



Informatik II

Languages, Compilers, and Theory

Lecture 5:

Parsing Review

Names and Binding



Review

- Last time:
 - Context free languages (CFLs) context free grammars (CFGs)
 - Key algorithms:
 - FIRST, FOLLOW, NULLABLE, and PREDICT sets for CFGs
 - Left recursion and common prefix elimination
 - Parsing
 - Using CFGs to specify programming language syntax
 - Construction of recursive descent parsers from CFGs
 - Automatic generation of parsers from CFGs



Context Free Grammars

- A context free grammar (CFG) is a recursive definition of a language with:
 - An alphabet Σ of **symbols**
 - A set of **productions** of the form
 - symbol \rightarrow symbol symbol ... symbol
 - A **start symbol**
 - A set of **non-terminal symbols** taken from the alphabet Σ which may appear on the left or right hand sides of productions (convention: written in all capital letters)
 - A set of **terminal symbols** taken from the alphabet Σ which may appear only on the right hand sides of productions (convention: written in all lower case letters)
- The set of all strings **generated** by a CFG G is said to be the **language of G** and is denoted $L(G)$



Derivations

- A **derivation** is a series of replacement operations that shows how to derive a string of terminals (tokens) from the grammar start symbol
- Assuming there is some production $X \rightarrow \gamma$, a single replacement operation, or **derivation step**, can be written $\alpha X \beta \Rightarrow \alpha \gamma \beta$, for arbitrary strings of grammar symbols α , β , and γ
- Shorthand notations:
 - $\alpha \Rightarrow^* \beta$ means β can be derived from α in 0 or more steps
 - $\alpha \Rightarrow^+ \beta$ means β can be derived from α in 1 or more steps
 - $\alpha \Rightarrow^n \beta$ means β can be derived from α in exactly n steps
- Leftmost derivations:
 - For each derivation step $\alpha X \beta \Rightarrow \alpha \gamma \beta$, X must be the leftmost non-terminal in the string of symbols $\alpha X \beta$
 - Used in LL(k) or top-down parsing
- Rightmost derivations:
 - For each derivation step $\alpha X \beta \Rightarrow \alpha \gamma \beta$, X must be the rightmost non-terminal in the string of symbols $\alpha X \beta$
 - Used in LR(k) or bottom-up parsing
 - Sometimes called **canonical derivations**



Derivation Example

- Grammar:

$\text{expr} \rightarrow (\text{sum}) \mid \text{INT}$
 $\text{sum} \rightarrow \text{expr} + \text{expr}$

- Input:

$(\text{INT} + (\text{INT} + \text{INT}))$

Leftmost derivation:

$\text{expr} \Rightarrow (\text{sum})$
 $\Rightarrow (\underline{\text{expr} + \text{expr}})$
 $\Rightarrow (\underline{\text{INT}} + \text{expr})$
 $\Rightarrow (\text{INT} + (\underline{\text{sum}}))$
 $\Rightarrow (\text{INT} + (\underline{\text{expr} + \text{expr}}))$
 $\Rightarrow (\text{INT} + (\underline{\text{INT}} + \text{expr}))$
 $\Rightarrow (\text{INT} + (\text{INT} + \underline{\text{INT}}))$

Rightmost derivation:

$\text{expr} \Rightarrow (\text{sum})$
 $\Rightarrow (\underline{\text{expr} + \text{expr}})$
 $\Rightarrow (\text{expr} + (\underline{\text{sum}}))$
 $\Rightarrow (\text{expr} + (\underline{\text{expr} + \text{expr}}))$
 $\Rightarrow (\text{expr} + (\text{expr} + \underline{\text{INT}}))$
 $\Rightarrow (\text{expr} + (\underline{\text{INT}} + \text{INT}))$
 $\Rightarrow (\underline{\text{INT}} + (\text{INT} + \text{INT}))$

Derivations and Parse Trees

expr

⇒ (sum)

⇒ (expr + expr)

