COEN 175

Lecture 1: Overview and Lexical Analysis

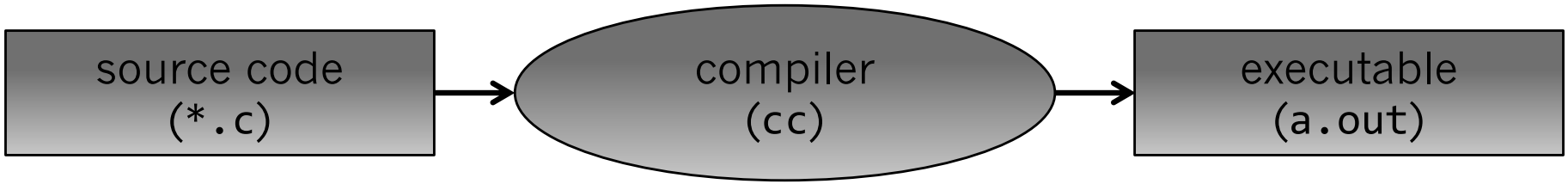
Read: Chapters 1, 2, 3.1–3.3

What is a Compiler?

- A **translator** is a program that takes as input a program written in one language, called the **source**, and produces as output an equivalent program in another language, called the **target**.
- If the source language is **high-level** and the target language is **low-level,** then we call the translator a **compiler**.
  - High-level languages include C, C++, Java, and Fortran.
  - Low-level languages include assembly, binary, and bytecode.

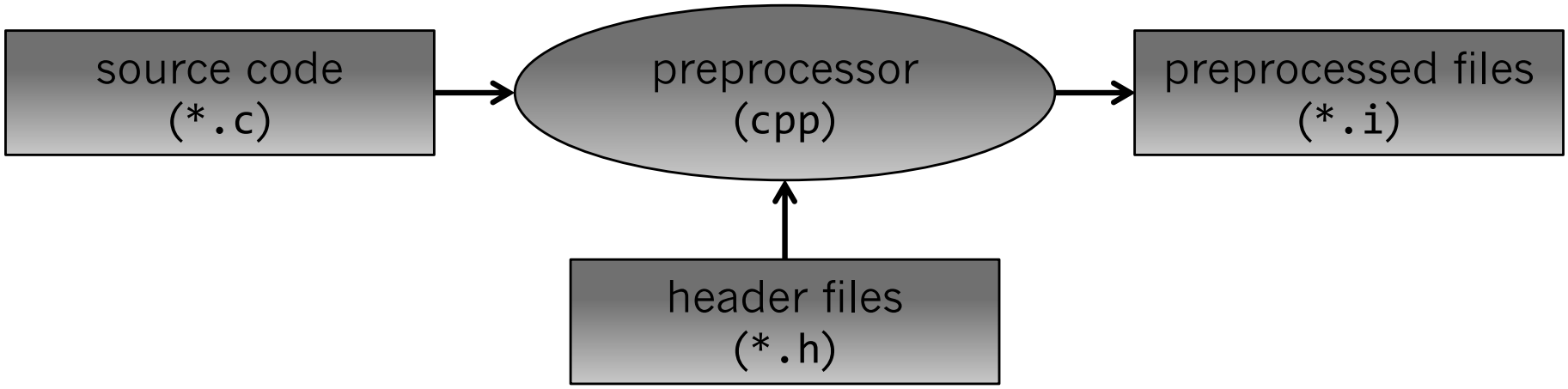
Anatomy of a Compiler

- From the simplest viewpoint, the compiler takes source code and produces an executable.
  - You may have used separate compilation before. But, in reality, there's still a lot more going on behind the scenes.



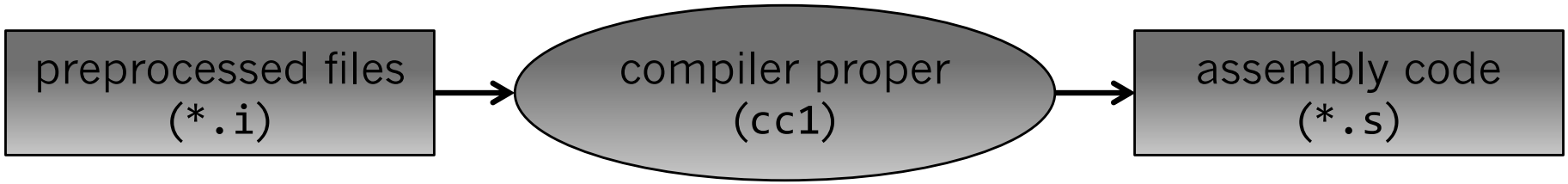
Anatomy of a Compiler

- Actually, the input files need to be preprocessed to expand macros and include files.
  - The actual C language does not contain `#define` or `#include` directives.



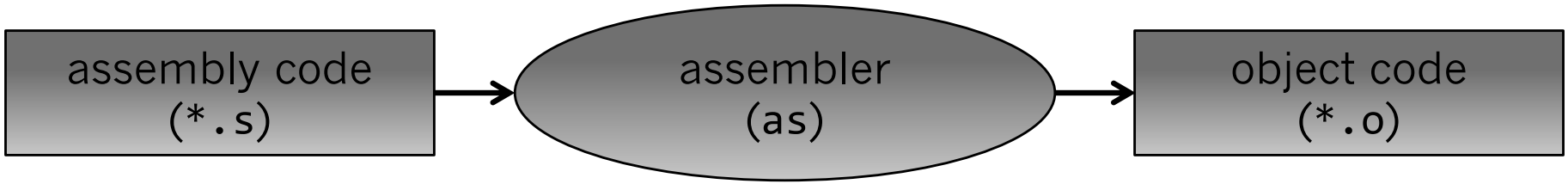
Anatomy of a Compiler

- The preprocessed files can now be compiled by the compiler proper.
  - The compiler proper does not produce binary files, only assembly code.



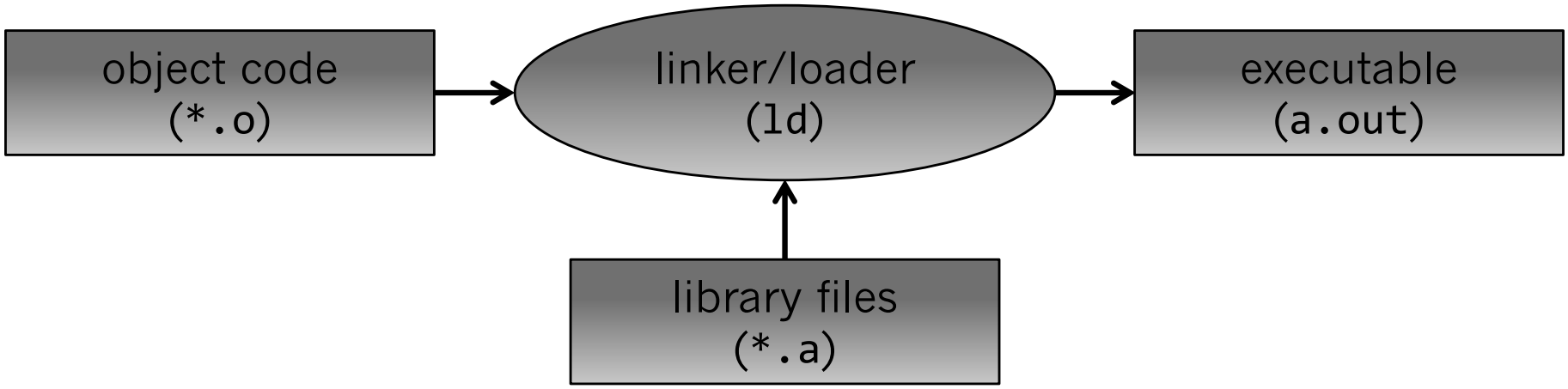
Anatomy of a Compiler

- The assembler produces binary object code, which is still not executable.
  - Many assemblers also let you define macros, but these are taken care of by the assembler itself.



Anatomy of a Compiler

- Finally, the linker/loader produces an executable using library files, which are packaged object files.
  - In dynamic linking, the actual library object files are not placed in the executable. Only a reference is placed there.



Phases of Compilation

- The compiler itself is broken into several phases:
  - Lexical analysis (scanning),
  - Syntax analysis (parsing),
  - Semantic analysis (static checking),
  - Optimization (optional),
  - Code generation (with storage allocation),
  - Instruction scheduling (optional).
- The first three phases are part of the **front end**, which is specific to the source language.
- The last two phases are part of the **back end**, which is specific to the target language.

Retargetable Compilers

- A **retargetable compiler** separates the front end and the back end.
- An **intermediate representation** (IR) connects the front and back ends.
  - An IR is at a higher level than assembly, but not so high as the source language.
  - GCC uses at least three intermediate representations: two tree structures and RTL (register transfer language).
- The optimizer often works on the IR (which means it is independent of the source and target) and is therefore “middle end.”

Lexical Analysis

- Lexical analysis is the process of reading the input program and grouping the characters into meaningful sequences called **tokens**.
- You are already experts in lexical analysis.
- Identify the tokens in the following program:

```
int main(void) {
    printf("Hello, world.\n");
}
```

Lexical Structure

- Specifying lexical structure in words is ambiguous.
- Example: an identifier consists of letters or an underscore followed by digits. Which are legal?
  - `a` ✗
  - `a5` ✗
  - `_` ✗
  - `_56` ✓
- Oops, I thought an identifier consists of **letters**, **or** an underscore followed by **digits**.
- Clearly, a more formal mechanism is needed.

What is a Language?

- A **language** is a set of strings.
  - A string is a sequence of symbols from an alphabet.
  - {0,1} is the binary alphabet.
  - ASCII and Unicode are also alphabets.
  - We let ϵ denote the empty string.
- If L_1 and L_2 are languages, then:
  - $L_1 \cup L_2$ is the set of all strings from L_1 or L_2.
  - $L_1 \cdot L_2$ is the set of all strings from L_1 followed by (**concatenated** with) any string from L_2.
  - L_1^* is the set of all strings from L_1 concatenated zero or more times.

Regular Expressions

- A **regular expression** denotes a set of strings.
- We have rules for building arithmetic expressions:
  - An integer or variable is an expression.
  - The sum or product of two expressions is an expression.
- We also have rules for building regular expressions. The base cases for expressions are:
  - ϵ denotes $\{\epsilon\}$ (the set consisting of the empty string).
  - a denotes $\{a\}$ (the set consisting of the symbol a).

Constructed Expressions

- If r_1 and r_2 are regular expressions for languages L_1 and L_2, respectively, then:
  - $r_1 | r_2$ denotes $L_1 \cup L_2$ (**alternation**).
  - $r_1 r_2$ denotes $L_1 \cdot L_2$ (**concatenation**).
  - r_1^* denotes L_1^* (**closure**).
- These are in order of increasing precedence:
  - *foo*|*bar* denotes (*foo*)|(*bar*) and not *fo* (*o*|*b*) *ar*.
  - *foo** denotes *fo* (*o**) and not (*foo*)*.
- We also introduce some shorthands:
  - r^+ denotes rr^* (one or more).
  - $r^?$ denotes $r \mid \epsilon$ (zero or one).
  - r^n denotes $rr \cdots r$ (exactly n times).

Character Classes

- A **character class** is another useful shorthand:
  - [*aeiou*] denotes *a*|*e*|*i*|*o*|*u*.
  - [*x*–*z*] denotes *x*|*y*|*z*.
  - [*abcx*–*z*] denotes *a*|*b*|*c*|*x*|*y*|*z*.
- A character class can also be **negated**:
  - [^*a*] denotes any symbol other than *a*.
  - [^0–9] denotes any symbol other than a digit.
- A character class can also be used with operators:
  - [*abc*]* denotes (*a*|*b*|*c*)*.

Example Expressions

- The keyword **for** in the C language.
  - Answer: *for*.
- The keyword **do** in Pascal, which is case-insensitive.
  - An answer: *do*|*Do*|*dO*|*DO*.
  - A better answer: (*d*|*D*)(*o*|*O*).
  - An even better answer: [*dD*][*oO*].
- An identifier in the C language.
  - Answer: [*a–zA–Z*_][*a–zA–Z*_0–9]*.
- A floating-point literal in the Pascal language.
  - Partial answer: [0–9]+.[0–9]+([*eE*][+–]?[0–9]+)?.

COEN 175

Lecture 2: Syntax Analysis and Grammars

Syntax Analysis

- Syntax analysis (i.e., parsing) is the process of taking the sequence of tokens and grouping them **structurally** into meaningful sentences.
- Syntax refers to structure, whereas semantics refers to meaning.
- Are regular expressions sufficient for describing the structure of a programming language?
- Consider matching balanced parentheses:
  - Actually, for clarity, we'll match balanced "*a*"s and "*b*"s.
  - Is the language $\{a^n b^n \mid n \geq 0\}$ regular?

Regular Languages

- Every regular expression describes a regular language, and vice versa.
- However, not all languages are regular:
  - $\{a^m b^n \mid m, n \geq 0\}$ is regular.
  - $\{a^n b^n \mid k \geq n \geq 0\}$ is regular.
  - $\{a^n b^n \mid n \geq 0\}$ is **not** regular.
- Informally, a regular expression can only "remember" a finite amount of information.
  - We cannot remember an arbitrary number of "*a*"s in order to make sure we match the same number of "*b*"s.

Context-Free Languages

- A context-free language is more powerful than a regular language.
- Each context-free language is described by a **context-free grammar** (CFG), and vice versa.
- A grammar consists of four parts:
  - A set, T, of **terminal** symbols.
  - A set, NT, of **nonterminal** symbols.
  - A special nonterminal, S, called the **start symbol**.
  - A set of rules or **productions** of the form $NT \rightarrow (T \cup NT)^*$.

Context-Free Grammars

- General typesetting conventions:
  - Nonterminals are uppercase letters and words.
  - Terminals are operators, lowercase letters, and bold words like **id** (often written `ID` on the board).
  - Greek letters are strings of terminals or nonterminals.
- Our first grammar:

  $L \rightarrow$ **id**
  $L \rightarrow L$ **,** **id**

  - OR -

  $L \rightarrow$ **id**
  | L **,** **id**

- Literally, a list, *L*, is either an identifier, or is another list followed by a comma and an identifier.

Derivations

- We say that $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a **derivation** if there is a production $A \rightarrow \gamma$.
  - In other words, we can replace a nonterminal *A* with the right-hand side of one of its productions.
- What sentences can we derive from our grammar?
  - $L \Rightarrow$ **id**
  - $L \Rightarrow L$ **, id** $\Rightarrow$ **id , id**
  - $L \Rightarrow L$ **, id** $\Rightarrow L$ **, id , id** $\Rightarrow$ **id , id , id**
- This grammar generates a non-empty, comma-separated list of identifiers.

Specific Derivations

- Usually, there is more than one possible derivation for the same sentence. Consider this grammar:

  $S \rightarrow A\ B$
  $A \rightarrow a$
  $B \rightarrow b$

- This grammar only generates one sentence, but can do so in two different ways:
  - $S \Rightarrow A\ B \Rightarrow a\ B \Rightarrow ab$.
  - $S \Rightarrow A\ B \Rightarrow A\ b \Rightarrow ab$.

- In a **leftmost** derivation, we always replace the leftmost nonterminal (and similarly for **rightmost**).

Expression Grammars

- Consider the following grammar for expressions:
  - $E \rightarrow E + E \mid E * E \mid \mathbf{id}$.
- Derive the input sentence **id** + **id** $*$ **id** using a leftmost derivation.
  - $E \Rightarrow E + E \Rightarrow \mathbf{id} + E \Rightarrow \mathbf{id} + E * E \Rightarrow \mathbf{id} + \mathbf{id} * E \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$.
  - $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow \mathbf{id} + E * E \Rightarrow \mathbf{id} + \mathbf{id} * E \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$.
- Which derivation should we use? It depends.
  - Show that the input **id** + **id** + **id** also has two different leftmost derivations.
  - Again, which should we choose? It depends.

Parse Trees

- A **parse tree** is a graphical representation of a derivation sequence.
  - The root of the tree is the start symbol.
  - Each nonleaf node is a nonterminal.
  - The leaves are the terminals of the input sentence.
  - If a nonleaf node, X, has children Y_1, Y_2, ..., Y_n (from left to right), then $X \rightarrow Y_1\ Y_2\ \ldots\ Y_n$ is a production.
- Top-down parsers construct parse trees from the root to the leaves.
- Bottom-up parsers construct parse trees from the leaves to the root.

Ambiguous Grammars

- A grammar is said to be **ambiguous** if, for **some** input sentence, **any** of the following are true:
  - There is more than one leftmost derivation.
  - There is more than one rightmost derivation.
  - There is more than one parse tree.
- Our compiler cannot use an ambiguous grammar.
  - How would you like it if the compiler decided arbitrarily the order of multiplication and addition?
- Therefore, we must first **disambiguate** the grammar without changing the language being defined.

Undecidability

- It would be nice if we had a tool that could tell us whether a grammar was ambiguous.
- Also, after disambiguation, we must be sure we have not changed the language being defined.
- It would be nice if we had a tool to tell us whether two grammars define the same language.
  - Unfortunately, these tools cannot exist.
- These two problems are **undecidable**: it is impossible to construct such an algorithm.

Recursive Grammars

- Draw parse trees for the input sentence **id , id , id** using each of the following grammars:
  - $L \rightarrow$ **id** | L , **id**
  - $L \rightarrow$ **id** | **id** , L
- The first grammar is said to be **left recursive**.
  - Left recursive grammars yield **left-heavy** parse trees.
  - We want left-heavy parse trees for **left associativity**.
- The second grammar is said to be **right recursive**.
  - Right recursive grammars yield **right-heavy** parse trees.
  - We want right-heavy parse trees for **right associativity**.

COEN 175

Lecture 3: Disambiguating Expression Grammars

Enforcing Precedence

- We want multiplication to have higher precedence than addition.
- We need multiplication lower (deeper) in the parse tree than addition.
- Parse trees grow when we apply productions.
- Therefore, we need a longer chain of productions to reach multiplication than to reach addition.
- We need addition to be a list of multiplications, not the other way around.

