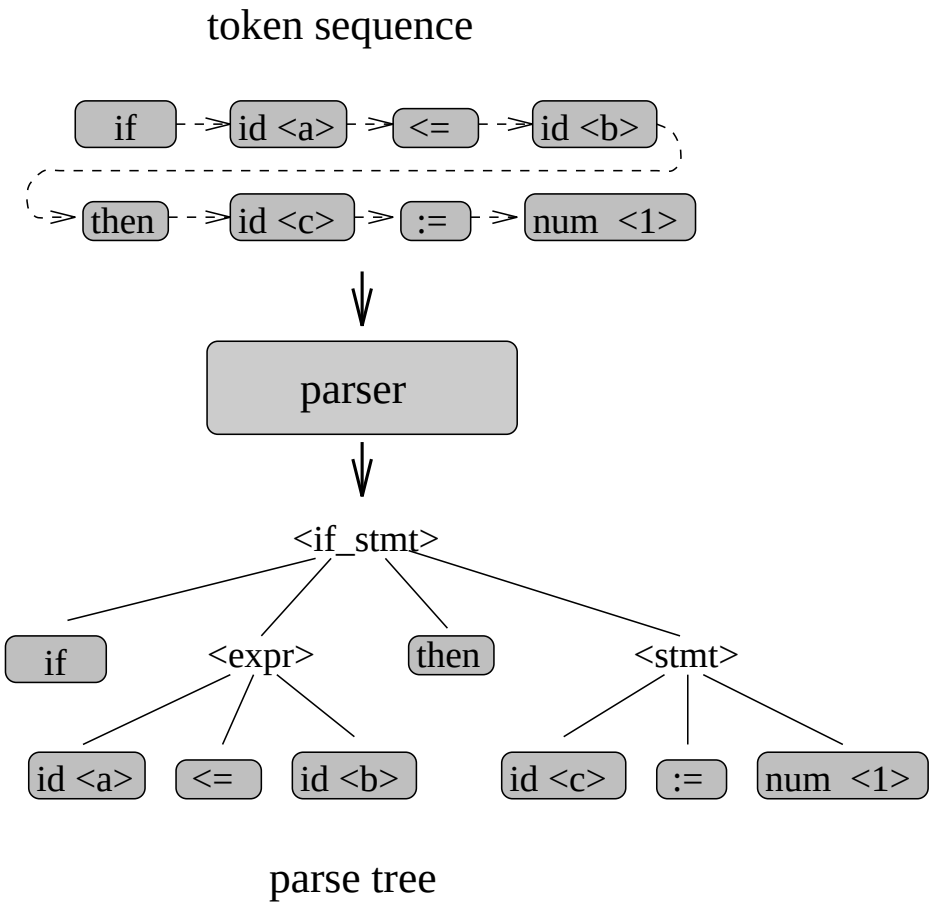
Class Announcements

- First homework deadline extension: Thursday, February 2 , 11:59pm
- Reminder: Please do not post answers or partial answers to homework questions on piazza.
- Everyone should have access to canvas and piazza, including students who signed up using SPNs.

Syntax Analysis (Scott 2.3 - 2.5 - skip 2.3.4)



Review - Context Free Grammars (CFGs)

- A formalism for describing languages
- CFGs are a quadruple $< T, N, P, S >$:
  1. A set T of terminal symbols (tokens)
  2. A set N of nonterminal symbols
  3. A set P production rules
  4. A special start symbol S

CFGs are rewrite systems with restrictions on the structure of rewrite (production) rules that can be used

All productions in P have the following structure:

$X \rightarrow Y$ where

$X \in N$ (single non-terminal symbol) and

$Y \in (N|T)^*$ (any combination of terminals and non-terminals, including ϵ)

Review - Language specified by a given CFG

Given a CFG (context-free grammar) G.

The language L(G) is the set of sentences $w \in T^*$ which can be derived from the unique start symbol S in finitely many rewrite steps (finitely many rule applications):

$$\mathrm{L(G)} = \{\ w \in T^* \mid S \Rightarrow^* w\}$$

The entire sequence of rewrite steps is called a **derivation** ($\Rightarrow$). $\Rightarrow^*$ means 0 or multiple rewrite steps.

To make a derivation "more systematic", an order on rule applications is imposed:

rightmost derivation ($\Rightarrow_R$) : always replace the rightmost non-terminal symbol next

leftmost derivation ($\Rightarrow_L$) : always replace the leftmost non-terminal symbol next

Review - CFGs specified as a BNF Grammar ($\mathcal{G}$)

Terminals letters, digits, `:=`

Nonterminals <letter> <digit> <identifier> <stmt>

Productions

1. <letter> ::= A | B | C | . . . | Z
2. <digit> ::= 0 | 1 | 2 | . . . | 9
3. <identifier> ::= <letter> |
   <identifier> <letter> |
   <identifier> <digit>
4. <stmt> ::= <identifier> := 0

Start Symbol <stmt>

BNF is a convenient way to specify CFGs, i.e., without using the quadruple $< T, N, P, S >$ formulation.

