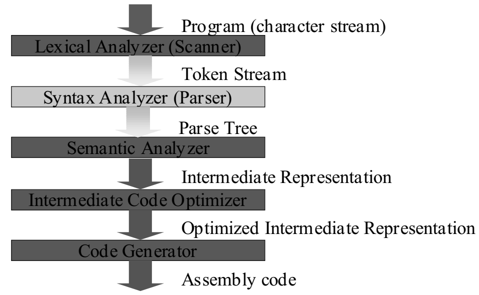
6.035

Fall 2000

Lecture 3: Introduction to Parsing

Formal Languages

Parser



Aspects of Languages

- Aspects of Language Structure
  - token (words)
  - syntactic (grammar)
  - semantic (meaning)
- Subjective View of Language Structure
  - Natural Languages
  - Formal Languages
  - Programming Languages

Aspects of Correctness

- Token Incorrect
  - asgd bjjte jkdk
- Token Correct, Syntax Incorrect
  - run jump has block five whatever zebra smiles
- Token Correct, Syntax Correct, Semantics Incorrect
  - Colorless green ideas sleep furiously

Job of Parser

- Takes A Token Stream as Input
  - already checked for token correctness
- Determine Syntactic Correctness
- Produce Program Representation that Supports
  - Semantic Correctness Checks
  - Further Analysis and Optimizations
- Distinction between syntactic and semantic correctness not completely clear cut - matter of convenience at boundary

Specifying Formal Languages

- Huge Triumph of Computer Science
  - Beautiful Theoretical Results
  - Practical Techniques and Applications
- Two Dual Notions
  - Grammar (generative approach)
  - Automaton (recognition approach)

Grammar Example

Regular Expression: (0|1)*.(0|1)*

Corresponding Regular Grammar:

Goal → *BinaryFloat*
BinaryFloat → 0 *BinaryFloat*
BinaryFloat → 1 *BinaryFloat*
BinaryFloat → . *BinaryTail*
BinaryTail → 0 *BinaryTail*
BinaryTail → 1 *BinaryTail*
BinaryTail → _

Notation:
Nonterminals
terminals

Applying Productions

Goal	Apply *Goal* → *BinaryFloat*
BinaryFloat	Apply *BinaryFloat* → 0 *BinaryFloat*
0 BinaryFloat	Apply *BinaryFloat* → 1 *BinaryFloat*
01 BinaryFloat	Apply *BinaryFloat* → . *BinaryTail*
01. BinaryTail	Apply *BinaryTail* → 0 *BinaryTail*
01.0 BinaryTail	Apply *BinaryTail* → 1 *BinaryTail*
01.01 BinaryTail	Apply *BinaryTail* → _
01.01	

Production Game

start with *Goal* nonterminal

loop until no more nonterminals

- choose a nonterminal
- choose a production with nonterminal in RHS
- replace nonterminal with LHS of production

generated string is in language

Note: different choices produce different strings

More Examples

Goal	*Goal*
BinaryFloat	*BinaryFloat*
0 BinaryFloat	*. BinaryTail*
01 BinaryFloat	*.1 BinaryTail*
011 BinaryFloat	*.11 BinaryTail*
011. BinaryTail	*.110 BinaryTail*
011.1 BinaryTail	*.110*
011.11 BinaryTail	
011.11	

Defining a Language

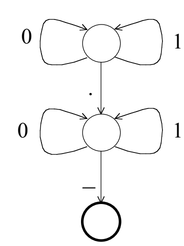
- Grammar
  - Generative approach
  - All strings that grammar generates (How many are there for grammar in previous example?)
- Automaton
  - Recognition approach
  - All strings that automaton accepts
- Different flavors of grammars and automata
- In general, grammars and automata correspond

Correspondence in Example

Grammar

Goal → *BinaryFloat*
BinaryFloat → 0 *BinaryFloat*
BinaryFloat → 1 *BinaryFloat*
BinaryFloat → . *BinaryTail*
BinaryTail → 0 *BinaryTail*
BinaryTail → 1 *BinaryTail*
BinaryTail → _

Automaton



Regular Languages

- Automaton Characterization
  - (S,A,F,s_0,s_F)
  - Finite set of states S
  - Finite Alphabet A
  - Transition function $F : S \times A \rightarrow S$
  - Start state s_0
  - Final states s_F
- Lanuage is set of strings accepted by Automaton

Regular Languages

- Grammar Characterization
  - (T,NT,S,P)
  - Finite set of Terminals T
  - Finite set of Nonterminals NT
  - Start Nonterminal S (goal symbol, start symbol)
  - Finite set of Productions $P: NT \rightarrow T \cup NT \cup T.NT$
- Language is set of strings generated by grammar

