CS 536 Announcements for Monday, February 19, 2024

Programming Assignment 2

- due Tuesday, February 20

Last Time

- syntax-directed translation
- abstract syntax trees

Today

- implementing ASTs end of Midterm I material

Thur Feb 29

Next Time

- Java CUP

SDT review

SDT = translating from a sequence of tokens into a sequence of actions/other form, based on underlying syntax

To define a syntax-directed translation

- augment CFG with ***translation rules*** lhs → rhs contains terms, non-terms, ε
  - define translation of LHS non-terminal as a function of:
    - constants 2, " "
    - translations of RHS non-terminals rhs.trans
    - values of terminals (tokens) on RHS TOKEN.value

To translate a sequence of tokens using SDT (conceptually)

- build parse tree
- use translation rules to compute translation of each non-terminal (bottom-up)
- translation of sequence of tokens = translation of parse tree's root non-terminal

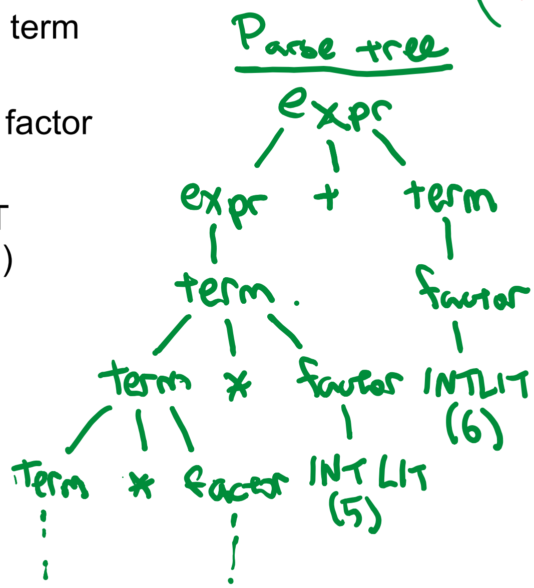
For parsing, we'll need to translate tokenized stream to **abstract-syntax tree (AST)**

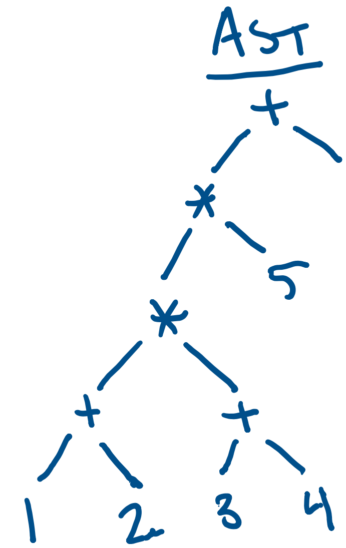
Example ((1+2) * (3+4)) * 5 + 6

expr → expr + term
      | term

term → term * factor
      | factor

factor → INTLIT
      | (expr)





translation rule:

$expr_1.trans = MkPlusNode(expr_2.trans, term.trans)$

AST for parsing

We've been showing the translation in two steps:

token stream → parse tree → AST   then throw away parse tree

In practice we'll do

token stream → AST

Why have an AST?

- captures essential structure
- easier to work with

AST implementation

expr → expr + term $\quad$ $expr_1.trans$ = mkPlusNode($expr_2.trans$, term.trans)

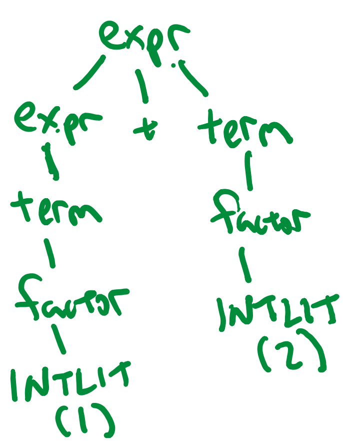
Define a class for each kind of AST node

Create a new node object in some rules

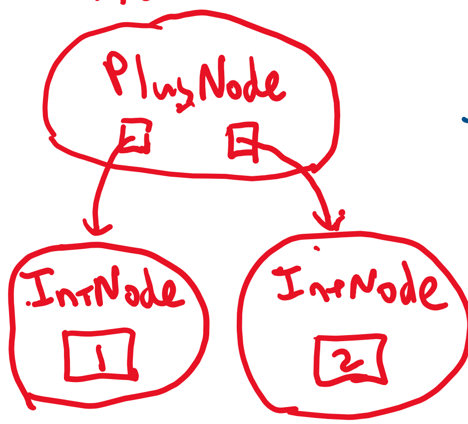
- new node object is the value of LHS.trans
- fields of node object come from translations of RHS non-terminals

