

Building a Parser I

CS164
3:30-5:00 TT
10 Evans

Prof. Bodik CS 164 Lecture 5

1

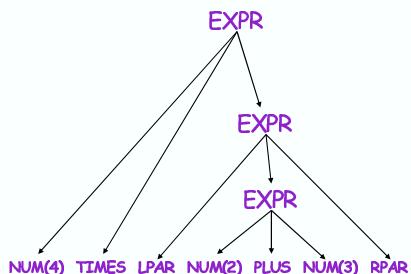
Recall: Syntactic Analysis

- 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.

Prof. Bodik CS 164 Lecture 5

7

Parse tree for the example

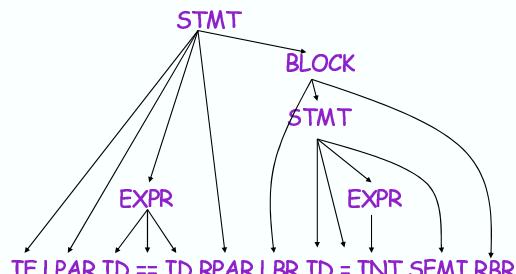


leaves are tokens

Prof. Bodik CS 164 Lecture 5

9

Parse tree for the example

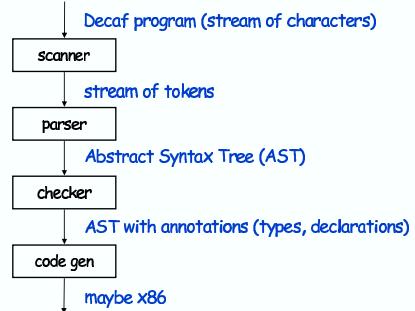


leaves are tokens

Prof. Bodik CS 164 Lecture 5

11

Recall: The Structure of a Compiler

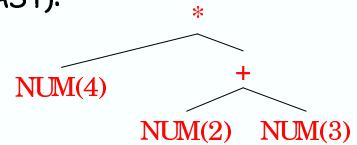


Prof. Bodik CS 164 Lecture 5

6

Example

- Decaf
 $4 * (2 + 3)$
- Parser input
 $\text{NUM}(4) \text{ TIMES } \text{IPAR } \text{NUM}(2) \text{ PILS } \text{NUM}(3) \text{ RPAR}$
- Parser output (AST):

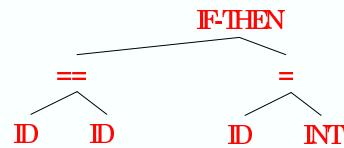


Prof. Bodik CS 164 Lecture 5

8

Another example

- Decaf
 $\text{if } (x == y) \{ a=1; \}$
- Parser input
 $\text{IF } \text{IPAR } \text{ID } \text{EQ } \text{ID } \text{RPAR } \text{IBR } \text{ID } \text{AS } \text{INT } \text{SEMI } \text{RBR}$
- Parser output (AST):



Prof. Bodik CS 164 Lecture 5

10

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

Prof. Bodik CS 164 Lecture 5

12

Comparison with Lexical Analysis

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

Prof. Bodik CS 164 Lecture 5

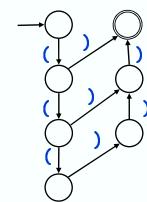
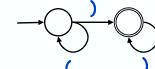
13

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?

Prof. Bodik CS 164 Lecture 5

17



18

Parser code preliminaries

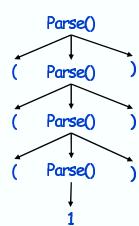
- Let TOKEN be an enumeration type of tokens:
 - INT, OPEN, CLOSE, PLUS, TIMES, NUM, LPAR, RPAR
- Let the global in[] be the input string of tokens
- Let the global next be an index in the token string

Prof. Bodik CS 164 Lecture 5

19

Where's the parse tree constructed?

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



Prof. Bodik CS 164 Lecture 5

21

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

Prof. Bodik CS 164 Lecture 5

14

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

Prof. Bodik CS 164 Lecture 5

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");
}
```

Prof. Bodik CS 164 Lecture 5

20

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");
}
```

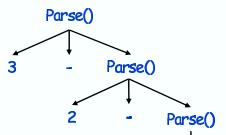
Prof. Bodik CS 164 Lecture 5

22

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?



Prof. Bodik CS 164 Lecture 5

23