Disambiguated Grammar

- Here is our disambiguated grammar:

  $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow$ **id**

- An expression, *E*, is a list of terms that are added together (from left to right).
- A term, *T*, is a list of factors that are multiplied together (from left to right).
- Draw the parse trees for the following sentences:
  - **id** + **id** $*$ **id**.
  - **Id** + **id** + **id**.

Expressions

- In order to disambiguate an expression grammar, we must know two things:
  - **Precedence** tells us the order of operations.
  - **Associativity** tells us how operators of the same precedence are grouped.
- A related term is **arity**, which is the number of operands an operator expects:
  - If the arity is one, the operator is called **unary**.
  - If the arity is two, the operator is called **binary**.
  - If the arity is three, the operator is called **ternary**.

Enforcing Precedence

- Operators of higher precedence belong lower in the parse tree.
- Therefore, we need a longer chain of productions to reach them.
- Thus, we must place productions for operators of lower precedence before those for operators of higher precedence.
- Operators at the same level of precedence, such as + and –, are grouped together.

Binary Operators

- If the operator is left associative, its grammar rule is left recursive. If the operator is right associative, its grammar rule is right recursive.
- Disambiguate the following grammar:
  - $E \rightarrow E * E \mid E + E \mid E - E \mid E / E \mid$ **id**.
- The correct, disambiguated grammar is:

  $E \rightarrow E + T \mid E - T \mid T$
  $T \rightarrow T * F \mid T / F \mid F$
  $F \rightarrow$ **id**
- Draw the parse tree for the input **id** + **id** $*$ **id** / **id**.

Unary Operators

- Unary operators are either prefix or postfix:
  - Prefix operators are always right associative.
  - Postfix operators are always left associative.
- Disambiguate the following grammar:
  - $E \rightarrow E + E \mid E * E \mid - E \mid \mathbf{id}$.
- The correct, disambiguated grammar is:

  $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow - F \mid \mathbf{id}$

- Draw the parse tree for the input **id** + – **id** $*$ **id** + **id**.

Parentheses

- Parentheses are used to override precedence:
  - In effect, they have the highest precedence.
  - Any expression is legal within the parentheses.
- Disambiguate the following grammar:
  - $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$.
- The correct, disambiguated grammar is:

  $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow (E) \mid \mathbf{id}$
- Draw the parse tree for the input **id** + (**id** + **id**) $*$ **id**.

COEN 175

Lecture 4: Top-Down Parsing of Expressions

Top-Down Parsing

- Top-down parsers construct parse trees from the root to the leaves.
- Top-down parsers are suited for LL(*k*) grammars.
  - LL = left-to-right scan, leftmost derivation
  - *k* = number of symbols of look-ahead
- Bottom-up parsers are suited for LR(*k*) grammars.
  - LR = left-to-right scan, rightmost derivation in reverse
- The easiest type of top-down parser to implement is a **recursive descent** parser.

Recursive Descent Parsing

- A recursive descent parser is implemented as a set of functions:
  - Each nonterminal has a corresponding function that is responsible for matching its right-hand side.
  - Each nonterminal on the right-hand side is replaced by a call to its corresponding function.
  - Each terminal on the right-hand side is matched with (i.e., compared against) the current token.
- A recursive descent parser is a **predictive parser** if it never has to backtrack. Thus, we can parse any program in linear time.

A Simple Example

- Implement a parser for the following grammar:

  $S \rightarrow A\ B$
  $A \rightarrow a$
  $B \rightarrow b$

- Sample implementation:

```
void S() {
    A();
    B();
}
```

```
void A() {
    match('a');
}

void B() {
    match('b');
}
```

```
int lookahead = lexan();

void match(int t) {
    if (lookahead == t)
        lookahead = lexan();
    else
        error();
}
```

Selecting Alternatives

- Implement a parser for the following grammar:

  $S \rightarrow a\ A \mid b\ B$
  $A \rightarrow a$
  $B \rightarrow b$

- Sample implementation:

```
void A() {
    match('a');
}

void B() {
    match('b');
}
```

```
void S() {
    if (lookahead == 'a') {
        match('a');
        A();
    } else {
        match('b');
        B();
    }
}
```

A Big Problem

- Consider the following grammar:

  $E \rightarrow E + T \mid T$

- In order to match *E*, we first may have to match *E*. This will result in infinite recursion.

- We must **eliminate left recursion** by rewriting the grammar to use only right recursion:

  $E \rightarrow T\ E'$
  $E' \rightarrow +\ T\ E' \mid \epsilon$

- In other words, an expression is a term followed by zero or more terms summed together.

Eliminating Left Recursion

- An LL(k) grammar cannot have left recursion.
- Consider the following grammar, where no α_i begins with A:

  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$

- The grammar without left recursion is:

  $A \rightarrow \beta\, A'$

  $A' \rightarrow \alpha_1\, A' \mid \alpha_2\, A' \mid \ldots \mid \alpha_n\, A' \mid \epsilon$

- When coding our parser, we can either explicitly eliminate the left recursion or implicitly do so by combining the functions for A and A'.

A First Implementation

- Implement a parser for the following grammar:

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid \epsilon$$

- A straightforward implementation:

```
void E() {
    T();
    E'();
}
```

```
void E'() {
    if (lookahead == '+') {
        match('+');
        T();
        E'();
    } else {
        /* nothing to do */
    }
}
```

Eliminating Tail Recursion

- Implement a parser for the following grammar:

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid \epsilon$$

- A better implementation:

```
void E() {
    T();
    E'();
}
```

```
void E'() {
    while (lookahead == '+') {
        match('+');
        T();
    }
}
```

With Function Inlining

- Implement a parser for the following grammar:

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid \epsilon$$

- An even better implementation:

```
void E() {
    T();

    while (lookahead == '+') {
        match('+');
        T();
    }
}
```

An Advanced Example

- Implement a parser for the following grammar:

  $E \rightarrow E + T \mid E - T \mid T$
  $T \rightarrow T * F \mid T / F \mid F$
  $F \rightarrow (E) \mid \textbf{id}$

- First, we eliminate left recursion:

  $E \rightarrow T\ E'$
  $E' \rightarrow + T\ E' \mid - T\ E' \mid \epsilon$
  $T \rightarrow F\ T'$
  $T' \rightarrow * F\ T' \mid / F\ T' \mid \epsilon$
  $F \rightarrow (E) \mid \textbf{id}$

- We now implement functions for the rules following the method we just established.

An Advanced Example

- Looking at the first two rules:

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid -\ T\ E' \mid \epsilon$$

- Our implementation:

```
void E() {
    T();

    while (lookahead == '+' || lookahead == '-') {
        match(lookahead);
        T();
    }
}
```

An Advanced Example

- Looking at the next two rules:

  $T \rightarrow F\ T'$

  $T' \rightarrow *\ F\ T' \mid /\ F\ T' \mid \epsilon$

- Our implementation:

```
void T() {
    F();

    while (lookahead == '*' || lookahead == '/') {
        match(lookahead);
        F();
    }
}
```

An Advanced Example

- Looking at the final rule:

  $F \rightarrow$ (E) | **id**

- Our implementation:

```
void F() {
    if (lookahead == '(') {
        match('(');
        E();
        match(')');
    } else
        match(ID);
}
```

COEN 175

Lecture 5: Parsing of Declarations and Statements

Using Right Recursion

- Many left-recursive grammar rules involve lists and can simply be made right-recursive:
  - A left-recursive rule for a list is $L \rightarrow L$ **,** **id** | **id**.
  - A right-recursive rule for a list is $L \rightarrow$ **id** **,** L | **id**.
- Often rules have actions that are performed when the rule is matched:

  $E \rightarrow T \mid E + T$ `{ cout << "add" << endl; }`
- We must be sure to carry along the action when we eliminate left recursion:

  $E \rightarrow T\ E'$

  $E' \rightarrow +\ T$ `{ cout << "add" << endl; }` $E' \mid \epsilon$

Another Problem

- Consider the following grammar:

  $S \rightarrow a\ A \mid a\ B$
  $A \rightarrow a$
  $B \rightarrow b$

- How do we decide which rule to take when both begin with the same prefix?

- We must rewrite the grammar to eliminate this problem by factoring out the common prefix:

  $S \rightarrow a\ S'$
  $S' \rightarrow A \mid B$

- This technique is called **left factoring**.

Left Factoring

- A grammar must be left-factored to be LL(k).
- Consider the following grammar, where each β_i begins with a different symbol:

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$.

- The left-factored grammar is therefore:

  $A \rightarrow \alpha\, A'$

  $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$.

- Unlike left recursion elimination, left factoring usually involves declarations and statements, rather than expressions.

Two Examples

- Left factor the following grammar:

  declarator → *pointers* **id** | *pointers* **id** [**integer**]

- After left factoring, the grammar becomes:

  declarator → *pointers* **id** *declarator'*
  declarator' → [**integer**] | ϵ

- Left factor the following grammar:

  stmt → **if** (*expr*) *stmt* | **if** (*expr*) *stmt* **else** *stmt* | *expr* ;

- After left factoring, the grammar becomes:

  stmt → **if** (*expr*) *stmt stmt'* | *expr* ;
  stmt' → **else** *stmt* | ϵ

Dangling-Else Ambiguity

- Actually, the previous grammar is ambiguous. Consider the following phrase:

  if (E_1) **if** (E_2) S_1 **else** S_2

- When is S_2 executed?
  - One interpretation: when E_1 is true and E_2 is false.
  - Another interpretation: when E_1 is false.

- We can resolve this ambiguity by rewriting the grammar. However, the result is quite ugly.
  - Instead, we simply associate the **else** with the nearest unmatched **if**. So, the first interpretation is used.

Looking Across Rules

- Consider the two expressions `(int) x` and `(x + y)`.
- What rule should be used at the left parenthesis?
  - The simplest solution here is to add look-ahead.

```
void prefixExpr() {
    ...
    else if (lookahead == '(') {
        match('(');
        if (isSpecifier(lookahead)) {
            ...
        } else
            postfixExpr(true);
    } else
        postfixExpr(false);
}
```

```
void postfixExpr(bool lp) {
    primaryExpr(lp);
    ...
}

void primaryExpr(bool lp) {
    if (lp) {
        expr();
        match(')');
    } else if (...)
        ...
}
```

A Problem in Simple C

- Consider the following input:

```
int *foo(int a);
int *foo(int a) { }
```

- At what point do we know that the function is being declared instead of being defined?
  - We don't know until the end of the parameters.
- Adding look-ahead will not help here.
  - Why? We can have an arbitrary number of pointer declarators.
  - The only solution is to left factor the grammar.

Rewriting the Grammar

- Consider the relevant grammar rules:

function-definition
→ *specifier pointers* **id** (*parameters*) { ... }

global-declaration
→ *specifier global-declarator-list* **;**

global-declarator-list
→ *global-declarator*
| *global-declarator* **,** *global-declarator-list*

global-declarator
→ *pointers* **id**
| *pointers* **id** (*parameters*)
| *pointers* **id** [**integer**]

Rewriting the Grammar

- We can pull out the first declared identifier revealing the necessary left factoring:

function-or-global

→ *specifier pointers* **id** (*parameters*) { ... }
| *specifier pointers* **id** (*parameters*) *remaining-decls*
| *specifier pointers* **id** [**integer**] *remaining-decls*
| *specifier pointers* **id** *remaining-decls*

remaining-decls

→ **;**
| **,** *global-declarator remaining-decls*

Left-Factored Grammar

- After initial left factoring:

function-or-global
→ *specifier pointers* **id** *function-or-global'*

function-or-global'
→ (*parameters*) { ... }
| (*parameters*) *remaining-decls*
| [**integer**] *remaining-decls*
| *remaining-decls*

remaining-decls
→ **;**
| **,** *global-declarator remaining-decls*

COEN 175

Lecture 6: Syntax-Directed Translation and Semantic Analysis

Syntax-Directed Translation

- **Syntax-directed translation** is the principle that we use the syntax, or structure, of the input program to guide its translation.
- Thus, the parser is “in charge”:
  - The parser calls the lexer when it needs a new token.
  - The parser will also invoke functions or actions to perform semantic analysis.
  - In a single-pass compiler, the parser would output the target language.
  - In a multi-pass compiler, the parser would construct the intermediate representation.

Attributes

- Each grammar rule can have an action that is executed when it is matched.
- Often these actions need to communicate information between them.
- This communication takes the form of **attributes**.
  - An attribute is simply any value or object that we associate with a grammar symbol.
  - For a grammar symbol *X*, we simply use to *X.name* to refer to the attribute *name.*
  - We don't communicate using global variables because many grammar rules are recursive.

Example: A Calculator

- As we parse an expression, suppose we want to calculate its value.
- We could use an attribute, called `val`, to hold the value of an expression.
- We will assume that the lexer sets the attribute for a number to be the value of the number.
- We will have to compute the value for additions, multiplications, etc.

Attribute Grammar

- Add attributes to the following grammar:

  $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow (E) \mid \textbf{num}$

- The attribute grammar is simply:

$E \rightarrow E + T$		`{ E.val = E.val + T.val; }`
$\mid T$		`{ E.val = T.val; }`
$T \rightarrow T * F$		`{ T.val = T.val * F.val; }`
$\mid F$		`{ T.val = F.val; }`
$F \rightarrow (E)$		`{ F.val = E.val; }`
$\mid$ **num**		`{ F.val = num.val; }`

Kinds of Attributes

- If an attribute of a nonterminal is computed from the attributes of its children, it is **synthesized**.
  - In a recursive-descent parser, a synthesized attribute is the return value of a function.
  - Our calculator used synthesized attributes exclusively.
- If an attribute of a symbol is computed from the attribute of its parent (or ancestors), it is **inherited**.
  - In a recursive-descent parser, an inherited attribute is a parameter to a function.
  - A declaration in C would use inherited attributes since the type specifier must be passed down into the declarator.

Abstract Syntax Trees

- A parse tree is also called a concrete syntax tree.
  - A parse tree has every token in the input program and every production used, many of which are not needed.
  - For example, expression grammars often have long chains of productions used only to enforce precedence.
- An **abstract syntax tree**, or AST, omits any information that we don't need in later analysis.
  - An AST is a common intermediate representation.
  - There is no one, definitive AST. It is a design choice as to which symbols are included.
  - Although, there are some common conventions, especially using **expression trees** for expressions.

Semantic Analysis

- **Semantic analysis** is the process of determining the meaning of the well-formed input program.
  - Most of semantic analysis is **semantic checking**.
- Semantic checks are either **static** or **dynamic**:
  - "Static": compile-time (things aren't changing)
  - "Dynamic": run-time (things are changing)
- Checking that a variable is declared before being used is a static semantic check.
- Checking that an array index is within the bounds of the array is a dynamic semantic check.

Static Semantic Checks

- Uniqueness checks: verifying that there is one and only one definition of an object or identifier.
  - Is a variable declared before being used?
  - This is the third phase of the project.
- Type checks: verifying that the operands to an operator are of the correct type.
  - Can you add a pointer and an integer?
  - This is the fourth phase of the project.
- Control checks: verifying that the control structure of a program is consistent.
  - Is a `continue` inside a loop?

COEN 175

Lecture 7: Symbol Tables

Uniqueness Checks

- At the time of its declaration, information about an identifier must be recorded for later use.
- There should be one and only one declaration.
- The central repository for all such information is called the **symbol table**.
- A **symbol** is simply any named object we store information about:
  - A variable such as `x`
  - A function such as `main`
  - A named type such as `uint8_t`

Basic Table Operations

- What kinds of operations do we need on our table?
  - `insert(symbol)`: add a symbol to the table
  - `find(name)`: return a symbol given its name
- At declaration time, we do an `insert` operation, which will fail if the name is already in use.
- Upon seeing a use, we do a `find` operation, which will return the associated symbol for our use.
- Unfortunately, most languages have **scoping** that complicates our design.

Scoping

- A **scope** is simply a region of code over which a given name is valid. You can think of it as the lifetime of the name.
- A name is said to be **bound** to a declaration.
- Two commonly used kinds of scoping:
  - **Static scoping**: the declaration to which a name is bound is known at compile time
  - **Dynamic scoping**: the declaration to which a name is bound is not known until run time
- Most languages, including C, use static scoping.

Static Nested Scoping

- Most languages implement **static nested scoping**:
  - Scopes can be nested hierarchically within each other.
  - An inner scope must be contained completely within an outer scope. It cannot "span" two different outer or enclosing scopes.
- Most languages also allow **shadowing**:
  - A name may be redeclared within an inner scope.
  - When a name is used, it is bound to the nearest or "innermost" declaration.
  - Any object with that name declared in an outer scope will be inaccessible.

Example: Shadowing

```
int x, y;

int f(int y, int z) {          // shadowing of global y
    int a, b;

    {
        int x, y;              // shadowing of x and y
        x = a + y;             // assigns to local x
    }

    x = f(a, y);               // assigns to global x
}

int g(int f) {                 // shadowing of global f
    f = 1;                     // assigns to parameter f
}
```

Symbol Table Design

- Supporting static nested scoping requires a redesign of our symbol table.
  - Before we had one mapping from names to symbols.
  - We now need a mapping for each scope.
- How can we model the nesting of scopes?
- We can use a **stack**:
  - When a scope is created or "**opened**" then we **push** a new scope onto the stack.
  - When a scope is destroyed or "**closed**" then we **pop** it from the stack.

Symbol Table Operations

- Our new design requires a new set of operations:
  - `open(scope)`: create and push a new scope, which is linked to the given enclosing scope
  - `close()`: pop the topmost scope
  - `insert(scope, symbol)`: insert a symbol into a scope
  - `find(scope, name)`: search a given scope for a name
  - `lookup(scope, name)`: starting with a given scope, search it **and all enclosing** scopes for a name
- The key operation is `lookup`, which will find the nearest declaration.

Reporting Errors

- If a `lookup` operation fails, then the name is **undeclared**.
- If an `insert` operation fails, then the name is **previously declared**.
- Some languages may allow redeclarations:
  - In C (and Simple C), a global name may be declared multiple times so long as all such declarations have identical types.
  - We must first do a `find` operation and compare types.
  - However, a global object may be defined only once.