Given 1+2

Parse tree



AST




```
class PlusNode {
  IntNode left;
  IntNode right;
}

class IntNode {
  int value
}
```

Need class hierarchy

& make these subclasses of ExpNode

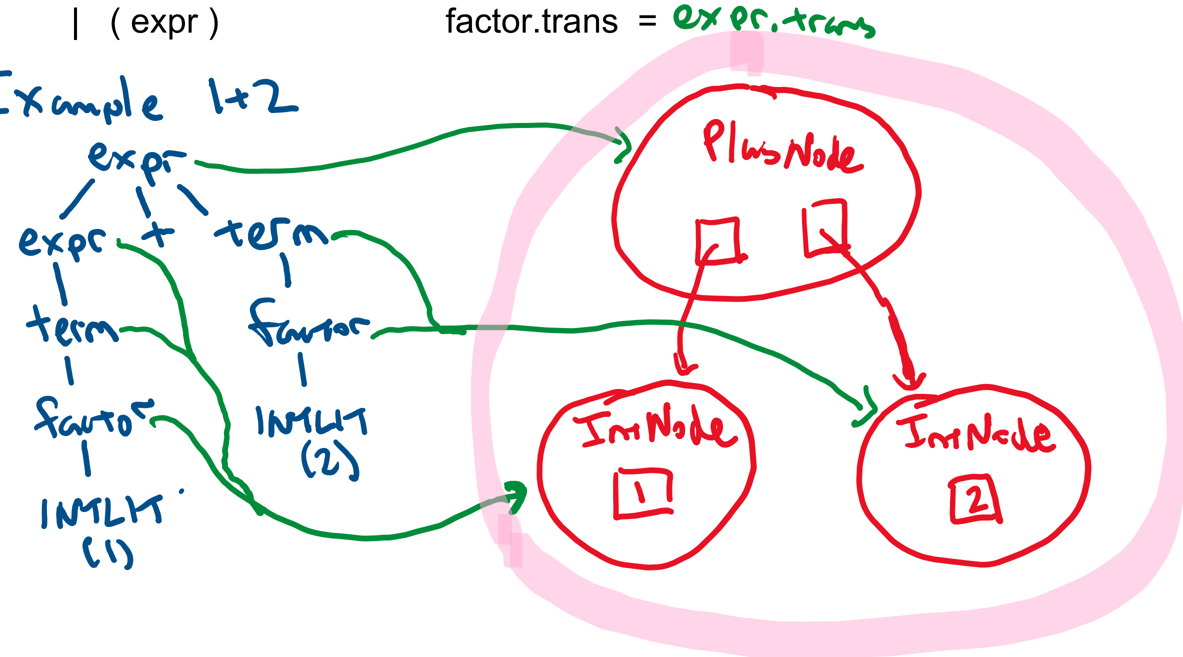
```
class PlusNode extends ExpNode {
    ExpNode left;
    ExpNode right;
}
class IntNode extends ExpNode {
    int value;
}
```

put into ExpNode

Translation rules to build ASTs for expressions

CFG			Translation rules
expr	→	expr + term	$expr_1$.trans = new PlusNode($expr_2$.trans, term.trans)
	|	term	expr.trans = term.trans
term	→	term * factor	$term_1$.trans = new TimesNode($term_2$.trans, factor.trans)
	|	factor	term.trans = factor.trans
factor	→	INTLIT	factor.trans = new IntNode(INTLIT.value);
	|	(expr)	factor.trans = expr.trans

