Introduction to Compiler Design

A **compiler** is a software that translates a program written in a high-level programming language (like C, C++, or Java) into machine code or intermediate code that a computer can understand. Compilers play a crucial role in making programming languages practical by converting user-written programs into instructions that hardware can execute.

Structure of a Compiler

A compiler is typically structured into several phases that perform different tasks. These phases can be grouped into **front-end**, **middle-end**, and **back-end** components.

1. **Front-end**:
   - **Lexical Analysis (Scanning)**: Breaks down the source code into tokens.
   - **Syntax Analysis (Parsing)**: Organizes tokens into a parse tree based on grammar rules.
   - **Semantic Analysis**: Checks for semantic errors, ensuring the program's meaning is correct.
2. **Middle-end**:
   - **Intermediate Code Generation**: Converts the parse tree into an intermediate representation.
   - **Optimization**: Optimizes the intermediate code to improve performance or reduce resource consumption.
3. **Back-end**:
   - **Code Generation**: Converts the intermediate code into machine code.
   - **Code Optimization**: Further optimizations specific to the target machine architecture.
   - **Code Linking and Assembly**: Produces the final executable code.

Lexical Analysis

Lexical Analysis is the first phase of the compiler. Its job is to scan the source code from left to right and break it into **tokens**. Tokens are the smallest units of the program, such as keywords, operators, identifiers, or punctuation marks.

Role of Lexical Analyzer

The Lexical Analyzer (also called a **scanner**) performs the following tasks:

1. **Tokenization**: Converts the input source code into tokens, which are meaningful sequences of characters.
2. **Remove Whitespaces and Comments**: The lexical analyzer ignores whitespace, comments, and extra formatting.
3. **Error Handling**: Identifies errors like invalid tokens or unsupported characters.
4. **Communicate with Syntax Analyzer**: After creating tokens, the lexical analyzer passes them to the next phase—syntax analysis.

Input Buffering

Input buffering is used to make reading from the source program more efficient. The source program is usually read from a file and can be large, so reading it character by character can be slow. Instead, **buffers** are used to read a block of the source code at a time.

- **Two-Buffer Scheme**: A common technique where two buffers are used, allowing one to be read while the other is being filled.
- **Sentinels**: Special characters placed at the end of a buffer to indicate when to switch buffers.

This method speeds up the scanning process by reducing the number of I/O operations.

Specification of Tokens

Each programming language has its own set of tokens, and they are usually specified by **regular expressions**. Tokens can be classified into categories such as:

1. **Keywords**: Reserved words in the language (e.g., if, while, return).
2. **Identifiers**: Names for variables, functions, classes, etc.
3. **Literals**: Constant values like 5, 3.14, or "hello".
4. **Operators**: Symbols like +, -, *, /, etc.
5. **Punctuation**: Characters like ;, ,, and {} that structure the code.

Regular expressions describe patterns for these tokens. For example:

- An identifier in C may be specified as [a-zA-Z_][a-zA-Z_0-9]*.

Recognition of Tokens

The Lexical Analyzer reads the input stream character by character and matches it against the regular expressions that define the tokens. **Finite Automata** (often **Deterministic Finite Automata**, DFA) are used to recognize tokens.

Steps for Token Recognition:

1. **Start State**: Begin in the initial state of the DFA.
2. **Transition**: Move to a new state based on the input character.
3. **Accepting State**: If an accepting state is reached, a token is recognized.
4. **Backtracking**: If no accepting state is reached, backtrack to the last valid token.

Lex: A Lexical Analyzer Generator

Lex is a tool used to automatically generate a lexical analyzer (scanner). Given a specification of tokens (usually via regular expressions), Lex generates C code that can recognize those tokens.

Steps to Use Lex:

1. **Specification**: Write the Lex specification, which includes token definitions and associated actions.
2. **Lex Program**: Use the Lex program to generate a C file with the lexical analyzer code.
3. **Compile and Link**: Compile the C file and link it with the rest of the compiler.

Lex Structure:

- The Lex program is divided into three sections:
    - **Definitions**: Define regular expressions for tokens.
    - **Rules**: Map regular expressions to actions (e.g., what to do when a token is recognized).
    - **Auxiliary Code**: Additional C code for integration.

