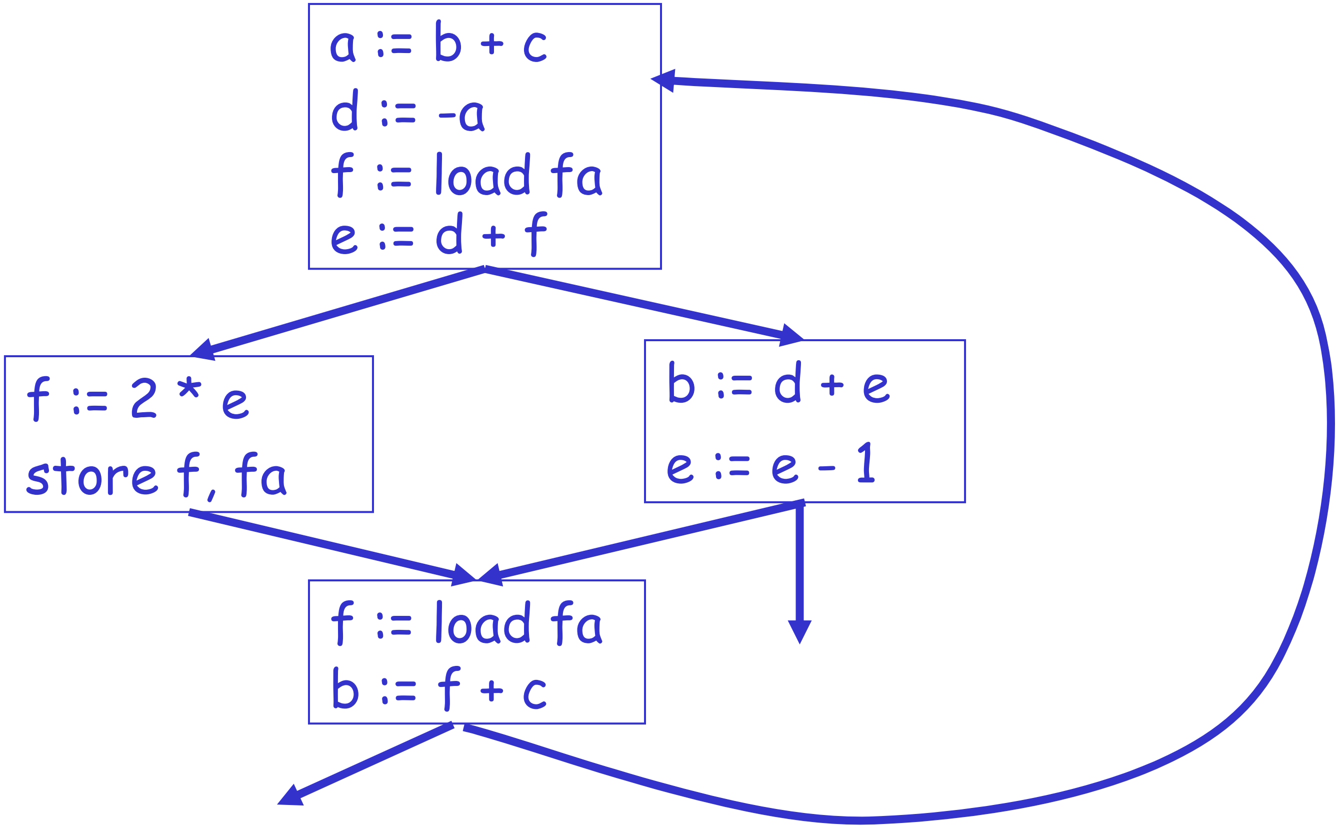
  f := load fa

- After each operation that defines f, insert

  store f, fa

Spilling: Example

- This is the new code after spilling f



```
a := b + c
d := -a
f := load fa
e := d + f
```