Example: Scoping Errors

```
int x, a[10];
int f(int a, int b);

int f(int x, int y) {
    int y;                  // redeclaration of y

    {
        int w;
        w = x + z;          // z undeclared
        y = g();            // error? it depends if implicit
    }                       // declarations are allowed

    w = a[x];               // w undeclared
}

int a[5];                   // conflicting types for a
int f(int x, int y) {}      // redefinition of f
```

Necessary Data Types

- Our symbol table requires three primary data types:
  - `Type`, which holds simple C information;
  - `Symbol`, which holds a name and its type;
  - `Scope`, which holds the symbols.
- The symbol table itself is a **stack** of scopes.
  - We explicitly link each scope to the next scope.
  - Thus, we create a linked list of scopes.
- The symbol table can be viewed as a **tree** of scopes.
  - We explicitly link each scope to the parent scope.
  - Thus, we create a hierarchy of scopes.

COEN 175

Lecture 8: Data Type Modeling

Types in Simple C

- We don't want to start writing C++ code just yet.
- Instead, we want to identify what information a type should hold.
- In other words, we need the requirements.
- A type consists of a **specifier** and **declarators**.
  - A specifier is one of `int`, `char`, or `double`.
  - A declarator can be a pointer, array, or function.
- In what ways can these be combined?

Examples

- Which of these declarations are legal in Simple C?

```
int x;            // legal: int
double *p;        // legal: pointer to double
char a[10];       // legal: array of char
int **q;          // legal: pointer to pointer to int
double *b[10];    // legal: array of pointers to double
char (*c)[10];    // illegal: pointer to array of char
int **d[10];      // legal: array of pointers to pointers
int *(*e)[10];    // illegal: pointer to array of pointers
```

More Examples

- Which of these declarations are legal in Simple C?

```
int f(int x);        // legal: func returning int
double *g(void);     // legal: func returning ptr to double
char a(void)[10];    // illegal (also illegal in C)
int b[10](void);     // illegal (also illegal in C)
double **h(void);    // legal: func returning ptr to ptr
char (*p)(int x);    // illegal: ptr to func returning char
int *(*q)(int x);    // illegal: ptr to func returning ptr
int f();             // illegal: Simple C doesn't allow it!
```

Types in Simple C

- Pointer declarators are always allowed, but always have the lowest precedence:
  - Array of pointers, not pointer to array;
  - Function returning pointer, not pointer to function.
- At most one array or function declarator is allowed, and it always has the highest precedence.
- We can choose to model types as a triple:
  - Specifier;
  - Indirection (number of pointers);
  - Kind: function, array, or scalar (no declarators).

Examples

- What are the triples for these declarations?

```
int x;                  // (INT, 0, SCALAR)
double *p;              // (DOUBLE, 1, SCALAR)
char a[10];             // (CHAR, 0, ARRAY)
int **q;                // (INT, 2, SCALAR)
double *b[10];          // (DOUBLE, 1, ARRAY)
int **d[10];            // (INT, 2, ARRAY)
int f(int x);           // (INT, 0, FUNCTION)
double *g(int x);       // (DOUBLE, 1, FUNCTION)
```

Computing Information

- Should we start writing C++ code yet? No.
  - We still need to make sure we can gather the necessary information.
  - No point in writing a class if we can't test it!
- Let's modify our parser to compute the necessary information.
  - We'll need to pass information between rules.
  - To do that we will need to use inherited and synthesized attributes.

What Rules Are Affected?

- Let's look at the relevant grammar rules:

declaration	→	*specifier declarator-list* **;**
declarator-list	→	*declarator*
	|	*declarator* **,** *declarator-list*
declarator	→	*pointers* **id**
	|	*pointers* **id** [**integer**]
pointers	→	ϵ
	|	* *pointers*

- All the actions will take place in *declarator*.
  - Pointer declarators will be a **synthesized** attribute.
  - The specifier will be an **inherited** attribute.

Computing Indirection

- Let's modify the code for `pointers()` to compute the number of levels of indirection.

```
// modified code

unsigned pointers() {
    unsigned count = 0;

    while (lookahead == '*') {
        match('*');
        count ++;
    }

    return count;
}
```

Computing the Specifier

- Let's modify the code for `specifier()` to return the type specifier.

```
// modified code

int specifier() {
    int typespec = lookahead;

    if (lookahead == INT)
        match(INT);
    else if (lookahead == DOUBLE)
        match(DOUBLE);
    else
        match(CHAR);

    return typespec;
}
```

Inheriting the Specifier

- Let's modify the code for `declaration()` to pass in the type specifier.

```
// modified code

void declaration() {
    int typespec = specifier();
    declarator(typespec);

    while (lookahead == ',') {
        match(',');
        declarator(typespec);
    }

    match(';');
}
```

Inheriting the Specifier

- Finally, let's modify the code for `declarator()` to accept the inherited attribute.

```
// modified code

void declarator(int typespec) {
    unsigned indirection = pointers();
    match(ID);

    if (lookahead == '[') {
        match('[');
        match(INTEGER);
        match(']');
        cout << "(" << typespec … << ", ARRAY)" << endl;
    } else
        cout << "(" << typespec … << ", SCALAR)" << endl;
}
```

Missing Information

- Our requirements analysis of types in Simple C has left out two important pieces of information:
  - Array length
  - Function parameters
- Clearly, we can model these as follows:
  - Array length is simply an unsigned value.
  - Function parameters are a list (i.e., vector) of types.
  - We can either create separate C++ types (i.e., subclasses) for arrays and functions, or just store them all as one C++ type since the overhead is minimal.

COEN 175

Lecture 9: C++ Coding

C++ Guidelines

- Use a `typedef` to shorten a long type declaration and make it easier to understand.
  - `typedef std::vector<std::string> strings;`
- Use `nullptr`, and not `NULL`, when referring to a pointer type.
  - `NULL` is just zero and can sometimes be confused with an integer: `void f(int x)` vs. `void f(int *x)`.
- Pick a naming convention and stick to it!
  - Class names in Pascal Case, Camel Case, etc.
  - Member variables begin with an underscore.

C++ Classes

- Class declarations are placed in the header file.
- Public variables are usually a bad idea.
  - There's usually always a reason to encapsulate (i.e., hide behind an interface) the state of a class.
- Member function definitions should be placed in the source file.
  - If the function is small then it can be written inside the class declaration if you want.
- Make life easy on yourself and overload the output stream operator. It'll be useful for debugging.

C++ Header Files

- Always `#include` any headers that define any types that your class needs.
  - Otherwise, you rely on the client to do it for you. Bad idea!
- Protect your header file against multiple inclusions with `#ifndef` and `#endif`.
- Do not open the standard namespace.
  - Otherwise, the client might end up with name conflicts.
- Be consistent in the order of declarations.
  - Private, then protected, then public.
  - Constructors, then accessors, then other member functions.

C++ Source Files

- It's okay to open the standard namespace here.
- Define your member functions in the same order as you declared them in the header file.
- It's okay to write non-member functions as well, especially utility functions.
  - Place them before the member functions.
  - Declare them static so they aren't visible outside the file.
- Use the C++ initialization syntax in constructors.
  - It's required in some cases and always a good idea.

Simple C Types

- Last time we identified the requirements for our `Type` class to represent Simple C types.
- Every type has a specifier, indirection, and kind.
  - An array type also has its length.
  - A function type also has its parameters.
- We will overload the == and != operators so that checking for identical types would be easy.
- Note that types will be **immutable** as we will only provide accessors and not mutators.

Symbols

- Once we have developed and tested our `Type` class, we can move on to our `Symbol` class.
- For now, a symbol will just hold a name and a type.
  - In later phases of the project, it will hold more information.
- Like types, symbols are more or less immutable.
  - Once created, the information doesn't change since neither the name nor type can change.
  - Thus, we will provide accessors but not mutators.

Scopes

- A `scope` is simply a container for symbols.
  - Each scope holds a collection of symbols.
  - Each scope also has a link to its enclosing scope.
- A scope will need to support basic ADT operations:
  - `insert(symbol)`: add the symbol to the scope
  - `remove(name)`: remove the symbol with the given name
  - `find(name)`: return the symbol with the given name
  - `lookup(name)`: return the nearest symbol with the name
- To preserve declaration order, we will simply store the symbols in a vector.

Architectural Design

- We are building our semantic checker as a **layered architecture**.
- The lower layer is an **object-oriented design**.
  - We have our three classes: `Type`, `Symbol`, and `Scope`.
  - Very little at this layer deals with the language semantics.
- The upper layer is a **functional design**.
  - We will have functions to open and close scopes.
  - We will have functions to manage symbols and report errors according to our language semantics.

The Semantic Checker

- The semantic checking logic belongs in its own module, `checker.cpp`.
- In this module, we will have functions to manage scopes and symbols.
  - `openScope()`, `closeScope()`
  - `declareFunction()`, `defineFunction()`
  - `declareVariable()`, `checkIdentifier()`, `checkFunction()`
- These functions will be called by the parser to implement the required semantic checks.

COEN 175

Lecture 10: Type Systems

Type Systems

- The **type system** of a language is a set of rules that assign types to expressions.
- Sometimes the rules are obvious:
  - Adding two integers in C yields an integer.
  - Subtracting two integers in C yields an integer.
- Sometimes the rules are not so obvious:
  - Can you add a pointer and an integer in C?
  - Yes, the pointer is moved by the integer number of objects.
  - Can you add two pointers in C?
  - No, this is illegal (even in C).

Type System Specification

- We can specify a type system in a variety of ways.
- We can use English text.
  - No explanation needed, but often ambiguous.
- We can use type tables.
  - These are like truth tables, but for types.
  - Simple for single operators, but no way of composing them to determine types of larger expressions.
- We can use type expressions.
  - A formal way of describing a type system.
  - Can be composed algebraically.

Overloaded Operators

- An operator is said to be **overloaded** if it does **different operations** depending upon the number or types of its operands.
- As an example, & is overloaded in C:
  - In its unary form, it takes the address.
  - In its binary form, it performs bitwise-and.
- As another example, * is overloaded in Simple C:
  - In its unary form, it performs a dereference.
  - In its binary form, it performs multiplication.

Polymorphic Operators

- An operator is said to be **polymorphic** if it does the **same operation** regardless of the type or number of its operands.
- As an example, consider the address operator:
  - It operates on any type of operand, but performs the same operation regardless of the type.
- True polymorphic functions don't really exist in C.
  - Macros in C and especially templates in C++ come close, though they are examples of compile-time polymorphism.
  - Most functional languages have run-time polymorphism.

Type Conversions

- Languages allow you to convert an object of one type to an object of another type.
- An **implicit** type conversion is called a **coercion**.
  - Adding a `float` and an `int` in C results in the `int` being converted to a `float` without your interaction.
  - Some languages perform run-time checks on coercions.
- An **explicit** type conversion is called a **type cast**.
  - In C, the desired type is written in parentheses before the expression.
  - In other languages, an explicit function or method call (often a constructor) is required.

Type Equivalence

- What does it mean for two types to be equivalent?
  - Type equivalence is important as it is used for assignment, which includes passing parameters by value.
  - We can only assign “apples to apples.”
- Under **name equivalence**, two types are equivalent if and only if they have the same name.
- Under **structural equivalence**, two types are equivalent if and only if they have the same structure.

Name Equivalence

- Which objects have equivalent types under name equivalence?

```
typedef int height;
struct foo { int x, y; } s1;
struct bar { int x, y; } s2;
height x;
int y;
```

- None of these objects have equivalent types.
  - We cannot even initialize `x` with a value such as 123.

Structural Equivalence

- Which objects have equivalent types under structural equivalence?

```
typedef int height;
struct foo { int x, y; } s1;
struct bar { int x, y; } s2;
height x;
int y;
```

- The variables `x` and `y` have equivalent types.
- The variables `s1` and `s2` have equivalent types.

Type Equivalence in Practice

- Pure name equivalence is too restrictive.
- Pure structural equivalence is too expensive and does not work on recursive types.
- Most languages use a compromise.
- C uses structural equivalence for everything but structures, for which it uses name equivalence.
  - In essence, any `typedef` is expanded.

Static vs. Dynamic Typing

- A language can use **static** or **dynamic** typing.
- Statically typed languages such as C perform type checking at compile time.
- Dynamically typed languages such as PhP and Python perform type checking at run time.
- C++ is still statically type-checked, but does some run-time type lookups for polymorphic functions.
- Static type checking lets us catch errors early.

Type Tables

- Type tables are like truth tables but types are used instead of Boolean values.
- Consider a language with types `integer` and `real`. What would the table for addition be?

+	integer	real
integer	integer	real
real	real	real

Handling Errors

- We can introduce an error type to make handling of errors explicit.
- Consider a language with types `integer` and `real` along with the error type. What would the table for addition be?

+	integer	real	error
integer	integer	real	error
real	real	real	error
error	error	error	error

Summary

- Type tables are simple and intuitive.
- However, they do not scale well if a language has a rich set of operators and types.
- More importantly, they do not let us manipulate types in an algebraic way.
- Type expressions are a compact specification that let us manipulate types algebraically.
- However, like regular expressions, type expressions take some getting used to.

COEN 175

Lecture 11: Type Expressions

Type Expressions

- A **type expression** or **type signature** denotes the type of an expression.
- Any built-in or “atomic” type in the language is a legal type expression.
- If *S* and *T* are type expressions, then:
  - $S \rightarrow T$ denotes a mapping from type *S* to type *T*
  - $S \times T$ denotes a (Cartesian) product of type *S* and type *T*
  - pointer(*T*) denotes a pointer to type *T*
  - array(*T*, length) denotes an array of type *T*

Examples

- Give type expressions for the following declarations:

```
int x;            // int
double *p;        // pointer(double)
char a[10];       // array(char, 10)
int **q;          // pointer(pointer(int))
double *b[10];    // array(pointer(double), 10)
char (*c)[10];    // pointer(array(char), 10)
int **d[10];      // array(pointer(pointer(int)), 10)
long *(*e)[10];   // pointer(array(pointer(long)), 10)
```

More Examples

- Give type expressions for the following declarations:

```
void f(int);              // int -> void
double g(int *);          // pointer(int) -> double
char h(int, int);         // int x int -> char
int (*p)(int);            // pointer(int -> int)
double **x(int);          // int -> pointer(pointer(double))
char *(*q)(void);         // pointer(void -> pointer(char))
int (*a[4])(double);      // array(pointer(double -> int), 4)
char y(int (*)(long));    // pointer(long -> int) -> char
```

Operators vs. Functions

- An operator is just a special way of writing a function with a different syntax and name.
- In languages such as Scheme, operators are in fact functions with the same syntax.
  - We don't write `a + b` but rather `(+ a b)` just as we would write `(cons a b)`.
- An operator such as + is just a binary function.
- Thus, we can write type expressions for operators.

Example: Division

- What are the type expressions for `/` in Simple C?
  - `int` × `int` → `int`
  - `double` × `double` → `double`
- Thus, the `/` operator is overloaded.
  - Either integer or floating-point division is performed.
- Two questions should come to mind:
  - What about characters?
  - What about division with mixed-type operands?

Type Promotions

- In C and Simple C, all characters are coerced to integers before any operation.
- This coercion is called a **promotion**.
- There are two automatic promotions in Simple C:
  - A character is promoted to an integer.
  - An array is promoted to a pointer.
- Promotions help reduce the total number of cases.
  - In the original C standard, a `float` was always promoted to a `double`, but a more recent standard eliminated it.

Type Coercions

- Type coercions also reduce the number of cases.
- Whereas a processor will have both integer and floating-point division, it likely won't have a mixed-type operation.
- An `int` can be converted to a `double` with little or no loss in precision, so it can be coerced.
- C includes coercions for all built-in types, both signed and unsigned, and integral and floating-point.

Type Coercions

- Type coercions are just operations or functions that are implicitly performed for us.
- Therefore, we can write type expressions for them:
  - `char` → `int`
  - `int` → `double`
  - array(α, *n*) → pointer(α)
- In this last expression, we use α to denote any type.
  - Thus, this last coercion is itself polymorphic.

Another Example

- What are the type expressions for * in Simple C?
  - `int` × `int` → `int`
  - `double` × `double` → `double`
  - pointer(α) → α
- The first two expressions are for the binary case of multiplication.
- The last expression is for the unary case of pointer dereference.
  - Given a pointer to an object of some type, the result is an object of that type.

COEN 175

Lecture 12: More Type Expressions

Review

- A **type expression** or **type signature** denotes the type of an expression.
- Any built-in or “atomic” type in the language is a legal type expression.
- If *S* and *T* are type expressions, then:
  - $S \rightarrow T$ denotes a mapping from type *S* to type *T*
  - $S \times T$ denotes a (Cartesian) product of type *S* and type *T*
  - pointer(*T*) denotes a pointer to type *T*
  - array(*T*, length) denotes an array of type *T*

Example: Addition

- What are the type expressions for + in Simple C?
  - `int` × `int` → `int`
  - `double` × `double` → `double`
  - pointer(α) × `int` → pointer(α)
  - `int` × pointer(α) → pointer(α)
- The first two expressions are for addition.
- The last two are for pointer arithmetic.
  - We have two expressions since addition is commutative.

Example: Subtraction

- What are the type expressions for `-` in Simple C?
  - `int` × `int` → `int`
  - `double` × `double` → `double`
  - pointer(α) × `int` → pointer(α)
  - pointer(α) × pointer(α) → `int`
- The first two expressions are for subtraction.
- The last two are for pointer arithmetic.
  - If adding an offset to a pointer yields a new pointer, then we should be able to subtract the two pointers to get the offset.

Example: Address

- What is the type expression for & in Simple C?
  - $\alpha \rightarrow$ pointer(α)
- Given an object of some type, the result is a pointer to that object.
- What was the type expression for dereference?
  - pointer(α) $\rightarrow \alpha$
- We see that the address and dereference operators are **inverses**.
  - For example, *&x is the same as just writing x!