Example of a Lex rule:

```
[0-9]+   { printf("NUMBER token\n"); }

[a-zA-Z_][a-zA-Z0-9_]*  { printf("IDENTIFIER token\n"); }
```

Conclusion

Lexical analysis is crucial as it simplifies the rest of the compilation process by breaking the source code into meaningful tokens. The efficiency of this phase can significantly affect the overall performance of the compiler. With tools like **Lex**, automating the token recognition process becomes easier, improving productivity and reliability in compiler design.

Syntax Analysis in Compiler Design

Syntax Analysis, also known as **parsing**, is the second phase of a compiler, following lexical analysis. The goal of this phase is to take the tokens generated by the lexical analyzer and determine their grammatical structure using the language's grammar rules. The output of the syntax analysis phase is typically a **parse tree** or an **abstract syntax tree (AST)**, which represents the hierarchical structure of the source code.

Role of Parser

The parser is responsible for verifying whether the sequence of tokens conforms to the grammatical rules of the programming language. It performs the following key tasks:

1. **Grammatical Check**: The parser ensures that the tokens generated by the lexical analyzer form a valid syntactic structure according to the grammar rules of the language.
2. **Parse Tree Generation**: It builds a parse tree or AST that represents the hierarchical structure of the program.
3. **Error Reporting**: If there is any deviation from the grammar, the parser reports syntax errors.
4. **Interface with Semantic Analyzer**: The output from the parser is passed to the next phase, the semantic analyzer, which ensures that the meaning of the code is valid.

Challenges of Parsing

There are several challenges that a parser needs to handle effectively, including **ambiguous grammar**, **left recursion**, and various parsing strategies.

1. Ambiguous Grammar

An **ambiguous grammar** is a grammar where a string can have more than one valid parse tree. This can lead to confusion during parsing. Ambiguous grammars are problematic because they do not provide a single, clear structure for sentences, leading to different interpretations of the code.

Example of ambiguous grammar:

$E \rightarrow E + E$

$E \rightarrow id$

The expression id + id + id can have two parse trees, one grouping the first two id terms and the other grouping the last two.

To resolve ambiguities, grammar can be modified, or disambiguation techniques like operator precedence and associativity rules are applied.

2. Left Recursion

Left recursion occurs when a non-terminal in a grammar refers to itself as the first symbol in its production rule. This causes issues in some parsing methods, particularly **top-down parsers**, leading to infinite recursion.

Example:

A → Aα | β

Here, the non-terminal A is left-recursive because it appears on the left side of its own production rule. To remove left recursion, the grammar is rewritten in a non-left-recursive form.

Top-Down Parsing

Top-down parsing attempts to construct a parse tree starting from the root and working down to the leaves. It tries to match the input tokens to the productions of the grammar in a top-to-bottom, left-to-right manner.

1. Recursive Descent Parser

A **recursive descent parser** is a type of top-down parser that uses a set of recursive procedures to process the input. Each non-terminal in the grammar is implemented as a procedure, and parsing decisions are made based on the current input token.

- **Advantages**: Easy to implement, especially for small grammars.
- **Disadvantages**: Cannot handle left recursion directly and may need backtracking, making it inefficient for complex grammars.

Example: Consider the following grammar for arithmetic expressions:

E → T + E | T

T → int | int * T

The recursive descent parser would have procedures to handle each non-terminal (E and T) and recursively call these procedures.

2. Predictive Parser

A **predictive parser** is a special type of recursive descent parser that does not require backtracking. It uses **lookahead** (usually one token) to make parsing decisions, making it more efficient than general recursive descent parsers.

To ensure that a grammar is suitable for predictive parsing, it must be **LL(1)**—meaning that it can be parsed with a single token of lookahead without ambiguity.

Predictive parsers use **predictive parsing tables**, where each table entry tells the parser which production rule to apply based on the current input symbol and non-terminal.

Bottom-Up Parsing

Bottom-up parsing starts from the leaves of the parse tree (i.e., the input tokens) and works up toward the root. It attempts to reduce a sequence of tokens into a grammar's start symbol by applying production rules in reverse.

1. Operator Precedence Parser

An **operator precedence parser** is a simple bottom-up parser that can handle a limited class of grammars, particularly those with well-defined operator precedence. The idea is to define precedence and associativity rules to decide how to reduce tokens when an operator is encountered.