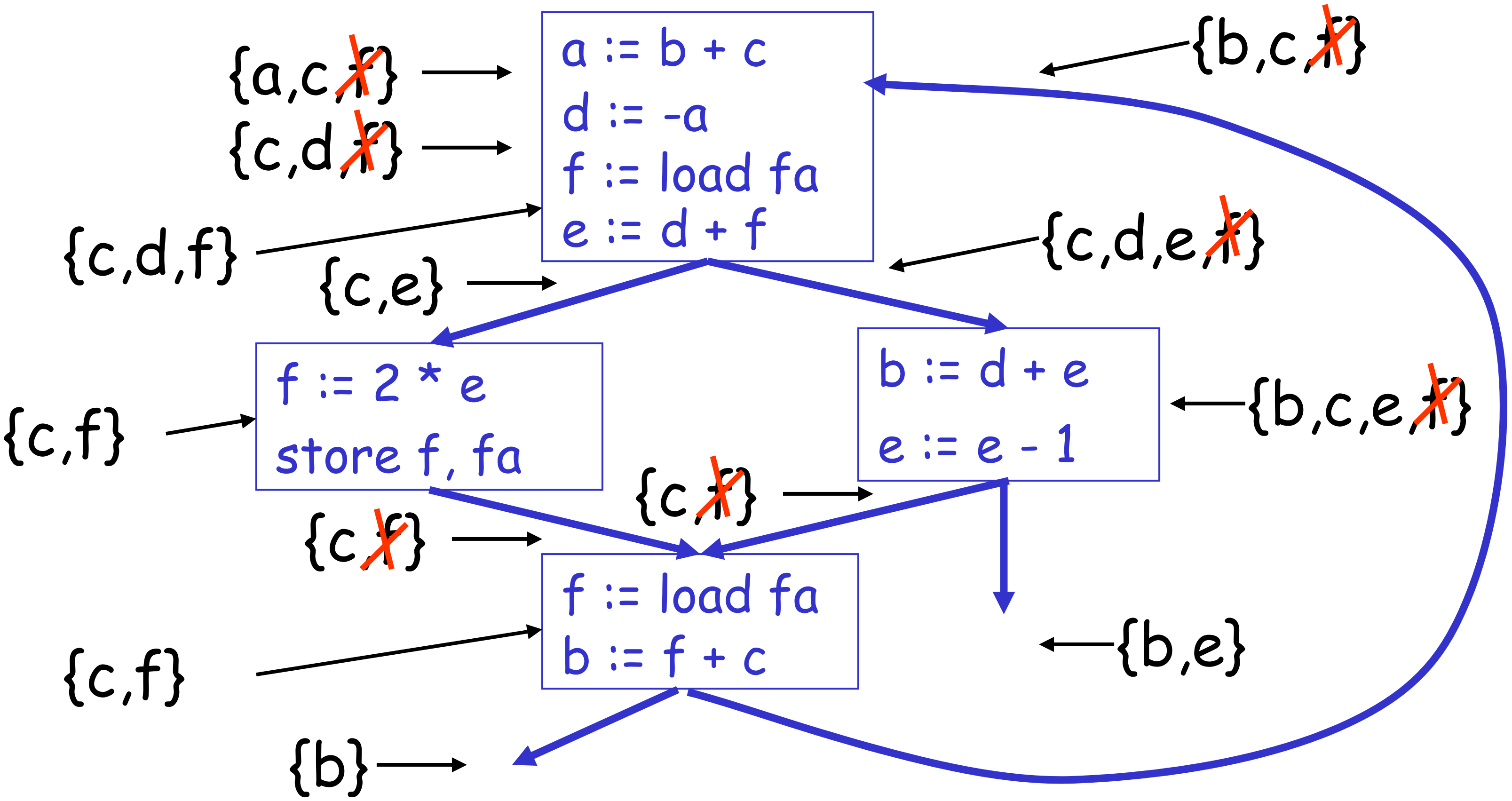
```
f := 2 * e
store f, fa
```

```
b := d + e
e := e - 1
```

```
f := load fa
b := f + c
```

Recomputing Liveness Information

- The new liveness information after spilling:



{a,c,~~f~~}

{c,d,~~f~~}

{c,d,f}

```
a := b + c
d := -a
f := load fa
e := d + f
```

{b,c,~~f~~}

{c,d,e,~~f~~}

{c,e}

{c,f}

```
f := 2 * e
store f, fa
```

```
b := d + e
e := e - 1
```

{b,c,e,~~f~~}

{c,~~f~~}

{c,~~f~~}

```
f := load fa
b := f + c
```

{c,f}

{b,e}

{b}

Recomputing Liveness Information

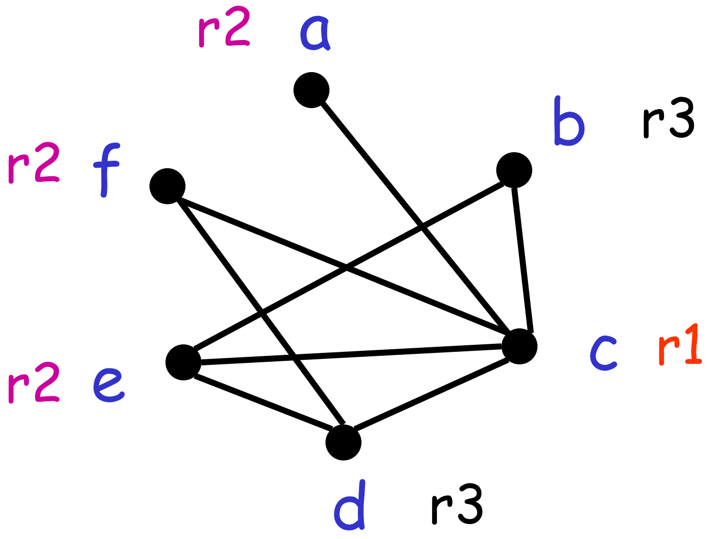
- New liveness information is almost as before.

- f is live only:
  - Between a f := load fa and the next instruction.
  - Between a store f, fa and the preceding instruction.

- Spilling reduces the live range of f.
  - And thus reduces its interferences.
  - Which results in fewer RIG neighbors for f.

Recompute RIG After Spilling

- The only changes are in removing some of the edges of the spilled node.
- In our case f now interferes only with c and d.
- And now the resulting RIG is 3-colorable.



Spilling Notes

- Additional spills might be required before a coloring is found.
- The tricky part is deciding what to spill.
- Possible heuristics:
  - Spill temporaries with most conflicts.
  - Spill temporaries with few definitions and uses.
  - Avoid spilling in inner loops.
- Any heuristic is correct.

Precolored Nodes

- Precolored nodes are nodes which are *a priori* bound to actual machine registers.

- These nodes are usually used for some specific (time-critical) purpose, e.g.:
  - for the frame pointer;
  - for the first N arguments (N=2,3,4,5).

Precolored Nodes (Cont.)

- For each color, there should be only one precolored node with that color; all precolored nodes usually interfere with each other.
- We can give an ordinary temporary the same color as a precolored node as long as it does not interfere with it.
- However, we cannot simplify or spill precolored nodes; we thus treat them as having "infinite" degree.

Effects of Global Register Allocation

Reduction in % for MIPS C Compiler

Program	cycles	total loads/stores	scalar loads/stores
boyer	37.6	76.9	96.2
diff	40.6	69.4	92.5
yacc	31.2	67.9	84.4
nroff	16.3	49.0	54.7
ccom	25.0	53.1	67.2
upas	25.3	48.2	70.9
as1	30.5	54.6	70.8
Geo Mean	28.4	59.0	75.4

Managing Caches

- Compilers are very good at managing registers.
  - Much better than a programmer could be.

- Compilers are not good at managing caches.
  - This problem is still left to programmers.
  - It is still an open question whether a compiler can do anything general to improve performance.

- Compilers can, and a few do, perform some simple cache optimization.

Cache Optimization

- Consider the loop:

```
for (j = 1; j < 10; j++)
  for (i = 1; i < 1000000; i++)
    a[i] *= b[i]
```

- This program has terrible cache performance.
  - Why?

Cache Optimization (Cont.)

- Consider now the program:

```
for (i = 1; i < 1000000; i++)
  for (j = 1; j < 10; j++)
     a[i] *= b[i]
```

  - Computes the same thing.
  - But with much better cache behavior.
  - Might actually be more than 10x faster!

- A compiler can perform this optimization
  - called *loop interchange.*

Concluding Remarks

- Register allocation is a "must have" optimization in most compilers:
  - Because intermediate code uses too many temporaries.
  - Because it makes a big difference in performance .

- Graph coloring is a powerful register allocation scheme (with many variations on the heuristics).

- Register allocation is more complicated for CISC machines.