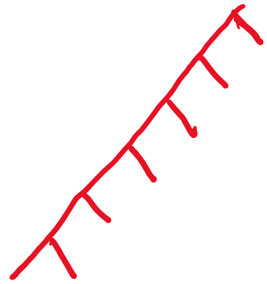
Example 1+2



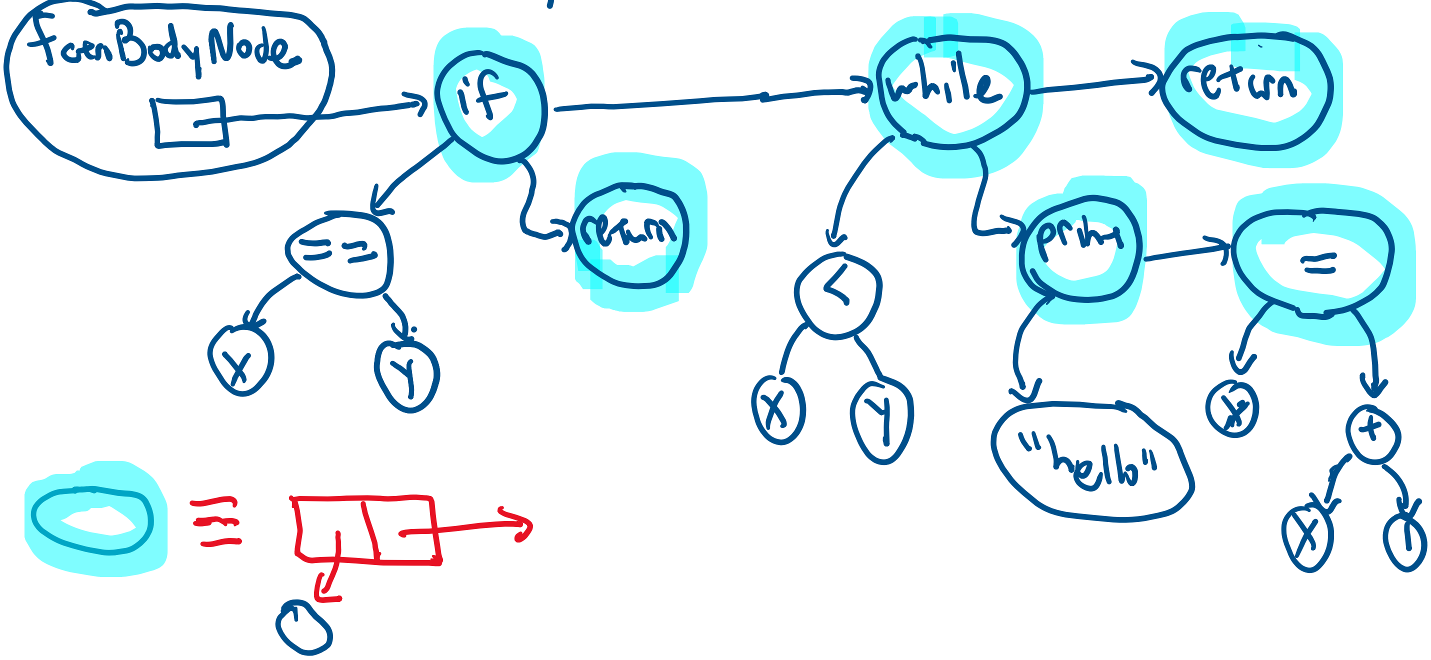
ASTs for non-expressions

Example

```
void foo(int x, int y) {
    if (x == y) {
        return;
    }
    while (x < y) {
        cout << "hello";
        x = x + 1;
    }
    return;
}
```



AST for function body



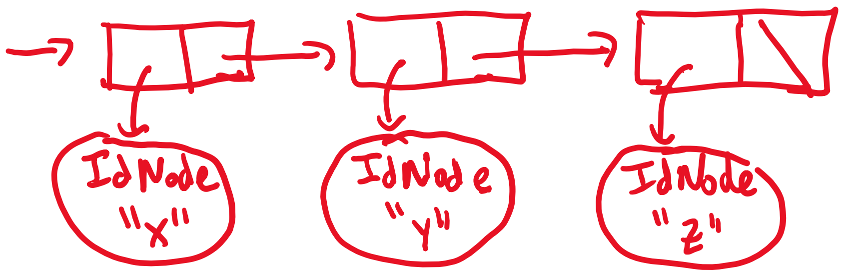
ASTs for lists

<u>CFG</u>

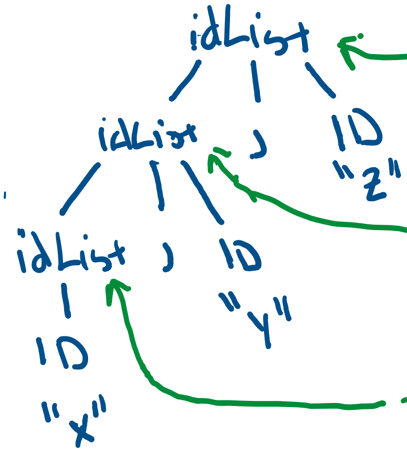
idList → idList COMMA ID
        | ID

Input   x, y, z

Want AST to be



Parse tree



translation adds IdNode for "z" to end of list

translation adds IDNode for "y" to end of list

translation is a new list with IDNode for "x"

The bigger picture

Scanner

- **Language abstraction**: regular expressions
- **Output**: token stream
- **Tool**: JLex
- **Implementation**: interpret DFA using table (for δ), recording most_recent_accepted_position & most_recent_token

Parser

- **Language abstraction**: CFG
- **Output**: AST (by way of a syntax-directed translation)
- **Tool**: JavaCUP ← next time
- **Implementation**: ??? ← next couple weeks