What is Intermediate Code Generation?

Imagine you're building a house. Before you can start building, you need a blueprint, right? Similarly, in the process of translating a high-level programming language (like Python or C++) into machine code that a computer can execute, we need a blueprint too. This is where intermediate code generation comes into play.

Intermediate code generation is like creating a rough sketch or blueprint of the program, called intermediate code, which captures the essential structure and operations of the program. It happens after the program's meaning and structure have been analyzed (semantic phase) but before the final machine code is generated.

Benefits of Intermediate Code:

1. Machine Independence:

Think of intermediate code as a universal language understood by different types of computers. Just like how architects use a common blueprint language regardless of the construction site, compilers can generate intermediate code that's not tied to any specific computer architecture. This makes it easier to adapt the compiler for different processors or platforms.

2. Proximity to Target Machine:

While high-level source code is human-readable and understandable, it's quite far from the language that computers understand directly. Intermediate code, however, is closer to the low-level instructions that computers can execute. This closeness makes it simpler to translate into the final machine code that the computer executes.

3. Optimization Opportunities:

Intermediate code provides a middle ground where optimization techniques can be applied effectively. Just like how an architect might refine a blueprint to make the house more efficient or cost-effective, compilers can optimize intermediate code to improve program performance or reduce memory usage. These optimizations are often independent of the specific machine, making them more widely applicable.

4. Integration with Parsing:

Intermediate code generation is often seamlessly integrated into the parsing process. This means that as the compiler parses the source code (like understanding the sentences in a language), it simultaneously generates the intermediate code (like drawing the blueprint). This integration streamlines the compilation process and makes it more efficient.

Related posts:

1. What are the types of passes in compiler ?
2. Discuss the role of compiler writing tools. Describe various compiler writing tools.
3. What do you mean by regular expression ? Write the formal recursive definition of a regular expression.
4. How does finite automata useful for lexical analysis ?
5. Explain the implementation of lexical analyzer.
6. Write short notes on lexical analyzer generator.
7. Explain the automatic generation of lexical analyzer.
8. Explain the term token, lexeme and pattern.
9. What are the various LEX actions that are used in LEX programming ?
10. Describe grammar.

11. Explain formal grammar and its application to syntax analyzer.
12. Define parse tree. What are the conditions for constructing a parse tree from a CFG ?
13. Describe the capabilities of CFG.
14. What is parser ? Write the role of parser. What are the most popular parsing techniques ? OR Explain about basic parsing techniques. What is top-down parsing ? Explain in detail.
15. What are the common conflicts that can be encountered in shift-reduce parser ?
16. Differentiate between top-down and bottom-up parser.Under which conditions predictive parsing can be constructed for a grammar ?
17. Differentiate between recursive descent parsing and predictive parsing.
18. What is the difference between S-attributed and L-attributed definitions ?
19. Define parse tree. Why parse tree construction is only possible for CFG ?
20. Discuss symbol table with its capabilities ?
21. What are the symbol table requirements ? What are the demerits in the uniform structure of symbol table ?