CO20-320241

# Computer Architecture and Programming Languages

CAPL

## Lecture 22 & 23

Dr. Kinga Lipskoch

Fall 2019

## Symbol-Table Management

- The function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name
- They may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned
- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name
- Should allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly

## Compiler-Construction Tools (1)

- The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, etc.
- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler

## Compiler-Construction Tools (2)

- Parser generators automatically produce syntax analyzers from a grammatical description of a programming language
- Scanner generators produce lexical analyzers from a regular-expression description of the tokens of a language
- Syntax-directed translation engines produce routines for walking a parse tree and generating intermediate code
- Code-generator generators produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language
- Data-flow analysis engines facilitate the gathering of information about how values are transmitted from one part of a program to each other part, it is part of code optimization
- Compiler-construction toolkits provide an integrated set of routines for constructing various phases of a compiler

## Evolution of Programming Languages

- First electronic computers appeared in the 1940's and were programmed in machine language by sequences of 0's and 1's
- First step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's
- Major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientific computation, Cobol for business data processing, and Lisp for symbolic computation
- Many more languages were created with innovative features to help make programming easier, more natural, and more robust

## Classification of Programming Languages by Generation

- **First-generation**: machine languages
- **Second-generation**: assembly languages
- **Third-generation**: higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java
- **Fourth-generation**: designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting
- **Fifth-generation**: logic- and constraint-based languages like Prolog and OPS5

## Another Classification of Programming Languages

- **Imperative language**: a program specifies how a computation is to be done like C, C++, C#, and Java
- **Declarative language**: a program specifies what computation is to be done like ML, Haskell, Prolog
- **Von Neumann language**: computational model is based on the von Neumann computer architecture like Fortran and C
- **Object-oriented language**: a program consists of a collection of objects that interact with one another like Simula 67, Smalltalk, C++, C#, Java, and Ruby
- **Scripting language**: interpreted languages with high-level operators designed for "gluing together" computations (originally called scripts) like Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl

## Impacts on Compilers

- Compiler writers have to track new language features, they also had to devise translation algorithms that would take maximal advantage of the new hardware capabilities
- The performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built
- A compiler by itself is a large program, moreover, many modern language-processing systems handle several source languages and target machines within the same framework
- The problem of generating the optimal target code from a source program is undecidable, in general
- Compiler writers must evaluate trade-offs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code

## Programming Language Basics: Static/Dynamic Distinction

- If a language allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time
- A policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time
- **Example**: The scope of a declaration of `x` is the region of the program in which uses of `x` refer to this declaration
- A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program
- Otherwise, it uses dynamic scope: as the program runs, the same use of `x` could refer to any of several different declarations of `x`
- Most languages, such as C and Java, use static scope

## Programming Language Basics: Environments and States

- Changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data
- **Example**: an assignment such as `x = y + 1` changes the value denoted by the name `x`
- The association of names with locations in memory (the store) and then with values can be described by two mappings that change as the program runs

$$\begin{tikzcd}
\text{names} \arrow[r, bend left, "\textit{environment}"] & \text{locations (variables)} \arrow[r, bend left, "\textit{state}"] & \text{values}
\end{tikzcd}$$

## Environment Change

Environments change according to the scope rules of a language

```
...
int i;    /* global i */
void f(...) {
  int i;    /* local i */
  ...
  i=3;    /* use of local i */
  ...
}
...
x = i + 1;    /* use of global i */
```

## Change Exceptions

The environment and state mappings are dynamic, but there are a few exceptions:

- Static versus dynamic binding of names to locations: Most binding of names to locations is dynamic, some declarations, like the global `i`, can be given a location in the store once and for all, as the compiler generates object code
- Static versus dynamic binding of locations to values: The binding of locations to values is generally dynamic as well, since we cannot tell the value in a location until we run the program, constants are an exception

## Names, Identifiers, and Variables

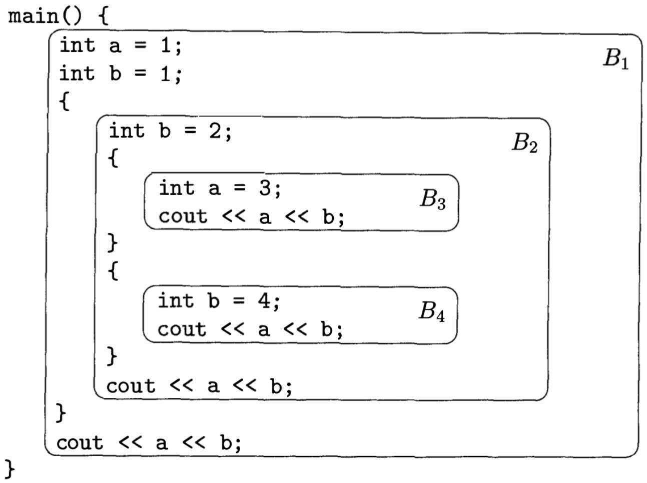
Although the terms often refer to the same thing, we use them to distinguish between compile-time names and the run-time locations denoted by names