Grammar and Automata Correspondence

Grammar	Automaton
Regular Grammar	Finite-State Automaton
Context-Free Grammar	Push-Down Automaton
Context-Sensitive Grammar	Turing Machine

Context-Free Grammars

- Grammar Characterization
  - (T,NT,S,P)
  - Finite set of Terminals T
  - Finite set of Nonterminals NT
  - Start Nonterminal S (goal symbol, start symbol)
  - Finite set of Productions $P: NT \rightarrow (T.NT)^*$
- RHS of production can have any sequence of terminals or nonterminals

Push-Down Automata

- DFA Plus a Stack
  - (S,A,V, F,s_0,s_F)
  - Finite set of states S
  - Finite Input Alphabet A, Stack Alphabet V
  - Transition relation $F : S \times (A \cup \{\varepsilon\}) \times V \rightarrow S \times V^*$
  - Start state s_0
  - Final states s_F
- Each configuration consists of a state, a stack, and remaining input string

CFG Versus PDA

- CFGs and PDAs are of equivalent power
- Grammar Implementation Mechanism:
  - Translate CFG to PDA, then use PDA to parse input string
  - Foundation for bottom-up parser generators

Context-Sensitive Grammars and Turing Machines

- Context-Sensitive Grammars Allow Productions to Use Context
  - *P: (T.NT)+* → *(T.NT)**
- Turing Machines Have
  - Finite State Control
  - Two-Way Tape Instead of A Stack

Grammars Versus Automata

- Usually easier to specify grammar than corresponding automaton
- But we have a recognition problem, not generation problem
- One solution for parsers:
  - Automatically convert grammar to corresponding automaton

Programming Language Syntax

- Regular languages suboptimal for specifying programming language syntax
- Why? Constructs with nested syntax
  - (a+(b-c))*(d-(x-(y-z)))
  - if (x < y) if (y < z) a = 5 else a = 6 else a = 7
- Regular languages lack state required to model nesting
- Canonical example: balanced parentheses

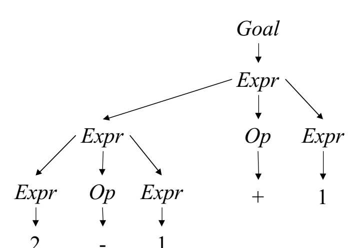
Canonical Example

Goal → *Expr*
Expr → *Expr Op Expr*
Expr → 0
Expr → 1
Expr → 2
Op → +
Op → -
Op → *
Op → /

Sample Derivation

Goal
Expr
Expr Op Expr
Expr Op Expr Op Expr
2 *Op Expr Op Expr*
2 - *Expr Op Expr*
2 - 1 *Op Expr*
2 - 1 + *Expr*
2 - 1 + 1

Parse Tree for 2-1+1



Corresponds to (2-1)+1

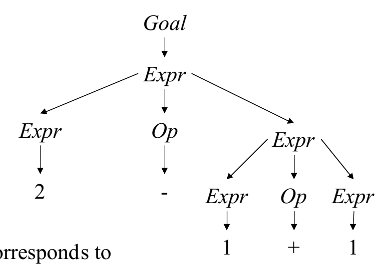
Parse Tree

- Internal Nodes: Nonterminals
- Leaves: Terminals
- Edges:
  - From Nonterminal from LHS of production
  - To Nodes from RHS of production

Parser goal

- Produce parse tree for program
- Subsequent passes use parse tree as starting point
- Will Use Context-Free Grammars
  - Powerful enough (more powerful than regular grammars)
  - Not Too Powerful (don't want to use a context-sensitive grammar)

Another Parse Tree for 2-1+1



Corresponds to
2-(1+1), NOT (2-1)+1

Amiguous Grammar

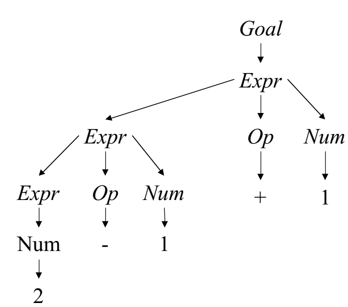
- Two parse trees for same string
- Parse tree related to semantics of program
  - first parse tree, 2-1+1 = 2
  - second parse tree, 2-1+1 = 0
- Ambiguity considered bad
- Recommended solution: hack the grammar

Hacked Grammar

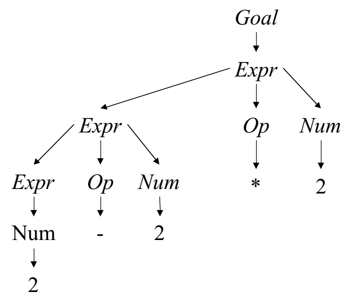
Goal ➔ *Expr*
Expr ➔ *Expr Op Num*
Expr ➔ *Num*
Num ➔ 0
Num ➔ 1
Num ➔ 2

