Ambiguity and Parsing Algorithms

CSCI 3136: Principles of Programming Languages

Agenda

- Ambiguous Grammars
- Left and Right Parse Tree Derivations
- LL(K) and LR(K) Parsing
- S-Grammars
- LL(1) Parsing
- LR(1) Parsing

Recall: A CFG for Expressions

- V = {E, Op} Non-Terminals / Variables
- Σ = {identifier, number, (,), +, −, X, /} Terminals / Alphabet
- P={ Productions

  E → E Op E LHS → RHS

  E → −E

  E → (E)

  E → number

  E → identifier

  Op → +

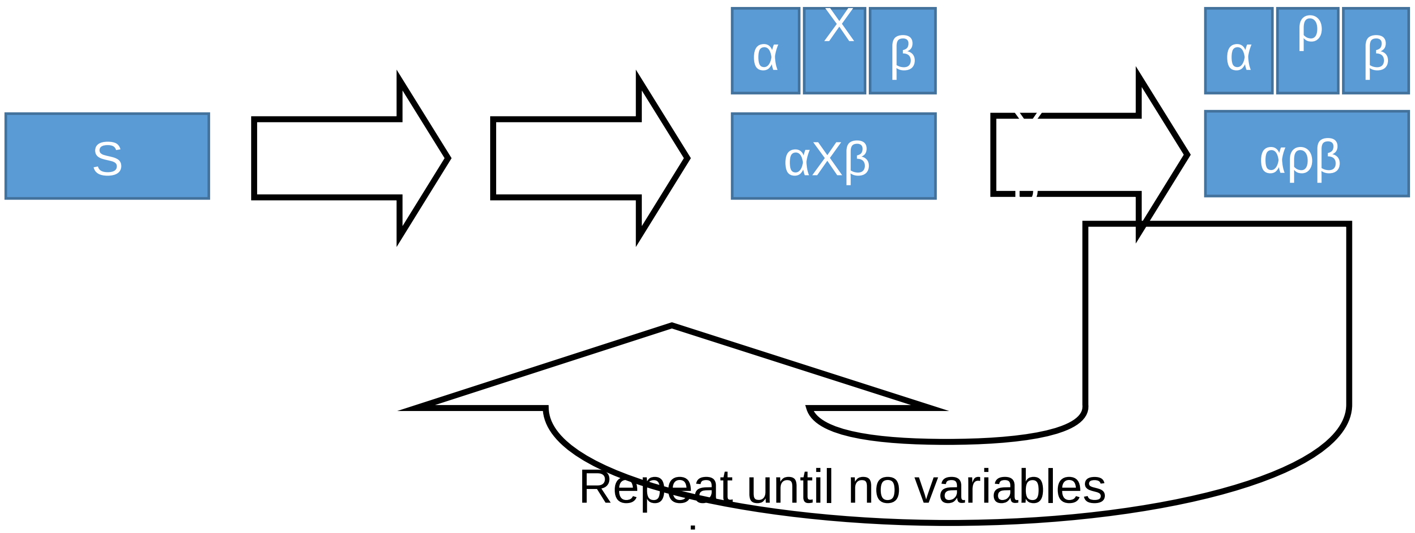
  Op → −

Derivations

- An input string is a stream or sequence of elements from the Σ, the Alphabet (i.e., input is a sequence of non-terminals).
- The act of matching the input string to the grammar is called "parsing."
- The result of parsing is a "derivation" – sequence of grammar rule applications that produce the input string.
- A derivation is expressed as a parse tree or AST (abstract syntax tree).

Derivations in a Nutshell

Start with S.
Replace a variable (rule LHS) with its rule RHS, matching terminals as we go.
Repeat until no variables remain.
Problem: Which rule do we use?



Repeat until no variables remain.