Example: Indexing

- What is the type expression for `[]` in Simple C?
  - pointer(α) × `int` → α
- Why do we use a pointer and not an array?
  - All arrays are promoted to pointers.
- The true semantics in C are more interesting.
  - E_1`[`E_2`]` is defined by the C standard as `*(`E_1 `+` E_2`)`.
  - By that definition, `a[i]` is equivalent to `*(a + i)`, which is equivalent to `*(i + a)`, which is equivalent to `i[a]`!

Example: Logical Or

- What are the type expressions for `||` in Simple C?
  - `int` × `int` → `int`
  - `int` × `double` → `int`
  - `int` × pointer(α) → `int`
  - `double` × `int` → `int`
  - `double` × `double` → `int`
  - `double` × pointer(α) → `int`
  - pointer(α) × `int` → `int`
  - pointer(α) × `double` → `int`
  - pointer(α) × pointer(β) → `int`
- Why so many? Short-circuit evaluation.

Are These Errors?

- Assume that `x` is declared as type `int`.
- The statement `x = 1` is certainly legal, as `1` also has type `int`.
- Is the statement `1 = x` legal? Why or why not?
- Assume that `p` is declared as a pointer to an `int`.
- The statement `p = &x` is again certainly legal.
- Is the statement `&x = p` legal? Why or why not?

Lvalues vs. Rvalues

- The statement `x = y` says to take the value of `y` and place it into the location denoted by `x`.
- The problem we had earlier is that both `1` and `&x` do not have locations.
- An expression that denotes a location is an **lvalue**.
  - It is so called because it can be used on the left-hand side of an assignment statement.
- An expression that only denotes a value is an **rvalue**.
  - Both `1` and `&x` are not lvalues; they are rvalues.

Lvalues

- Not every identifier denotes an lvalue.
  - Scalar variables are lvalues.
  - Functions and arrays are **not** lvalues (in Simple C).
  - You could consider them to be **constant** lvalues in C.
- Most expressions do not yield lvalues, but some do.
  - A dereference does: `*p = 1`.
  - An array index does: `a[i] = 1`.
- Some expressions require lvalues.
  - Assignment does: `x = 1` is legal, but `1 = x` is not.
  - Address does: `p = &x` is legal, but `p = &1` is not.

Imperative Languages

- Formally including lvalues in our specification would require some new notation.
- In an imperative language such as C:
  - Names are bound to declarations;
  - Declarations are mapped to locations;
  - Locations store values.
- In functional languages, declarations are mapped directly to values.
- A graduate level class in formal semantics would cover these topics in a lot more depth.

Summary

- A type system is a set of rules that assign types to expressions.
- Type expressions or signatures are a formal and compact way of specifying those rules.
- C is a statically typed language, albeit weakly typed.
  - C++ is somewhat stronger, but still not much.
  - Ada is a strongly, statically typed language.
- Overloading, polymorphism, coercion, and casting are all necessary evils of type systems.

COEN 175

Lecture 13: Storage Allocation

Storage Allocation

- The compiler must ensure that storage will be allocated at run time for the program's data.
- Note that the compiler does not itself allocate space.
  - The compiler does not use `malloc` or `new`.
  - Rather, the compiler must generate code that will allocate the space at run time.
  - The compiler must also ensure that its generated code uses this space correctly.
  - Each object should be assigned its own memory location in accordance with the rules of the language.

Access Patterns

- Languages provide different allocation mechanisms depending upon how objects are used.
  - How many copies of an object exist?
  - What is the lifetime of an object?
  - How do we refer to objects?
- Simpler mechanisms are easier to implement and might be faster, but might be too restrictive.
- The three commonly used allocation mechanisms are **static**, **stack**, and **dynamic** allocation.

Static Allocation

- There is one and only one copy of an object.
- The lifetime of the object is the lifetime of the running program.
- Objects are simply referred to by name.
- Because of these properties, space for a statically allocated object can be laid out at **compile time**.
- What are some common examples?
  - Global variables and local variables declared as `static`
  - Code (your program is stored in memory too!)

Stack Allocation

- There can be multiple copies of an object. Each copy is called an **activation**.
- However, only the **most recently** allocated copy is (directly) accessible.
- Objects are still referred to by name.
- Each function is responsible for allocating and deallocating memory to hold these objects.
- What are some common examples?
  - Local variables and parameters

Dynamic Allocation

- There can be multiple copies of an object.
- The lifetime of objects is directly controlled by the programmer.
- Since the programmer can declare only a finite number of names, how do we refer to such objects?
  - We refer to such objects **indirectly** (i.e., using pointers).
- What are some common examples?
  - Objects created from `malloc` and `new`.

Language Support

- Does the C language support static allocation?
  - Yes ... global variables and code are statically allocated.
- Does the C language support stack allocation?
  - Yes ... C calls it “automatic” allocation.
  - Stack allocation is necessary to support recursion.
- Does the C language support dynamic allocation?
  - Surprisingly, no ... the **language** does not.
  - For that matter, the C language does not support I/O. Huh?
  - The C **standard library** does; these design decisions were key in allowing C to be easily ported to new platforms.

Memory Segments

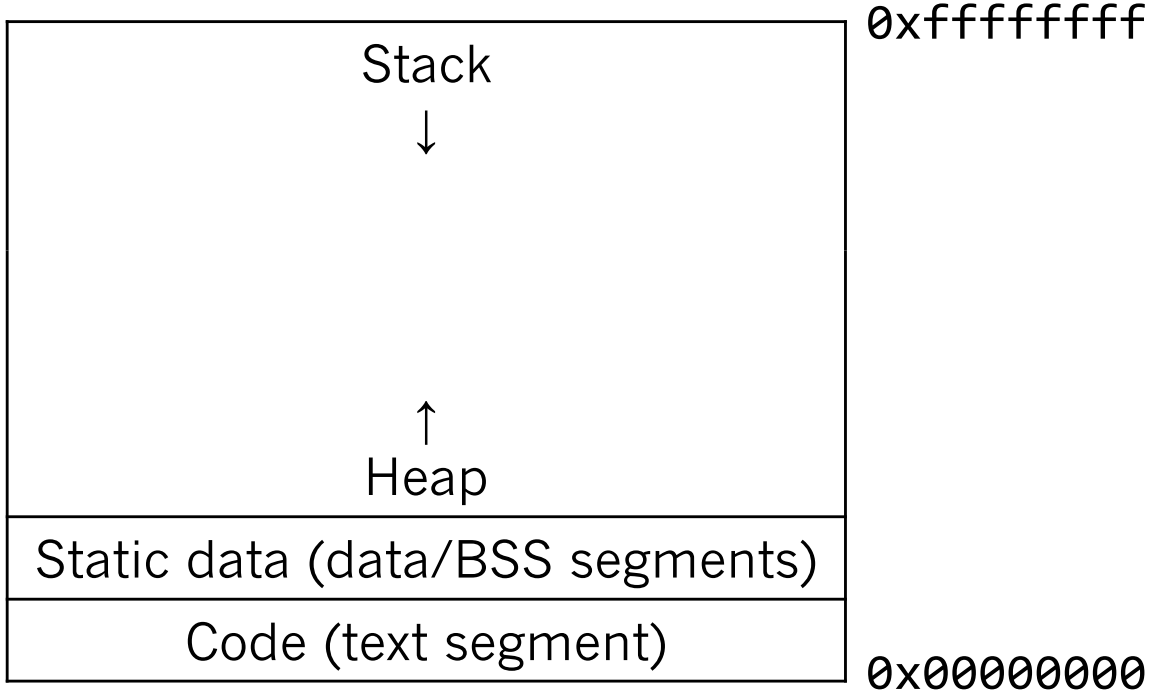
- Suppose we need to support static, stack, and dynamic allocation of data in our running program.
- How many different memory segments do we need?
  - Static data requires a segment.
  - Stack data requires a segment (“the run-time stack”).
  - Dynamic data requires a segment (“the heap”).
  - The code itself requires a segment.
- Don’t forget that your code is in memory as well!
- Trying to access an address not in one of your allocated segments will cause a **segmentation fault**.

Memory Layout

- Which segments have fixed size during execution?
  - The static data and code segments don't change size.
  - The stack and heap can grow and shrink.
- Suppose we have an entire 32-bit (or 64-bit) address space available to us.
- How might we place the four segments in memory?
  - Place the two fixed-size segments at either or both ends.
  - Place the other two segments at opposite ends in the remaining memory and have them grow towards each other.

Example Memory Layout

- The code and static data are placed at the start of the address space and the stack grows down.



Example

- Identify how each object in the following C program is allocated:

```
int x, *p;                  // x, p: static

int f(int y) {              // f: static, y: stack
    static int z, *q;       // z, q: static

    p = malloc(10);         // *p: dynamic
    q = &z;                 // *q: static
}
```

- Note that `*q` and `z` refer to the same object.
  - We say that `*q` and `z` are **aliases** for each other.

Static Allocation

- Easiest of all allocation mechanisms because it can be done at compile time.
- There is only one copy of an object and the lifetime is that of the running program.
- The compiler simply uses an assembler directive to reserve space for an object.
  - Such a directive is also called a “pseudo-op.”
- The object is simply referred to by its name.

Example

- Using standard AT&T assembler syntax:
  - `.byte` *x* – reserve a byte with initial value *x*
  - `.word` *x* – reserve a 16-bit “word” with initial value *x*
  - `.long` *x* – reserve a 32-bit “long word” with initial value *x*
  - `.quad` *x* – reserve a 64-bit “quad word” with initial value *x*
- Usually each directive is given a **label** based on the name of the variable.
  - Support we have a declaration `int ival;`
  - We would output: `ival: .long 0`
  - Note that an “`int`” in C is a “`long`” in the assembler.

An Easier Way

- By default symbols declared in the assembler are not visible outside the source file.
  - We can mark symbols as visible using the `.globl` directive.
  - This directive tells the assembler to mark the symbol as global in the link map for the object file.
  - Declaring a global function or variable `static` in C just means no `.globl` directive is issued.
- An easier way to declare statically allocated data is to use the `.comm` directive:
  - `.comm` *name*, *size* – reserve "common" space with the given *name* and *size*

COEN 175

Lecture 14: Stack Frames

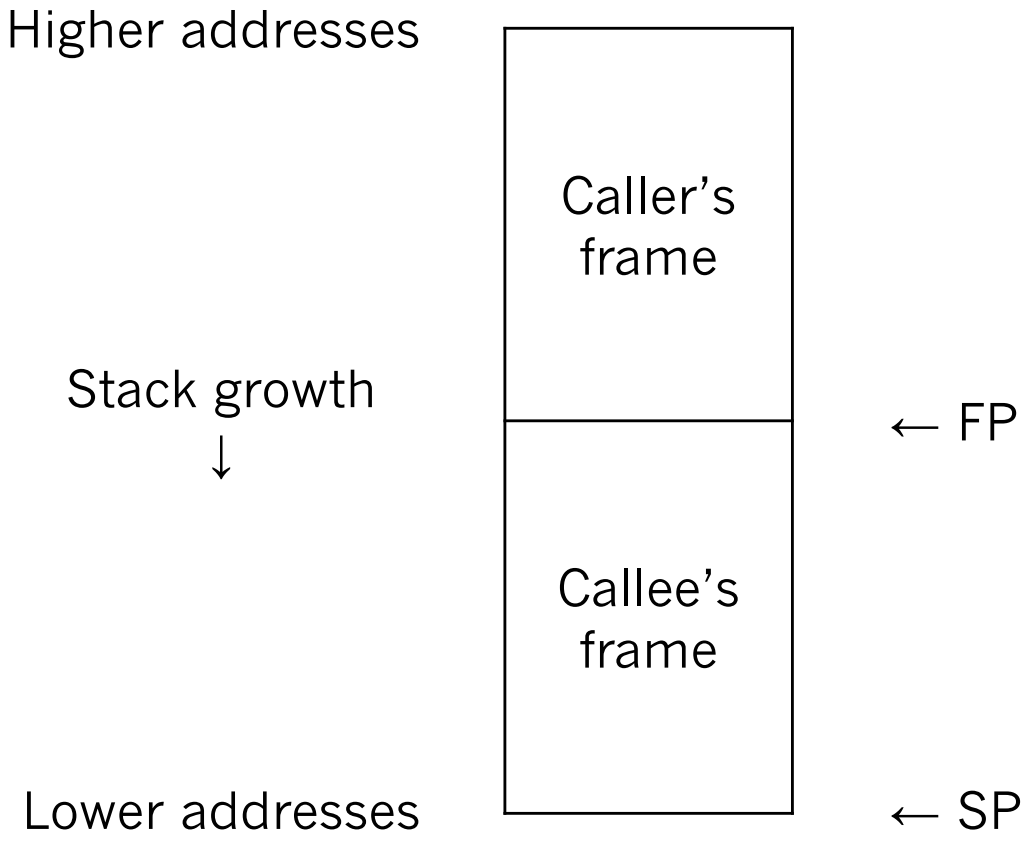
Stack Allocation

- Each function is responsible for allocating and deallocating memory for its stack-allocated data.
- Memory management is done per function, not per scope within a function.
- Stack-allocated data can include:
  - Local variables
  - Parameters
  - Intermediate results from computations
  - Saved program state such as registers

Terminology

- A function's block of memory on the stack is called an activation record or **stack frame**.
- The function making the function call is the **caller**, and the function being called is the **callee**.
- The processor reserves two registers for our use.
  - The **stack pointer** (SP) points to the top of the stack.
  - The **frame pointer** (FP) points to the start of the frame.
  - Since stacks typically grow down, the "top" of the stack is actually the lowest memory address.

Example: Stack Growth



Important Tasks

- Stack frame management
  - Who allocates the callee's stack frame?
  - Who deallocates the callee's stack frame?
- Transfer of control
  - How is control transferred to the callee?
  - How is control transferred back to the caller?
- Parameter passing
  - How are parameters passed to the callee?
  - How does the callee return a value to the caller?

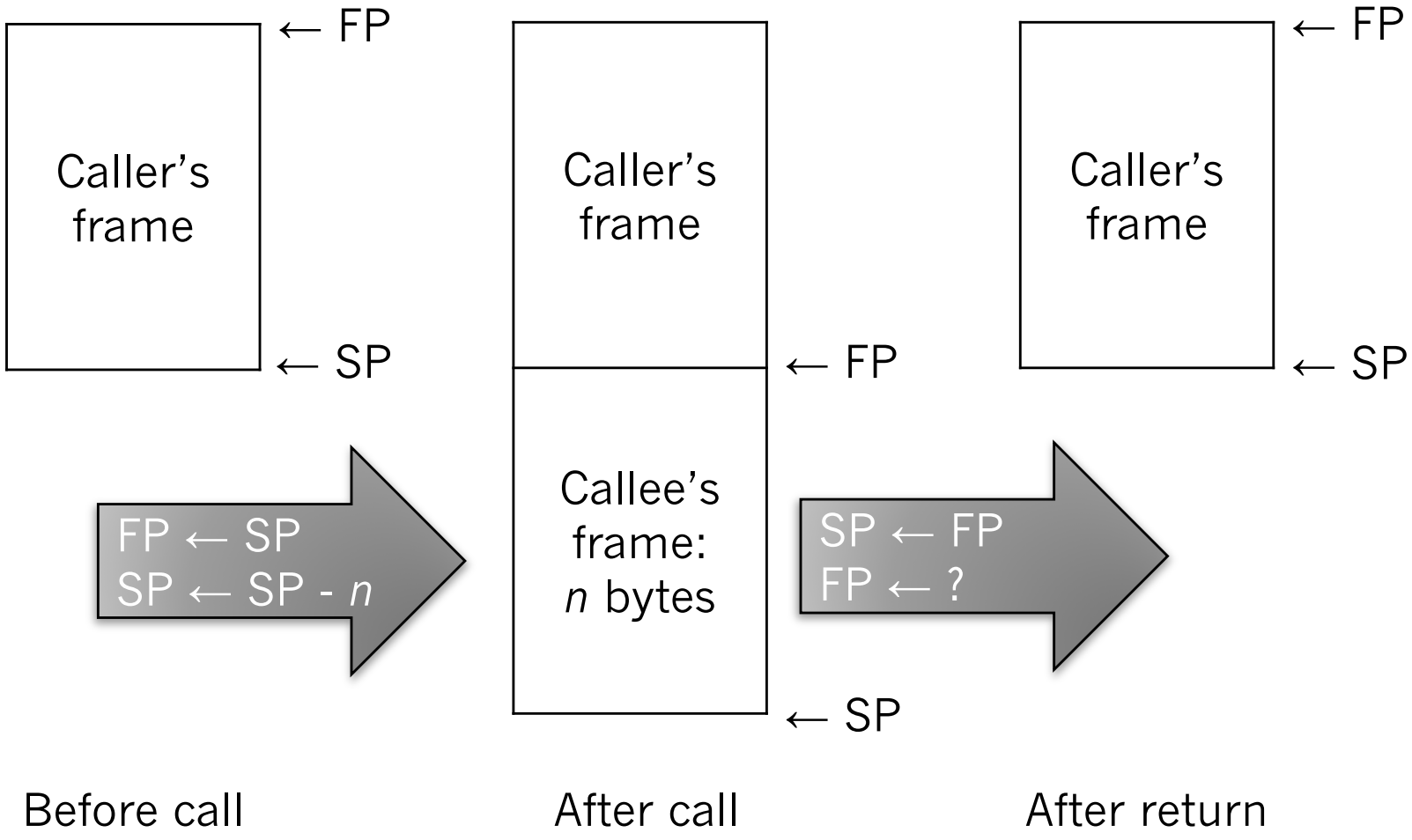
Frame Management

- Who allocates the callee's stack frame?
  - The callee does: it is the only one who knows how much space it requires.
- Who deallocates the callee's stack frame?
  - The callee does: only it knows how much space it has allocated.
- The callee does not know the identity of the caller.
  - Separate compilation of source files means that the source code of the caller is not available.

Assembly Code Layout

- A function begins execution by allocating its frame.
  - This piece of assembly code is called the **prologue**.
  - Thus, a call to the function does not in fact immediately begin execution of the function body.
- Next comes the body of the function.
  - This is the code that the programmer actually wrote.
- A function ends execution by deallocating its frame.
  - This piece of assembly code is called the **epilogue**.
  - Thus, a return from a function is actually a jump to the epilogue of the function.