For example:

- * and / have higher precedence than + and -.
- Left-to-right associativity means that an expression like a - b - c is parsed as (a - b) - c.

2. Simple LR (SLR) Parser

An **SLR (Simple LR)** parser is a type of **LR parser** (a bottom-up parser) that uses a **canonical collection of LR(0) items** and a **parsing table** to decide how to reduce tokens into grammar symbols. It is simpler to implement than more advanced LR parsers, but it cannot handle all types of grammars, particularly those with conflicts in the parsing table.

3. Canonical LR(1) Parser

A **canonical LR(1) parser** is a more powerful type of LR parser that uses **lookahead** (1 token) to resolve parsing decisions. It builds a larger parsing table than SLR, but it can handle more complex grammars, especially those that would cause conflicts in SLR parsing.

- **LR(1) grammars** are more expressive and can handle left-recursive and ambiguous grammars better than SLR.

4. Look-Ahead LR (LALR) Parser

An **LALR parser** (Look-Ahead LR) is an optimization of the canonical LR(1) parser. It combines states with similar LR(0) items, resulting in a smaller parsing table with fewer states while retaining most of the parsing power of canonical LR(1). Most modern parsers, such as **YACC**, use the LALR algorithm.

YACC (Yet Another Compiler Compiler)

YACC is a tool used to generate a **parser** from a given grammar specification. It is widely used in combination with **Lex** for building the front end of compilers. YACC takes a **context-free grammar (CFG)** as input and generates a **LALR parser** for that grammar.

Steps to Use YACC:

1. **Define Grammar**: Write a YACC specification that includes grammar rules and associated actions.
2. **Generate Parser**: YACC generates C code for the parser.
3. **Compile and Link**: The parser code is compiled and linked with the rest of the compiler.

YACC Structure:

YACC programs are divided into three sections:

- **Definitions**: Definitions of tokens and data types.
- **Rules**: Grammar rules with actions (e.g., building a parse tree).
- **Auxiliary Code**: Additional C code for integration.

Example:

```
%token NUMBER

%%

expr : expr '+' term { $$ = $1 + $3; }

    | term;

term : NUMBER;

%%
```

Conclusion

Syntax analysis is a crucial part of the compiler, responsible for verifying the structure of source code and constructing a parse tree or AST. Different parsing techniques, such as **top-down** and **bottom-up parsing**, have their own advantages and limitations. Tools like **YACC** and **Lex** simplify the construction of parsers, allowing for efficient handling of complex grammars.

Intermediate Code Generation

Intermediate Code Generation is a key phase in the compiler design process that translates the high-level source code into an intermediate form that is easier to optimize and translate into target machine code. The intermediate representation (IR) is independent of the target machine, making it easier to perform optimizations and code generation.

Syntax Directed Definitions (SDD)

Syntax Directed Definitions (SDDs) define how the semantic actions (related to the meaning or translation of code) are associated with the grammar productions. These actions are driven by **syntax-directed translation** (SDT) rules.

In an SDD:

- Each grammar symbol is associated with attributes.
- Semantic rules define how these attributes are computed.

There are two types of attributes:

1. **Synthesized Attributes**: Computed from the children of a node in a parse tree.
2. **Inherited Attributes**: Computed from the parent or siblings of a node.

Example: For the grammar rule E → E1 + T, an SDD might look like this:

E.val = E1.val + T.val

Here, E.val, E1.val, and T.val are synthesized attributes representing the value of expressions.

Evaluation Orders for Syntax Directed Definitions

There are two common orders for evaluating the semantic rules of SDDs:

1. **Dependency Graphs**: Used to determine the order in which attributes should be evaluated. Each attribute is represented as a node, and an edge is drawn if one attribute depends on another.
   - **Acyclic Dependency Graph**: Ensures that attributes can be evaluated in a sequence without circular dependencies.
2. **Postorder Traversal**: For synthesized attributes, the evaluation order is typically **postorder traversal** of the parse tree (evaluating child nodes before the parent).

Syntax Directed Translation (SDT)

Syntax Directed Translation (SDT) extends the concept of SDDs by adding actions to the grammar rules. These actions are performed as the parser processes the input, often generating intermediate code or performing transformations.