Parse Tree of a Derivation (42+13)*11

σ = **E**

□ **E** Op E

□ (**E**) Op E

□ (**E** Op E) Op E

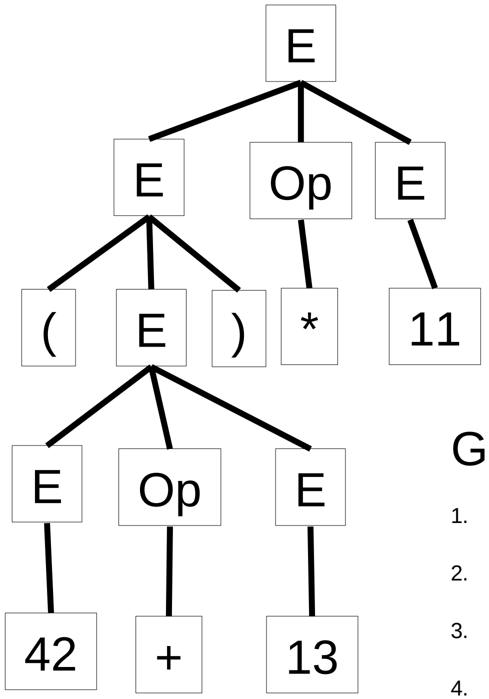
□ (42 **Op** E) Op E

□ (42 + **E**) Op E

□ (42 + 13) **Op** E

□ (42 + 13) X **E**

□ (42 + 13) X 11



Grammar

1. E → E Op E
2. E → −E
3. E → (E)
4. E → number
5. E → identifier
6. Op → +
7. Op → −
8. Op → X
9. Op → /

Another Example: 1 + 2 * 3

Grammar

1. E → E Op E
2. E → −E
3. E → (E)
4. E → number
5. E → identifier
6. Op → +
7. Op → −
8. Op → X
9. Op → /

This is ambiguous!

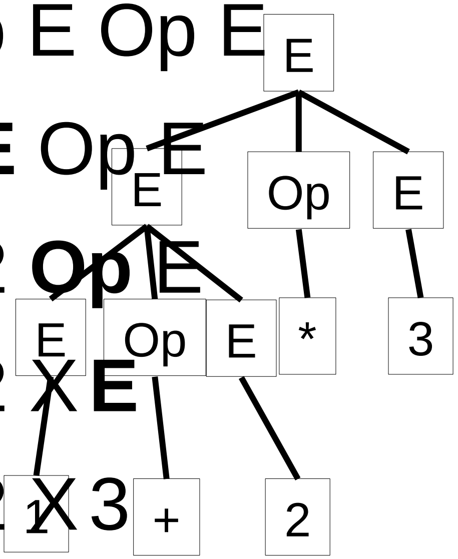
σ = **E**

- **E** Op E
- **E** Op E Op E
- 1 **Op** E Op E
- 1 + **E** Op E
- 1 + 2 **Op** E
- 1 + 2 X **E**
- 1 + 2 X 3

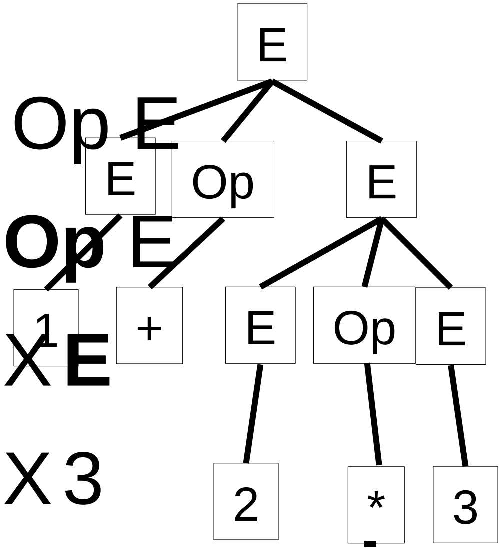
σ = **E**

- **E** Op E
- 1 **Op** E
- 1 + **E**
- 1 + **E** Op E
- 1 + 2 **Op** E
- 1 + 2 X **E**
- 1 + 2 X 3

(1 + 2) * 3



1 + (2 * 3)



Ambiguity

- Observations:
  - There are infinitely many grammars to specify the same language.
  - There may be multiple parse trees (derivations) for the same input string!
- Definition: If multiple parse trees can be generated by a grammar, G, for the same input string, then G is *ambiguous*.
- Definition: If L does not have an unambiguous grammar, then L is *inherently ambiguous*
  - Usually not the case for programming languages!