- An identifier is a string of characters, typically letters or digits, that refers to (identifies) an entity, such as a data object, a procedure, a class, or a type
- All identifiers are names, but not all names are identifiers
- Names can also be expressions (e.g., `x.y`)
- A variable refers to a particular location of the store, the same identifier to be declared more than once; each such declaration introduces a new variable

## Static Scope and Block Structure

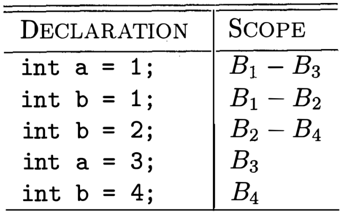
- Most languages, including C and its family, use static scope
- C++, Java, and C#, also provide explicit control over scopes through the use of keywords like public, private, and protected
- A block is a grouping of declarations and statements
- The static-scope rule for variable declarations in a block-structured languages is as follows
- If declaration `D` of name `x` belongs to block `B`, then the scope of `D` is all of `B`, except for any blocks `B'` nested to any depth within `B`, in which `x` is redeclared

## Example: Blocks in a C++ Program



```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

## Example: Scopes of Declarations



| DECLARATION | SCOPE |
|---|---|
| `int a = 1;` | $B_1 - B_3$ |
| `int b = 1;` | $B_1 - B_2$ |
| `int b = 2;` | $B_2 - B_4$ |
| `int a = 3;` | $B_3$ |
| `int b = 4;` | $B_4$ |

## Explicit Access Control

- Classes and structures introduce a new scope for their members
- The scope of a member declaration `x` in a class `C` extends to any subclass `C`', except if `C`' has a local declaration of the same name `x`
- The visibility keywords support encapsulation by restricting access
- A class definition may be separated from the definitions of some or all of its methods
- Regions inside and outside the scope may alternate, until all the methods have been defined

## Dynamic Scope

- Any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes
- Dynamic scope: a use of a name `x` refers to the declaration of `x` in the most recently called procedure with such a declaration
- Two examples of dynamic policies:
  - macro expansion in the C preprocessor
  - method resolution in object-oriented programming

```c
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b(); c(); }
```

- Dynamic scope resolution is also essential for polymorphic procedures

## Analogy Between Static and Dynamic Scoping

- The dynamic rule is to time as the static rule is to space
- The static rule asks us to find the declaration whose unit (block) most closely surrounds the physical location of the use
- The dynamic rule asks us to find the declaration whose unit (procedure invocation) most closely surroundings the time of the use

## Parameter Passing Mechanisms

- All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments
- Actual parameters = the parameters used in the call of a procedure
- Formal parameters = those used in the procedure definition
- The great majority of languages use either call-by-value or call-by-reference or both

## Call-by-Name

- Used in the early programming language Algol 60
- It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct)
- When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today

## Aliasing

- There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value
- It is possible that two formal parameters can refer to the same location; such variables are said to be aliases of one another
- As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well
- Understanding aliasing and the mechanisms that create it is essential if a compiler is to optimize a program

## Example: Aliasing

- Suppose `a` is an array belonging to a procedure `p`, and `p` calls another procedure `q(x, y)` with a call `q(a, a)`
- Suppose also that parameters are passed by value, but that array names are really references to the location where the array is stored, as in C or similar languages
- Now, `x` and `y` have become aliases of each other
- The important point is that if within `q` there is an assignment `x[10] = 2`, then the value of `y[10]` also becomes `2`

## A Simple Syntax-Directed Translator

- Illustrates the techniques by developing a working; Java program that translates representative programming language statements into three-address code, an intermediate representation
- The emphasis is on the front end of a compiler, in particular on lexical analysis, parsing, and intermediate code generation
- We start small by creating a syntax-directed translator that maps infix arithmetic expressions into postfix expressions
- We then extend this translator to map code fragments into three-address code

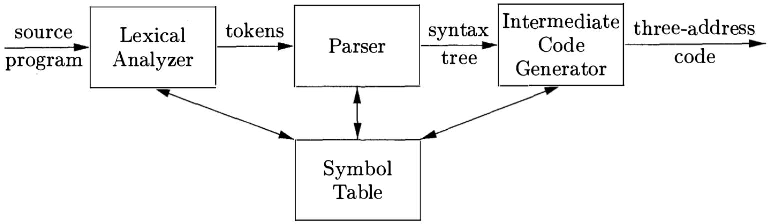
## Code Fragment to be Translated

