CS 160 Notes: Compilers

Compiler: program that translates a program written in one language (source) into another language (target)

- reports errors in the source program
- should be efficient and apply optimizations since programs must go through compiler
- usually translates to a lower level of abstraction

Interpreter: read an executable program and produce the results

- C is already compiled
- Scheme is typically interpreted
- Java is compiled into bytecodes, and then interpreted

Steps of Compilation

1. **Lexical Analysis (Scanning):** scan input file to produce stream of tokens; each lexeme (character string) corresponds to a token
    - use a set of patterns to specify valid tokens
    - each pattern specified as a regular expression built as a finite automata
    - need to get rid of white space and comments
    - need to print error messages for invalid strings
2. **Syntax Analysis (Parsing):** recognize the structure of the program based on context free grammars
    - apply rules of CFG to produce a parse tree
    - need to print error messages for invalid strings
3. **Intermediate Representations:** connect frontend (understand meaning of program) and backend (translate to equivalent program) of compilers
    - step for significant optimization
    - parse tree representation has too many details (relevant to remove ambiguities)
    - generate abstract representation from a parse tree with only necessary details
    - Abstract Syntax Trees (AST): nodes represent operators and children represent operands
4. **Semantic (Context-Sensitive) Analysis:** need to check if the semantics make sense
    - variables declared before they are used, valid variable types on operands with operator
    - can use symbol table to determine variable declarations and types

5. **Runtime Environment:** create efficient implementation of programming abstractions
6. **Code Generation:** generate lower level representations (may be still an intermediate representation)
    - need to create mappings between source and target code
    - improve inefficient code to make assembling code more efficient
    - there are limited number of registers on real machines

Desirable Properties of Compilers:

- generate correct executable code
- output program should be more efficient than the input program (time, energy)
- compiler should be efficient itself
- need good diagnostics for programming errors to aid debugging
- should support separate compilation; should work well with debuggers
- optimizations should be consistent and predictable; when optimizations are applied, they should improve performance on many of the inputs without degrading performance on other inputs

Why Build Compilers?

- provide interface between applications and architectures
- higher level programming languages provide better productivity, maintenance, and portability
- lower level machine code involves many details from instructions, pipelines, registers, and cache

Lexical Analysis (Scanner): maps stream of character into tokens, discarding white space and comments

- frontend stage of compiler
- only stage that touches every character of source code
- reports errors and correlated information (line number, etc)

Lexical Concepts:

- Token: basic unit of syntax which is the syntactic output of the scanner
    - Examples: keywords, operators, identifiers, numbers
- Pattern: rules that describe set of strings that correspond to token
- Lexeme: instance of pattern that matches to token

Specifying Lexical Patterns:

- Keywords and Operators are easy, match as literal patterns
- Identifiers and Numbers are more complex, need to represent using regular expressions (which are translated into DFAs to accept/reject patterns)

Regular Expressions: describe regular languages

- Empty string ε is a RE denoted by the set $\{\varepsilon\}$
- If a string s is in Σ, then s is a RE denoted by the set $\{s\}$
- If x and y are REs denoting languages L(x) and L(y), then the following operations are closed the class of REs: union, concatenation, kleene star

Extensions to Regular Expressions

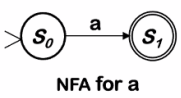
- $x+ = x\,x^*$ denotes $L(x)^+$
- $x? = x \mid \varepsilon$ denotes $L(x) \cup \{\varepsilon\}$
- [abc] = a | b | c matches one character in the square bracket
- a-z = a | b | c | ... | z range
- [0-9a-z] = 0 | 1 | 2 | ... | 9 | a | b | c | ... | z
- [^abc] ^ means negation
  matches any character except a, b or c
- . = [^\n] dot matches any character except the newline
  \n means newline so dot is equivalent to [^\n]
- "[" matches left square bracket, meta-characters in double quotes become plain characters
- \[matches left square bracket, meta-character after backslash becomes plain character

