CS 536 Announcements for Wednesday, February 14, 2024

Programming Assignment 2

- due Tuesday, February 20

Hw 2 is available

Last Time

- Makefiles
- ambiguous grammars
- grammars for expressions
  - precedence
  - associativity
- grammars for lists

Today

- syntax-directed translation
- intro to abstract syntax trees

Next Time

- implementing ASTs

Recall our expression grammar

Write an unambiguous grammar for integer expressions involving only addition, multiplication, and parentheses thate correctly handles precedence and associativity.

```
expr   →  expr PLUS term
       |  term
term   →  term TIMES factor
       |  factor
factor →  INTLIT
       |  LPAREN expr RPAREN
```

exponentiation (^)
- has highest precedence (of +, *, ^)
- right associative

Extend this grammar to add exponentiation (POW)

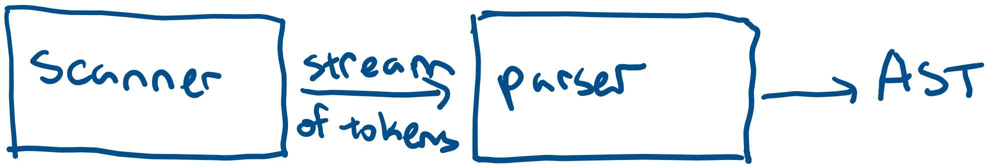
$2 \wedge 3 \wedge 4 = 2 \wedge (3 \wedge 4)$ $\quad 2^{3^4}$

Add exponentiation (POW) to this grammar, with the correct precedence and associativity.

```
factor   → ~~factor~~ exponent POW factor     ← for associativity
         | exponent

exponent → INTLIT
         | ( expr )
```

Overview of CFGs



CFGs for language definition

- the CFGs we've discussed can generate/define languages of valid strings

start by building parse tree & end with some valid string $w \in L(G)$

CFGs for language recognition

start with string w & end with yes/no answer depending on whether $w \in L(G)$

CFGs for parsing

start with string w & end with parse tree for w if $w \in L(G)$

generally use AST instead of parse tree

– need to translate sequence of tokens (w)

Syntax-directed translation (SDT)

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

Could be: AST, value, type, etc.

To define a syntax-directed translation

Augment CFG with ***translation rules*** (at most 1 rule per production) ↳ LHS → RHS

- define translation of LHS non-terminal as a function of
  - constants
  - translations of RHS non-terminals
  - values of tokens (terminals) on RHS

To translate a sequence of tokens using SDT *

- build parse tree
- use translation rules to compute translation of each non-terminal in parse tree bottom-up ← handle children of node before node
- translation of sequence of tokens is the translation of the parse tree's root non-terminal (ie, start symbol)

The **type** of the translation can be anything: numeric, string, set, tree, ...

*Note: above is how to understand the translation, not how a compiler actually does it

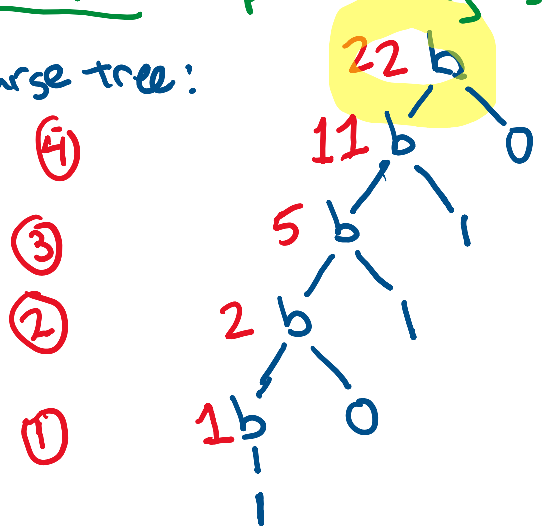
Example: grammar for language of binary numbers

SDT to compute the decimal equivalent of a binary number

CFG	translation rules
b → 0	b.trans = 0
b → 1	b.trans = 1
b → b 0	b_1.trans = b_2.trans * 2
b → b 1	b_1.trans = b_2.trans * 2 + 1

Example: input string is 10110

Parse tree:



Translation is 22

Example: grammar for language of variable declarations

CFG			Translation rules
declList	→	ε	declList.trans = ""
	|	decl declList$_2$	declList$_1$.trans = decl.trans + " " + declList$_2$.trans
decl	→	type ID ;	decl.trans = ID.value
type	→	INT	
	|	BOOL	

string concatenation

Write a syntax-directed translation for the CFG given above so that the translation of a sequence of tokens is a string containing the ID's that have been declared.

