CSC 425 - Principles of Compiler Design I

Overview

The Implementation of Programming Languages

- Two major strategies:
  - Interpreter: A program that reads a source program and produces the results of executing this source
  - Compiler: A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) (Aho *et al*).
- Interpreters run programs directly
- Compilers do extensive preprocessing

Structure of a Compiler

- Compilers are typically divided into two main parts:
  - Front end: (analysis) read the source language program and understand its structure
  - Back end: (synthesis) generate an equivalent target language program and optionally optimize the code without changing its behavior.

Properties of a Compiler

- Recognize legal programs
- Generate correct code (most important)
- Conform to the specification of the source language
- Manage runtime storage of all variables/data

Intermediate Representations

- The phases of a compiler communicate via Intermediate Representations (IR)
- The front end maps the source language into an IR
- The back end maps an IR to the target language
- Often multiple IRs are produced by different phases of the front and back ends

Typical Phases of a Compiler

- Lexical Analysis (Scanner): converts a character stream to a token stream
- Parser: converts a token stream to an IR, typically an abstract syntax tree (AST).
- Semantic analysis: attempt to understand the meaning of the program (this is difficult) – perform limited analysis to catch inconsistencies, for example, type checking.
- Optimization (optional): modify programs based on some metric, for example, execution time or size of executable.
- Code generation: generate the target language, typically assembly code.

Lexical Analysis Example

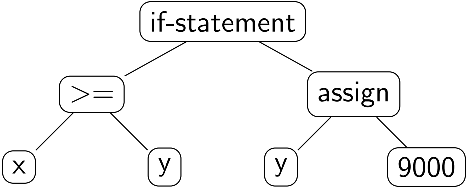
- Input text:

```
// This is a comment
if (x >= y) y = 9000;
```

- Token stream:
  IF, LPAREN, ID(x), GEQ, ID(y,) RPAREN, ID(y), ASSIGN, INT(9000), SEMICOLON
- Note: tokens are atomic objects, not character strings; comments and whitespace are typically not tokens

Parser Example

- Input token stream:
  IF, LPAREN, ID(x), GEQ, ID(y,) RPAREN, ID(y), ASSIGN, INT(9000), SEMICOLON
- Output Abstract Syntax Tree:



- Note: an AST is a tree where nodes are operations and children are operands

Semantic Analysis Example

- Compilers perform many semantic checks
- Example C++ variable scope:

```
int x = 3;
{
    int x = 4;
    cout << x; // prints 4, not 3
}
```

- Example C++ type checking:

```
int y = 4;
string z = "Bob";
x + z; // this is an error
```

Optimization Example

- Optimization improves the code in some fashion
- Example common subexpression elimination:
  `(x + y) * (x + y)` → `t = x + y; t * t;`
- Example constant folding
  `(1 + 2) * x` → `3 * x`

Issues

- Compilers and interpreters are almost this simple, but there are many pitfalls
- Example: How are bad programs handled?
- Language design determines the difficulty in implementing a compiler

Why Study Compilers?

- Become a better programmer
  - insight into the interaction between high-level source languages, compilers, and hardware
  - understand implementation techniques
  - better intuition about what your code does
  - understanding optimization allows you to write code that is easier for the compiler to optimize

Why Study Compilers?

- Compiler techniques are everywhere
  - Parsing ("little" languages, XML, ...)
  - Software tools (verifiers, checkers, ...)
  - Database engines and query languages
  - Text processing

Why Study Compilers?

- Blend of theory and engineering
  - Lots of interesting theory around compilers
  - But also interesting engineering challenges and tradeoffs
  - And some difficult problems (NP-hard or worse)

Why Study Compilers?

- Draws ideas from many parts of computer science
  - AI: greedy algorithms, heuristic search
  - Algorithms: graph algorithms, dynamic programming, approximation algorithms
  - Theory: grammars, deterministic finite automata, fixed point algorithms
  - Systems: interaction with OS, runtime systems
  - Architecture: pipelines, instruction set use, memory hierarchy management, locality