Deterministic Finite Automata (DFA):

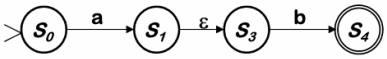
- Defined by a set of states S, a set of inputs Σ, a transition function delta : S x Σ -> S, a start state s_0, and a set of final accepting states F
- Takes linear time to simulate with respect to input string and 2^r space complexity
- Want to build scanner using a DFA to minimize time complexity
- Hopcroft's Algorithm: find a representative set of non distinguishable states for minimal DFA

Nondeterministic Finite Automata (NFA):

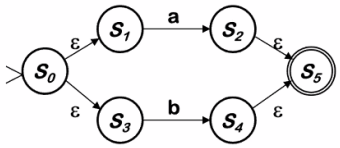
- Thompson's Construction for RE -> NFA: use closure properties
- An NFA accepts a string if there exists any path that results in an accepting state
- To turn a NFA into a DFA, we take the power set of the NFA to generate the DFA
- Takes O(x * r) time, with O(r) space complexity



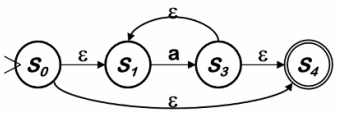
NFA for a



NFA for ab



NFA for a | b



NFA for a*

Scanner Construction:

- Identify all relevant tokens
- Write the RE that expresses each token pattern
- Translate RE to NFA to DFA; optimize the DFA (state minimization algorithms)
- Translate into code

Building Fast Scanners:

- no need to encode transitions in a table
- encode state and actions directly in code (generates ugly, but effective code)

```
state = s0;
string = ε;
char  = get_next_char();
while (char != eof) {
  state = δ(state,char);
  string = string + char;
  char = get_next_char();
}
if  (state in Final) then
  report acceptance;
else
   report failure;
```

Limits of Regular Languages: cannot recognize recursive patterns

Parsing: uses context free syntax to build an intermediate representation of our tokenized source program

- Input: sequence of tokens representing the source program
- Output: a parse tree (abstract syntax tree)

Context Free Grammar: a four tuple (terminal symbols, non-terminal symbols, start symbol, production rules)

- Terminal symbols (T): terminating tokens returned by the scanner
- Nonterminal symbols (N): variables substituted during a derivation (denotes sets of substrings)
- Start symbol (S): strings in a grammar is derived from its start symbol
- Production rules (P): $N \rightarrow (N \cup T)^*$
- can express recursive nature of most programming languages

Vocabulary:

- Sentence of G: string of terminals in L(G)
- Sentential Form of G: string of non-terminals and terminal from which a sentence of G can be derived
- Derivation: a sequence of applied production rules
- Production: a rule mapping non-terminals to a string of non-terminals and terminals
- Parsing: determining the derivation for a sentence

Derivations:

- Each step has 2 choices: choose a non-terminal to replace or a production rule to apply
- Leftmost Derivation: replace leftmost non-terminal at each step
- Rightmost Derivation: replace rightmost non-terminal at each step
- Both methods of derivation does not reduce the expressiveness

Parse Tree: shows how a sentence is parsed (but no strict ordering)

- root node is the start symbol, leaves are terminal symbols, internal nodes are non-terminals
- for each parent node in the tree and its children, there is a corresponding production rule

Ambiguous Grammars:

- can be generated non uniquely using the derivation technique
- must ensure our grammar is unambiguous (no sentence has 2 parse trees)
- add precedence by creating a non-terminal for each level of precedence
- have higher precedence generated from the right to enforce left associativity

Top Down Parsers: (LL(1), Recursive Descent Parsers)

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to derivation)
- Pick a production rule and try to match the input
- Need to backtrack if a bad production rule is picked
- Some grammars are backtrack-free (Predictive Parsing)