Op ➔ +
Op ➔ -
Op ➔ *
Op ➔ /

New Parse Tree for 2-1+1



Parse Tree for 2-2*2



Precedence Violations

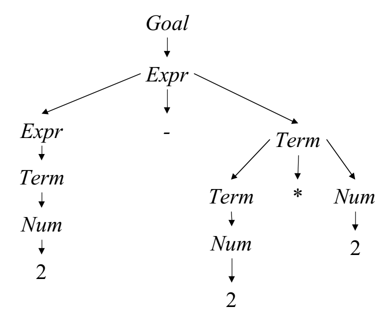
- All operators associate to left
- Violates precedence of * over +
- Recommended Solution: Hack Grammar

Hacked Grammar

Goal ➔ *Expr*
Expr ➔ *Expr* + *Term*
Expr ➔ *Expr* - *Term*
Expr ➔ *Term*
Term ➔ *Term* * *Num*
Term ➔ *Term* / *Num*
Term ➔ *Num*

Num ➔ 0
Num ➔ 1
Num ➔ 2

New Parse Tree for 2-2*2



General Idea

- Group Operators into Precedence Levels
  - * and / are at top level, bind strongest
  - + and - are at next level, bind next strongest
- Nonterminal for each Precedence Level
  - *Expr* is nonterminal for + and -
  - *Term* is nonterminal for * and /
- Can make operators left or right associative within each level
- Generalizes for arbitrary levels of precedence

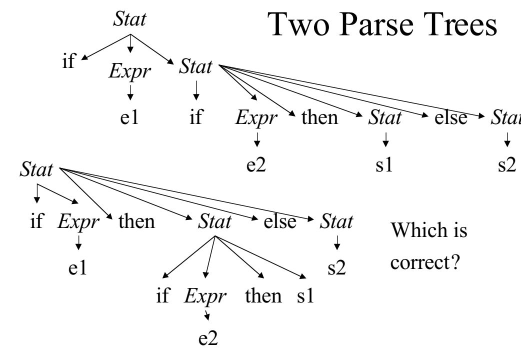
Handling If Then Else

Goal ➔ *Stat*
Stat ➔ if *Expr* then *Stat* else *Stat*
Stat ➔ if *Expr* then *Stat*
Stat ➔ ...

Parse Trees

- Consider Statement if e_1 then if e_2 then s_1 else s_2

Two Parse Trees



Which is correct?

Alternative Readings

- Parse Tree Number 1
  - if e_1
    - if e_2 s_1
    - else s_2
- Parse Tree Number 2
  - if e_1
    - if e_2 s_1
  - else s_2

Grammar is ambiguous

Why not use indentation to determine which one programmer wants?

Hacked Grammar

Goal ➔ *Stat*
Stat ➔ *WithElse*
Stat ➔ *LastElse*
WithElse ➔ if *Expr* then *WithElse* else *WithElse*
WithElse ➔ ...
LastElse ➔ if *Expr* then *Stat*
LastElse ➔ if *Expr* then *WithElse* else *LastElse*

Hacked Grammar

- Basic Idea: control carefully where an if without an else can occur
  - Either at top level of statement
  - Or as very last in a sequence of if then else if then ... statements

Problem With Hacked Grammars

- Hacked grammars more complicated than original “intuitive” grammar
- Parse trees more complicated
  - Harder for subsequent passes to process
  - Larger than necessary

Solution

- Abstract versus Concrete Syntax
  - Abstract syntax corresponds to “intuitive” way of thinking of structure of program
    - Omits details like superfluous keywords that are there to make the language easily parsable
    - May be ambiguous
  - Concrete Syntax corresponds to full grammar used to parse the language
- Parsers are often written to produce abstract syntax trees.

Interaction with Lexical Analyzer

- Examples have full language in grammar
- In practice terminals are produced by lexer
- Single kind of terminal for constructs like
  - integer numeric constants: 2, 5, 8
  - floating numeric constants: 2.345, 7.68
  - variables: x, y, z
- Specific value is stored in terminal structure returned by lexer

Grammar Vocabulary

- Leftmost derivation
  - Always expands leftmost remaining nonterminal
  - Similarly for rightmost derivation
- Sentential form
  - Partially or fully derived string from a step in valid derivation
  - 0 + *Expr Op Expr*
  - 0 + *Expr* - 2

Summary

- Defining Formal Languages
  - generative (grammar) vs. recognition (automata)
- Context-Free versus Regular Grammars
- Ambiguity and Eliminating Ambiguity
  - Hacked Grammars
  - Precedence
- Abstract versus Concrete Syntax