SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

Building Compiler For Functional Programming Language

Guide by
Dr.G.MICHAEL

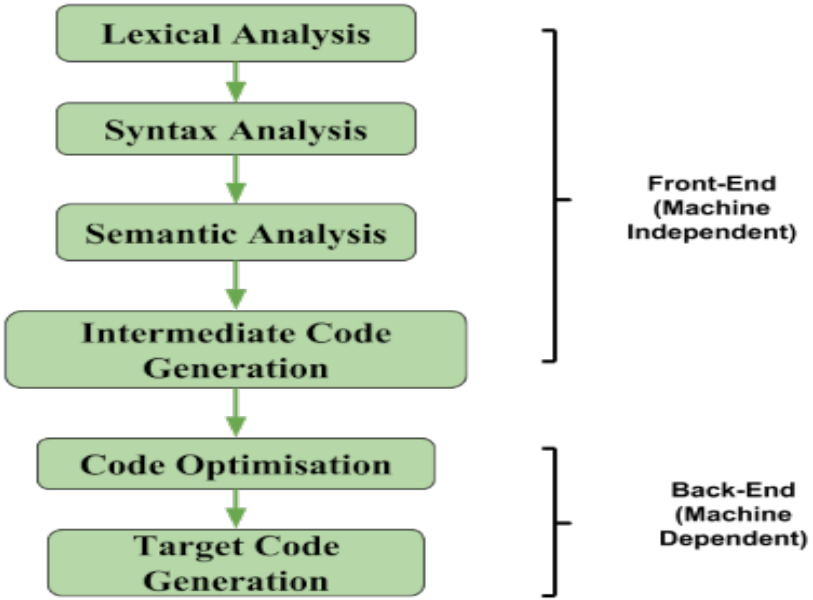
BY
V. Mano Karthik (192225077)
D. Teja(192210626)
G. Ajay Kumar(192210590)

Introduction to Building Compiler for Functional Programming Language

Compilers play a crucial role in translating high-level programming languages into machine code.

Functional programming languages emphasize the use of pure functions and immutable data.

Building a compiler for a functional programming language involves parsing, semantic analysis, optimization, and code generation.

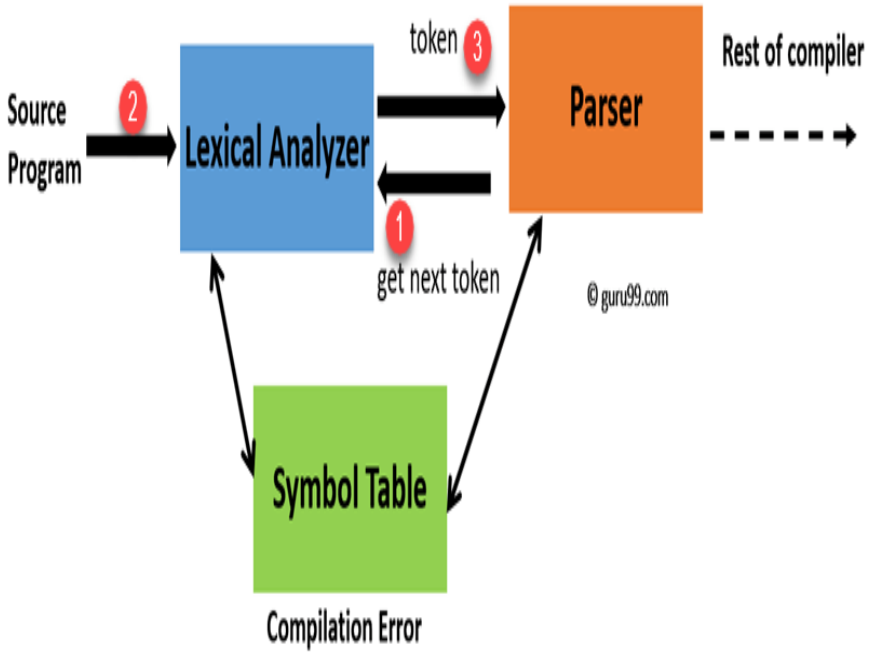


Lexical Analysis in Compiler Design

Lexical analysis involves breaking the source code into tokens such as keywords, identifiers, and symbols.

Tokenization is an essential step in the compilation process to simplify parsing.