Example of SDT: For the production E → E1 + T, if the action is to generate code:

E → E1 + T { E.code = E1.code || T.code || "add" }

Here, || denotes string concatenation, and the action generates code to add the results of E1 and T.

Intermediate Language

The **intermediate language** (IL) is a representation of the source code that lies between the high-level language and machine code. It is easier to manipulate and optimize. Common intermediate forms include:

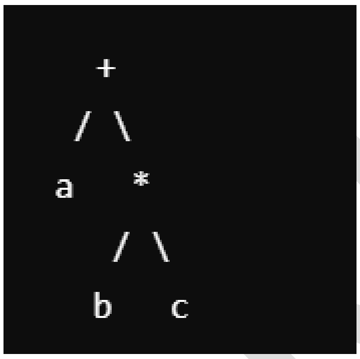
1. **Abstract Syntax Trees (AST)**: Represent the hierarchical structure of the program without unnecessary details like punctuation.
2. **Three-Address Code (TAC)**: A form where each instruction has at most three operands (two operands and one result).
3. **Control Flow Graphs (CFG)**: Represent the flow of control between different blocks of code (basic blocks).

Intermediate code is often machine-independent, which facilitates portability across different architectures.

Syntax Tree

A **syntax tree** (or **abstract syntax tree**, AST) is a simplified representation of the parse tree that captures the essential structure of the source code without redundant details. Each internal node represents an operation, and the leaf nodes represent operands.

For example, the arithmetic expression a + b * c can be represented as:



```
    +
   / \
  a   *
     / \
    b   c
```

Syntax trees are useful for **semantic analysis** and **code generation** as they abstract away low-level syntactic details, focusing on the structure of operations and expressions.

Three-Address Code (TAC)

Three-Address Code (TAC) is a popular intermediate representation that breaks down complex expressions into simple instructions where each instruction involves at most three operands. The three-address code can be thought of as a sequence of simple operations.

Example: For the expression a = b + c * d, the corresponding TAC might be:

t1 = c * d

t2 = b + t1

a = t2

Here, t1 and t2 are temporary variables introduced to store intermediate results.

TAC is widely used because:

1. It makes **optimization** easier by breaking down expressions into simple steps.
2. It can be easily translated into **assembly code**.

Types and Declarations

During intermediate code generation, the **types** of variables and expressions need to be checked to ensure correctness. **Declarations** provide the information about variable types, sizes, and memory allocation requirements.

- **Type Declarations**: Describe the type of variables (e.g., int, float, char).
- **Type Compatibility**: Ensures that operations are performed on compatible types.
- **Symbol Table**: Stores information about variables, types, and scope, aiding in type checking and code generation.

Example: In a declaration like int x;, the intermediate code must allocate memory for the variable x and ensure it is treated as an integer in subsequent expressions.

Translation of Expressions

When translating expressions into intermediate code, the compiler needs to consider the type of operations and how to evaluate the expression efficiently. This involves generating intermediate code for:

- **Arithmetic expressions**: Use TAC to break down into simple steps.
- **Logical expressions**: Translate into jump instructions for control flow (e.g., if-else conditions).
- **Relational expressions**: Convert relational operators like <, >, and == into comparisons that control jumps in the code.

Example:
For the expression a = b + c * d, the intermediate code generation might look like:

t1 = c * d

t2 = b + t1

a = t2

Type Checking

Type Checking is a critical part of both semantic analysis and intermediate code generation. The compiler ensures that:

- Operations are performed on compatible types.
- Type conversions (casting) are handled correctly.
- Array bounds are respected (where applicable).

Type checking can be done at two stages:

1. **Static Type Checking**: Done during compilation. It ensures that type rules are followed without running the program.
2. **Dynamic Type Checking**: Done during runtime (not common in statically typed languages).

Control Flow

Control flow in intermediate code generation involves representing:

- **Loops** (for, while): Translate into basic blocks with jump statements.
- **Conditionals** (if, else): Translate into comparisons followed by conditional jumps.

The **Control Flow Graph (CFG)** is a representation of the program where each node is a basic block of code, and edges represent the flow of control between blocks. It is used in optimization and code generation.