Review - Elements of BNF Syntax

BNF (Backus-Naur Form): A formal notation for describing a context-free grammar.

Terminal Symbol: **Symbol-In-Boldface**

Non-Terminal Symbol: *Symbol-In-Angle-Brackets*

Production Rule:

Non-Terminal ::= Sequence of Symbols

or

Non-Terminal ::= Sequence | Sequence | ...

Alternative Symbol: |

Empty String: ϵ

Derivation in a Grammar

Is `X2 := 0` $\in L(G)$, i.e., can `X2 := 0` be derived in G?

leftmost derivation		rule applied
<stmt>	$\Rightarrow_L$	4
<identifier> := 0	$\Rightarrow_L$	3c
<identifier><digit> := 0	$\Rightarrow_L$	3a
<letter><digit> := 0	$\Rightarrow_L$	1
X <digit> := 0	$\Rightarrow_L$	2
X2 := 0		

rightmost derivation		rule applied
<stmt>	$\Rightarrow_R$	4
<identifier> := 0	$\Rightarrow_R$	3c
<identifier><digit> := 0	$\Rightarrow_R$	2
<identifier> 0 := 0	$\Rightarrow_R$	3a
<letter> 0 := 0	$\Rightarrow_R$	1
X2 := 0		

Parsing in a Grammar

Can we **recognize** `X2 := 0` as being in L(G)?

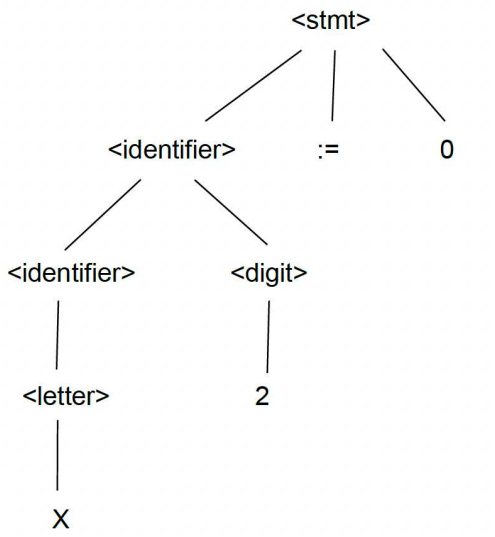
In other words, can we write a program, called **a parser for L(G)**, that given an input string of terminal symbols (tokens) can construct a derivation (leftmost or rightmost) for the input using rules in G? If a deriviation does not exist, the program should report an error.

Note: Different parsing techniques, i.e., the automatic recognition sentences $w \in L(G)$ will be discussed in more detail in `198:415 Compilers`.

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

Parse Trees

A *parse tree* of `X2 := 0` in G:



Each internal node is a nonterminal; its children are the RHS of a production for that nonterminal.

The parse tree demonstrates that the grammar generates the terminal string on the frontier.

Note: Different parsing techniques, i.e., the automatic recognition sentences $w \in L(G)$ will be discussed in `198:415 Compilers`.

Grammars are not Unique

Consider $\mathcal{G}'$:

Terminals letters, digits, `:=`

Nonterminals <letter> <digit> <ident> <stmt>
<letterordigit>

Productions

1. <letter> ::= A | B | C | . . . | Z
2. <digit> ::= 0 | 1 | 2 | . . . | 9
3. <ident> ::= <letter> |
   <ident> <letterordigit>
4. <stmt> ::= <ident> := 0
5. <letterordigit> ::= <letter> | <digit>

Start Symbol <stmt>

$\mathcal{G}$ and $\mathcal{G}'$ generate the same language, but yield different parse trees.

Example: A *parse tree* of `X2 := 0` in $\mathcal{G}'$.

Grammars and Programming Languages

Many grammars may correspond to one programming language.

Good grammars:

- capture the logical structure of the language,
- use meaningful names,
- are easy to read,
- should be unambiguous if we want to attach meaning to structure
- suitable for different parsing strategies (LL(k), LR(k), LALR(1))
- ...

Simple Statement Grammar

<start> ::= <stmt>

<stmt> ::= <if-stmt> | <assgn>

<if-stmt> ::= **if** <expr> **then** <stmt> |
if <expr> **then** <stmt> **else** <stmt>

<assgn> ::= <id> := <d>

<expr> ::= <id> = 0

<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<id> ::= a | b | c | . . . | z

Dangling Else Ambiguity

How are nested **if** statements parsed with this grammar?

if x = 0 **then if** y = 0 **then** z := 1 **else** z := 2

Definition of Ambiguous Grammars

"Time flies like an arrow; fruit flies like a banana."

A grammar $\mathcal{G}$ is ambiguous iff there exist a $w \in L(\mathcal{G})$ such that there are

1. two distinct parse trees for w, or
2. two distinct leftmost derivations for w, or
3. two distinct rightmost derivations for w.

We want a unique semantics of our programs, which typically requires a unique syntactic structure.

Ambiguity

How to deal with ambiguity?

