Building a Parser I

CS164
3:30-5:00 TT
10 Evans

PA2

- in PA2, you'll work in pairs, no exceptions
  - except the exception if odd # of students
- hate team projects? form a "coalition team"
  - team members work alone, but
    - discuss design, clarify the handout, keep a common eye on the newsgroup, etc
    - share some or all code, at the very least their test cases!
  - a win-win proposition:
    - work mainly alone but hedge your grade
    - each member submits his/her project, graded separately
    - score: the lower-scoring team member gets a bonus equal to half the difference between his and his partner's score

Administrativia

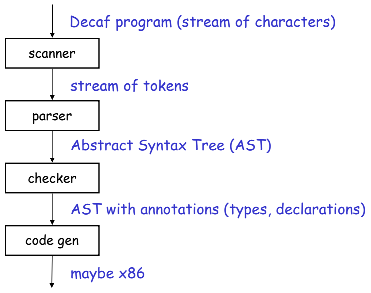
- Section room change
  - 3113 Etcheverry moving next door, to 3111 Etch.
  - starting 9/22.

Overview

- What does a parser do, again?
  - its two tasks
  - parse tree vs. AST
- A hand-written parser
  - and why it gets hard to get it right

What does a parser do?

Recall: The Structure of a Compiler


Decaf program (stream of characters)
scanner
stream of tokens
parser
Abstract Syntax Tree (AST)
checker
AST with annotations (types, declarations)
code gen
maybe x86

Recall: Syntactic Analysis

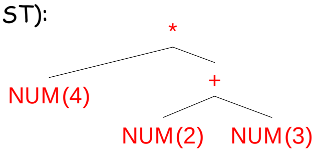
- **Input:** sequence of tokens from scanner
- **Output:** abstract syntax tree
- Actually,
  - parser first builds a parse tree
  - AST is then built by translating the parse tree
  - parse tree rarely built explicitly; only determined by, say, how parser pushes stuff to stack
  - our lectures first focus on constructing the parse tree; later we'll show the translation to AST.

Example

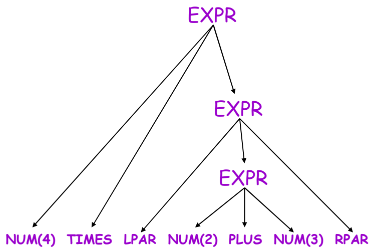
- Decaf

  4*(2+3)
- Parser input

  NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR
- Parser output (AST):



Parse tree for the example



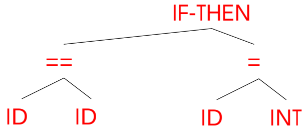
leaves are tokens

Another example

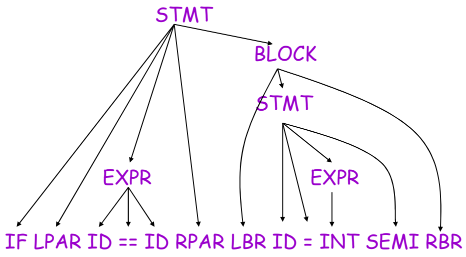
- Decaf

  if (x == y) { a=1; }
- Parser input

  IF LPAR ID EQ ID RPAR LBR ID AS INT SEMI RBR
- Parser output (AST):



Parse tree for the example



leaves are tokens

Parse tree vs. abstract syntax tree

- Parse tree
  - contains all tokens, including those that parser needs "only" to discover
    - intended nesting: parentheses, curly braces
    - statement termination: semicolons
  - technically, parse tree shows concrete syntax
- Abstract syntax tree (AST)
  - abstracts away artifacts of parsing, by flattening tree hierarchies, dropping tokens, etc.
  - technically, AST shows abstract syntax

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	AST, built from parse tree

Summary

- Parser performs two tasks:
  - syntax checking
    - a program with a syntax error may produce an AST that's different than intended by the programmer
  - parse tree construction
    - usually implicit
    - used to build the AST

How to build a parser for Decaf?

Writing the parser

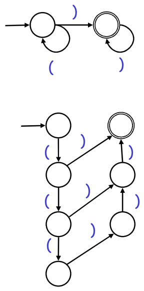
- Can do it all by hand, of course
  - ok for small languages, but hard for Decaf
- Just like with the scanner, we'll write ourselves a parser generator
  - we'll concisely describe Decaf's syntactic structure
    - that is, how expressions, statements, definitions look like
  - and the generator produces a working parser
- Let's start with a hand-written parser
  - to see why we want a parser generator

First example: balanced parens

- Our problem: check the syntax
  - are parentheses in input string balanced?
- The simple language
  - parenthesized number literals
  - Ex.: 3, (4), ((1)), (((2))), etc
- Before we look at the parser
  - why aren't finite automata sufficient for this task?

Why can't DFA/NFA's find syntax errors?

- When checking balanced parentheses, FA's can either
  - accept all correct (i.e., balanced) programs but also some incorrect ones, or
  - reject all incorrect programs but also reject some correct ones.
- Problem: finite state
  - can't count parens seen so far



Parser code preliminaries

- Let TOKEN be an enumeration type of tokens:
  - INT, OPEN, CLOSE, PLUS, TIMES, NUM, LPAR, RPAR
- Let the global in[] be the input string of tokens
- Let the global next be an index in the token string

Parsers use stack to implement infinite state

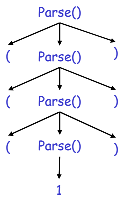
Balanced parentheses parser:

```
void Parse() {
   nextToken = in[next++];
   if (nextToken == NUM) return;

   if (nextToken != LPAR)  print(“syntax error”);
   Parse();
   if (in[next++] != RPAR) print(“syntax error”);
}
```

Where’s the parse tree constructed?

- In this parser, the parse is given by the call tree:
- For the input string (((1))) :



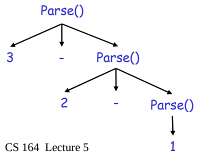
Second example: subtraction expressions

The language of this example:
1, 1-2, 1-2-3, (1-2)-3, (2-(3-4)), etc

```
void Parse() {
   if (in[++next] == NUM) {
         if (in[++next] == MINUS)  { Parse(); }
   } else if (in[next] == LPAR) {
         Parse();
         if (in[++next] != RPAR) print(“syntax error”);
   } else print(“syntax error”);
}
```

Subtraction expressions continued

- Observations:
  - a more complex language
    - hence, harder to see how the parser works (and if it works correctly at all)
  - the parse tree is actually not really what we want
    - consider input 3-2-1
    - what’s undesirable about this parse tree’s structure?



We need a clean syntactic description

- Just like with the scanner, writing the parser by hand is painful and error-prone
  - consider adding +, *, / to the last example!
- So, let’s separate the what and the how
  - what: the syntactic structure, described with a context-free grammar
  - how: the parser, which reads the grammar, the input and produces the parse tree