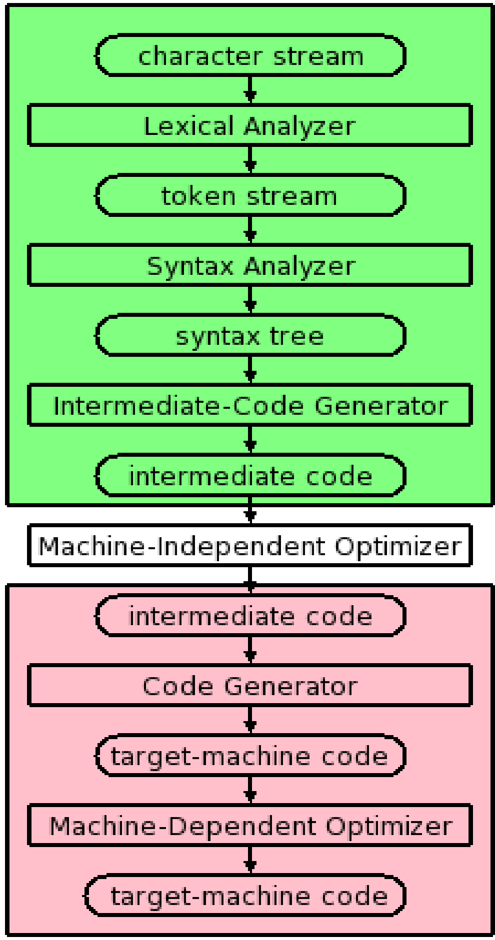
CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- The Structure of a Compiler
- Lexical Analysis (or Scanning) (We use the C/ Pascal like style)
- Syntax analysis (parsing)
- Semantic Analysis
- Intermediate code generation (in Part2)
- Code optimization (in Part2)
- Code generation (in Part2)
- Symbol-Table Management (in Part2)

The Structure of a Compiler



- Modern compilers contain two (large) parts
- Getting closer and look at those boxes!
  - These two parts are the *front end*,

  shown in green on the right and the *back end*,

  shown in pink.
- The front end *analyzes* the source program, determines its constituent parts, and constructs an intermediate representation of the program.
- Typically the front end is independent of the target language.

The Structure of a Compiler

- The back end *synthesizes* the target program from the intermediate representation produced by the front end.
- Typically the back end is independent of the source language.
- Conceptually, the input to each phase is the output of the previous.
- Sometimes, a phase changes the representation of the input.
  - For example, the lexical analyzer converts a character stream input into a token stream output.
- Sometimes, the representation is unchanged.
  - The machine-dependent optimizer transforms target-machine code into (hopefully improved) target-machine code. (Last 3 steps in the pink box)
- The front and back end are themselves each divided into multiple *phases*

The Structure of a Compiler

- The green box, can be roughly classified as 3 phases. Each of these phases changes the representation of the program being compiled.
  - *lexical analysis* or *scanning*, which transforms the program from a string of characters to a string of tokens;
  - *syntax analysis* or *parsing*, which transforms the program from a string of tokens to some kind of syntax tree;
  - semantic analysis, which decorates the tree with semantic information.

Lexical Analysis

- The first phase when compiler scans the source code
- This process can be left to right, character by character, and group these characters into tokens.
- The input character stream (which the compiler reads in) is grouped into meaningful units called **lexemes**, which are then mapped into **tokens**
- It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase.

Lexical Analysis

- Jobs for Lexical Analysis
  - Identify the lexical units in a source code
  - Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
  - Identify token which is not a part of the language

Example:

x = y + 10

Tokens

X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

Lexical Analysis

- An Example
  - x3 := y + 3; would be grouped into the lexemes x3, :=, y, +, 3, and ;
  - token is a <token-name,attribute-value> pair. For example
  - The lexeme x3 would be mapped to a token such as <id,1>
    - id means the “identifier”
    - The value 1 is the index of the entry for x3 in the symbol table produced by the compiler.
    - This table is used gather information about the identifiers and to pass this information to subsequent phase
  - The lexeme y is mapped to the token <id,2>
  - The lexeme + is mapped to the token <+>
  - The lexeme ; is mapped to the token <;>
- Lexemes are often described by *regular expressions*

Lexical Analysis

- Note that non-significant blanks are normally removed during scanning
- In C, most blanks are non-significant.
- That does not mean the blanks are unnecessary.
- Blanks inside strings are part of the lexeme and the corresponding token

Syntax Analysis (or Parsing)