1. Change the language to include **delimiters**
2. Change the grammar to impose **associativity** and **precedence**

Example: Changing the Language to Include Delimiters

Algol 68 if statement:

<if-stmt> ::= **if** <expr> **then** <stmt> **fi** |
if <expr> **then** <stmt>
else <stmt>
fi

Let's look at this issue in the context of an arithmetic expression grammar.

Arithmetic Expression Grammar

<start> ::= <expr>

<expr> :: = <expr> + <expr> |

<expr> - <expr> |

<expr> * <expr> |

<expr> / <expr> |

<expr> ^ <expr>|

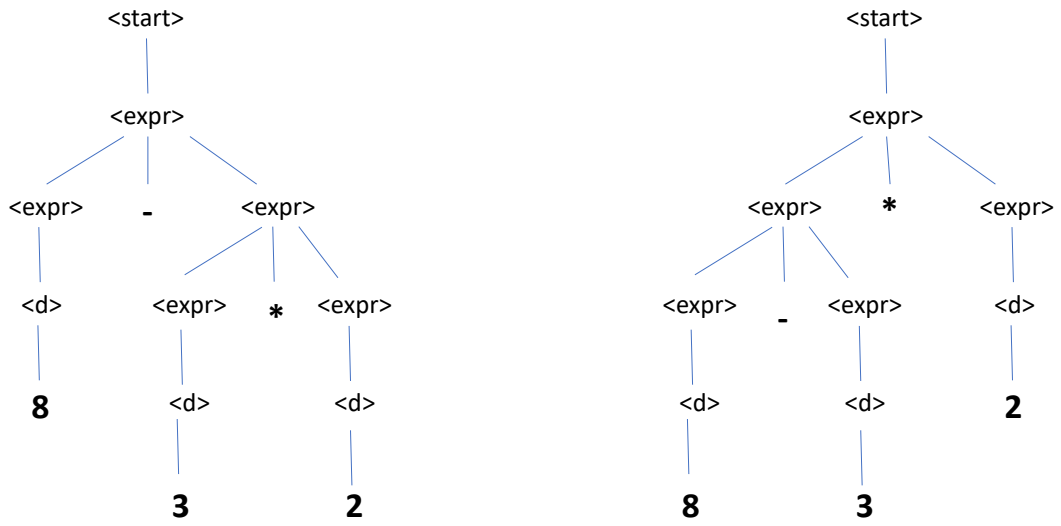
<d> | <l>

<d> :: = 0 | 1 | 2 | 3 | . . . | 9

<l> :: = a | b | c | . . . | z

Possible Parse Trees

Parse "$8 - 3 * 2$":



There are two parse trees for the same input string. Each parse tree is represented by a unique leftmost and unique rightmost derivation. Therefore, there are two leftmost and two rightmost derivations.

Changing the Language to Include Delimiters

<expr> ::= (<expr>) − (<expr>) |

(<expr>) ∗ (<expr>) |

<l> | <d>

```
(8)-((5)*(2))
((8)-(5))*(2)
```

Pretty ugly, isn't it? Is there any other way to disambiguate our expression grammar?

Changing the Grammar to Impose Precedence

<expr> ::= <expr> − <expr> |

<term>

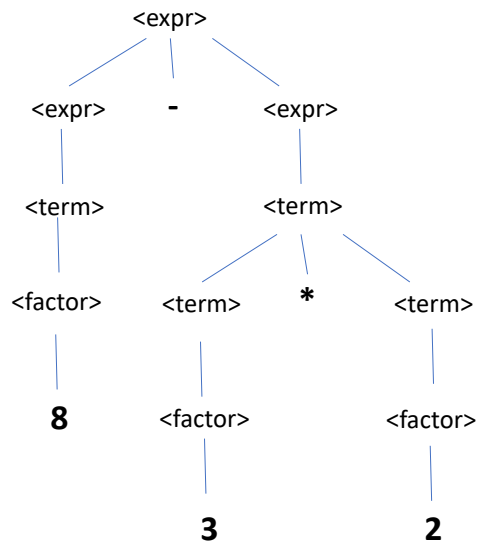
<term> ::= <term> ∗ <term> |

<factor>

<factor> ::= 0 | 1 | 2 | 3 | . . . | 9

Grouping In Parse Tree Now Reflects Precedence

Parse "$8 - 3 * 2$":



There is only a single possible parse tree for the input string. By changing the grammar, we got a parse tree structure that is (1) unique and (2) matches the operator precedence that we want.

Precedence

- Low Precedence:

  Addition + and Subtraction −
- Medium Precedence:

  Multiplication ∗ and Division /
- Highest Precedence:

  Exponentiation ^

⇒ Ordered lowest to highest in grammar.

Next Lecture

More on ambiguity

Regular and context-free languages

Top-down parsing, FIRST and FOLLOW sets, LL(1) grammars, recursive descent parsing

Things to do:

- read Scott, Ch. 2.3 - 2.5 (skip 2.3.3 Bottom-up Parsing)