Detecting Ambiguity

- There is no algorithm to detect ambiguity in a grammar (undecidable, equivalent to the PCP).
- It has to be proved by creating an input which has multiple derivations.
- Done by trial and error, experience, intuition.
- Usually not too difficult from examining the grammar.
- Same thing for inherent ambiguity – no algorithm exists.
- Example of inherently ambiguous language:

[illegible]

An Unambiguous Grammar for Expressions

Grammar

1. $E \rightarrow T$
2. $E \rightarrow E + T$
3. $E \rightarrow E - T$
4. $T \rightarrow F$
5. $T \rightarrow T \times F$
6. $T \rightarrow T / F$
7. $F \rightarrow$ number
8. $F \rightarrow$ identifier

- Try deriving 1 + 2 + 3
- The "Trick" – we only allow one recursive symbol in the RHS of a production.

Derivation Order

- Derivation orders refer to the order in which variables are replaced in the current partial derivation.
- The two most common ones are:
  - **Leftmost derivation** replaces the leftmost variable in each step
  - **Rightmost derivation** replaces the rightmost variable in each step
- Different orders for derivations do not lead to ambiguity ... they generate the SAME result (if G is not ambiguous) just in different ways

Leftmost Derivation Example 1+2*3

E ⇒ **E** + T

⇒ **T** + T

⇒ **F** + T

⇒ 1 + **T**

⇒ 1 + **T** X F

⇒ 1 + **F** X F

⇒ 1 + 2 X **F**

⇒ 1 + 2 X 3

Grammar

1. E → T
2. E → E + T
3. E → E − T
4. T → F
5. T → T X F
6. T → T / F
7. F → number
8. F → identifier

Rightmost Derivation Example 1+2*3

E □ E + **T**

□ E + T X **F**

□ E + **T** X 3

□ E + **F** X 3

□ **E** + 2 X 3

□ **T** + 2 X 3

□ **F** + 2 X 3

□ 1 + 2 X 3

Grammar

1. E → T
2. E → E + T
3. E → E − T
4. T → F
5. T → T X F
6. T → T / F
7. F → number
8. F → identifier

Sentential Forms

- Consider the grammar:

({S, B}, {a, b}, S, {S → aS, S → B, B → bB, B → ε})

(V, Σ, S, P)

- A (leftmost) derivation of abb might look like:

S □ aS □ aB □ abB □ abbB □ abb

- Each of {S, aS, aB, abB, abbB, abb} is a "sentential" form."
- That is, a sentential form is a partial derivation.
- Formally, it is any string from (V Ç T)* that is

So Far ...

- Languages can have many grammars
- Some grammars are ambiguous, some are not
- A derivation occurs when we parse an input string by matching it to the grammar
- Derivations can be done systematically by replacing the left-most (or right-most) variable in each step

And in this all means ...

- CFGs are used to specify programming language syntax
- Parsing finds the parse tree of the program (token stream)
- CFGs for programming languages must unambiguously capture the language's syntax (i.e., its structure).
- Parsers must be efficient:
  - A parser can be generated from a CFG that runs in O(n3) time
  - We prefer (require) linear time; O(n)

Aside: Linear Time

- A (parsing) algorithm is linear if:

  The time that the algorithm takes to run is linearly dependent on the size of the input.

- i.e., the parsing algorithm will take (about) twice as long if the input (to parse) is twice the size.
- Running time is related to input size by a constant factor (e.g., 2 milliseconds per line).
- This is generally a good situation and is considered efficient.
- Running times are pretty reasonable.