Top Down Parsing Algorithm

- Construct root node of parse tree, label with start symbol, and set current node to root node
- Repeat the following until all the input is consumed (frontier of parse tree matches input string)
    - If the label of the current node is a non-terminal node A, select a production with A on its LHS and for each symbol on the RHS, construct the appropriate child
    - If the current node is a terminating symbol,
        - If it matches input string, consume it (advance to next input pointer)
        - If it does not match the input string, backtrack
    - Set the current node to the next node in the frontier of the parse tree
        - If there is node node left in the frontier of the parse tree and the input is not consumed, backtrack
- Key step is picking correct production rule, guided by the input string
- Note: Backtracking algorithm not efficient, using Predictive/LL Parser
- Note: Choosing excess production (left recursion) would result in a nontermination problem

Left Recursion:

- Top-down parsers cannot handle left-recursive grammars
- Left recursive: non terminal A creates a derivation A α for some string α in (NT ∪ T)*
- Recursion must be right recursion (can turn left recursive to equivalent right recursive grammar

Eliminating Left Recursion:

- Immediate recursion can be eliminated trivially
- For nonimmediate recursion, follow this algorithm:
    - Arrange NTs in some order A_1, A_2, ..., A_n
    - For i in range(1, n+1), for j in range(1, i):
        - replace each production A_i -> A_j γ with A_i -> δ_1 γ | δ_2 γ | ... | δ_n γ
        - where A_j -> δ_1 | δ_2 | ... | δ_n are the production rules of A_j
    - Eliminate immediate recursion rules trivially
- Ensures no inner loop because we delete production in correct order, which turns into self loops

Picking the Right Production:

- In general, need Cocke-Younger, Kasami, or Earley's algorithm which is $O(x^3)$
- Some special subcases can look ahead in linear complexity

Predicting Parsing:

- Basic idea: given A -> α | β, the parser should be able to choose α or β based on peeking at the next token in the stream
- FIRST sets: define FIRST(α) as the set of tokens that appear as the first terminal symbol in some string that derives from α.
- Must find all tokens that can be at the beginning of a string that is derived from α
- LL(1) Property: if A -> α and A -> β both appear in the grammar, we would like FIRST(α) and FIRST(β) to be disjoint. This allows the look ahead of exactly one symbol.

Left Factoring:

- For grammars without LL(1) property, we need to apply the left-factoring algorithm
- Algorithm: For all nonterminals A, find the longest prefix α that occurs in two possible productions of A
    - If α is nonempty, then we want to extract α by adding an intermediate production

Recursive Descent Parsing:

- top down parsing method (descent downward)
- Use a set of mutually recursive procedures (one procedure for each non terminal symbol)
    - Start parsing process by calling procedure that corresponds with start symbol
    - Each production becomes one clause in procedure
- Predictive parsing is a special type of recursive descent using a look-ahead symbol

FIRST Sets:

- Set of all terminals we could possibly see when starting to parse S
- We need look-ahead token to tell us with 100% confidence which production rule to apply
- Anytime we have a production rule like A -> α | β, we need to make sure FIRST(α) is distinct from FIRST(β)

Computing FIRST Sets: for FIRST(X)...

- Set of tokens that appear as the first symbol in some string that derives from α
- If X is a terminal, return X
- If X is a non-terminal and X -> ε is a production, include ε
- If X is a non-terminal and X -> t is a production where t is terminal, include t
- If X is a non-terminal and X -> Y_1 | ... | Y_k is a production where Y_i is a non-terminal, include all FIRST(Y_i)

  If X is a non-terminal and X -> Y_1 ... Y_k is a production where Y_i is a non-terminal, include FIRST(Y_1) except ε
    - If ε in all Y_i for all i <= j, include FIRST(Y_i)
    - If ε in all Y_i for all i <= k, include ε

Epsilon in FIRST Sets:

- Want to make sure epsilon not in FIRST Set
- Otherwise, must compute FOLLOW Set which is the set of characters that follow the first non-terminal

FOLLOW Sets: for FOLLOW(A)