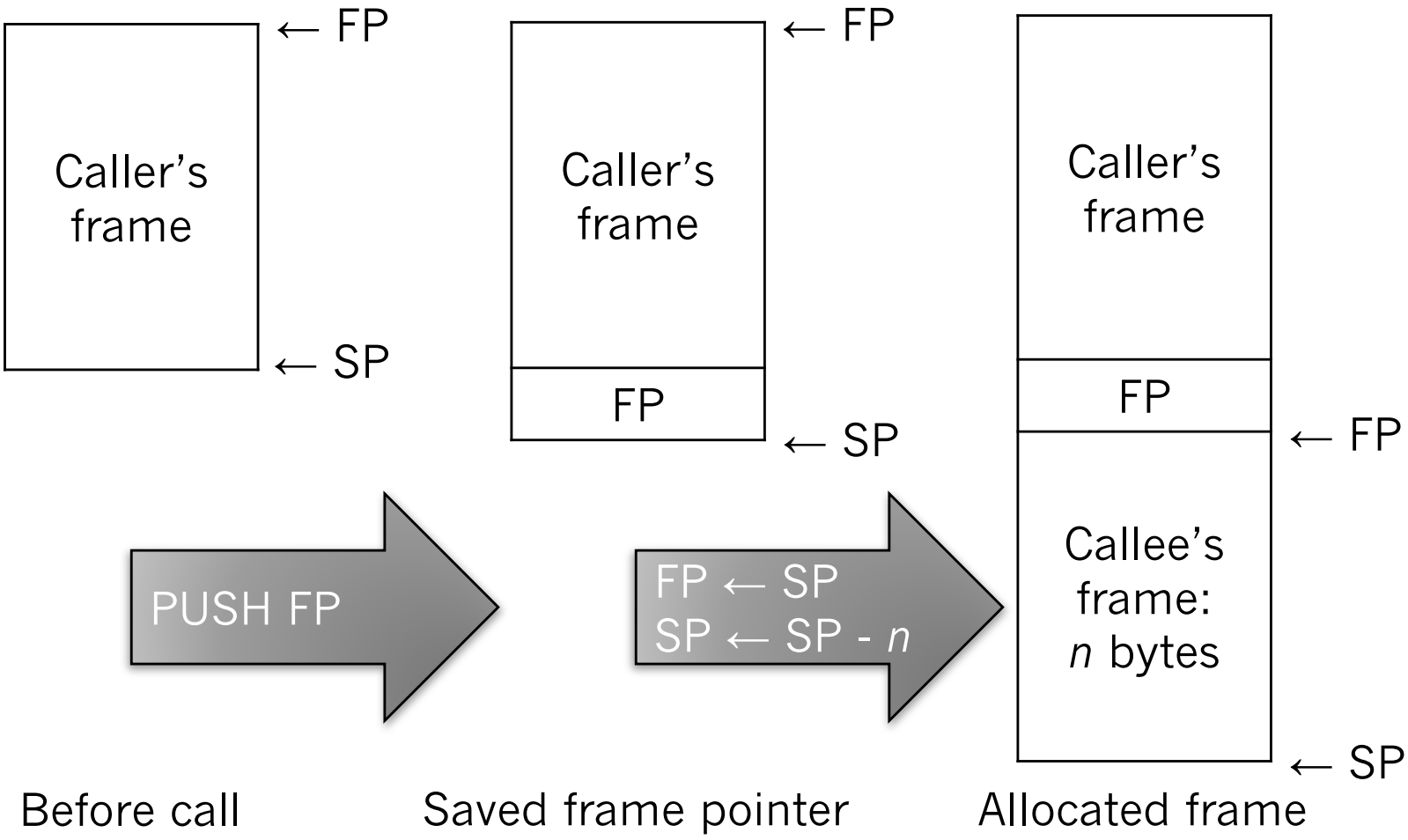
Example: Call and Return



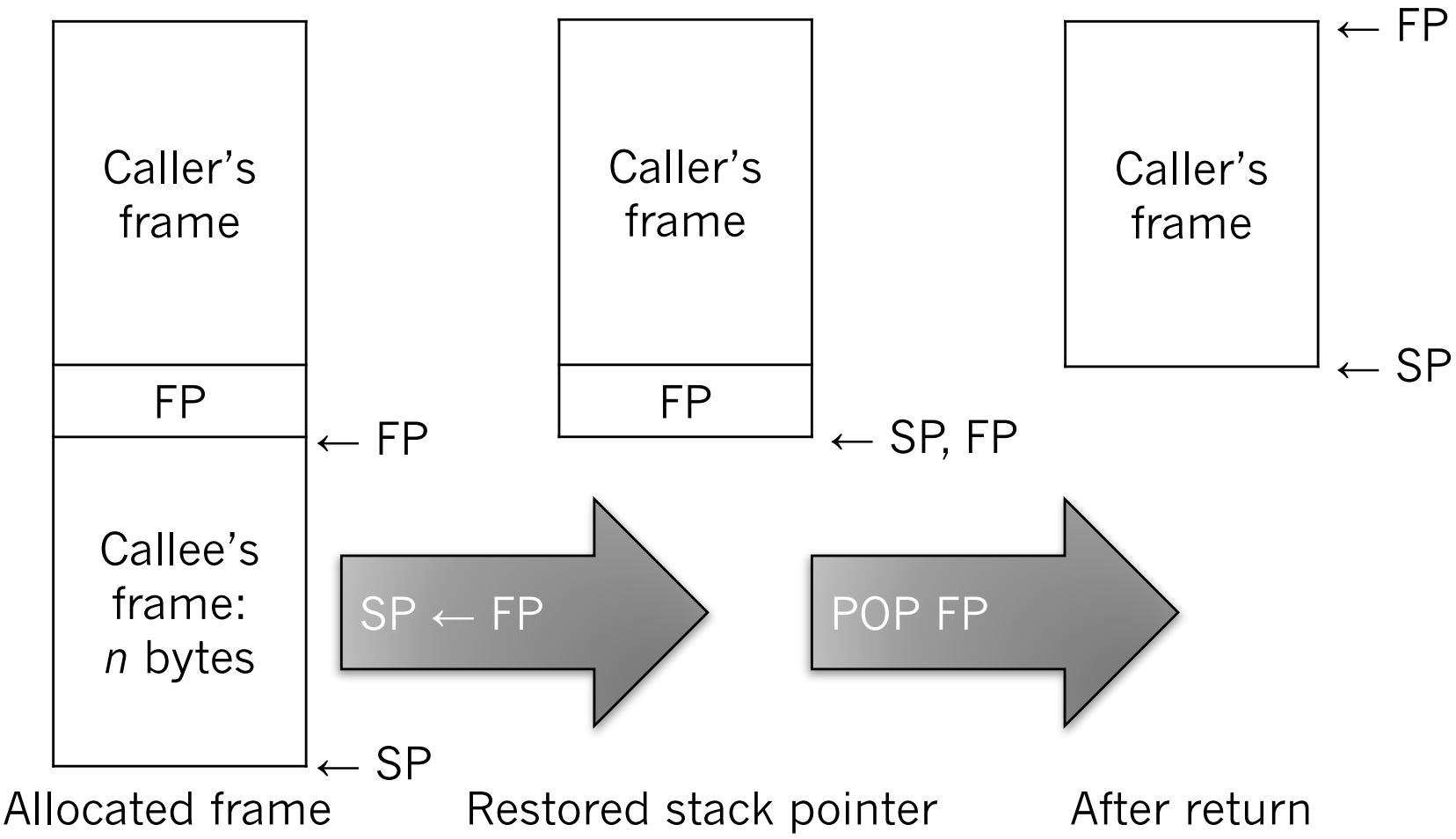
Saving the Frame Pointer

- Deallocating the stack frame requires us to restore the old frame pointer, which we have lost.
  - The solution is to first save the old frame pointer.
  - Where do we save it? On the stack, of course.
  - Note that we cannot use static memory because our function might be recursive.
- The prologue will first push the frame pointer.
- The epilogue will pop the old value before returning.

Revised: Call and Return



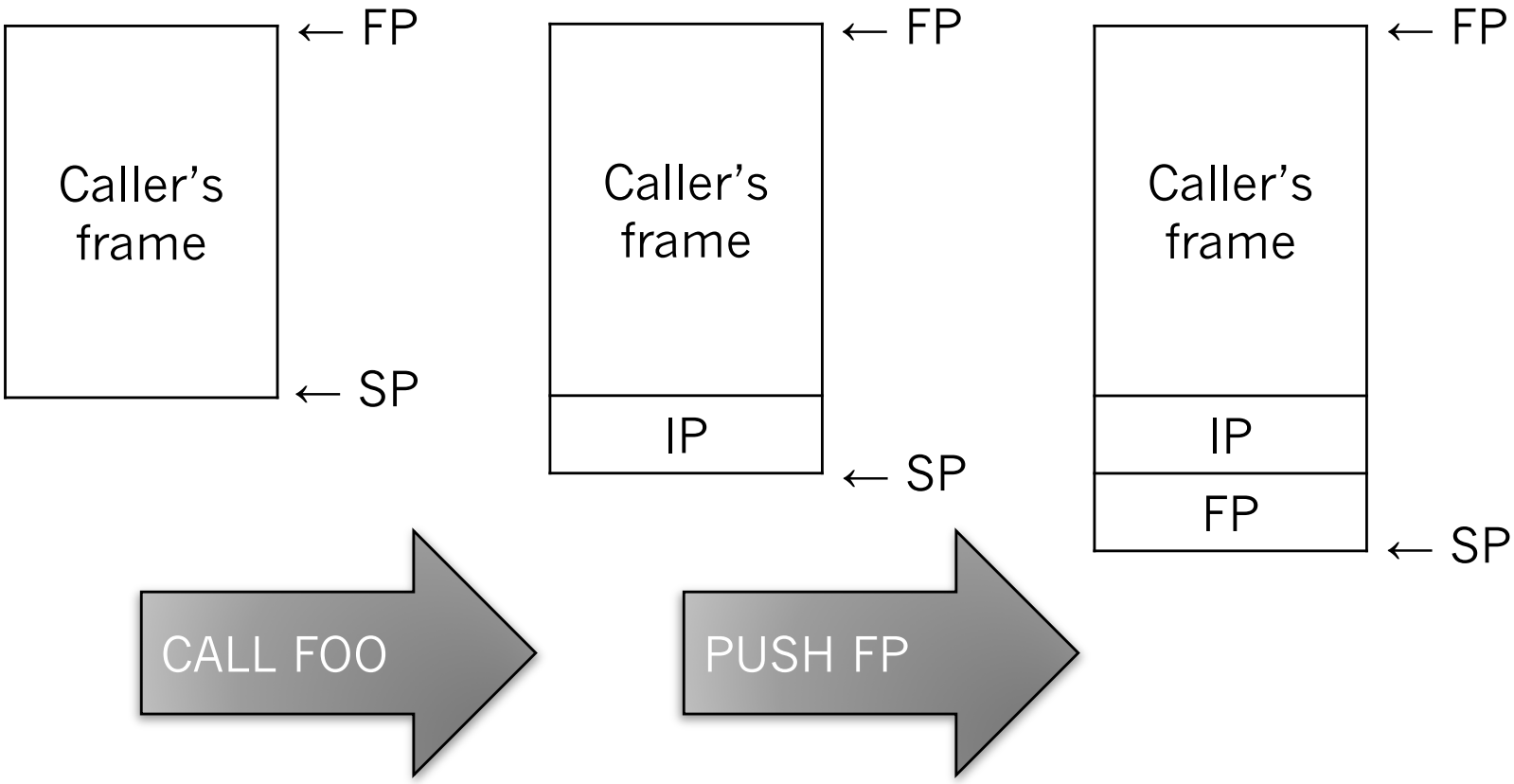
Revised: Call and Return



Transfer of Control

- How is control transferred to the callee?
  - The address of the current instruction is stored in a special register called the program counter or **instruction pointer**.
  - The caller uses the `call` instruction to change the value of the instruction pointer (IP).
  - The `call` instruction must first save the current value of the instruction pointer on the stack.
- How is control transferred back to the caller?
  - The callee uses the `return` instruction to pop the instruction pointer and return control.

Reality: Call and Return

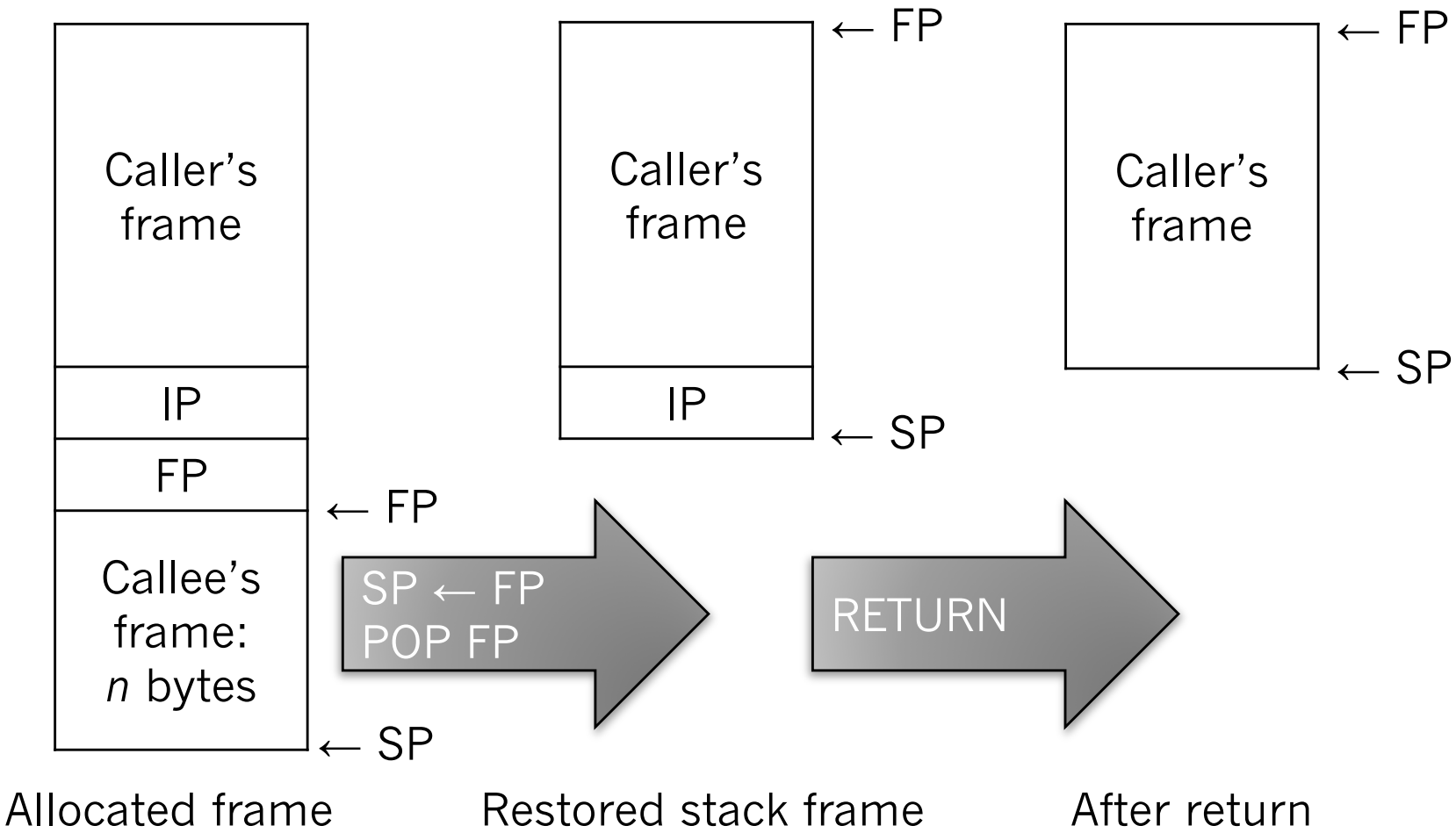


Before call

Saved instruction pointer

Saved frame pointer

Reality: Call and Return



Assembler Syntax

- Registers are prefixed with a percent sign.
- Immediate values are prefixed with a dollar sign.
- Opcodes are suffixed to indicate operand size:
  - `b` – byte (8-bit) operand
  - `w` – word (16-bit) operand
  - `l` – long (32-bit) operand
  - `q` – quad (64-bit) operand
- Base offset addressing is denoted by *offset*(*register*).
- The destination operand is on the right.

Intel Registers

- The Intel 32-bit architecture has 8 registers.
  - The stack pointer is called `%esp`.
  - The frame or **base pointer** is called `%ebp`.
- In the original 16-bit architecture the stack pointer was simply `%sp`.
- The 32-bit register is called `%esp` (for “extended”) and `%sp` is simply the lower 16-bits.
- On the 64-bit architecture, the 64-bit register is called `%rsp`, with `%esp` being the lower 32-bits of it.

Intel Assembly

- Standard prologue on 32-bit Intel architecture, where *n* is the number of bytes required:

```
pushl   %ebp
movl    %esp, %ebp
subl    $n, %esp
```

- Standard epilogue on 32-bit Intel architecture:

```
movl    %ebp, %esp
popl    %ebp
ret
```

COEN 175

Lecture 15: Stack-Allocated Variables

Parameter Passing

- Who evaluates the arguments to the called function?
  - The caller evaluates them since they are passed by value.
- How are arguments communicated to the callee?
  - Some arguments may be placed in registers.
  - Other arguments will be placed on the stack.
- Who evaluates the return value?
  - The callee evaluates the return value.
- How is the return value communicated to the caller?
  - The value is typically placed in a register (assuming it fits).

Calling Conventions

- The **calling conventions** of the system dictate how the registers and the stack are used.
  - Which registers are used to pass arguments?
  - What if an argument is too large for a single register?
  - Where are extra arguments placed on the stack?
  - Which register is used to return a value?
  - Where is a return value placed if it does not fit in a register?
  - Which registers can be modified by the callee?
- These conventions depend on the OS used.
  - Unix and Microsoft systems have different conventions.

