CS502: Compilers & Programming Systems

Introduction and
Lexical Analysis

Zhiyuan Li
Department of Computer Science
Purdue University, USA

PURDUE
UNIVERSITY

Course Web Sites

- All course contents and discussions, except grades, are on Piazza:

https://piazza.com/purdue/fall2014/cs502/home

Grades are stored on Purdue's Blackboard Learn, due to student information privacy regulations.

Please take the course interest survey posted on Piazza

PURDUE UNIVERSITY

Course Outline

- This course studies how to mechanically translate programs which are written in a certain programming language.
  - The syntax analysis can be used to translate documenting languages as well, e.g. html
- A programming language defines the components of programs.
  - It defines the syntax form for each of such components.
  - It assigns the meaning (i.e. the semantics) to such syntax forms.
    - Assignment statements
    - Branch statements
    - Loops
    - Function calls and returns

- A **translator** for the programming language analyzes a given program, and then
  - transforms it into another semantically equivalent program, or
  - directly performs the semantic actions specified in the program.
- In other words, a translator **implements** a programming language.

PURDUE
UNIVERSITY

- Two main classes of language translators:
  - **Interpreters:** Analyze a statement and execute it immediately. (Example: java virtual machines, UNIX shells, mobile scripts.)
  - **Compilers**: Analyze the whole program, then generates an equivalent program for later execution. This approach results in more efficient and more reliable codes,
    - Especially useful for performance-critical programs which will be executed many times and have long life-time.
- Conventionally, a compiler analyzes a program written in a high-level language and generates an equivalent program in a low-level language, e.g. machine code, or intermediate code.

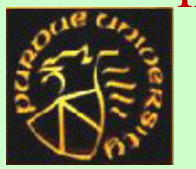
Abstract Syntax Tree

- At the center of a modern compiler, there is the internal representation (IR) of the program, which takes the form of **abstract syntax trees** (ASTs), or in short, **syntax trees**.

- The ASTs define the operations of a program.
  - Information about the identifiers is stored in the symbol table.

Main Phases in a Compiler

- Syntax analysis
  - converting the source code into ASTs and the symbol table
    - We give a quick coverage in this course
- Semantic analysis
  - type checking, memory allocation, dataflow analysis
- Code generation
  - Optimization
    - Memory
    - Speed
    - Reliability
- We emphasize dataflow analysis, code optimization and impact of parallelism and memory locality



PURDUE UNIVERSITY

Why Study Compiler Techniques

- To better understand the designs of programming languages.
  - Understand compilation error messages better
- To better understand program execution mechanism.
  - Better able to diagnose program execution errors
  - How does my code interact with the library routines?
    - Does the program error occur in my code or elsewhere?
  - Better understand the vulnerability of program execution
    - How are strings implemented, e.g.
- To understand limitations of compilers
  - Write programs in a way that their performance is not hampered by such limitations
- To be able to develop sophisticated user interfaces for software tools.

PURDUE UNIVERSITY

An Open Source Compiler

- GNU gcc compiler and its supporting tools are well known and have extensive community support
- Many industrial labs and computer/software vendors use it for R&D and for their shipped products
- Many university research projects use it as an experimentation platform
- NOTE: LLVM is another compiler tool that gains wide usage and support

PURDUE
UNIVERSITY

Lexical Analysis

- The first phase of the compiler is the *lexical analyzer*, also known as the scanner,
  - which recognizes the basic language units, called *tokens*.
- The exact characters in a token is called its *lexeme*.
- Tokens are classified by token types,
  - e.g. identifiers, constant literals, strings, operators, punctuation marks, and key words.
- Different types of tokens may have their own *semantic attributes* (or values) which must be extracted and stored in the *symbol table*.