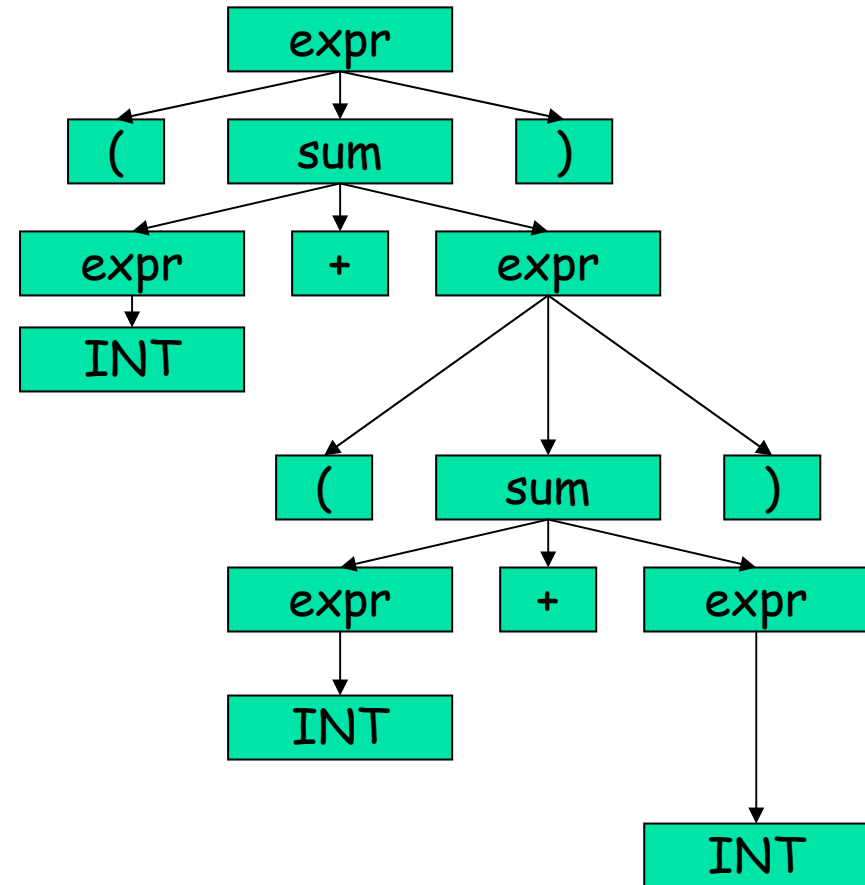
⇒ (INT + expr)

⇒ (INT + (sum))

⇒ (INT + (expr + expr))

⇒ (INT + (INT + expr))

⇒ (INT + (INT + INT))





LL Parsers

- Match an input of terminal symbols (tokens) to productions in a grammar by making leftmost derivations
- Given a set of productions for a non-terminal, $X \rightarrow \gamma_1 | \dots | \gamma_n$, and a leftmost derivation step $\alpha X \beta \Rightarrow \alpha \gamma_i \beta$, we must be able to determine which γ_i to choose by looking only at the next k input tokens
- Note:
 - Given a set of leftmost derivation steps from the start symbol $S \Rightarrow \alpha X \beta$, the string of symbols α will consist only of terminals (tokens) and represents the portion of the input matched to grammar productions so far



Problems with LL Parsing

- Left recursion
 - Productions of the form:
 - $A \rightarrow A\alpha$
 - $A \rightarrow \beta$
 - LL parsers will go into an infinite loop when trying to make leftmost derivations in such grammars
 - Can be eliminated by rewriting as:
 - $A \rightarrow \beta A'$
 - $A' \rightarrow \alpha A' \mid \epsilon$



Informal Justification

- Original grammar:

- $A \rightarrow Aa$
- $A \rightarrow \beta$

- Derives:

$$A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow^* \beta aaa \dots$$

- Rewritten Grammar:

- $A \rightarrow \beta A'$
- $A' \rightarrow aA' \mid \epsilon$

- Derives:

$$A \Rightarrow \beta A' \Rightarrow \beta a A' \Rightarrow \beta a a A' \Rightarrow^* \beta a a a \dots$$



Problems with LL Parsing

- Common Prefixes
 - Productions of the form:
 - $A \rightarrow ba$
 - $A \rightarrow b\beta$
 - LL(1) parsers can't decide which right hand side to choose when replacing A in a leftmost derivation step because both right hand sides begin with the same terminal symbol (token)
 - Can be eliminated by factoring:
 - $A \rightarrow bA'$
 - $A' \rightarrow a \mid \beta$

Building a Recursive Descent Parser

- Each non-terminal in the grammar has a subroutine which when called corresponds to a single leftmost derivation step

- Example:

- factor \rightarrow (expr)
- factor \rightarrow [sexpr]

```
void factor (void) {  
    switch(next_token()) {  
        case '(':  
            expr(); match('('); break;  
        case '[':  
            sexpr(); match(']'); break;  
    }  
}
```

- Hard part:

- Figuring out which tokens label the 'case' arms of the switch statement

Building a Recursive Descent Parser

■ PREDICT Sets

- Tell us which production right hand side to choose in a leftmost derivation when there are several choices
- Defined in terms of FIRST, FOLLOW, and NULLABLE sets
 - $\text{NULLABLE}(X)$ is true if $X \Rightarrow^* \epsilon$
 - $\text{FIRST}(\gamma)$ is $\{b : \gamma \Rightarrow^* b\gamma\}$
 - $\text{FOLLOW}(X)$ is $\{b : \gamma \Rightarrow^+ aXb\beta\}$
- $\text{PREDICT}(A \rightarrow \gamma) = \text{FIRST}(\gamma) \cup \text{FOLLOW}(A)$ if $\text{NULLABLE}(\gamma)$
- $\text{PREDICT}(A \rightarrow \gamma) = \text{FIRST}(\gamma)$ if not $\text{NULLABLE}(\gamma)$

A Recursive Descent Parser Example

- Grammar:
 - $S \rightarrow E \$ \$$
 - $E \rightarrow E * T \mid T$
 - $T \rightarrow id$
 - $T \rightarrow id (E)$
- Problem #1: left recursion in E
- Rewrite:
 - $S \rightarrow E \$ \$$
 - $E \rightarrow T E'$
 - $E' \rightarrow * T E' \mid \epsilon$
 - $T \rightarrow id$
 - $T \rightarrow id (E)$

A Recursive Descent Parser Example

- Grammar:
 - $S \rightarrow E \$\$$
 - $E \rightarrow T E'$
 - $E' \rightarrow * E' \mid \epsilon$
 - $T \rightarrow id$
 - $T \rightarrow id (E)$
- Problem #2: common prefixes in T
- Rewrite:
 - $S \rightarrow E \$\$$
 - $E \rightarrow T E'$
 - $E' \rightarrow * T E' \mid \epsilon$
 - $T \rightarrow id T'$
 - $T' \rightarrow (E) \mid \epsilon$

A Recursive Descent Parser Example

- Grammar:
 - $S \rightarrow E \$ \$$
 - $E \rightarrow T E'$
 - $E' \rightarrow * T E' \mid \epsilon$
 - $T \rightarrow \text{id } T'$
 - $T' \rightarrow (E) \mid \epsilon$
- Compute PREDICT sets
 - Easy ones first:
 - $\text{PREDICT}(E' \rightarrow * T E') = \{*\}$
 - $\text{PREDICT}(T \rightarrow \text{id } T') = \{\text{id}\}$
 - $\text{PREDICT}(T' \rightarrow (E)) = \{()\}$

A Recursive Descent Parser Example

- Grammar:
 - $S \rightarrow E \$\$$
 - $E \rightarrow T E'$
 - $E' \rightarrow * T E' \mid \epsilon$
 - $T \rightarrow \text{id } T'$
 - $T' \rightarrow (E) \mid \epsilon$
- Compute PREDICT sets
 - What about:
 - PREDICT($S \rightarrow E \$\$$) ?
 - PREDICT($E \rightarrow T E'$) ?
 - PREDICT($E' \rightarrow \epsilon$) ?
 - PREDICT($T' \rightarrow \epsilon$) ?

A Recursive Descent Parser Example

- Grammar:

- $S \rightarrow E \$ \$$

- $E \rightarrow T E'$

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

- $T \rightarrow \text{id } T'$

- $T' \rightarrow (E) \mid \epsilon$

- $\text{PREDICT}(S \rightarrow E \$ \$) = \text{FIRST}(E) = \text{FIRST}(T) = \{\text{id}\}$

- $\text{PREDICT}(E \rightarrow T E') = \text{FIRST}(T) = \{\text{id}\}$

- $\text{PREDICT}(E' \rightarrow \epsilon) = \text{FOLLOW}(E')$

- $\text{PREDICT}(T' \rightarrow \epsilon) = \text{FOLLOW}(T')$

A Recursive Descent Parser Example

- Grammar:
 - $S \rightarrow E \$ \$$
 - $E \rightarrow T E'$
 - $E' \rightarrow * T E' \mid \epsilon$
 - $T \rightarrow \text{id } T'$
 - $T' \rightarrow (E) \mid \epsilon$
- $\text{PREDICT}(E' \rightarrow \epsilon) = \text{FOLLOW}(E')$
 - $\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{\$ \$\}$

