🌂

CMP 407 Mr Barka’s presentation

ROLES

- Aromolaran Adenike Elizabeth - `Frontend` : 1, 2 & 3
- Chukwu Daniel Nonso - `Frontend` : 4 & 5
- Anointing Edube Dauda - `Middle End`: 1
- Daniel e Ojiawuna - `Backend` : 1 & 2
- Jolly D Joseph - `Backend`: 3 & 4

TABLE OF CONTENT

- Compilation Phases Explanation
- Compilation phases using a program `a = 10 + 5`

NOTE

- Go through ur specific roles on both sections in the TABLE OF CONTENT
- Also just incase Mr. Barka tries to be spontaneous, I would advise we all go through everything and understand all the phases

Haskell Programming Language - `Compilation Phases Explanation`

> ## Front End:
>
> 1. **Lexical Analysis (or the lexer):**

- Breaks the source code into tokens. Tokens are the smallest units of meaning in the language (i.e. keywords, identifiers, literals, and operators).

2. **Syntax Analysis (or the parser):**
    - The parser takes the tokens generated by the lexical analyzer and organizes them into a hierarchical structure called the Abstract Syntax Tree (AST). The AST represents the syntactic structure of the program.
    - In summary, it analyzes the tokens to create an AST, which represents the program's structure.
3. **Renaming:**
    - Assigns unique names to variables and expressions to avoid conflicts.
4. **Type Checking:**
    - Haskell is a statically-typed language, meaning that type information is checked at compile-time. This phase involves checking if the types used in the program are consistent and inferring types for expressions that don't have explicit type annotations.
    - In summary, it verifies that the program adheres to Haskell's type system, ensuring type safety
5. **Desugaring:**
    - Haskell supports a range of syntactic sugar, which is more convenient syntax that is translated into the core language.

The desugaring phase involves converting these sugar-coated expressions into their simpler, core equivalents.

- In summary, it transforms syntactic sugar (e.g., do notation, list comprehensions) into simpler core language constructs.

Middle End:

1. **Optimization:**

    - Applies various transformations to improve the code's efficiency and performance, including:
        - Demand analysis (generalization of strictness analysis)
        - Unfolding (inlining)
        - Let-floating
        - Unboxing
        - Constructed product result analysis
        - Specialization of overloaded functions
        - Constant folding
        - Beta reduction

Back End:

1. **STG Machine:**

    - Translates Core code into STG (Spineless Tagless G-machine), an intermediate language designed for efficient graph reduction.

2. **Code Generation:**

- Produces C-- code (an internal representation) from STG.
- **C--** (pronounced *C minus minus*) is a C-like programming language, designed to be generated mainly by compilers for high-level languages rather than written by human programmers.

3. **Backends:**
   - C-- code can be:
     - Printed as C code for compilation with GCC
     - Converted directly into native machine code
     - Converted to LLVM IR for compilation with LLVM
4. **Linking:**
   - If the program is composed of multiple modules, the linker combines them into a single executable or library. This phase resolves references between different parts of the code.
   - In summary, it combines generated code with the GHC runtime system (RTS) to create an executable.

Key Points:

- The GHC compiler is highly optimizing, aiming to produce efficient code.
- The STG language is a crucial component for managing Haskell's lazy evaluation model.

> - The RTS provides essential runtime services, such as garbage collection and memory management.
> - Understanding these phases can aid in debugging, performance optimization, and exploring compiler internals.

Haskell Programming Language - `Compilation Phases using below program`

a = 10 + 5

> **Front End:**
>
> 1. **Lexical Analysis:**
>   - The code is broken into tokens: `a`, `=`, `10`, `+`, `5`.
> 2. **Parsing:**
>   - Tokens are arranged into an AST:
>   `Assign (Variable "a") (BinaryOp "+" (Number 10) (Number 5))`
> 3. **Renaming:**
>   - No renaming needed as there are no conflicts.
> 4. **Type Checking:**
>   - Types are inferred: `a` is `Integer`.
> 5. **Desugaring:**
>   - No desugaring needed as there's no syntactic sugar.
>
> **Middle End:**

> 1. **Optimization:**
>
>   - Potential optimizations:
>   - Constant folding: `10 + 5` might be evaluated to `15` at compile time.
>
> **Back End:**
>
> 1. **STG Machine:**
>
>   - Core code is translated to STG for graph reduction.
>
> 2. **Code Generation:**
>
>   - C-- code is produced, representing machine instructions.
>
> 3. **Backends:**
>
>   - C-- code is compiled to machine code or LLVM IR.
>
> 4. **Linking:**
>
>   - Generated code is linked with GHC RTS to form an executable.
>
> **Execution:**
>
> 1. **Runtime:**
>
>   - The executable runs, evaluating `a` to `15`.
>   - The final value `15` is associated with the variable `a`.
>
> **Key Points:**
>
> - Optimization might simplify the expression to `a = 15`.
> - The STG machine efficiently handles lazy evaluation.

- The RTS manages memory and garbage collection.
- Understanding these phases aids in debugging, performance optimization, and compiler exploration.