Regular expressions and finite automata are commonly used techniques in lexical analysis.

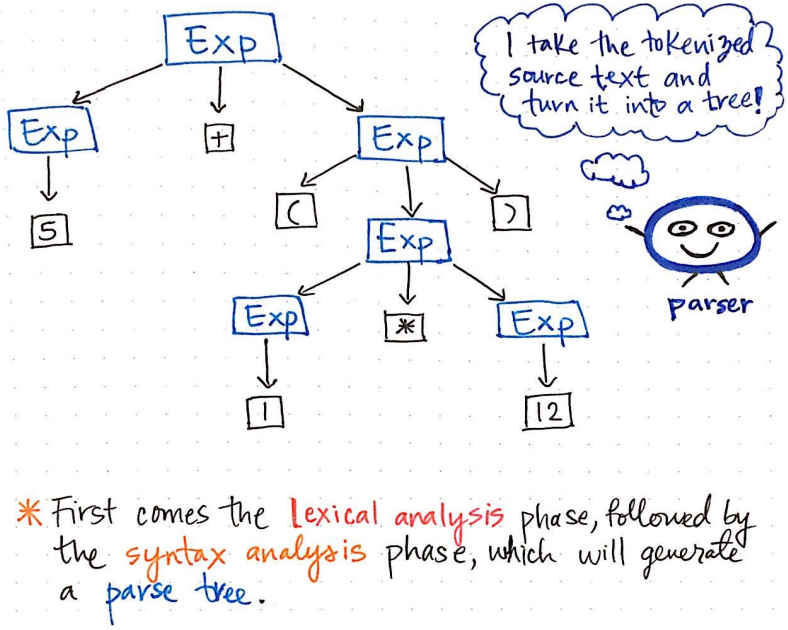


Parsing and Syntax Analysis

Parsing involves analyzing the syntax of the source code to build a parse tree or abstract syntax tree (AST).

Context-free grammars and parser generators like YACC and ANTLR are used for parsing.

Syntax analysis ensures that the source code adheres to the grammar rules of the programming language.

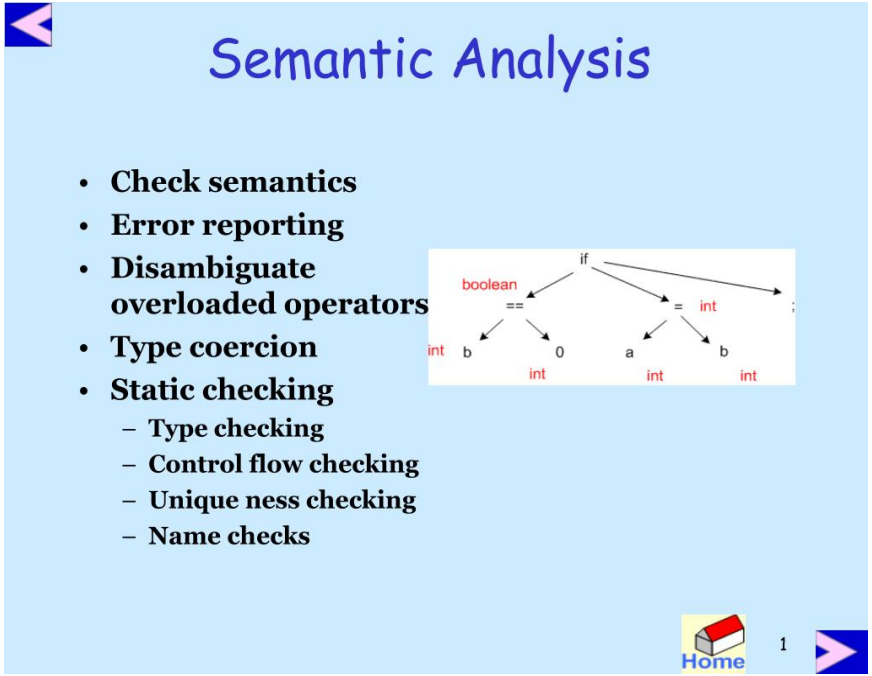


Semantic Analysis and Type Checking

Semantic analysis checks the meaning and correctness of the source code beyond its syntax.

Type checking ensures that variables are used in a manner consistent with their defined types.

Functional programming languages often have strong type systems that require rigorous type checking during compilation.

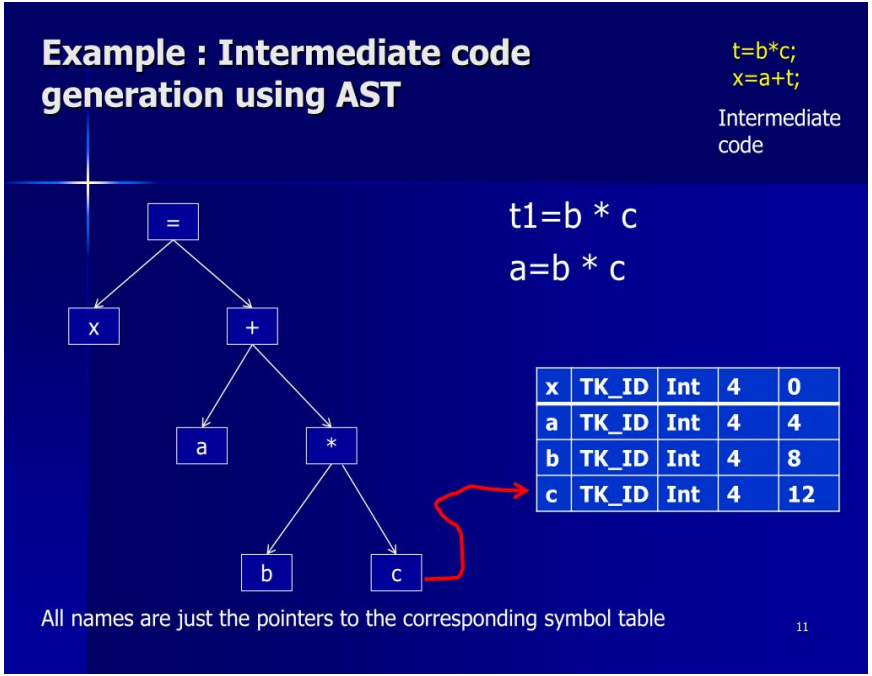


Intermediate Code Generation

Intermediate code generation involves transforming the AST into an intermediate representation (IR) for optimization.

Common IR forms include three-address code, static single assignment (SSA) form, and abstract stack machines.

IR facilitates various optimizations before generating the target code.

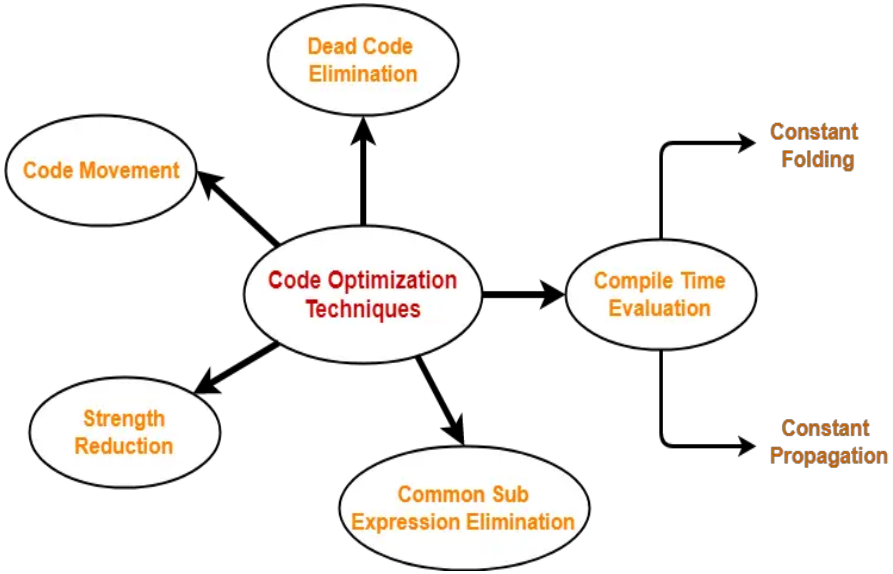


Optimization Techniques in Compiler Design

Optimization aims to improve the efficiency and performance of the generated code.

Common optimization techniques include constant folding, loop optimization, and inlining.

Functional programming languages benefit from optimizations that leverage immutability and higher-order functions.

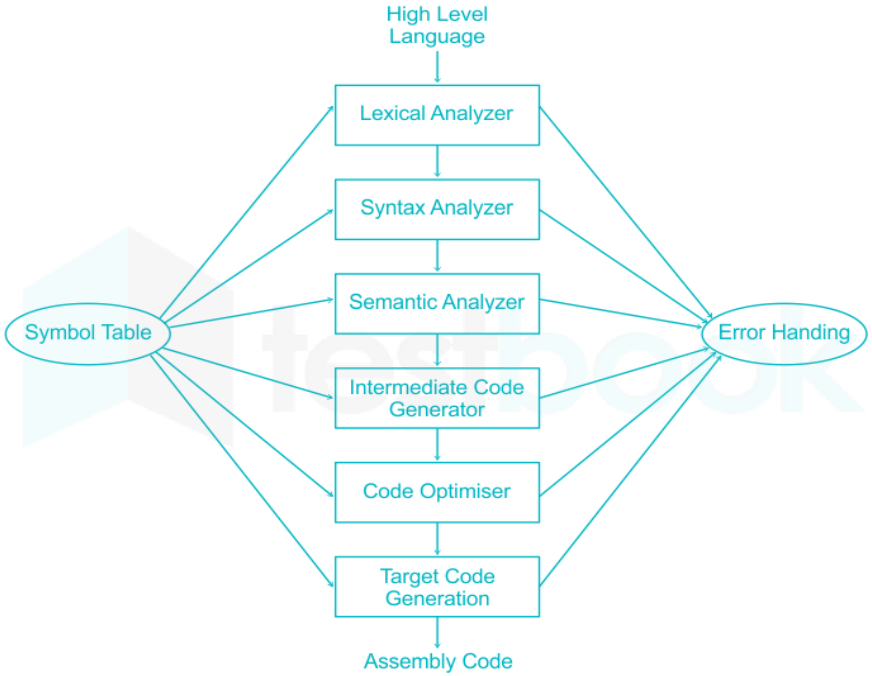


Code Generation for Functional Programming Languages

Code generation translates the optimized IR into machine code specific to the target platform.

Functional programming languages may target different execution models such as the stack-based or register-based model.

Tail call optimization is a key technique for efficient code generation in functional programming languages.

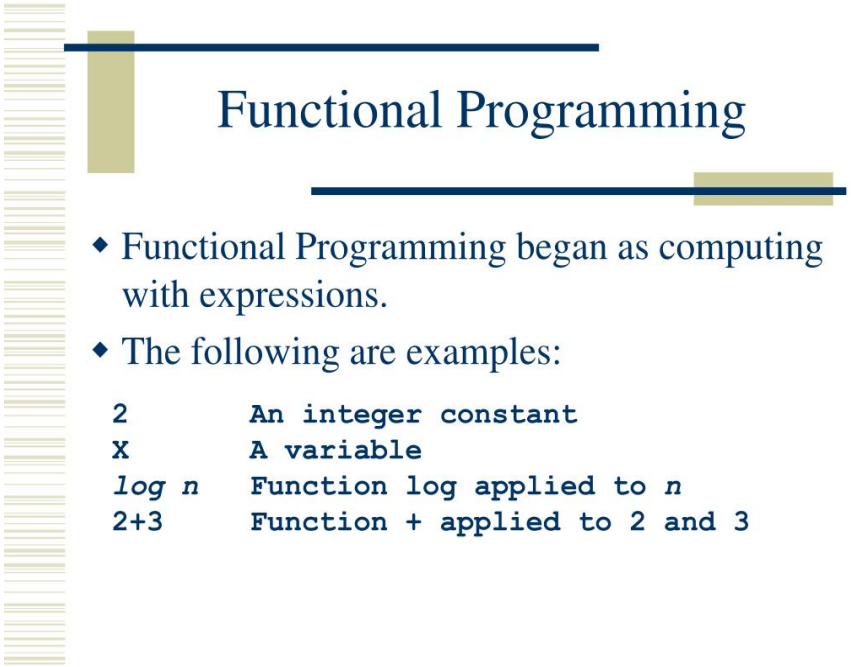


Handling Recursion in Functional Programming Language Compilation

Recursion is a fundamental feature in functional programming languages like Haskell and Lisp.

Compiler optimization techniques such as tail call optimization help manage recursion efficiently.

Proper handling of recursion is crucial for maintaining performance and stack space usage.