```
{
  int i; int j; float[100] a;
  float v; float x;
  while(true) {
    do i = i + 1; while(a[i] < v);
    do j = j - 1; while (a[j] > v);
    if (i >= j) break;
    x = a[i]; a[i] = a[j]; a[j] = x;
  }
}
```

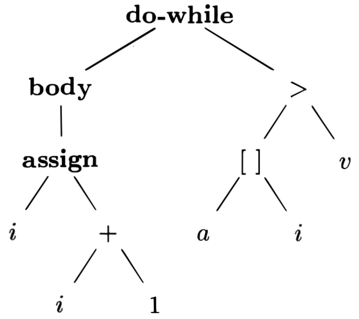
## Simplified Intermediate Code for the Program Fragment

```
1    i = i + 1
2    t1 = a[i]
3    if t1 < v goto 1
4    j = j - 1
5    t2 = a[j]
6    if t2 > v goto 4
7    ifFalse i >= j goto 9
8    goto 14
9    x = a[i]
10   t3 = a[j]
11   a[i] = t3
12   a[j] = x
13   goto 1
```

## Model of a Compiler Front End



## Intermediate Code for `do i = i + 1; while(a[i] < v);`



(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

## Syntax Definition

- We introduce a notation - the context-free grammar or grammar for short - that is used to specify the syntax of a language
- A grammar naturally describes the hierarchical structure of most programming language constructs
- **Example**: an `if-else` statement in Java can have the form
  `if (expression) statement else statement`
- Using the variable `expr` to denote an expression and the variable `stmt` to denote a statement, this structuring rule can be expressed as
  `stmt` $\rightarrow$ `if (expr) stmt else stmt`

## Rules

- `stmt` $\rightarrow$ `if (expr) stmt else stmt`
- The arrow may be read as “can have the form”
- Such a rule is called a production
- In a production, lexical elements like the keyword `if` and the parentheses are called terminals
- Variables like `expr` and `stmt` represent sequences of terminals and are called nonterminals

## Context-Free Grammars

Have four components:

1. A set of terminal symbols, sometimes referred to as “tokens”
2. A set of nonterminals, sometimes called “syntactic variables”
3. A set of productions, where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminals, called the body or right side of the production
4. A designation of one of the nonterminals as the start symbol

## Example

- Strings such as $9 - 5 + 2$, $3 - 1$ , or 7
- Lists of digits separated by plus or minus signs
- The following grammar describes the syntax
  1. `list` $\rightarrow$ `list` + `digit`
  2. `list` $\rightarrow$ `list - digit`
  3. `list` $\rightarrow$ `digit`
  4. `digit` $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- The bodies of the three productions with nonterminal list as head equivalently can be grouped

  `list` $\rightarrow$ `list` + `digit` | `list - digit` | `digit`
- The terminals of the grammar are the symbols + - 0 1 2 3 4 5 6 7 8 9
- The string of zero terminals, written as $\epsilon$, is called the empty string

## Derivations

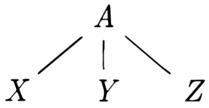
- A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal
- The terminal strings that can be derived from the start symbol form the language defined by the grammar
- Example: Java or C++ function call `max(x, y)`
- Grammar identifying these function calls:
  1. $\texttt{call} \rightarrow \texttt{id(optparams)}$
  2. $\texttt{optparams} \rightarrow \texttt{params} \mid \epsilon$
  3. $\texttt{params} \rightarrow \texttt{params}, \texttt{param} \mid \texttt{param}$

## Parsing

- Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar
- If it cannot be derived from the start symbol of the grammar
- Reporting syntax errors within the string
- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language

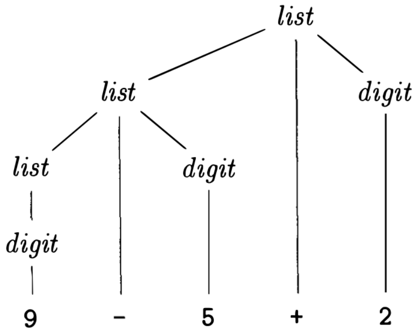
## Parse Trees

- If nonterminal $A$ has a production $A \rightarrow XYZ$, then a parse tree may have an interior node labeled $A$ with three children labeled $X$, $Y$, and $Z$, from left to right:



1. The root is labeled by the start symbol
2. Each leaf is labeled by a terminal or by $\epsilon$
3. Each interior node is labeled by a nonterminal
4. If $A$ is the nonterminal labeling some interior node and $X_1$, $X_2$, ..., $X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1X_2...X_n$. If $A \rightarrow \epsilon$ is a production, then a node labeled $A$ may have a single child labeled $\epsilon$