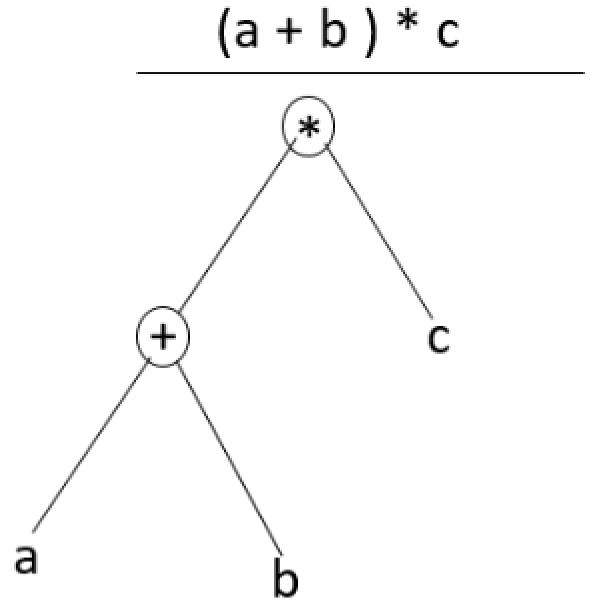
- Syntax analysis is all about discovering structure in code.
- It determines whether or not a text follows the expected format.
- The aim of this phase is to make sure that the source code was written by the programmer is correct or not. (See? The compiler complains if you put some other syntax in other languages into C++ compiler)
- The AST tree is built with the help of tokens (form the previous stage)

Syntax Analysis (or Parsing)

- A couple of jobs has to be done in this phase
  - Obtain tokens from the lexical analyzer
  - Checks if the expression is syntactically correct or not
  - Report all syntax errors
  - Construct a hierarchical structure which is known as a parse tree

Syntax Analysis (or Parsing)

- In the parse tree
  - Ensure that the components of the program fit together meaningfully
  - Gathers type information and checks for type compatibility
  - Checks operands are permitted by the source language



Syntax Analysis (or Parsing)

- Parsing involves a further grouping in which tokens are grouped into grammatical phrases which are often represented in a parse tree
- For example,
  - x3 := y + 3;
- The parsing into this kind of tree might

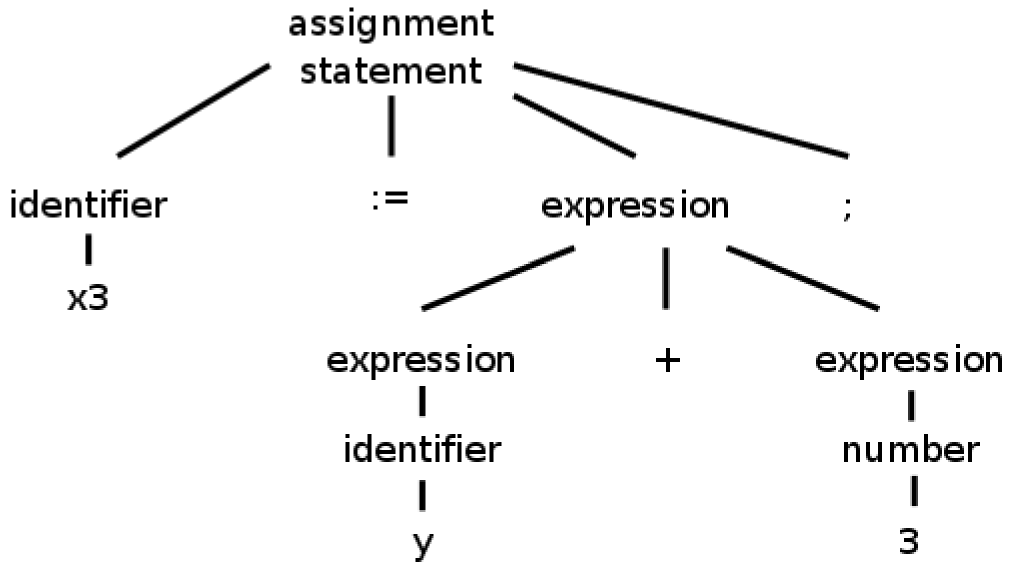
resulting from the grammar such as,

asst-stmt → id := expr ;

expr → number

| id

| expr + expr



Syntax Analysis (or Parsing)

- The division between scanning and parsing is somewhat arbitrary, in that some tasks can be accomplished by either.
- However, if a recursive definition is involved (as it is above for expr, it is considered parsing not scanning.

Semantic Analysis

- Semantic analysis checks the semantic consistency of the code.
  - It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent.
  - It also checks whether the code is conveying an appropriate meaning.
  - Semantic Analyzer will check for:
    - Type mismatches,
    - incompatible operands,
    - a function called with improper arguments,
    - an undeclared variable, etc.

Semantic Analysis

- Primary jobs for this stage (type checking)
  - Helps you to store type information gathered and save it in symbol table or syntax tree
  - Allows you to perform type checking (see the typecast example)
  - In the case of type mismatch, if there are no exact type correction rules which satisfy the desired operation, a semantic error is shown
  - Collects type information and checks for type compatibility
  - Checks if the source language permits the operands or not
  - [Example] the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication

```
float x = 20.2;
float y = x*30;
```

Semantic Analysis

- In this stage, the compiler needs semantic information, e.g., the types (integer, real, pointer to array of integers, etc) of the objects involved. This enables checking for semantic errors and inserting type conversion where necessary.
- Another example:
- x3 := y + 3
  - y is a real
  - x3 is an integer
  - We will need to insert very special (high level idea in this example) conversion operator
  - We can trace that from "3" and "bottom-up"