09 - Parsing

Midterm next Thursday (Mar 14th)

- Interpreters, program transformation, x86, IRs, lexing, parsing
- Lectures covered up to March 7th (this Thursday)
- Can look at last year's Compilers website
- How to prepare:
  - Start on HW3
  - Review slides
  - Review example code from lectures (try re-implementing)

Overview of parsing

- Takes lexer's stream of tokens => builds abstract syntax tree
- Responsible for reporting syntax errors if token stream can't be parsed
- Variable scoping, type checking, etc. is handled **later** (semantic analysis)
- AST: represents syntactic structure of source code.
  - E.g. following have same ASTs:
    - x + y * z
    - x + (y * z)

How to implement parser?

- By hand: (recursive descent)
  - Clang, gcc since 3.4
  - Libraries can make this easier (e.g. parser combinators like parsec)
  - Biggest benefit is quality of error messages
- Option 2: use a parser generator
  - Much easier to get right ("specification is the implementation")
    - Parser generator warns of ambiguities, ill-formed grammars, etc.
  - gcc (before 3.4), Glasgow Haskell Compiler, OCaml compiler
  - Parser generators: Yacc → C, Bison → C++, ANTLR, menhir → OCaml

Defining syntax

- Recall:

    - Alphabet Σ is finite set of symbols
    - Word (or string) is a sequence over Σ
    - Language over Σ is a set of words over Σ
- Set of syntactically valid programs in a programming language is a language
    - In practice: this language is over token types (lexer's higher-level view is easier to work with)
- This language is often specified by a context-free grammar
    - e.g.
        - <expr> ::= <int> | <var> | <expr>+<expr> | <expr>*<expr> | (<expr>)
- Why not use regular expressions?
    - Programming languages are typically not regular... typically because regular languages can't count
    - For contrradiction: suppose L(<expr>) is regular
    - There is a DFA such that L(A) = L(<expr>)
    - Suppose A has N states
    - Consider the word (((...(
        - consisting of N+1 left parentheses
        - By pigeonhole, some state in DFA must be repeated while reading these left parens at paren #m and paren #r
        - The language accepts ((((... m starting parentheses followed by)))... m ending parentheses
        - But it also accepts (((... r starting parentheses followed by)))... m ending parentheses (as must be in same state after m starts as after r starts)
        - Thus it accepts unbalanced parentheses,
        - which <expr> specifies is wrong.

Context-free grammar

- $G = (N, \Sigma, R, S)$ consists of
    - N: finite set of non-terminal symbols
    - Σ: finite alphabet (or set of terminal symbols)
    - $R \subseteq N \times (N \cup \Sigma)^*$: finite set of rules or productions
        - Rules often written A $\rightarrow$ w
        - A is a non-terminal (left-hand side)
        - w is a word over N and Sigma (right-hand side)
    - $S \in N$: the starting non-terminal
- Backus-Naur form: specialized syntax for writing CFGs

- non-terminal symbols written between <>
- e.g. how we wrote expr:
  - <expr> ::= <int> | <var> | <expr>+<expr> | <expr>*<expr> | (<expr>)

Derivations

- Consists of finite sequence of words $w1, ..., wn \in (N \cup \Sigma)^*$ such that w1 = S and for each i, wi+1 is obtained from wi by replacing a non-terminal symbol with the right-hand-side of one of its rules
- E.g.
  - Grammar: <S> ::= <S><S> | (<S>) | eps
  - Derivations:
    - <S> => (<S>) => ()
- The set of all strings w over Σ such that G has a derivation of w is the language of G, written L(G)
- A particular word may have multiple derivations in the grammar, but
  - A derivation is leftmost if we always substitute the leftmost non-terminal, and rightmost if we always substitute the rightmost non-terminal.
  - A word can have multiple rightmost or leftmost derivations

Parse trees

- Tree representation of a derivation
- Every leaf node is a terminal, each internal node is a non-terminal
  - If internal node has label X its children (read left-to-right) are RHS of rule with X as LHS
- Root is labeled with start symbol
- Parse tree corresponds to many derivations,
  - but exactly one leftmost derivation (and exactly one rightmost derivation)

Multiple parse trees corresponding to the same string is bad... leads to ambiguity.

Ambiguity

- A CFG is ambiguous if there are two different parse trees for the same word.
  - Equivalently: if some word has two different left-most derivations
- For example: x + 2*y*z + 3*x + 7

    - We all have an intuitive, unambiguous understanding of how this should be parsed: as a sum of products.
    - So we're going to make sure the pluses are higher in the tree.
    - <expr> ::= <expr> + <expr> | <term>
    - <term> ::= <term> * <term> | <factor>
    - <factor> ::= <int> | <var> | (<expr>)
    - Now it is no longer valid to have an addition as a child of a multiplication.
    - This is actually still an ambiguous grammar! 2 * y * z could be (2 * y) * z or 2 * (y * z)
    - We want to associate to right or left consistently...
        - Change "term" rule to <term> ::= <factor> * <term> | <factor>
        - This forces us to recurse in one direction.
    - Now + and * associate to the left (recursive case right of operator)
    - has higher precedence than + (* is farther from start symbol)
- Some languages are inherently ambiguous: context-free, but every grammar that accepts the language is ambiguous
    - E.g. $\{a^i b^j c^k : i = j \text{ or } j = k\}$
        - <S> ::= <AB><C> | <A><BC>
        - <AB>::= a<AB>b | eps
        - <BC> ::= b<BC>c | eps
        - <A> ::= a<A> | eps
        - <C> ::= c<C> | eps
        - any word with equal numbers of 'a's, 'b's, and 'c's will be ambiguous.
    - In programming languages, we always want unambiguous grammars.

Regular languages are context-free

- Suppose that L is a regular language
- Then there is an NFA A = (Q, Sigma, Delta, s, F) such that L(A) = L
- How can we construct a CFG that accepts L?
    - Take nonterminals N = Q (states of automaton)
    - Start nonterminal S? Start state s
    - Rules? R = { <q> ::= a<q'> exactly when (q, a, q') \in Delta }
        - 'a' is a terminal symbol here.
        - Also need to handle accept states...
        - For any transition to terminal state, allow <q> ::= eps (when q \in F)
- Consequence: could fold lexer definition into grammar defn

- Why not?
  - Separation of concerns
  - Ambiguity is easily understood at lexer level, not parser level
  - Parser generator only handle some context-free grammars
    - Non-determinism is easy at lexer level (NFA → DFA conversion)
    - Non-determinism is hard at parser level (deterministic CFL ≠ non-deterministic CFL)

Pushdown automata

- PDAs recognize context-free languages
- PDA basically like having NFA + a stack
- Parser generator compiles (restricted) grammar to (restricted) PDA
- rather than finite state machine, have "finite control, infinite memory"
- Specify transitions as (a, b → c) where a is what we're reading as input, b is current top of stack, c is pushed onto stack
  - x → eps : pop x
  - epx → x : push x
- Don't need to know these in full generality, but it's useful to have them because other parsing strategies we'll look at are basically deterministic cases of PDA (nondeterminism is hard to implement with stack)
- Can look at formal defn in lecture slides

End of lecture examples

- Using menhir for parsing into AST for language with only parens
- Then an example that builds off of the lexer from last lecture