## Parse Tree Example

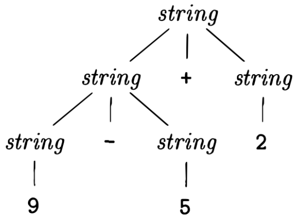


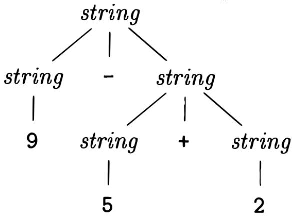
## Ambiguity

- We have to be careful in talking about the structure of a string according to a grammar
- A grammar can have more than one parse tree generating a given string of terminals
- Such a grammar is said to be ambiguous
- Since a string with more than one parse tree has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities

## Ambigious Grammars

- Suppose we used a single nonterminal `string` and did not distinguish between `digits` and `lists`
- $\texttt{string} \rightarrow \texttt{string} + \texttt{string} \mid \texttt{string} - \texttt{string} \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$





## Associativity of Operators

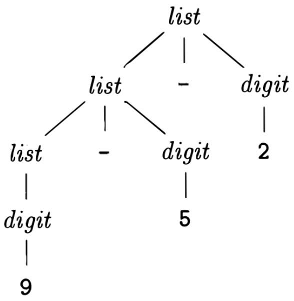
- `9+5+2` is equivalent to `(9+5)+2` and `9-5-2` is equivalent to `(9-5)-2`
- When an operand like `5` has operators to its left and right, conventions are needed for deciding which operator applies to that operand
- We say that the operator + associates to the left, because an operand with plus signs on both sides of it belongs to the operator to its left
- In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative

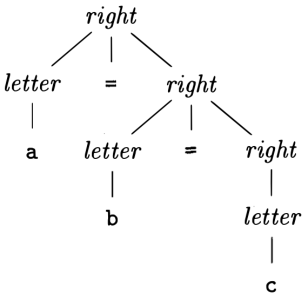
## Right-Associative Operators

- Some common operators such as exponentiation are right-associative
- The assignment operator = in C and its descendants is right-associative
- The expression `a=b=c` is treated in the same way as the expression `a=(b=c)`
- `a=b=c` can be generated by the following grammar:
  1. $\texttt{right} \rightarrow \texttt{letter} = \texttt{right} \mid \texttt{letter}$
  2. $\texttt{letter} \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{...} \mid \texttt{z}$

## Left- and Right-Associative Grammars

The parse tree for `9-5-2` grows down towards the left, whereas the parse tree for `a=b=c` grows down towards the right





## Precedence of Operators

- Consider the expression `9+5*2`
- Two possible interpretations: `(9+5)*2` or `9+(5*2)`
- Rules defining the relative precedence of operators are needed
- We create two nonterminals `expr` and `term` for the two levels of precedence, and an extra nonterminal `factor` for generating basic units in expressions
- $\texttt{factor} \rightarrow \texttt{digit} \mid \texttt{( expr )}$
- $\texttt{term} \rightarrow \texttt{term} * \texttt{factor} \mid \texttt{term} \,/\, \texttt{factor} \mid \texttt{factor}$
- $\texttt{expr} \rightarrow \texttt{expr} + \texttt{term} \mid \texttt{expr} - \texttt{term} \mid \texttt{term}$

## A Grammar for a Subset of Java Statements

```
(1) stmt → id = expression;
         | if (expression) stmt
         | if (expression) stmt else stmt
         | while(expression) stmt
         | do stmt while(expression);
         | { stmts }
(2) stmts → stmts stmt
          | ϵ
```

## Syntax-Directed Translation

- Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar
- Example: `expr` $\rightarrow$ `expr1 + term`
- We can translate `expr` by exploiting its structure, as in the following pseudo-code:

```
translate expr1;
translate term;
handle +;
```

## Attributes

- An attribute is any quantity associated with a programming construct
- Examples of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct, among many other possibilities
- Since we use grammar symbols (nonterminals and terminals) to represent programming constructs, we extend the notion of attributes from constructs to the symbols that represent them

## (Syntax-Directed) Translation Schemes

- A translation scheme is a notation for attaching program fragments to the productions of a grammar
- The program fragments are executed when the production is used during syntax analysis
- The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied