05 - Compiler Construction

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Outline

1. Compiler Construction

2. Compiler Layers

Outline

1 Compiler Construction

2 Compiler Layers

Compiler Phases and Passes

- Three Main Phases:

Compiler Phases and Passes

- Three Main Phases:
    1. Lexical Analysis

Compiler Phases and Passes

- Three Main Phases:
  1. Lexical Analysis
  2. Syntax Analysis

Compiler Phases and Passes

- Three Main Phases:
  1. Lexical Analysis
  2. Syntax Analysis
  3. Code Generation

Compiler Phases and Passes

- Three Main Phases:
  1. Lexical Analysis
  2. Syntax Analysis
  3. Code Generation
- The phases may require more than one pass.

Compiler Phases and Passes

- Three Main Phases:
  1. Lexical Analysis
  2. Syntax Analysis
  3. Code Generation
- The phases may require more than one pass.
- A recursive descent compiler typically requires only one pass.

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.
- Function calls are mutually recursive.

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.
- Function calls are mutually recursive.
- That is, functions call each other as they parse the program.

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.
- Function calls are mutually recursive.
- That is, functions call each other as they parse the program.
- For example: `while <clause> do <clause>`

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.
- Function calls are mutually recursive.
- That is, functions call each other as they parse the program.
- For example: **`while`** `<clause>` **`do`** `<clause>`
  1. The parser sees the keyword `while`, and so it invokes the `while()` function.

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.
- Function calls are mutually recursive.
- That is, functions call each other as they parse the program.
- For example: **`while`** `<clause>` **`do`** `<clause>`
  1. The parser sees the keyword `while`, and so it invokes the `while()` function.
  2. `while` then calls the `clause()` function.

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.
- Function calls are mutually recursive.
- That is, functions call each other as they parse the program.
- For example: **`while`** `<clause>` **`do`** `<clause>`
  1. The parser sees the keyword `while`, and so it invokes the `while()` function.
  2. `while` then calls the `clause()` function.
  3. Once `clause()` returns, `while` checks to see if there is a `do` keyword.

Recursive Descent Compiler Design

- Each non-terminal has a corresponding function.
- Function calls are mutually recursive.
- That is, functions call each other as they parse the program.
- For example: **`while`** `<clause>` **`do`** `<clause>`
    1. The parser sees the keyword `while`, and so it invokes the `while()` function.
    2. `while` then calls the `clause()` function.
    3. Once `clause()` returns, `while` checks to see if there is a `do` keyword.
    4. `while` then calls the `clause()` function once more.

Non-Terminal Production Function Design

- The function checks the next lexical symbol.

Non-Terminal Production Function Design

- The function checks the next lexical symbol.
- Based on the symbol, it then either consumes the symbol or it selects a non-terminal production.

Non-Terminal Production Function Design

- The function checks the next lexical symbol.
- Based on the symbol, it then either consumes the symbol or it selects a non-terminal production.
- Should an unexpected symbol arise, the function should report an error.

Non-Terminal Production Function Design

- The function checks the next lexical symbol.
- Based on the symbol, it then either consumes the symbol or it selects a non-terminal production.
- Should an unexpected symbol arise, the function should report an error.
- Let's try designing the functions for the G grammar! (The $\mathrm{LL}(1)$ variant):

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow TE' \\
E' &\rightarrow \lambda | + TE' \\
T &\rightarrow FT' \\
T' &\rightarrow \lambda | * FT' \\
F &\rightarrow (E) | U \\
U &\rightarrow 0|1|2|3|4|5|6|7|8|9
\end{aligned}
$$

Non-Terminal Production Function Design

- The function checks the next lexical symbol.
- Based on the symbol, it then either consumes the symbol or it selects a non-terminal production.
- Should an unexpected symbol arise, the function should report an error.
- Let's try designing the functions for the *G* grammar! (The $\mathrm{LL}(1)$ variant):

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow TE' \\
E' &\rightarrow \lambda | + TE' \\
T &\rightarrow FT' \\
T' &\rightarrow \lambda | * FT' \\
F &\rightarrow (E) | U \\
U &\rightarrow 0|1|2|3|4|5|6|7|8|9
\end{aligned}
$$

- Let's step through some valid and invalid sentences.

Outline

1 Compiler Construction

2 Compiler Layers

Stepwise Refinement

- The compiler process is as follows:

Stepwise Refinement

- The compiler process is as follows:
  1. Read in the Source

Stepwise Refinement

- The compiler process is as follows:
  1. Read in the Source
  2. Check the Syntax

Stepwise Refinement

- The compiler process is as follows:
  1. Read in the Source
  2. Check the Syntax
  3. Generate the Code

Stepwise Refinement

- The compiler process is as follows:
  1. Read in the Source
  2. Check the Syntax
  3. Generate the Code
- Each phase of compiler design and construction refines these steps, adding more detail as we go.

Stepwise Refinement

- The compiler process is as follows:
  1. Read in the Source
  2. Check the Syntax
  3. Generate the Code
- Each phase of compiler design and construction refines these steps, adding more detail as we go.
- The easiest approach is to treat view the compiler as an ogre (it has layers).

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.
2. Write a lexical analyzer.

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.
2. Write a lexical analyzer.
3. Add the context free error diagnosis and recovery.

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.
2. Write a lexical analyzer.
3. Add the context free error diagnosis and recovery.
4. Add the type checking and type handler.

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.
2. Write a lexical analyzer.
3. Add the context free error diagnosis and recovery.
4. Add the type checking and type handler.
5. Add the environment handler and scope checker.

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.
2. Write a lexical analyzer.
3. Add the context free error diagnosis and recovery.
4. Add the type checking and type handler.
5. Add the environment handler and scope checker.
6. Add the context sensitive error reporting.

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.
2. Write a lexical analyzer.
3. Add the context free error diagnosis and recovery.
4. Add the type checking and type handler.
5. Add the environment handler and scope checker.
6. Add the context sensitive error reporting.
7. Add the data and code address calculation.

Writing a Recursive Descent Compiler

1. Write a pure syntax analyzer.
2. Write a lexical analyzer.
3. Add the context free error diagnosis and recovery.
4. Add the type checking and type handler.
5. Add the environment handler and scope checker.
6. Add the context sensitive error reporting.
7. Add the data and code address calculation.
8. Write the code generation.

Syntax Analysis

- The syntax analyzer is responsible for turning the input into a string of basic symbols.

Syntax Analysis

- The syntax analyzer is responsible for turning the input into a string of basic symbols.
- This part of the compiler must be aware of terminals, and **keywords**.

Syntax Analysis

- The syntax analyzer is responsible for turning the input into a string of basic symbols.
- This part of the compiler must be aware of terminals, and **keywords**.
- A keyword is a fixed terminal string, such as `while`, `if`, etc.

Lexical Analysis

- The lexical analyzer classifies groups of symbols into basic constructs.

Lexical Analysis

- The lexical analyzer classifies groups of symbols into basic constructs.
- This is the phase that identifies literals and keywords.

Lexical Analysis

- The lexical analyzer classifies groups of symbols into basic constructs.
- This is the phase that identifies literals and keywords.
- The lexical analyzer reduces the sentence to a series of symbols over the $N \cup T$ alphabet.

Context Free Error Diagnosis and Recovery

- This phase basically consists of checking for unexpected symbols.

Context Free Error Diagnosis and Recovery

- This phase basically consists of checking for unexpected symbols.
- This is a fairly trivial exercise if we have an $\mathrm{LL}(1)$ language (or one close to it).

Type Checking

- Type checking validates types used in program expressions.

Type Checking

- Type checking validates types used in program expressions.
- Incompatible types generate errors.

Environment and Scope Checking

- This is symbol table checking.

Environment and Scope Checking

- This is symbol table checking.
- Verify that all variables are defined in the scope in which they are used.

Context Sensitive Error Reporting

- These are errors caused by programs which parse, but are meaningless.

Context Sensitive Error Reporting

- These are errors caused by programs which parse, but are meaningless.
- Other examples include duplicate names, and other such non-syntax related errors.

Machine Abstraction and Code Generation

- An abstract machine is used to compute addresses of variables and the like.

Machine Abstraction and Code Generation

- An abstract machine is used to compute addresses of variables and the like.
- This where concepts such as “stack” and “heap” come into play.

Machine Abstraction and Code Generation

- An abstract machine is used to compute addresses of variables and the like.
- This where concepts such as “stack” and “heap” come into play.
- Eventually, the abstract machine definition of the code is mapped to the real machine during code generation.

Machine Abstraction and Code Generation

- An abstract machine is used to compute addresses of variables and the like.
- This where concepts such as “stack” and “heap” come into play.
- Eventually, the abstract machine definition of the code is mapped to the real machine during code generation.
- These final two layers are the only one with any awareness of the underlying computer. Hence they are typically well separated to ensure language portability.

Conclusion

- The process of writing a compiler is about stepwise refinement.

Conclusion

- The process of writing a compiler is about stepwise refinement.
- The layers are inter-related, however we typically can write them through an iterative process.

Conclusion

- The process of writing a compiler is about stepwise refinement.
- The layers are inter-related, however we typically can write them through an iterative process.
- In the coming weeks, we will study how we make each layer work, adding details as we go.