A Recursive Descent Parser Example

- Grammar:
 - $S \rightarrow E \$\$$
 - $E \rightarrow T E'$
 - $E' \rightarrow * T E' \mid \epsilon$
 - $T \rightarrow \text{id } T'$
 - $T' \rightarrow (E) \mid \epsilon$
- $\text{PREDICT}(T' \rightarrow \epsilon) = \text{FOLLOW}(T')$
 - $\text{FOLLOW}(T') = \text{FOLLOW}(T) = \text{FIRST}(E') \cup \text{FOLLOW}(E)$
 - $\text{FOLLOW}(E) = \{\$\$\}$
 - $\text{FIRST}(E') = \{*\} \cup \text{FOLLOW}(E')$
 $= \{*\} \cup \text{FOLLOW}(E) = \{*, \$\$ \}$

A Recursive Descent Parser Example

- All Predict Sets:
 - $\text{PREDICT}(S \rightarrow E \$\$) = \{\text{id}\}$
 - $\text{PREDICT}(E \rightarrow T E') = \{\text{id}\}$
 - $\text{PREDICT}(E' \rightarrow * T E') = \{*\}$
 - $\text{PREDICT}(E' \rightarrow \epsilon) = \{\$\$ \}$
 - $\text{PREDICT}(T \rightarrow \text{id } T') = \{\text{id}\}$
 - $\text{PREDICT}(T' \rightarrow (E)) = \{(\}$
 - $\text{PREDICT}(T' \rightarrow \epsilon) = \{*, \$\$ \}$
- No duplicate terminals in PREDICT sets for productions of the same non-terminal
 - Therefore, grammar is LL(1)!

A Recursive Descent Parser Example

- Building the recursive descent function for E'
 - Predict sets for E'
 - $\text{PREDICT}(E' \rightarrow * T E') = \{*\}$
 - $\text{PREDICT}(E' \rightarrow \epsilon) = \{\$\$\}$

```
void E_prime (void) {  
    switch(next_token()) {  
        case `*`:  
            T(); E_prime(); break;  
        case `$$`:  
            break;  
    }  
}
```



Worksheet (1)

- Write a context free grammar which generates the language of all parenthesized integer arithmetic expressions using operators $+$, $-$, $*$, $\%$. Don't worry about precedence.
- Example strings in the language:
 - 10
 - $2 + 4$
 - $(1 + (4 * 5) \% 2)$



Worksheet (2)

- Write a context free grammar which generates the same strings as the following regular expressions
 - $a.b.b.a$
 - $a|b|c$
 - $a.(b.c)^*$
 - $(a.b^*. (c|d))^*$



Worksheet (3)

- Eliminate (immediate) left recursion and common prefixes for the following grammar
 - $S \rightarrow S a B c$
 - $S \rightarrow a$
 - $B \rightarrow d B d$
 - $B \rightarrow d S .$



Worksheet (4)

- Build PREDICT sets for the following LL(1) grammar
 - $S \rightarrow A \$\$$
 - $A \rightarrow aA \mid bBCd$
 - $B \rightarrow b \mid \epsilon$
 - $C \rightarrow c \mid \epsilon$



Names, Scopes, and Bindings

- Names

- A mnemonic character name used to represent something else
 - Usually identifiers
- Essential for abstraction
 - Allow programmers to name values thereby eliminating the need for directly manipulating addresses, register names, etc.
 - Example: Don't need to know whether variable foo is stored in register \$t0 or at memory location 10000.
 - Allow the programmer to let a simple name stand for potentially complex program fragment
 - Example: $foo = a*a + b*b + c*c + d*d$;
 - Both cases reduce conceptual complexity



Names, Scopes, and Bindings

- Bindings

- An association between two things
 - A variable name to a value
 - A variable name to a specific memory location
 - A type and its representation or layout in memory
- **Binding time** is the time at which such an association is made

