CS 415: Lecture 2

- Compiler Organization

---

Compilers

- A program that translates a program in one language into a program in another language
- A compiler typically (but not always) lowers the level of abstraction of the translated program
  - High-level language (e.g., C++) → Low-level language (e.g., assembly)
Compilers

- Recognize legal (and illegal) source programs
- Generate correct target program
- For usual case of HLL -> assembly/machine code
  - Manage storage of all variables and code
  - Need format for assembly/machine code
  - For separate compilation, need the help of a linker
    - Why?
    - Ever heard of a loader? What’s it for? What does it do?

Compiler Structure

source code \[\rightarrow\] front end \[\rightarrow\] il \[\rightarrow\] back end \[\rightarrow\] machine code

- Implications
  - Intermediate language (il)
  - Front end maps legal source code into il
  - Back end maps il onto target machine code
  - Simplifies retargeting
  - Allows multiple front ends
UNCOL: Does it work?

FORTRAN → front end → back end → target 1
C++ → front end → back end → target 2
Ada → front end → back end → target 3
Modula → front end

UNCOL

- Can we build \( n \times m \) compilers with \( n + m \) components
  - Must represent all required features in one il
  - Must encode all the knowledge in each front end
  - Must handle all the features in each back end
- Limited success with low-level ils
  - One real-life example is Mips
Compiler Structure - More Details

Front End Responsibilities

- Recognize legal (and illegal) programs
- Report errors
- Produce intermediate code
- Preliminary storage map
- Shape the code for the back end

- Much of the front end construction can be automated
  - Flex and yacc
Scanner (Lexical Analyzer)

Maps characters into tokens - the basic unit of syntax
- $x := x + y$ becomes
- $<id, > := <id, > + <id, >$
- Character string for a token is a lexeme
- Typical tokens: number, id, +, -, *, /, do, end
- Eliminate white space (tabs, blank, comments)
- Most often automatically generated using a tool like lex

Specifying Patterns

- A scanner must recognize various parts of the language’s syntax
- Some parts are easy:
  - White space
    - some combination of $<$tab$>$, tab, and $<$cr$>$
  - Keywords and operators
    - literal patterns: +, do, end, &
  - Comments
    - opening and closing delimiters: /* ... */
Specifying Patterns

- Other parts are much harder
  - Identifiers
    - Alphabetic character followed by k alphanumeric characters
  - Numbers
    - Integers: 0 or digit from 1-9 followed by digits from 0-9
    - Decimals: integer "." digits from 0-9
    - Reals: (integer or decimal) "E" (+ or -) digits from 0-9
    - Complex: "(" real "," real ")"

- We need a powerful notation to specify these patterns

Formal Languages

- **Alphabet**: finite set of symbols
- **String**: finite sequence of symbols from an alphabet
- **Language**: set of strings over an alphabet
- **Operations**
  - **Union**: $L_1 \cup L_2 = \{ s \mid s \in L_1 \lor s \in L_2 \}$
  - **Concatenation**: $L_1L_2 = \{ st \mid s \in L_1 \land t \in L_2 \}$
  - **Kleene closure**: $L^* = \bigcup_{i=0}^{\infty} L^i$
  - **Positive closure**: $L^+ = \bigcup_{i=1}^{\infty} L^i$
Regular Expressions

- Patterns are often specified as regular languages
- Regular expressions over an alphabet A:
  - $\epsilon$ is an RE denoting the set $\{\epsilon\}$
  - if $a \in A$, then $a$ is an RE denoting $\{a\}$
  - if $r$ and $s$ are REs, denoting $L(r)$ and $L(s)$, then:
    - $(r)$ is an RE denoting $L(r)$
    - $(r) \cup (s)$ is an RE denoting $L(r) \cup L(s)$
    - $(r)s$ is an RE denoting $L(r)L(s)$
    - $(r)^*$ is an RE denoting $L(r)^*$
- If we adopt a precedence for operators, can get rid of some parentheses
  - Order of precedence: closure, concatenation, alternation

RE Examples

- Identifier
  - letter $\rightarrow$ (a | b | c | ... | z | A | B | C | ... | Z)
  - digit $\rightarrow$ (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
  - id $\rightarrow$ letter ( letter | digit )*
- Numbers
  - integer $\rightarrow$ (+ | - | $\epsilon$) (0 | 1 | 2 | 3 | ... | 9) (digit)*
  - decimal $\rightarrow$ integer . (digit)*
  - real $\rightarrow$ (integer | decimal) E (+ | -) (digit)*
  - numbers can get even more complicated
- Most programming language tokens can be described with regular expressions
- We can use regular expressions to automatically build scanners
Parser (Syntax Analyzer)

- Recognize context-free syntax
- Guide context-sensitive analysis
- Construct il
- Produce meaningful error messages
- Attempt error correction

These days, parser generators mechanize much of the work

Grammar

- Context-free syntax is specified with a grammar
- Example
  - `<sheep noise>::= baa | baa <sheep noise>`
  - This grammar defines the set of noises that a sheep makes under normal circumstances
  - This format is called Backus-Naur form (BNF)
- Formally, a grammar \( G = (S, N, T, P) \)
  - \( S \) is the start symbol
  - \( N \) is a set of non-terminal symbols
  - \( T \) is a set of terminal symbols
  - \( P \) is a set of productions or rewrite rules \( (P : N \rightarrow N \cup T) \)
Grammar Example

- The following grammar defines simple expressions with addition and subtraction over the tokens id and number:
  - `<goal> ::= <expr>`
  - `<expr> ::= <expr><op><term> | <term>`
  - `<term> ::= number | id`
  - `<op> ::= + | -`
- id and number typically specified by REs

Derivation

- Given a grammar, valid sentences can be derived by repeated substitution:
  - `<goal>`
  - `<expr>`
  - `<expr> <op> <term>`
  - `<expr> <op> y`
  - `<expr> - y`
  - `<expr> <op> <term> - y`
  - `<expr> <op> 2 - y`
  - `<expr> + 2 - y`
  - `x + 2 - y`
- What do we do to recognize a valid sentence in some CFG?
Parse Tree

Abstract Syntax Tree

- Much more concise
- Abstract syntax tree (ASTs) are often used as an il between the front and back end
Back End

- Translate il into abstract machine code (e.g., three address)
  - Choose instructions for each each il operation
  - Allocate registers
- Optimize code
  - Make better/best use of cache and registers
  - Remove redundant code
- Generate final machine code
- Automation has been less successful here

Instruction Selection and Register Allocation

- Instruction selection
  - Produce compact, fast code
  - Use available instructions and addressing modes
  - Pattern matching problem
    - Ad hoc techniques
    - Tree pattern matching
    - String pattern matching
    - Dynamic programming
- Register allocation
  - Have value in a register when used
  - Limited resources
  - May change instruction choices
  - Optimal allocation is difficult! (NP-complete for 1 or k registers)
  - Modern allocators often use some form of graph coloring
Optimizer

- Modern optimizers are usually built as a set of passes
- Typical passes
  - Discover and propagate constant values
  - Reduction of operator strength
  - Common subexpression elimination
  - Redundant computation elimination
  - Move computation to less frequently executed place (e.g., out of loop)
- Recently, cache-related optimizations are gaining in importance
  - Loop tiling

Next Class

- Scanner construction
- Read ASU Chapter 3
- Read the manual page for flex