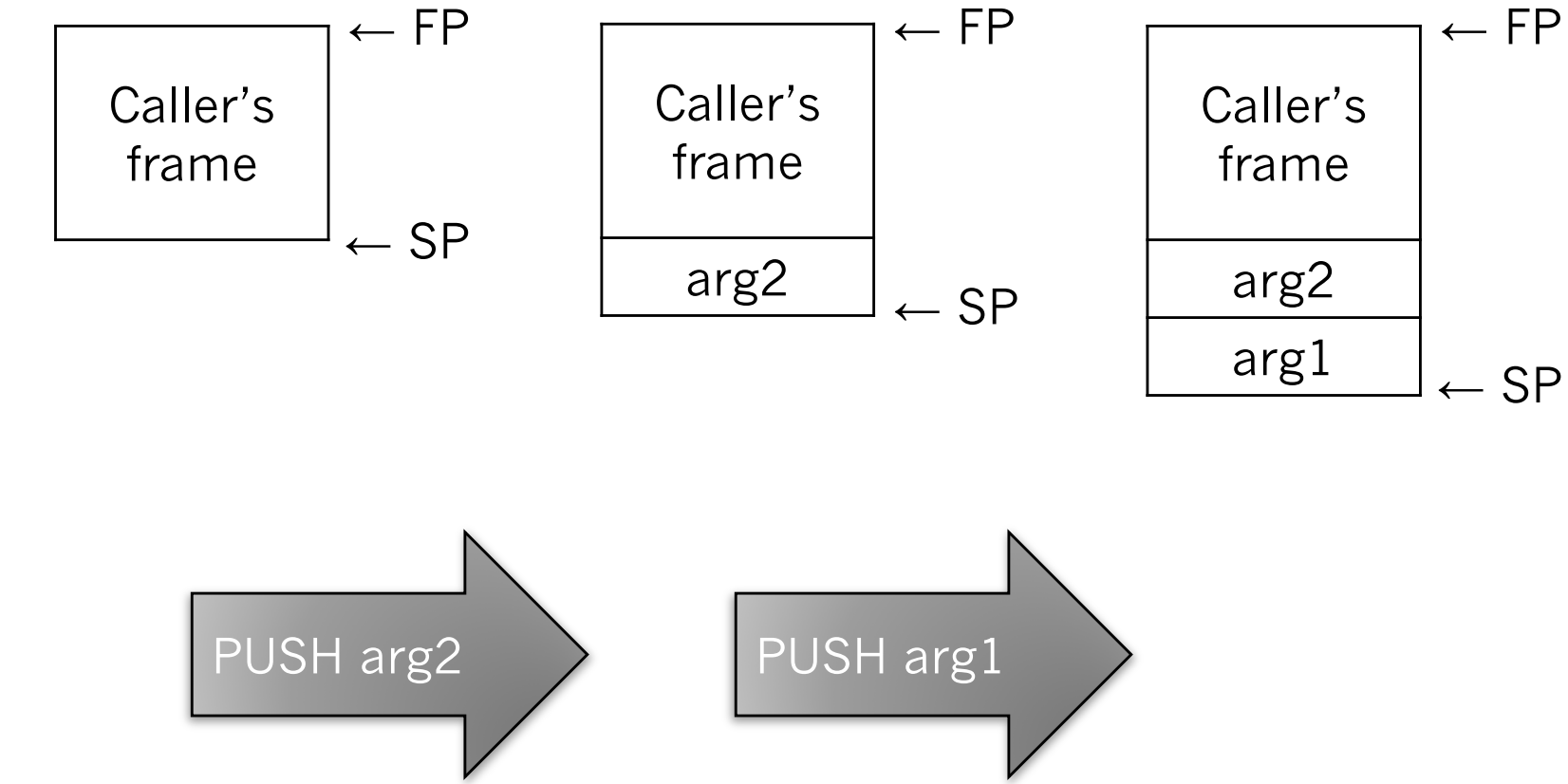
Intel Register Names

- The original 16-bit Intel register names:
  - `AX` – accumulator
  - `BX` – base register
  - `CX` – count register
  - `DX` – data register
  - `SI` – source index
  - `DI` – destination index
- The 32-bit versions are called `EAX`, `EBX`, etc.
- The 64-bit versions are called `RAX`, `RBX`, etc.
  - The additional 8 registers are simply `R8–R15`.
  - The 32-bit versions of these are called `R8d–R15d`.

Linux Calling Conventions

- No arguments are passed in registers.
- All arguments are placed on the stack.
  - Arguments are pushed on the stack from **right to left**.
  - Arguments must be a multiple of 4 bytes.
  - The stack must be kept aligned on a 4-byte boundary. (Note that OS X requires 16-byte alignment!)
  - The first argument is +8 bytes from the base pointer.
- An integer return value is placed in `EAX`.
  - A floating value is returned differently using the floating-point co-processor stack.

Stack-Based Parameters

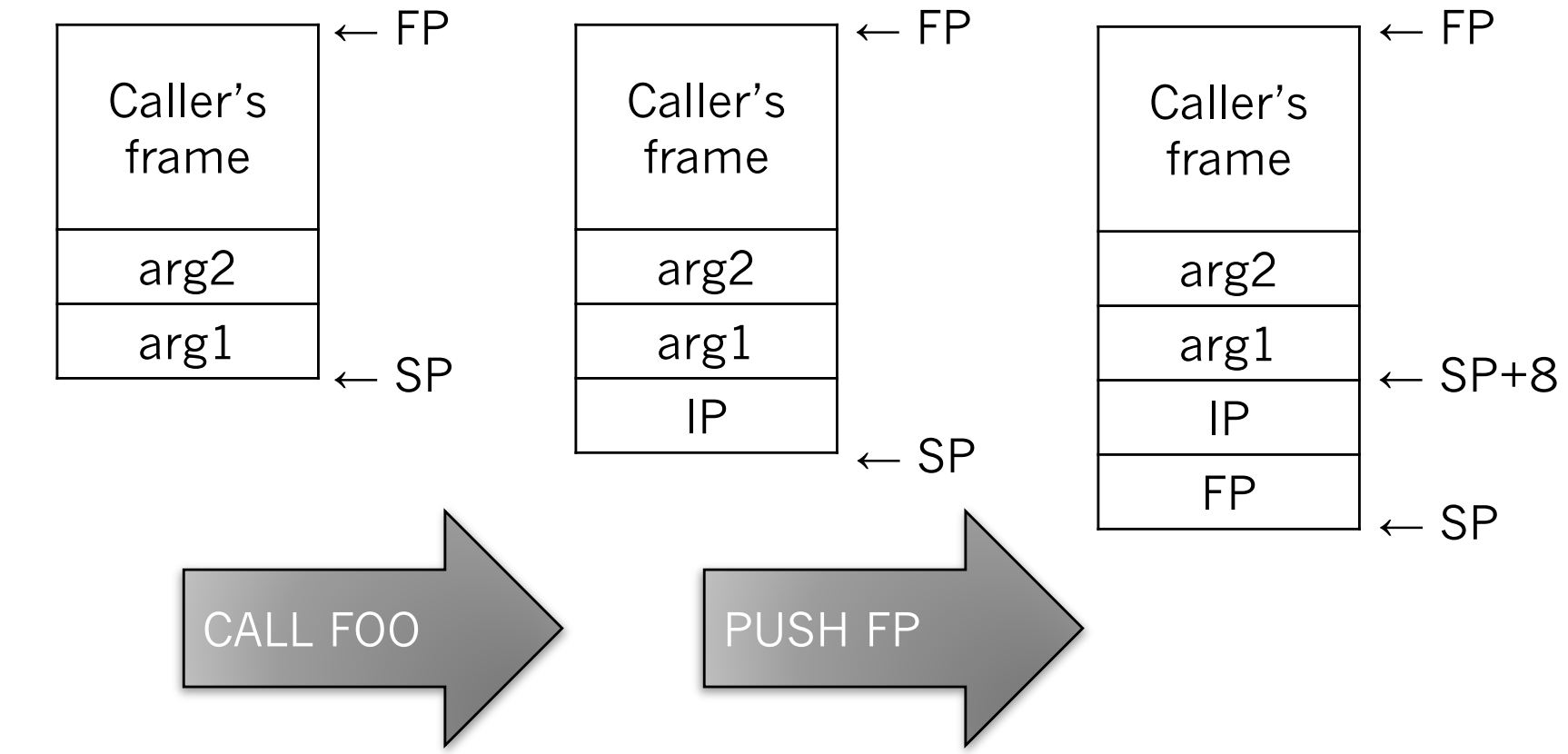


Before call to `foo(arg1,arg2)`

Push rightmost arg

Push next argument

Stack-Based Parameters



After arguments Saved instruction pointer Saved frame pointer

Referencing Locals

- Local variables are stored within the stack frame, which could be located anywhere in memory.
- Thus, we cannot use absolute or fixed addressing.
- Instead, we must use relative addressing.
  - Also called base-offset addressing.
- Relative to which base register? FP or SP?
  - SP might move during function execution.
  - FP will not move during function execution.
  - So, FP is a better choice.

Assigning Offsets

- Local variables will be referenced via negative offsets from the frame pointer.
- For simplicity, local variables are typically laid out within the frame in declaration order:
  - Keep track of the last offset in a global variable `offset`.
  - Assign the next local variable, `x`, its offset as follows:

```
offset = offset - x.size
x.offset = offset
```

  - For example, if the first variable requires 4 bytes, it would be at offset -4. If it required 8 bytes, it would be at offset -8.

Example: Offsets

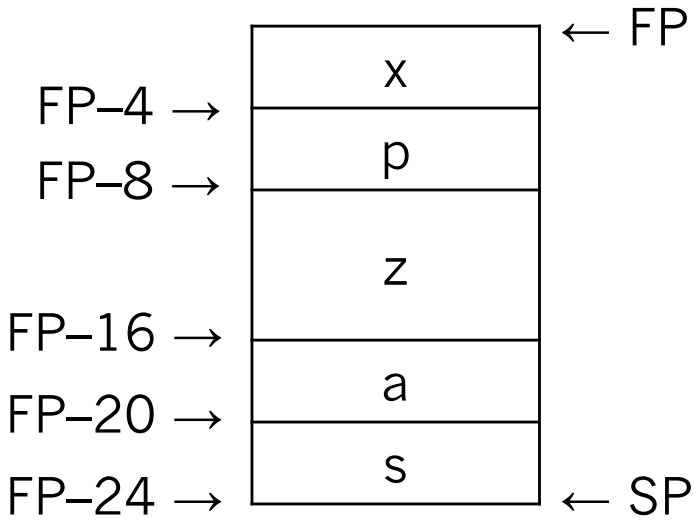
- Assign offsets to each variable:

```
int q, a[10];                 // none: q and a are static

int f(int i, int j) {         // i = +8, j = +12
    int x, *p;                // x = -4, p = -8
    double z;                 // z = -16
    char a[4], *s;            // a = -20, s = -24
}
```

Example: Offsets

- Assign offsets to each variable:

```
int q, a[10];

int f(int i, int j) {
    int x, *p;
    double z;
    char a[4], *s;
}
```



- What if `f()` were to call another function?
- What would happen to any registers in use?

Register Usage

- There is only one set of registers that must be shared amongst all functions.
- Most calling conventions divide up the registers:
  - **Caller-saved** – the register's value must be saved on the stack before a function call and restored after the call;
  - **Callee-saved** – the register's value must be saved before it is first used and restored before returning;
  - Any argument and return registers must be caller-saved.
- Intel 32-bit convention: `EAX`, `ECX`, and `EDX` are caller-saved; `EBX`, `ESI`, and `EDI` are callee-saved.

Another Example: Offsets

- Assign offsets to each variable:

```
int f(int i, int j) {       // i = +8, j = +12
    char x, y, *p;          // x = -1, y = -2, p = -6
    double z;               // z = -14
    char a[5], *s;          // a = -19, s = -23
}
```

- Some of our objects are not **aligned**.
  - Not all architectures require alignment.
  - An unaligned memory access will trigger a **bus error**.
  - Intel does not require alignment, but aligned accesses run faster as unaligned accesses require multiple fetches.

Another Example: Alignment

- Assign offsets to each variable:

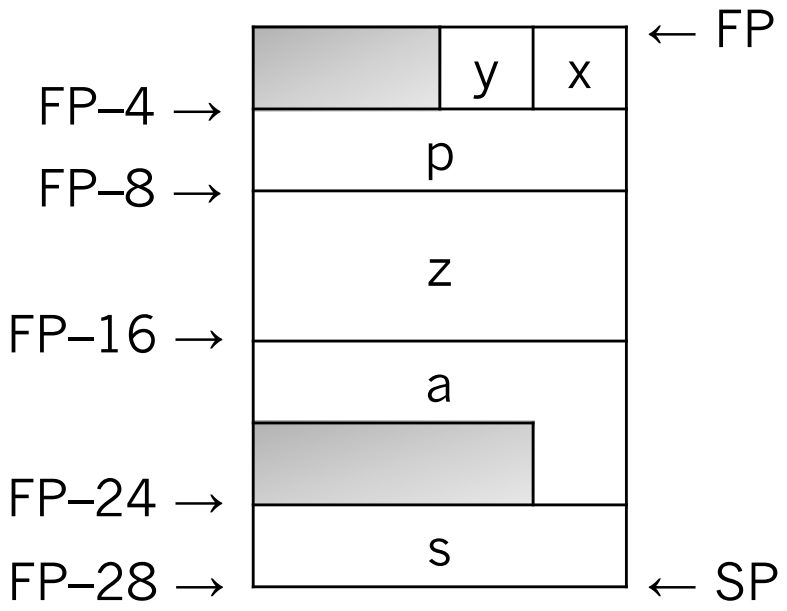
```
int f(int i, int j) {       // i = +8, j = +12
    char x, y, *p;          // x = -1, y = -2, p = -8
    double z;               // z = -16
    char a[5], *s;          // a = -21, s = -28
}
```

- Assume memory accesses should be aligned:
  - A 4-byte object should be aligned on a 4-byte boundary;
  - An 8-byte object should be aligned on an 8-byte boundary;
  - A byte object is always aligned.

Another Example: Alignment

- Assign offsets to each variable:

```
int f(int i, int j) {
    char x, y, *p;
    double z;
    char a[5], *s;
}
```



COEN 175

Lecture 16: Code Generation for Expressions

Code Generation

- Functions consist of statements, and statements consist of expressions.
- So, we will start with code generation techniques for expressions.
- Note that some expressions affect the flow of control and will be discussed together with statements.
  - Short-circuiting logical operators in C and Simple C.
  - The inline conditional operator in C.

Using Registers

- Ideally, we would keep all data in registers.
- However, this is not always possible.
  - We may not have enough registers.
  - A variable may be too large to fit in a register.
  - A variable might be a global variable.
  - A variable might have its address taken.
- The process of assigning variables and expressions to registers is known as **register allocation**.
  - Optimal register allocation is NP-complete.

Intermediate Results

- As a compromise, we can keep variables in memory but intermediate results of expressions in registers.
- This approach eliminates many problems:
  - Intermediate results will always fit in a register;
  - They cannot have their addresses taken;
  - They cannot be referenced globally.
- We still have to deal with the problem of not having enough registers.
- This approach is used by many compilers such as GCC and Clang unless optimizations are enabled.

Stack-Based Temporaries

- A final, naïve approach would be to store everything on the stack.
- This approach would be very slow, but would not have any of the problems we discussed earlier.
- Effectively, we would need to assign the result of each operation to a temporary variable.
- Nothing would be kept in registers beyond the lifetime of a single operation.

Three-Address Code

- We would effectively be building an intermediate representation known as **three-address code**.
- A three-address code statement can do at most one operation: *x* `:=` *y op z* or *x* `:=` *op y*.
- Translate `a * b + c * d` into three-address code.

```
t0 := a * b
t1 := c * d
t2 := t0 + t1
```

- Here, t_0, t_1, and t_2 are stack-based temporaries.

Terminology

- Moving a value from memory to a register is a **load**.
- Moving a value from a register to a named memory location is a **store**.
- Moving a value from a register to a temporary on the stack is a **spill**.
- Why might we have to spill?
  - We may not have enough registers.
  - We may need a dedicated register for an operation.
  - We need to call a function and the register is caller-saved.

Intel Instruction Set

- ARM and MIPS are **load-store architectures**.
  - Dedicated instructions are used to move values between memory and registers.
  - All other instructions require register operands.
- Intel is a **register-memory architecture**.
  - Any instruction can specify a memory reference as either a source or destination operand.
  - However, at most one operand can be a memory reference.
- Intel has a **two-operand** instruction set.
  - One of the operands is both a source and destination.

A Simple Algorithm

- Let's develop a simple algorithm to generate code using registers to hold intermediate results.
- Consider a binary expression *left op right*:
  1. Generate code for the left and right operands.
  2. If *left* is not in a register, then allocate a register and load it into that register.
  3. Perform the operation by emitting *opcode right*, *left* where *opcode* is the appropriate instruction.
  4. If *right* is a register, then that register is now available.
- Assume that code is generated during a left-to-right parse of the program.

Example 1

- Assume the registers are `%eax`, `%ecx`, `%edx`, etc.
- Assume that all variables are 32-bit integers and are global variables so they can be referred to by name.
- Generate code for `a * b + c * d`.

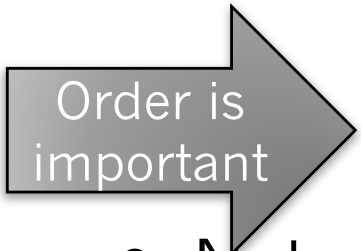
```
movl   a, %eax        # load: %eax allocated
imull  b, %eax
movl   c, %ecx        # load: %ecx allocated
imull  d, %ecx
addl   %ecx, %eax     # %ecx deallocated
```

- Here "`imul`" means integer, or signed, multiplication.

Example 2

- Generate code for `a + b * c – (a + b + c)`.

```
movl   b, %eax        # load: %eax allocated
imull  c, %eax
movl   a, %ecx        # load: %ecx allocated
addl   %eax, %ecx     # %eax deallocated
movl   a, %eax        # load: %eax allocated
addl   b, %eax
addl   c, %eax
subl   %eax, %ecx     # %eax deallocated
```



- Note that in evaluating `a + b * c`, we first evaluated `b * c` before applying our algorithm to the addition.