Names, Scopes, and Bindings

- Scopes

- The textual region of the program in which a binding is active

- Java Example:

```
public void foo (int a) {  
    int b;  
    while(a < n) {  
        int c;  
        c = a + a;  
        b = a * c;  
        a++;  
    }  
}
```

Global Scope

Method Scope

Block Scope

- Improve abstraction by controlling the visibility of bindings



Binding Times

- Language Design Time
 - Decisions made by the programming language designer
 - Typical examples:
 - Binding control structures (conditionals, loops, etc.) to their abstract meaning
 - "Statements in a while loop body execute until the while condition is no longer true"
 - Binding primitive type names (int, float, char) to their required representation
 - "Variables of type int hold signed 32-bit values"



Binding Times

- Language Implementation Time
 - Decisions made by the compiler writer
 - In other words, language implementation issues not specifically defined at Language Design Time
 - Typical examples:
 - Binding of primitive types to representation precision (number of bits)
 - "shorts are 16-bits, ints are 32-bits, longs are 64-bits"
 - Binding of file operations to operating system specific implementations of those operations
 - "open() is implemented with the SYS_open system call"



Binding Times

- Program Writing Time
 - Decisions made by the programmer
 - Typical examples:
 - Binding an algorithm to the programming language statements which implement it
 - Binding names to variables required by an algorithm
 - Binding data structures to language types



Binding Times

- Compile Time
 - Decisions made by the compiler
 - Typical examples:
 - Binding high-level constructs to machine code (includes optimization)
 - Binding statically defined data structures to a specific memory layout
 - Binding user defined types to a specific memory layout



Binding Times

- Link Time
 - Decisions made by the linker
 - Linkers combine program modules together
 - Typical examples:
 - Binding objects (subroutines and data) to specific locations in an executable file
 - Binding names which reference objects in other modules to actual references to locations



Binding Times

- Load Time

- Decisions made by the loader

- Loaders read executable files into memory

- Typical examples:

- Binding virtual addresses in the executable file to physical memory addresses

- Not really necessary in modern operating systems because the operating system binds virtual addresses to physical addresses using virtual memory hardware



Binding Times

- Run Time

- Decisions made during the execution of the program
- Typical examples:
 - Binding actual values to program variables
 - "a = a + 1;"
 - Binding references to dynamically allocated objects to memory addresses
 - Binding names to objects in languages with dynamic scope



Binding Times

- Similar binding decisions may be made at multiple binding times
 - Linking decisions may occur at link time, at load time (one type of dynamic linking), or at run-time (another type of dynamic linking)
 - Optimization decisions may occur at compile time, at link time, at load time, and even at run-time (dynamic optimization)
- Binding decisions may be made at different binding times in different languages
 - C binds variable names to the objects they reference at compile time
 - When we say "foo=bar;" in C, we know exactly whether or not foo and bar are global or local, what their types are, whether or not their types are compatible for assignment, etc.
 - Perl (and in fact all interpreted languages) bind variable names to the objects they reference at run-time
 - We can say "\$foo = \$bar;" in Perl, and if the name \$bar is not already bound to an object, one will be created, and then assigned to \$foo



Binding Times

- Static binding
 - Refers to all binding decisions made before run-time
- Dynamic binding
 - Refers to all binding decisions made at run-time
- Early binding times
 - Associated with greater efficiency
 - Compiled languages typically run very quickly because most binding decisions have been made at compile time
 - Once a binding decision is made though, we lose some flexibility
- Late binding times
 - Associated with greater flexibility
 - Interpreted languages allow most binding decisions to be made at run-time, allowing greater flexibility
 - For instance, most interpreted languages allow one program to dynamically generate and execute fragments of another program
 - Because binding decisions are made at run-time, interpreted languages can be slow



Scopes

- Textual region of a program during which a binding is active
- Two varieties:
 - Static scope
 - Can determine at compile time exactly which names refer to which objects at which points in the program
 - Dynamic scope
 - Bindings from names to objects depend on the flow of execution of the program



Scopes

- Elaboration

- The process by which a set of bindings become active when control enters a scope
 - For example, allocation of storage to hold the objects

- Referencing Environment

- The set of active bindings at a given point in a program's execution
- Determined by **scope rules** for the programming language



Static Scope

- Bindings between names and objects can be determined at compile time
- Simple
 - Early versions of BASIC had a single, global scope
- Complex
 - Nested subroutines and modules allowed by modern programming languages require more complicated scope rules