- Set of all terminals that could appear immediately after non-terminal A
- Put $ in FOLLOW(S) where $ is EOF symbol
- If there is a production A -> α B β, then include everything in FIRST(B) except ε
    - If ε in FIRST(β) then put everything in FOLLOW(A) in FOLLOW(B)
- If there is a production A -> α B, then put everything in FOLLOW(A) in FOLLOW(B)

LL(1) Grammars: left to right scan, leftmost derivation, 1 token-look ahead

- productions are uniquely predictable with a single token look ahead
- use recursive descent parser to implement LL(1) grammar, using look ahead symbol to determine production
- when no element in FIRST set matches, check the FOLLOW set
    - if look-ahead symbol is in FOLLOW set and there is an epsilon production, choose that
    - otherwise, terminate with parsing error

Stack-Based Table Driven Parsing:

- Two dimensional array with M[A, a] giving a production, A being a nonterminal and A being a terminal
- If the top of the stack is A and the look-ahead symbol is a, we can apply M[A, a]

LL(1) Parse Table Construction:

- For all productions A -> α,
    - For each terminal symbol a in FIRST(α), add A -> α to M[A, a]
    - If ε in FIRST(α), then for each terminal symbol b in FOLLOW(A), add A -> α to M[A, b]
    - Set undefined entries in M to ERROR

Top Down Parsing Recap:

- Simple to construct by hand
- Intuitive way to reason about parsing
- Predictive parsing is fast
- Really messy for complex grammars
- Does not handle left recursion nicely (becomes restrictive)

Bottom Up Parsers (LR(1), Shift-Reduce Parsers)

- Start at the leaves and grow toward the root
- Reducing input string to start symbol
- At each reduction step, a particular substring matching the right side of a production is replaced by the symbol on the left side of the production
- Bottom-up handle a larger class of grammars
- Only builds a small part a tree
- Can handle left-recursive grammars without modification

LL(k) has complete left context and the k terminals

LR(k) has complete left context, the reducible phrase, and the k terminals

How Bottom Up Parsing Works:

- find the rightmost derivations of a sentence by running productions backwards from sentence to the start symbol
- figure out what leads to w, then replace w with γ_n; then, figure out what leads to γ_n and with γ_{n-1}; continue until we reach the start symbol
- To derive γ_{i-1} from γ_i by using production α -> β, match β against γ_i and then replace β with α
- Nodes with no parent in a partial tree form its upper fringe
- Reduction: replace β with α to shrink the upper fringe

Handles and Reductions:

- Handle: substring β of a tree's fringe that matches some production α -> β in one step
- Does not need to scan past the handle (only a look ahead)

Shift-Reducing Parsing:

- Shift: eat input terminals and move them onto stack
- Reduce: apply some production in reverse
- Accept: stop parsing and report success
- Error: can an error reporting or recovery routine

LR Parsers:

- fast, and simple to implement
- need push down automata to handle recursive nature of grammar and recognize strings
- need pushdown automata in the form of a table to determine how to deal with handles

Handle Recognizing Machine:

- put special placeholder token to mark how far we went along the production
- build NFA, turn into DFA, then build ACTION GOTO table

ACTION GOTO Table: rows as states inputs as columns (Actions = Terminals; Goto = Non Terminals)

- Action: SX means shift and add X to the top of the stack
- Action: RX means reduce by production X
- Goto: shows the state to go after pushing left side symbol
- A means accept; empty square is an error
- LR(0): If action table has S/R conflict, must rewrite grammar to remove conflict

When to Reduce LR(0):

- If there is a dot at the end of the RHS, we can apply a reduction rule
- There may be reduce-reduce conflicts with 2 reductions using same character
- There may be shift-reduce conflict, but the solution should be to prefer shift

LR Parser Tables

- SLR, LALR, and LR(K) are different ways of automatically generating a state machine to capture handles encoded into the form of a table. Strings are recognized from a star.
- Action table tells shift/reduce/accept/throw error for terminals
- Goto table tells how to update the state on a reduce action