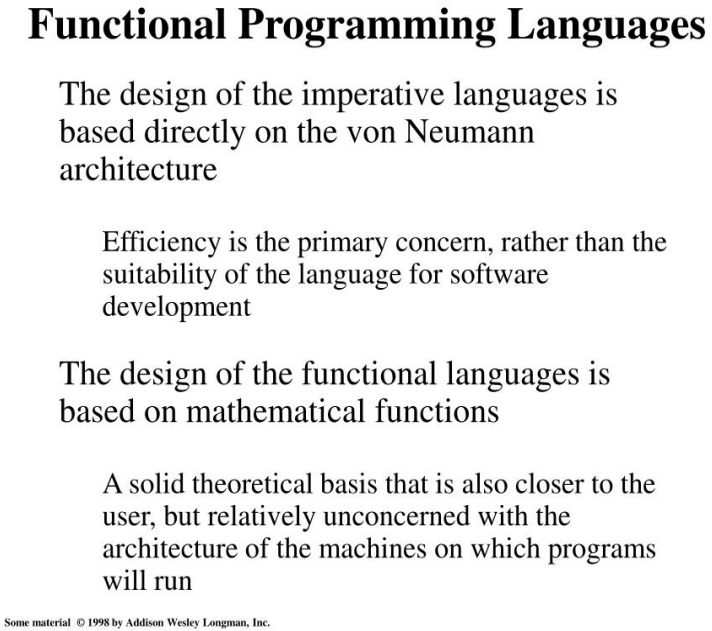


Challenges and Considerations in Building a Compiler for Functional Programming Language

Functional programming languages introduce unique challenges due to their emphasis on immutability and higher-order functions.

Handling lazy evaluation, purity, and type inference can be complex in compiler design.

Balancing between expressive language features and efficient compilation is a critical consideration.



Functional Programming Languages

The design of the imperative languages is based directly on the von Neumann architecture

Efficiency is the primary concern, rather than the suitability of the language for software development

The design of the functional languages is based on mathematical functions

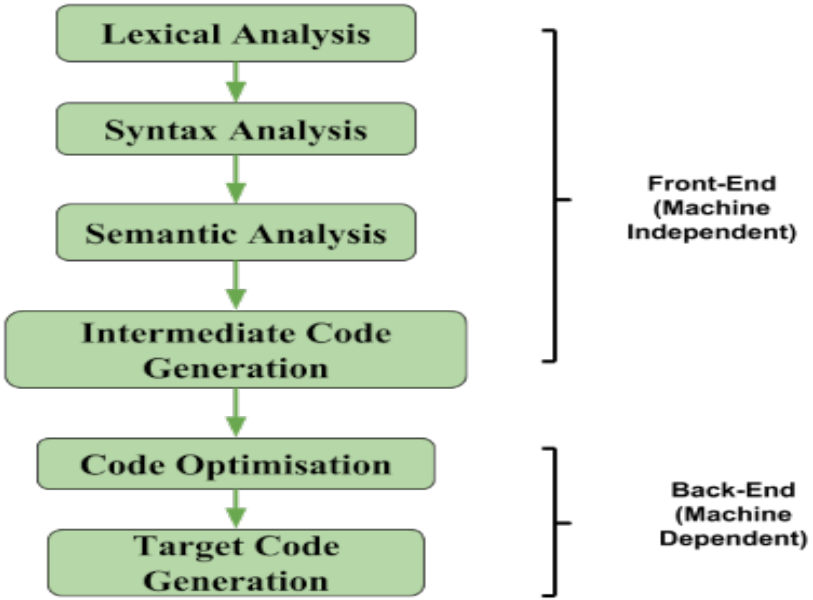
A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Conclusion and Future Directions

Building a compiler for a functional programming language requires a deep understanding of language semantics and compilation techniques.

Continuous research is focused on optimizing compilers for functional programming languages to improve performance.

The evolution of compiler technology will continue to shape the efficiency and scalability of functional programming languages.



References

- Aho, Alfred V., et al. "Compilers: Principles, Techniques, and Tools." Pearson Education, 2006.

- Appel, Andrew W. "Modern Compiler Implementation in ML." Cambridge University Press, 1998.

- Jones, Simon Peyton. "The Implementation of Functional Programming Languages." Prentice Hall, 1987.