Nested sub-routines

- Question:
 - In the body of function F1, which object does X refer to?
- Closest nested scope rule
 - Reference to variable refers to the object in the closest nested scope

```
procedure P1 (A1 : T1);
var X : real;
...
  procedure P2 (A2 : T2);
    ...
      procedure P3 (A3 : T3);
        ...
        begin
          ...      (* body of P3 *)
        end;
        ...
      begin
        ...      (* body of P2 *)
      end;
      ...
    procedure P4 (A4 : T4);
      ...
      function F1 (A5 : T5) : T6;
      var X : integer;
      ...
      begin
        ...      (* body of F1 *)
      end;
      ...
    begin
      ...      (* body of P4 *)
    end;
    ...
  begin
    ...      (* body of P1 *)
  end
```

A Problem Not Addressed by Nested Sub-routines

- Information hiding for complex abstract data types
- For simple ADTs, functions with static locals might work

```
/*  
  Place into *s a new name beginning with the letter l and continuing  
  with the ascii representation of an integer guaranteed to be distinct  
  in each separate call. s is assumed to point to space large enough  
  to hold any such name; for the short ints used here, seven characters  
  suffice. l is assumed to be an upper or lower-case letter.  
  sprintf 'prints' formatted output to a string.  
*/  
void gen_new_name (char *s, char l) {  
  static short int name_nums[52];  
  /* C guarantees that static local variables are initialized  
  to zeros */  
  int index = (l >= 'a' && l <= 'z') ? l-'a' : 26 + l-'A';  
  name_nums[index]++;  
  sprintf (s, "%c%d\0", l, name_nums[index]);  
}
```

- The variable `name_nums` retains its value across calls to `gen_new_name`
- Too simple for ADTs with multiple functions which need to share global state



Modules

- A module allows a collection of objects to be encapsulate so that
 - Objects inside the module are visible to one another
 - Objects on the inside are not visible on the outside unless explicitly exported
 - Objects outside are not visible on the inside unless explicitly imported

A module example

- A Stack abstraction in Modula
- We export push and pop functions
- Not visible outside of module
 - Top of stack pointer "top"
 - Stack array "s"

```
CONST stack_size = ...
TYPE element = ...
...
MODULE stack;
IMPORT element, stack_size;
EXPORT push, pop;
TYPE
    stack_index = [1..stack_size];
VAR
    s    : ARRAY stack_index OF element;
    top  : stack_index;      (* first unused slot *)


PROCEDURE error; ...

PROCEDURE push (elem : element);
BEGIN
    IF top = stack_size THEN
        error;
    ELSE
        s[top] := elem;
        top := top + 1;
    END;
END push;

PROCEDURE pop () : element; (* A Modula-2 function is just a *)
                             (* procedure with a return type. *)
BEGIN
    IF top = 1 THEN
        error;
    ELSE
        top := top - 1;
        RETURN s[top];
    END;
END pop;

BEGIN
    top := 1;
END stack;
```

```
VAR x, y : element;
...
push (x);
...
y := pop;
```



More module scope terminology

- Closed scopes
 - Modules into which names must be explicitly imported in order to be visible within the module
- Open scopes
 - Modules which do not require explicit imports in order for outside names to be visible



Dynamic Scope

- Bindings between names and object depend on the flow of control at run-time
 - Order in which sub-routines are called is important
- Dynamic scope rules are usually simple
 - The current binding between name and object is the one most recently encountered during execution



Static versus Dynamic Scope

- Static scoping
 - Program prints 1
- Dynamic scoping
 - Program prints 1 or 2 depending on value read at line 8
- Why?
 - Is the assignment to "a" at line 3 an assignment to the global "a" from line 1 or the local "a" from line 5

```
1: a : integer    -- global declaration
2: procedure first
3:   a := 1
4: procedure second
5:   a : integer  -- local declaration
6:   first ()
7:   a := 2
8:   if read_integer () > 0
9:     second ()
10:  else
11:    first ()
12:  write_integer (a)
```



Dynamic Scoping

- What are the advantages?
 - Makes it easy to customize sub-routines
 - Example:

```
begin --nested block
  print_base: integer := 16
  print_integer(n)
```

 - The variable `print_base` controls the base which `print_integer` uses to print numbers
 - `print_integer` can be set early in a global scope to a default value, and temporarily overridden in a global scope to some other base