- The lexical analyzer may perform *semantic actions* to extract semantic attributes and insert them in the symbol table.
- How to classify token types?
  - It mainly depends on what form of input is needed by the next compiler phase, the parser.
  - The parser takes a sequence of tokens as its input

Finite Automata (DFA)

- After we decide how to classify token types, we can use one of several ways to precisely express the classification.
- A common method is to use *a finite automaton* to define all character sequences (i.e. strings) which belong to a particular token type.
  - The states, *the starting state*, the *accepting states* of a finite automaton.
  - An accepting state is also called a *final state*.
  - Implemented by a *transition table*
- Given the definitions of different token types, it is possible for a string to belong to more than one type.
  - Such ambiguity is resolved by assigning priorities to token types. For example: Key words have a higher priority over identifiers.

- Finite automata for different token types are combined into a transition diagram for the lexical analyzer.
- Following the *"longest match" rule* –
  - keep scanning the next character until there is no corresponding transition. The longest string which matches an acceptance state during the scanning is the recognized token.
- Go back to the starting state of the transition diagram, ready to recognize the next token in the program.
- *Semantic actions* can be specified in the transition diagram.
- The lexical analyzer can also be used to remove *comments* from the program.

NFA

- Merging several transition diagrams into one may create the problem of *nondeterminism*.
- Two characteristics of an NFA
  - May exist edges which correspond to null input, called *ε-edges,* or *ε-transitions*
  - May exist more than one edge from the same state marked by the same input character
- A nondeterministic finite automaton (NFA) accepts an input string *x* if and only if there exists some path from the start state to **some** accepting state, such that the edge labels along the path spell out *x*.

NFA vs. DFA

- The DFA guarantees linear time complexity for scanning the input and determining whether to accept the input
- The NFA's implementation can take one of the following approaches:
  - Depth first search: may need to backtrack
    - Worst-case time exponential in the number of edges
  - Breadth first search: non-back tracking
    - On-the-fly simulation of NFA->DFA conversion
    - O(k(n+m)) time complexity
- There exist cases for which an NFA has significantly fewer states than any equivalent DFA

Regular Expressions

- Tokens can also be defined by declarations of *regular expressions*
- Regular expressions can be analyzed by compiler-like tools such that lexical analyzers can be constructed automatically to scan the input and extract the defined tokens
- Such a lexical-analyzer generator analyzes the declared regular expressions and generate a DFA transition table.
  - 1) For any given regular expression, there exist a DFA which accepts the same set of strings represented by the regular expression.
  - 2) For a given DFA, there exist a regular expression which represents the same set of strings accepted by the DFA.

- Regular expressions are composed by following a set of syntax rules:
  - Given an input alphabet Σ, a regular expression is a string of symbols from the union of the set Σ and the set { (,), ∗, |, ε}
  - The regular expression ε defines a language which contains the null string.
    - What is the DFA to recognize it?
  - The regular expression *a* defines the language { a }.
  - If regular expressions RA and RB define languages A and B, respectively, then
    - the regular expression (RA) | (RB) defines the language A ∪ B,
    - the regular expression (R_A)(R_B) defines the language AB (*concatenation*),
    - and the regular expression (RA)∗ defines the language A∗ (*Kleene closure).*

PURDUE UNIVERSITY

Several Definitions and Properties

- Two regular expressions are equivalent if they represent the exactly the same set of strings.
- There exist an algorithm to minimize a DFA
- There also exist algorithms which, for any DFA *M*, can construct a regular expression which represents the set of strings recognized by *M*.
  - (Unfortunately, sometimes the regular expression generated by such algorithms can be difficult to read.)
- There exist many languages, i.e. sets of input strings, which cannot be recognized by a DFA
  - DFA is too "primitive"
  - The "pumping" lemma is often an effective tool to prove that a certain set of strings cannot be recognized by a DFA

GCC's lexical analyzer

- The front end of GCC has a lexical analyzer and an LALR parser, generated by the lex (flex) and YACC tools.
- The *lex* tool takes regular expressions as its input and generates a DFA transition table.

PURDUE
UNIVERSITY