Backpatching

Backpatching is a technique used to handle **forward jumps** in intermediate code, such as those required for conditional or loop statements, where the target of the jump is not known until later in the code generation process.

Steps in backpatching:

1. Generate jump instructions with incomplete targets (holes).
2. Once the target is determined, go back and fill in the correct target address.

Example (for if-else):

if (a < b) goto L1

...

goto L2

L1: ...

```
L2: ...
```

Here, goto L1 and goto L2 might need backpatching as the exact locations of L1 and L2 are not known when the jumps are first generated.

Switch Statements

The translation of **switch statements** involves generating a sequence of conditional jumps. The general strategy is to:

1. Evaluate the switch expression.
2. Compare the result with each case value.
3. Jump to the corresponding block of code based on the comparison result.

Intermediate code for a switch statement might involve **jump tables** or **if-else chains**.

Example: For a switch statement:

```
switch (x) {
  case 1: statement1;
        break;
  case 2: statement2;
        break;
  default: statement3;
}
```

The corresponding intermediate code might look like:

```
if x == 1 goto L1
if x == 2 goto L2
goto L3 (default case)
L1: ... (code for case 1)
goto L_end
L2: ... (code for case 2)
goto L_end
L3: ... (code for default)
L_end: ...
```

Conclusion

Intermediate code generation is a crucial step in the compilation process, providing a machine-independent representation of the program. Concepts like **Syntax Directed Definitions (SDD)**, **Syntax Directed Translation (SDT)**, **three-address code (TAC)**, and **backpatching** play a key role in translating high-level language constructs into an intermediate form, which can then be optimized and further translated into machine code.

Run-Time Environment and Code Generation

The **run-time environment** is the environment in which a program executes, and it defines how a program's variables, functions, and control flow are managed at runtime. **Code generation** is the phase in a compiler where intermediate code is transformed into target machine code. This phase includes addressing storage organization, memory allocation, and instruction generation.

Storage Organization

During program execution, various types of data such as variables, function parameters, and return addresses need to be stored. The **run-time storage organization** is divided into the following areas:

1. **Static Memory**: This holds global variables, constants, and static local variables. The size is known at compile time and remains fixed throughout the execution of the program.
2. **Stack Memory**: This is used for dynamic memory management, particularly for storing local variables, function parameters, and return addresses. It grows and shrinks as functions are called and returned.
3. **Heap Memory**: Used for dynamically allocated memory (e.g., memory allocated using malloc in C or new in Java). It grows as needed during runtime.

Memory Layout:

```
------------------------------
|       Code Segment         |  (stores the program's machine code)
------------------------------
|    Static Data Segment     |  (global/static variables)
------------------------------
|           Heap             |  (dynamic memory allocation)
|      (grows upwards)       |
------------------------------
|           Stack            |  (function calls and local variables)
|      (grows downwards)     |
------------------------------
```

Stack Allocation Space

The **stack** is a key part of the run-time environment and is used for **function calls** and **local variables**. When a function is called, a **stack frame** (or activation record) is created and pushed onto the stack. When the function returns, its stack frame is popped off the stack.

A stack frame typically contains:

1. **Return Address**: The instruction to return to after the function call.
2. **Function Parameters**: The arguments passed to the function.
3. **Local Variables**: Variables declared within the function.
4. **Saved Registers**: Registers that need to be restored after the function returns.
5. **Control Link**: A pointer to the previous stack frame (the caller's frame).

Example:

```
int factorial(int n) {
   if (n == 0) return 1;
   return n * factorial(n - 1);
}
```

When factorial(3) is called, a new stack frame is created for each call, storing n, the return address, and other necessary information.

Access to Non-Local Data on the Stack

Non-local data refers to variables that are not local to the current function but need to be accessed. In programming languages with nested functions (like in Pascal or some functional languages), a function may need to access variables from an outer function (non-local variables).

There are two common strategies for accessing non-local data:

1. **Static Links**: A **static link** is a pointer to the stack frame of the function that is lexically enclosing the current function. This allows access to non-local variables by following the chain of static links.
2. **Display**: A **display** is an array of pointers to the stack frames of all active functions. The array index corresponds to the depth of the function in the call chain, and each entry points to the stack frame of the corresponding function.