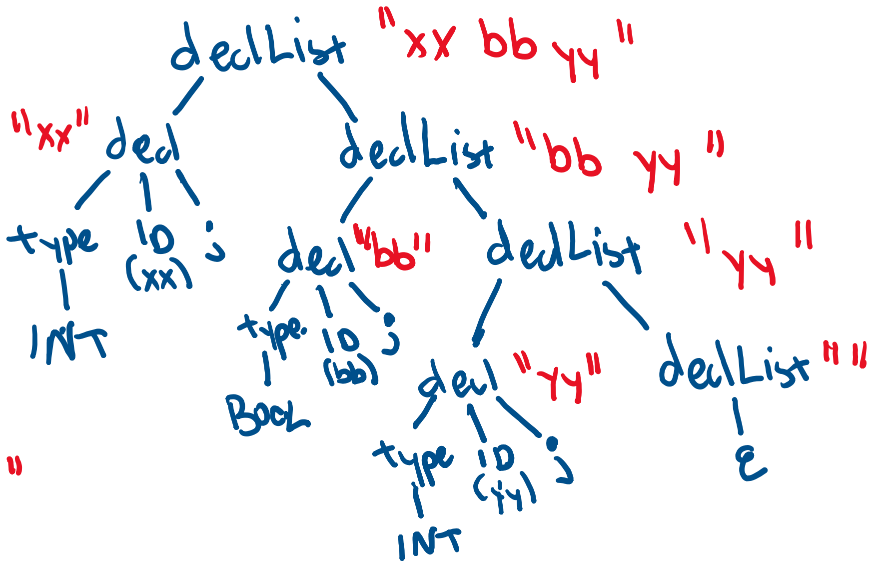
Example input

int xx;
bool bb;
int yy;

Output

"bb xx yy"
(in any order)

Parse tree



Translation is "xx bb yy"

Example: grammar for language of variable declarations

CFG			Translation rules
declList	→	ε	declList.trans = ""
	|	decl declList	$declList_1$.trans = decl.trans + " " + $declList_2$.trans
decl	→	type ID ;	decl.trans = type.trans ? ID.value : "" *
type	→	INT	type.trans = true
	|	BOOL	type.trans = false

Modify the previous syntax-directed translation so that only declarations of type `int` are added to the output string.

Example input

int xx;
bool bb;
int yy;

Output

"xx yy"
(in any order)

Note:
1) different nonterm can have different types as their translation
2) translation rules can be conditional

* x = a ? b : c; equiv to

```
if (a)
    x = b;
else
    x = c;
```

SDT for parsing

Previous examples showed SDT process assigning different types to the translation

- translate tokenized stream to an integer value
- translate tokenized stream to a string

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

Abstract syntax trees

AST = condensed form of parse tree

- Operators at internal nodes (not leaves)
- chains of productions are collapsed
- list are flattened
- syntactic details are omitted

  ↳ eg ; parens

AST Example

<u>CFG</u>

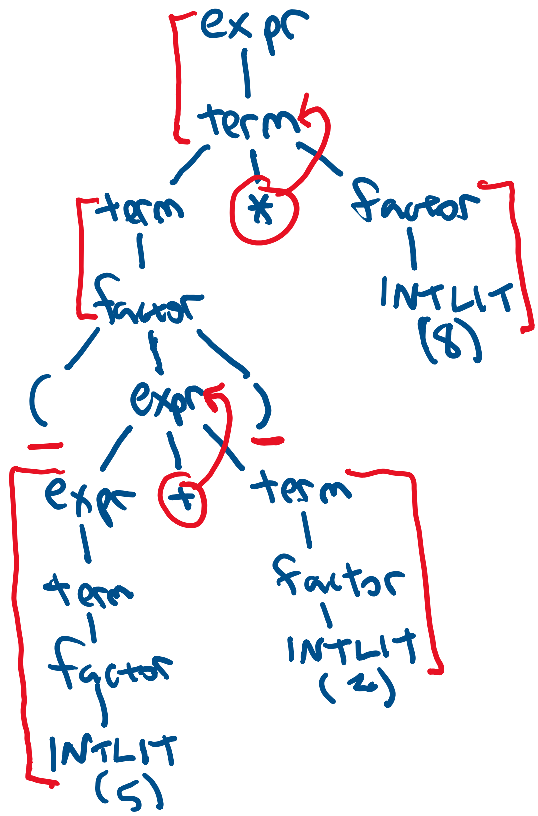
expr → expr PLUS term
    | term

term → term TIMES factor
    | factor

factor → INTLIT
    | LPAREN expr RPAREN

(5+2)*8

Parse tree



AST