Example 3

- Generate code for `a * b - (c * d + (a – b * c))`.

```
movl   a, %eax        # load: %eax allocated
imull  b, %eax
movl   c, %ecx        # load: %ecx allocated
imull  d, %ecx
movl   b, %edx        # load: %edx allocated
imull  c, %edx
movl   a, %ebx        # load: %ebx allocated
subl   %edx, %ebx     # %edx deallocated
addl   %ebx, %ecx     # %ebx deallocated
subl   %ecx, %eax     # %ecx deallocated
```

- This example required four registers! Would you believe we can do it using only two registers?

Example 3: Optimal Code

- Generate code for `a * b - (c * d + (a – b * c))`.

```
movl   b, %eax        # load: %eax allocated
imull  c, %eax
movl   a, %ecx        # load: %ecx allocated
subl   %eax, %ecx     # %eax deallocated
movl   c, %eax        # load: %eax allocated
imull  d, %eax
addl   %ecx, %eax     # %ecx deallocated
movl   a, %ecx        # load: %ecx allocated
imull  b, %ecx
subl   %eax, %ecx     # %eax deallocated
```

- We used only two registers; however, we generated code in an order different from parsing to do so.

COEN 175

Lecture 17: More Code Generation for Expressions

Sethi-Ullman Algorithm

- The **Sethi-Ullman algorithm** generates code for an expression using the fewest number of registers.
- The key insight is to evaluate the subtree that requires the most registers first.
- Thus, we must be able to generate code in an order different from the left-to-right parse order.
- The algorithm works on the abstract syntax tree (AST) and requires two passes over the tree:
  1. Label the abstract syntax tree.
  2. Generate optimal code based on the labeled tree.

Labeling

- Each node is assigned an integer **label** representing the number of registers it requires.
- The label of each node is computed bottom-up:
  - The label of a left leaf is 1.
  - The label of a right leaf is 0.
  - The label of a binary node, *parent*, with children *left* and *right* is computed as follows:

```
if label(left) = label(right) then
    label(parent) ← label(left) + 1
else
    label(parent) ← max(label(left), label(right))
```

Example 1: Labels

- Label the expression `a * b + c * d`.
  1. label(`a`) = 1
  2. label(`b`) = 0
  3. label(`a * b`) = max(1, 0) = 1
  4. label(`c`) = 1
  5. label(`d`) = 0
  6. label(`c * d`) = max(1, 0) = 1
  7. label(`a * b + c * d`) = 1 + 1 = 2
- The Sethi-Ullman algorithm tells us that two registers are required to evaluate this expression.

Example 2: Labels

- Label the expression `a + b * c – (a + b + c)`.
  1. label(a) = 1
  2. label(b) = 1
  3. label(c) = 0
  4. label(b * c) = max(1, 0) = 1
  5. label(a + b * c) = 1 + 1 + 2
  6. label(a) = 1
  7. label(b) = 0
  8. label(a + b) = max(1, 0) = 1
  9. label(c) = 0
  10. label(a + b + c) = max(1, 0) = 1
  11. label(a + b * c – (a + b + c)) = max(2, 1) = 2

Example 3: Labels

- Label the expression `a * b - (c * d + (a – b * c))`.
  1. label(a) = 1
  2. label(b) = 0
  3. label(a * b) = max(1, 0) = 1
  4. label(c) = 1
  5. label(d) = 0
  6. label(c * d) = max(1, 0) = 1
  7. label(a) = 1
  8. label(b) = 1
  9. label(c) = 0
  10. label(b * c) = max(1, 0) = 1
  11. label(a – b * c) = 1 + 1 = 2
  12. label(c * d + (a – b * c)) = max(1, 2) = 2
  13. label(a * b – (c * d + (a – b * c))) = max(1, 2) = 2

Generating Optimal Code

- To generate code requiring the fewest number of registers, we always generate code for the child with the **larger label first**.
  - If both children have the same label, the order is irrelevant.
- Suppose one child requires m registers and the other child requires n registers, where $m < n$.
  - The larger child requires n registers, but in the end the result is always held in a single register.
  - The smaller child requires $m + 1$ registers: m registers for itself and 1 register to hold the larger child's result.
  - Since $m < n$, we will use at most n registers.

Example 1: Code

- Generate code for `a * b + c * d`.
  - Since the two children of the addition have the same label, it does not matter which child we do first:

```
movl   a, %eax
imull  b, %eax
movl   c, %ecx
imull  d, %ecx
addl   %ecx, %eax
```

Left operand of addition done first

```
movl   c, %eax
imull  d, %eax
movl   a, %ecx
imull  b, %ecx
addl   %eax, %ecx
```

Right operand of addition done first

- Either choice requires two registers.

Example 2: Code

- Generate code for `a + b * c – (a + b + c)`.
  - Except for the tie between `a` and `b * c` in the left operand of the subtraction, there are no choices.

```
movl   b, %eax
imull  c, %eax
movl   a, %ecx
addl   %eax, %ecx
movl   a, %eax
addl   b, %eax
addl   c, %eax
subl   %eax, %ecx
```

- This is the same code we generated previously.

Example 3: Code

- Generate code for `a * b - (c * d + (a – b * c))`.
  - Except for the tie between `a` and `b * c` in the right operand of the addition, there are no choices.

```
movl   b, %eax
imull  c, %eax
movl   a, %ecx
subl   %eax, %ecx
movl   c, %eax
imull  d, %eax
addl   %ecx, %eax
movl   a, %ecx
imull  b, %ecx
subl   %eax, %ecx
```

Simple Arithmetic Operators

- Most binary operators follow the same template:
  1. Generate code for one child.
  2. Generate code for the other child.
  3. If the left child is not in a register, then load it.
  4. Perform the operation, overwriting the left child's register.
  5. Deallocate any register for the right operand.
- For addition, subtraction, and multiplication, performing the operation itself is simple.
  - Addition = `add`, subtraction = `sub`, multiplication = `imul`

Division and Remainder

- Division and remainder are slow operations.
  - Many compilers avoid division or remainder by a constant.
- Dividing two 32-bit operands requires a 64-bit dividend, and two 64-bit operands requires a 128-bit dividend.
- Intel requires special registers for division.
  - The `EDX:EAX` pair holds the 64-bit dividend.
  - Similarly, the `RDX:RAX` pair holds the 128-bit dividend.
  - After division, `EAX` (or `RAX` if 64-bit) holds the quotient and `EDX` (or `RDX`) holds the remainder.

Division: Example

- Generate code for `x / y + z`, using 32-bit operands.

```
movl    x, %eax         # load: %eax allocated
movl    %eax, %edx      # sign extend %eax into %edx
sarl    $31, %edx
idivl   y               # %edx:%eax / y
addl    z, %eax
```

- Note that the divide instruction, `idiv`, requires only the divisor as its single operand.
- The dividend is implicitly specified as the `%edx:%eax` register pair.

Division: Quirks

- Special instructions are available for sign-extension: `cltd` (32-bit to 64-bit) and `cqto` (64-bit to 128-bit).
- The Intel division instruction allows as an operand a register or memory reference, but not an immediate.
  - So, `x / 7` will not work using our template.
  - We must first load the immediate into a register.
- Generate code for `x / 7`.

```
movl   x, %eax
cltd
movl   $7, %ecx
idivl  %ecx
```

Comparison Operators

- The comparison operators in C yield 1 if the comparison is true, and 0 if the comparison is false.
- In Intel assembly, the `cmp` instruction sets internal condition codes or **flags**, which can then be tested:
  - `sete` – set if equal
  - `setne` – set if not equal
  - `setl` – set if less than
  - `setle` – set if less than or equal
  - `setg` – set if greater than
  - `setge` – set if greater than or equal

Comparison: Example

- The various `set` instructions require a byte register, which can then be zero-extended using `movzbl`.
- Generate code for `(x > y) + z`.

```
movl   x, %eax
cmpl   y, %eax    # test and set condition codes
setg   %al        # set %al based on condition codes
movzbl %al, %eax  # move zero-extend byte to long
addl   z, %eax
```

- Note that `AL` is the low byte of `AX`/`EAX`/`RAX`.
  - Not all registers have an analogous byte register.
  - `ESI` and `EDI` do have have a byte register.

Intel Register Summary

64-bit	32-bit	16-bit	8-bit
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8–R15	R8d–R15d	R8w–R15w	R8b–R15b

`R8–R15` are new registers available only in 64-bit mode.
`SIL`, `DIL`, `BPL`, and `SPL` are only available in 64-bit mode, not 32-mode.

COEN 175

Lecture 18: Code Generation for Unary Operators

Negation

- Arithmetic negation uses the `neg` instruction.
- Generate code for `-x + y`.

```
movl   x, %eax
negl   %eax
addl   y, %eax
```

- The logical negation of `x`, `!x`, is equivalent to `x == 0`.
- Generate code for `!x`.

```
movl   x, %eax
cmpl   $0, %eax
sete   %al
movzbl %al, %eax
```

Type Cast

- Converting to a smaller sized type is not necessary, as we simply use the appropriately sized register.
  - For example, we will simply use `AL` if we need to read the contents of `EAX` (or `RAX`) as a byte.
- Converting to a larger sized type requires sign extension, which is done by the `movs` instruction with the appropriate suffix.
- Generate code for `(int) c + 123`.

```
movsbl  c, %eax    # sign extend byte to long
addl    $123, %eax
```

Address

- The address operator has two cases.
- If its operand is a dereference expression then the operator does nothing as `&*p` is equivalent to `p`.
  - All that is necessary is to pass along its operand's location.
- If its operand is an identifier then we use the `lea` instruction to "load the effective address".
  - Note that addresses are always 32-bits.
- Generate code for `&x`.

```
leal   x, %eax
```

Strings

- To allocate space for a zero-terminated ASCII string, we use the `.asciz` assembler directive.

```
.L0:    .asciz "hello world"
```

- Labels beginning with a period are local to the file.
- When the string is used, the address of its first character must be computed using `lea` as before.
- Generate code for `"hello world" + 4`.

```
leal    .L0, %eax
addl    $4, %eax
```

Dereference: Load

- A dereference expression could be used as either an lvalue or an rvalue.
- If used as an rvalue, we read the memory whose address is specified by the operand.
- Generate code for `*p + x`.

```
movl   p, %eax
movl   (%eax), %eax
addl   x, %eax
```

- Every use of a dereference as an **rvalue** should have an indirect memory reference as a **source** operand.

Dereference: Store

- If a dereference is used as an lvalue then no operation is done by the dereference operator itself.
- Rather, the parent operation controls what is done:
  - As previously seen, the address operator simply ignores the dereference and passes along its operand.
  - An assignment statement will do a store.
- Every **assignment** via a dereference should result in an indirect memory reference as a **target** operand.
  - Note that in `**p = x`, the inner dereference is an rvalue and the outer dereference is an lvalue.

Assignment: Variable

- The left-hand side of an assignment statement is either a variable or a dereference.
- If it is a variable, then do the following:
  1. Generate code for both operands.
  2. If the right operand is a memory reference, then load it into a register.
  3. Move the right operand into the memory location given by the left operand using the `mov` instruction.
- In my implementation, a type cast has been inserted to truncate or extend the right operand.

Assignment: Dereference

- If the left-hand side is a dereference, then:
  1. Generate code for the right operand and the child of the left operand.
  2. If the right operand is a memory reference, then load it into a register.
  3. Load the **child** of the dereference expression into a register if necessary.
  4. Move the right operand into the indirect memory location given by the left operand. This step is where the dereference operation is actually performed.
- Again, a type cast has already been inserted to truncate or extend the right operand.

Example 1: Assignment

- Generate code for `x = y + z`.

```
movl   y, %eax
addl   z, %eax
movl   %eax, x
```

- Generate code for `*p = y + z`.

```
movl   y, %eax
addl   z, %eax
movl   p, %ecx
movl   %eax, (%ecx)
```

- What if all the operands are not the same size?

Example 2: Assignment

- Generate code for the following code snippet:

```
char *p, c;
int *q, x;

*p = *(q + c) + x;
```

```
movsbl  c, %eax
imull   $4, %eax
movl    q, %ecx
addl    %eax, %ecx
movl    (%ecx), %ecx
addl    x, %ecx
movl    p, %eax
movb    %cl, (%eax)
```

Advance by objects not bytes

`%cl` is the byte register of `%ecx`

- Note that my implementation does the following:
  - Coercions are represented as explicit type casts;
  - Adjustments in pointer arithmetic are made explicit.

Logical Operators

- The logical operators in C are **short-circuiting**.
  - If the left operand of a logical-OR is true, the right operand is not evaluated.
  - If the left operand of a logical-AND is false, the right operand is not evaluated.
- Like the comparison operators, these operators yield a 1 or 0 “truth value” as a result.
- However, we do not always evaluate both operands.
- Therefore, they are more like conditional statements.

Short-Circuiting Code

- The expression E_1 || E_2 is equivalent to:

```
if (E1 != 0)
    result = 1;
else if (E2 != 0)
    result = 1;
else
    result = 0;
```

- The expression E_1 && E_2 is equivalent to:

```
if (E1 == 0)
    result = 0;
else if (E2 == 0)
    result = 0;
else
    result = 1;
```

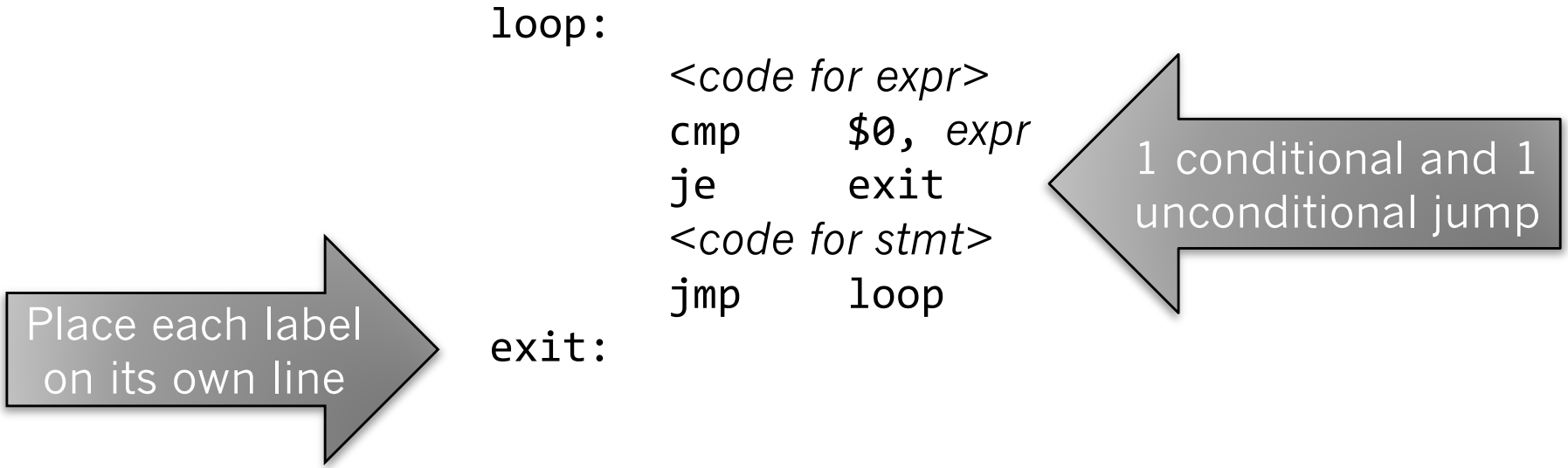
Example: Logical Or

- Generate code for `*p || y + z`.

```
        movl    p, %eax
        movl    (%eax), %eax
        cmpl    $0, %eax
        jne     .L1
        movl    y, %eax
        addl    z, %eax
        cmpl    $0, %eax
        jne     .L1
        movl    $0, %eax
        jmp     .L2
.L1:
        movl    $1, %eax
.L2:
```

While Statements

- The statement `while` (*expr*) *stmt* is translated as:



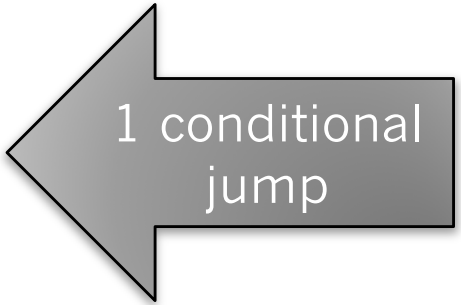
- It's important that each label is placed on its own line in case loops or conditionals are nested.

```
label1: label2:   # won't assemble
```

If Statements

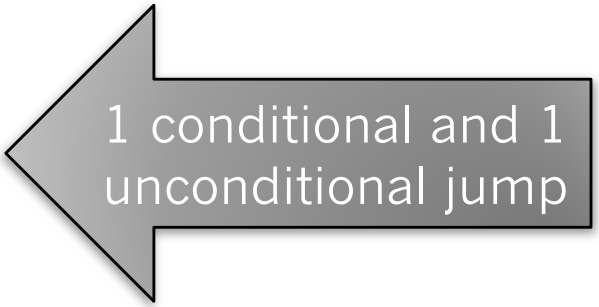
- The statement `if` (*expr*) *stmt* is translated as:

```
        <code for expr>
        cmp     $0, expr
        je      skip
        <code for stmt>
skip:
```



- And, `if` (*expr*) *stmt*$_1$ `else` *stmt*$_2$ is translated as:

```
        <code for expr>
        cmp     $0, expr
        je      skip
        <code for stmt1>
        jmp     exit
skip:
        <code for stmt2>
exit:
```



COEN 175

Lecture 19: Managing Registers

Managing Registers

- So far, we have not discussed how we allocate and deallocate registers while generating code.
- Suppose we have two C++ classes:
  - `Expression` – a base class for expressions;
  - `Register` – a class representing an Intel register.
- We need the following:
  - A way to find out if an expression is in a register;
  - A way to find out which expression is using a register;
  - A pool of available registers.

Linking the Two Classes

- Let's assume the following C++ class definitions:

```
class Expression {
    …
public:
    class Register *_register;
    std::string _operand;
};
```

```
class Register {
    …
public:
    class Expression *_node;
};
```

- Given an `Expression`, we can access its register, which will be `nullptr` if it is not loaded into one.
- Given a `Register`, we can access its expression, which will be `nullptr` if it is not being used by one.

Managing the Links

- Let's write a simple function to manage the links.

```
void assign(Expression *expr, Register *reg)
{
    if (expr != nullptr) {
        if (expr->_register != nullptr)
            expr->_register->_node = nullptr;

        expr->_register = reg;
    }

    if (reg != nullptr) {
        if (reg->_node != nullptr)
            reg->_node->_register = nullptr;

        reg->_node = expr;
    }
}
```

What Our Function Does

- The `assign` function does the following:
  - Disassociates any associated register from the node;
  - Disassociates any associated node from the register.
- The `assign` function does not:
  - Emit any assembly code to load values into registers;
  - Perform any spills if the register is already in use.
- All it does is maintain the proper mappings.
  - In other words, it is at the lowest level.
  - Policy decisions will be made at a higher level.

Register Allocation

- We need a pool of available registers.
- For simplicity, we will just use the caller-saved registers and allocate the first available register.

```
Register *eax = new Register("%eax", "%al");
vector<Register *> registers = { eax, … };

Register *getreg()
{
    for (unsigned i = 0; i < registers.size(); i ++)
        if (registers[i]->_node == nullptr)
            return registers[i];

    abort();
}
```

Printing Expressions

- Let's overload the output stream operator for expressions for convenience.
  - If the expression is in a register, then we use it. Otherwise, we use its `_operand` field which references memory.
  - Assume that our `Register` class has a function `name(`*n*`)` that returns the *n*-byte register name.

```
ostream &operator <<(ostream &ostr, Expression *expr)
{
    if (expr->_register == nullptr)
        return ostr << expr->_operand;

    unsigned size = expr->type().size();
    return ostr << expr->_register->name(size);
}
```

Register Loads

- Finally, we need a function to load an expression into a given register.

```
void load(Expression *expr, Register *reg)
{
    if (reg->_node != expr) {
        assert(reg->_node == nullptr);

        if (expr != nullptr) {
            unsigned size = expr->type().size();
            cout << "\tmov" << suffix(expr) << expr;
            cout << ", " << reg->name(size) << endl;
        }

        assign(expr, reg);
    }
}
```

Fail if register is allocated

Use `suffix(x)` to return the correct opcode suffix for `x`

Putting It All Together

- Let's write a complete function for addition.

```
void Add::generate()
{
    // Generate code for both the left child
    // and the right child

    // If the left child is not in a register,
    // then allocate a register and load it

    // Perform the operation such as
    // "addl right, left"

    // If the right operand is in a register,
    // then deallocate it
}
```

Putting It All Together

- Let's write a complete function for addition.

```
void Add::generate()
{
    _left->generate();
    _right->generate();

    if (_left->_register == nullptr)
        load(_left, getreg());

    cout << "\tadd" << suffix(_left);
    cout << _right << ", " << _left << endl;

    assign(_right, nullptr);
    assign(this, _left->_register);
}
```

Unresolved Issues

- Our new infrastructure works well and is very intuitive as it closely matches our algorithm.
- But, we still have a number of issues:
  - What if we call `getreg()` and no register is available?
  - What if we call `load()` to load an expression into a specific register and it is already allocated?
  - What happens to the caller-saved registers when we make a function call?
- Fortunately, we can fix all these issues at once by introducing spills.

Adding Temporaries

- To introduce spills, we will need to be able to create temporaries on the run-time stack.
- For simplicity, we will just assign temporaries the next available offset on the stack, just like locals.

```
void assigntemp(Expression *expr)
{
    stringstream ss;

    offset = offset – expr->type().size();
    ss << offset << "(%ebp)";
    expr->_operand = ss.str();
}
```

Smarter Loads

- Now let's modify `load()` to perform spills.

```
void load(Expression *expr, Register *reg)
{
    if (reg->_node != expr) {
        if (reg->_node != nullptr) {
            unsigned size = reg->_node->type().size();

            assigntemp(reg->_node);
            cout << "\tmov" << suffix(reg->_node);
            cout << reg->name(size) << ", ";
            cout << reg->_node->_operand << endl;
        }

        /* rest of function same as before */
    }
}
```

Smarter Allocation

- We can now write a smarter version of `getreg()`.
- If no register is available, then we will simply spill the first register and return it.

```
Register *getreg()
{
    for (unsigned i = 0; i < registers.size(); i ++)
        if (registers[i]->_node == nullptr)
            return registers[i];

    load(nullptr, registers[0]);
    return registers[0];
}
```

COEN 175

Lecture 20: Code Generation Improvements

Example

- Generate code for the body of the function:

```
int b;

int f(int a)
{
    return a + b * 3;
}
```

```
movl   b, %eax
imull  $3, %eax
movl   8(%ebp), %ecx
addl   %eax, %ecx
movl   %ecx, %eax
jmp    f.exit
```

- After we load `EAX` with the return value, we need to jump to the epilogue of the function.
  - For simplicity, we will just label the epilogue of a function named `f` as `f.exit`.
- Let's look at the expression `a + b * 3` in detail.

Example: Step-By-Step

Code Executed	Node Changes	Output Produced
`Add::generate()`		
`  Identifier::generate()`	`a::_operand = "8(%ebp)"`	
`  Multiply::generate()`		
`    Identifier::generate()`	`b::_operand = "b"`	
`    Number::generate()`	`3::_operand = "$3"`	
`    load(left,eax)`	`b::_register = eax`	`movl  b, %eax`
`    cout << … << endl`		`imull $3, %eax`
`    assign(right,nullptr)`		
`    assign(this,eax)`	`b::_register = nullptr` `*::_register = eax`	
`  load(left,ecx)`	`a::_register = ecx`	`movl  8(%ebp), %ecx`
`  cout << … << endl`		`addl  %eax, %ecx`
`  assign(right,nullptr)`	`*::_register = nullptr`	
`  assign(this,ecx)`	`a::_register = nullptr` `+::_register = ecx`	

Testing Expressions

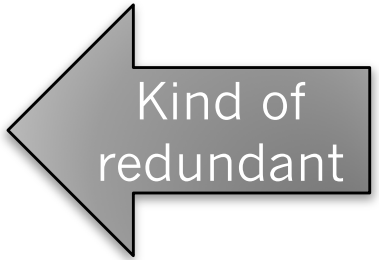
- Sometimes the result of an expression is used solely as a truth value.
- This situation frequently arises with the comparison and logical operators.
  - For example, in Java, these operators return a `bool` rather than an `int`, and all conditional tests require a `bool`.
- Currently, we do a lot of work to compute a 0 or 1 as the result of a comparison or logical operator.
- If we are merely going to test the truth value of the result next, we can streamline this process.

Example

- Generate code for the body of the function:

```
int a, b;

int f(void)
{
    while (a < b)
        a = a + b;
}
```

```
.L0:
        movl    a, %eax
        cmpl    b, %eax
        setl    %al
        movzbl  %al, %eax
        cmpl    $0, %eax
        je      .L1
        movl    a, %eax
        addl    b, %eax
        movl    %eax, a
        jmp     .L0
.L1:
```



Label: Header File

- Let's introduce a `Label` class to make things easier.

```
class Label {
    static unsigned _counter;
    unsigned _number;

public:
    Label();
    unsigned number() const;
};

ostream &operator <<(ostream &ostr, const Label &label);
```

Label: Source File

- Let's introduce a `Label` class to make things easier.

```
unsigned Label::_counter = 0;

Label::Label() {
    _number = _counter ++;
}

unsigned Label::number() {
    return _number;
}

ostream &operator <<(ostream &ostr, const Label &label) {
    return ostr << ".L" << label.number();
}
```

Label: Description

- Our `Label` class is essentially a wrapper around a single integer, which represents the label number.
- However, since it is a new type, we can define a constructor for it and also overload operators.
- Our constructor simply assigns it the next label number in sequence.
- The stream operator simply outputs the label number prefixed by the standard label prefix.
  - This is useful if we decide to use a different prefix.

A New Function

- Now, let's add a function `test` to `Expression`.

```
void Expression::test(const Label &label, bool ifTrue)
{
    generate();

    if (_register == nullptr)
        load(this, getreg());

    cout << "\tcmpl\t$0, " << this << endl;
    cout << (ifTrue ? "\tjne\t" : "\tje\t") << label << endl;

    assign(this, nullptr);
}
```

Explanation of Our Function

- Our test function does the following:
  1. Generates code for the expression.
  2. Compares the result against zero.
  3. Branches to the given `label` depending on the status of the `ifTrue` parameter.
- Other code generation functions can use our new function to test the truth value of an expression.
- Since our function is defined in the base class, it will be inherited by all subclasses of `Expression`.
  - Subclasses can provide their own versions, as we shall see.

Using the New Infrastructure

- We can easily write a function to generate code for a **while** statement using our new infrastructure.

```
void While::generate()
{
    Label loop, exit;

    cout << loop << ":" << endl;

    _expr->test(exit, false);
    _stmt->generate();
    release();

    cout << "\tjmp\t" << loop << endl;
    cout << exit << ":" << endl;
}
```

Releasing Registers

- We will write a convenience function to release all registers just in case we forgot somewhere.
- We can use this new function after we generate code for a statement.

```
void release()
{
    for (unsigned i = 0; i < registers.size(); i ++)
        assign(nullptr, registers[i]);
}
```

Specialized Functions

- We can write specialized versions of `test` for any subclass of `Expression`.

```
void LessThan::test(const Label &label, bool ifTrue)
{
    _left->generate();
    _right->generate();

    if (_left->_register == nullptr)
        load(_left, getreg());

    cout << "\tcmpl\t" << _right << ", " << _left << endl;
    cout << (ifTrue ? "\tjl\t" : "\tjge\t") << label << endl;

    assign(_left, nullptr);
    assign(_right, nullptr);
}
```

Example: Improved

- Generate code for the body of the function:

```
int a, b;

int f(void)
{
    while (a < b)
        a = a + b;
}
```

```
.L0:
        movl    a, %eax
        cmpl    b, %eax
        jge     .L1
        movl    a, %eax
        addl    b, %eax
        movl    %eax, a
        jmp     .L0
.L1:
```

Optional Improvements

- We could select a different register to spill than the first one in `getreg()`.
  - Which register to choose? Random? LRU?
- Instead of always spilling in `load()`, we could try to move the value to another register.
  - This gets tricky ... there's no point in moving a value in a caller-saved register to another one when doing a call.
- We can use the callee-saved registers.
  - GCC will do this to save caller-saved registers before making a function call. Clang just spills the caller-saved registers.

COEN 175

Lecture 21: Floating Point

Intel Floating Point

- Intel originally used a coprocessor for floating-point.
- The coprocessor uses a stack for computation, so an expression is evaluated as if written in postfix.
- Consider the expression `a * b + c * d`:

```
fldl   a      # load/push a
fldl   b      # load/push b
fmulp         # multiply and pop
fldl   c      # load/push c
fldl   d      # load/push d
fmulp         # multiply and pop
faddp         # add and pop
```

- Any return value is left on the top of the stack.

Modern Floating Point

- Modern Intel processors (since the Pentium III) use SSE (Streaming SIMD Extensions) for floating point.
  - The SSE register names are `%xmm0` – `%xmm7`.
  - They are all **caller-saved** registers.
- The arithmetic opcodes are:
  - `movsd` – move scalar, double-precision
  - `addsd` – add scalars, double-precision
  - `subsd` – subtract scalars, double-precision
  - `mulsd` – multiply scalars, double-precision
  - `divsd` – divide scalars, double-precision
  - `ucomisd` – (unordered) compare scalars, double-precision

Example: Addition

- Assume the registers are `%xmm0`, `%xmm1`, etc.
- Assume that all variables are 64-bit reals and are global variables so they can be referred to by name.
- Generate code for `a * b + c * d`.

```
movsd   a, %xmm0          # load: %xmm0 allocated
mulsd   b, %xmm0
movsd   c, %xmm1          # load: %xmm1 allocated
mulsd   d, %xmm1
addsd   %xmm1, %xmm0      # %xmm1 deallocated
```

- Compare this with our code for integer arithmetic.

Putting It All Together

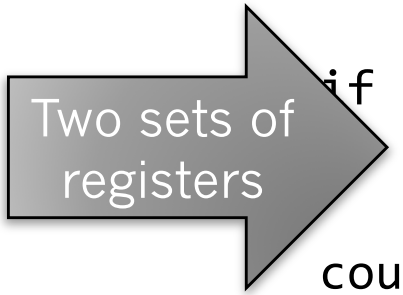
- Let's write a complete function for addition.

```
void Add::generate()
{
    // Generate code for both the left child
    // and the right child

    // If the left child is not in a register,
    // then allocate a register and load it

    // Perform the operation such as
    // "addl right, left"

    // If the right operand is in a register,
    // then deallocate it
}
```

Putting It All Together

- Let's write a complete function for addition.

```
void Add::generate()
{
    _left->generate();
    _right->generate();

    if (_left->_register == nullptr)
        load(_left, FP(_left) ? fp_getreg() : getreg());

    cout << "\tadd" << suffix(_left);
    cout << _right << ", " << _left << endl;

    assign(_right, nullptr);
    assign(this, _left->_register);
}
```



Utility Functions

- Let's write the utility functions we've used.

```
# define FP(expr)     ((expr)->type().isReal())
# define BYTE(expr)   ((expr)->type().size() == 1)

static string suffix(Expression *expr) {
    return FP(expr) ? "sd\t" : (BYTE(expr) ? "b\t" : "l\t");
}

Register *fp_getreg() {
    for (unsigned i = 0; i < fp_registers.size(); i ++)
        if (fp_registers[i]->_node == nullptr)
            return fp_registers[i];

    load(nullptr, fp_registers[0]);
    return fp_registers[0];
}
```

Floating-Point Comparison

- The `ucomisd` instruction sets the flags register based on the floating-point status word.
- The zero and carry flag are set, but not the sign flag.
- Therefore, we must use the **unsigned** `set` and `jump` instructions:
  - `setb/jb` – set/jump if below
  - `seta/ja` – set/jump if above
  - `setbe/jbe` – set/jump if below or equal
  - `setae/jae` – set/jump if above or equal

Example: Comparison

- Assume all variables are 64-bit reals.
- Generate floating-point code for `x > y * z`.

```
movsd      y, %xmm0        # %xmm0 allocated
mulsd      z, %xmm0
movsd      x, %xmm1        # %xmm1 allocated
ucomisd    %xmm0, %xmm1    # %xmm0 and %xmm1 deallocated
seta       %al             # %eax allocated
movzbl     %al, %eax
```

- Note that the result has type `int` and is stored in an integer register, not a floating-point register.

Floating-Point Conversion

- To convert a 32-bit integer to a 64-bit real:
  - `cvtsi2sd` – convert scalar integer to scalar double
- To convert a byte to a 64-bit real, we must first sign extend the byte into a 32-bit integer register.
- To convert a 64-bit real to a 32-bit integer:
  - `cvttsd2si` – convert scalar double to scalar integer
- To convert a 64-bit real to a byte, we first convert the real into a 32-bit integer register and then use the corresponding byte register.

Example: Conversion

- Assume `n` is a 32-bit integer and `x` is a 64-bit real.
- Generate code for `n = (char) (x + n)`.
  - With no coercions: `n = (int) (char) (x + (double) n)`.

```
cvtsi2sd   n, %xmm0
movsd      x, %xmm1
addsd      %xmm0, %xmm1
cvttsd2si  %xmm1, %eax
movsbl     %al, %eax
movl       %eax, n
```

- Note that the cast to a `char` generates no code beyond the conversion to `int`, but we do use `%al` for the next operation.

Floating-Point Literals

- Floating-point literals must be stored in memory and referenced using an assembler label.
- The `.double` directive is used for this purpose:

```
.L0:    .double    3.14159
.L1:    .double    2.989792e+8
```

- Note that like the `.asciz` directive for strings, the `.double` directive must be used in the `.data` section of the assembly file.

Floating-Point Zero

- A floating-point value of zero is useful for:
  - Determining a truth value;
  - Negating a floating-point value.
- Although floating-point literals must be stored in memory, there is a simple way to generate zero:

```
pxor    %xmm0, %xmm0
```

- Generate code for –x + y.

```
pxor    %xmm0, %xmm0
subsd   x, %xmm0
addsd   y, %xmm0
```

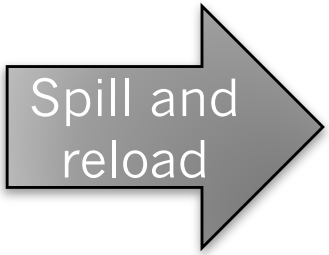
Floating-Point Arguments

- Floating-point arguments are pushed on the stack.
  - However, we cannot simply “push” them because they are 8-byte values, not 4-byte values.
  - Instead, we must first adjust the stack pointer and move them ourselves, effectively mimicking a push instruction.
- Generate code for `f(x + y, z)`.

```
movsd   z, %xmm0
subl    $8, %esp
movsd   %xmm0, (%esp)
movsd   x, %xmm0
addsd   y, %xmm0
subl    $8, %esp
movsd   %xmm0, (%esp)
call    f
addl    $16, %esp
```

Floating-Point Return

- For compatibility, a floating-point return value is still placed on the top of the coprocessor stack.
- Since there is no direct path between the SSE registers and the coprocessor, we must use memory.




```
# return x + y

movsd   x, %xmm0
addsd   y, %xmm0
movsd   %xmm0, t0
fldl    t0
jmp     f.exit
```

```
# x * f() + y

call    f
fstpl   t1
movsd   x, %xmm0
mulsd   t1, %xmm0
addsd   y, %xmm0
```

Summary

- Within our framework, floating-point values are treated no differently than integer values.
  - Our `load()` function is intelligent enough to handle either integer or floating-point loads.
  - We just need to tell it the correct register.
  - The arithmetic and comparison code generation functions require little to no modification.
- Code generation for type conversion is tricky.
- Function calls and returns require using the coprocessor stack and reloading from memory.