These strategies allow nested functions to access variables that are not local to the current function but are defined in an enclosing scope.

Heap Management

The **heap** is used for dynamic memory allocation at runtime. Heap memory is allocated when needed and freed when no longer required. Managing the heap involves:

1. **Memory Allocation**: Functions like malloc() in C or new in Java allocate a block of memory from the heap. The size of the allocation is determined at runtime.
2. **Memory Deallocation**: When a block of memory is no longer needed, it is returned to the heap (e.g., using free() in C or automatic garbage collection in Java).

Heap management must handle fragmentation and ensure efficient memory allocation.

Garbage Collection

Garbage collection is an automatic process that reclaims memory that is no longer used by the program, preventing memory leaks.

Common garbage collection algorithms include:

1. **Reference Counting**: Each object has a counter that tracks the number of references to it. When the reference count reaches zero, the object is deallocated.
   - **Advantage**: Simple to implement.
   - **Disadvantage**: Cannot handle cyclic references.
2. **Mark-and-Sweep**: In this approach, the garbage collector marks all objects that are reachable from the root (starting point) and sweeps (deallocates) all unmarked objects.

- **Advantage**: Handles cyclic references.
- **Disadvantage**: Requires a full scan of memory, leading to pause times.

3. **Generational Garbage Collection**: The heap is divided into generations based on the age of objects. Younger objects are collected more frequently, as most objects tend to die young.
   - **Advantage**: Efficient for typical usage patterns.
   - **Disadvantage**: More complex to implement.

Issues in Code Generation

Code generation converts the intermediate representation of a program into machine code or assembly code. Several issues must be considered:

1. **Instruction Selection**: The code generator must choose the appropriate machine instructions for each intermediate operation.
   - **Example**: An addition operation in the intermediate code might be translated to an ADD instruction in assembly.
2. **Register Allocation**: Efficiently allocating **registers** is critical for performance. There are typically fewer registers than variables, so the compiler must manage which variables are stored in registers at any given time.
   - **Register spilling**: When there are not enough registers, some variables must be spilled to memory, which can degrade performance.
3. **Instruction Scheduling**: Instructions may need to be reordered to avoid **pipeline hazards** or improve performance on modern CPUs. For example, delaying memory access instructions while waiting for a cache to load data.
4. **Handling Control Flow**: Translating control flow structures such as **if-else**, **loops**, and **function calls** involves generating conditional branches, jump instructions, and return instructions.

Design of a Simple Code Generator

A simple code generator translates intermediate code (such as **three-address code (TAC)**) into machine code or assembly code. The steps involved are:

1. **Input Intermediate Code**: The code generator takes intermediate code as input, which is typically machine-independent (e.g., three-address code).
2. **Register Allocation**: The code generator assigns variables to registers. When there are not enough registers, it spills variables to memory.

3. **Instruction Selection**: Each intermediate code operation is translated into one or more machine instructions. For example:
   - Intermediate code: t1 = a + b
   - Target assembly: ADD R1, R2, R3
4. **Handling Memory Access**: Variables that cannot fit in registers are stored in memory, and the code generator must insert instructions to load and store these variables as needed.
5. **Control Flow Translation**: The code generator translates control flow statements into **branch** and **jump** instructions. For example, an if statement in intermediate code might generate a CMP (compare) instruction followed by a conditional JMP.
6. **Function Call Translation**: Function calls are translated into instructions that push arguments onto the stack, set up the return address, and transfer control to the callee.

Example of a Simple Code Generator

Consider the intermediate code for an assignment statement:

x = y + z

The simple code generator might translate this into the following machine instructions:

```
MOV R1, y      ; Load y into register R1
MOV R2, z      ; Load z into register R2
ADD R1, R1, R2 ; Add y and z, store in R1
MOV x, R1      ; Store result in x
```

If y and z are already in registers, the code generator might skip the MOV instructions, optimizing the code.

Conclusion

The **run-time environment** provides the necessary infrastructure for memory management and control flow during program execution. The **stack** and **heap** are used for different types of memory allocation, with **garbage collection** ensuring that memory is efficiently managed. The **code generation** phase of the compiler involves several challenges, including instruction selection, register allocation, and handling control flow. By addressing these issues, the compiler produces efficient machine code that can be executed by the target processor.