Regular Grammars: A Brief Aside

- A CFG is *right-linear* if all productions are of the form
  - A → σB, σEΣX, BEV
  - A → σ, σEΣX
  - Idea: all variables are RIGHT of terminals in production RHS
- A CFG is *left-linear* if all productions are of the form
  - A → Bσ, σEΣX, BEV
  - A → σ, σEΣX
  - Idea: all variables are LEFT of terminals in

Parsing Regular Grammars

- Regular grammars are too weak to specify most programming languages
- E.g., you can't make $E \rightarrow (E)$ left or right linear
- But, parsers generated from them run in linear time!
  - Why?
  - Intuition: they duplicate a FSA and read the input once only, just changing state for each symbol.
  - Think of making transitions ≡ productions:

$(Q1, a) \rightarrow Q2 \equiv Q1 \rightarrow aQ2$

LL and LR Grammars

Two kinds of unambiguous grammars that can be parsed efficiently:

1. LL(k) grammars
   - Are scanned Left-to-right and generate a Leftmost derivation
   - If the first letter in the current sentential form is a variable, k tokens look-ahead in the input suffice to decide which production to use to expand it.
2. LR(k) grammars
   - Are scanned Left-to-right and generate a rightmost derivation
   - The next k tokens in the input suffice to choose the

S-Grammars

- First let’s consider a very simple grammar
- An *S-grammar* or *simple grammar* is a special case of an LL(1)-grammar
- A CFG is an S-grammar if
  - Every production's RHS starts with a terminal
  - Productions for the same LHS start with different terminals on the RHS

  E.g., If G contains A→aA and A→a then G is not simple!
- Idea: When using S-Grammars, selecting which rule to apply is easy.

Notation

- "Normal" math uses infix notation:

  e.g., 3 + 4

- Infix means the operator (+) goes in between the operands
- To assist in parsing, many languages use an alternative ordering
- Prefix notation (also called Polish Notation) puts the operator first – LISP, Scheme:

  e.g., + 3 4

- Can be more complicated:

  e.g., + - 3 4 5 ≡ (3 – 4) + 5

Example: LL(1) Parsing Grammar for Polish Notation

1. S → + SS
2. S → − SS
3. S → X SS
4. S → / SS
5. S → neg S
6. S → integer

Expression:

```
- * + 1 2 3 4
```

- How do we derive

```
- * + 1 2 3 4
```

S □ − **S** S

□ − X **S** S S

□ − X + **S** S S S

□ − X + 1 **S** S S

□ − X + 1 2 **S** S

□ − X + 1 2 3 **S**

□ − X + 1 2 3 4

LL(1) Parsing of S-Grammars

Left scan, Left-most derivation, 1 symbol lookahead

```
/* Use a stack to store the current
sentential form (start with S) */
push(S)
Loop until no more tokens:
    t = next_token()
    x = pop()
    if x == t:
         /* ToS matches next token */
        continue
    else if x E V:
         /* use t to select RHS of rule for
x */
        select production x → t α
```

Grammar

1. S → + SS
2. S → − SS
3. S → X SS
4. S → / SS
5. S → neg S
6. S → integer

This takes linear time!

Parse Expression:

* + 1 2 3 4

Example: LR(1) Parsing Grammar for Polish Notation

1. S → + SS
2. S → − SS
3. S → X SS
4. S → / SS
5. S → neg S
6. S → integer

PN Expression:

```
- * + 1 2 3 4
```

- How do we derive

```
- * + 1 2 3 4
```

S □ − S **S**

□ − **S** 4

□ − * S **S** 4

□ − X **S** 3 4

□ − X + S **S** 3 4

□ − X + **S** 2 3 4

□ − X + 1 2 3 4

LR(1) Parsing of S-Grammars

Left scan, Right-most derivation, 1 symbol lookahead

```
/* Use a stack to store what has
been seen so far, starting with first
token */
push( next_token() )
Loop until no more tokens:
    if ∃(P → α)  such that α = ToS
        /* reduce operation */
        pop(α)
        push(P)
        add children α to node P
    else:
        /*shift operation */
```

Grammar

1. $S \rightarrow + SS$
2. $S \rightarrow - SS$
3. $S \rightarrow X SS$
4. $S \rightarrow / SS$
5. $S \rightarrow$ neg S
6. $S \rightarrow$ integer

Parse Expression:

This takes linear time!

Building Parsers

- We now have some intuition about parsing algorithms
- But …
  - The algorithms, so far, are for S-Grammars (too simple)
  - Want to generate parser given a CF grammar
- So ...
  - Assume that we will be using more complex grammars
  - How do we generate the parser?

Building an LL(1) Parser

Left scan, **L**eft-most derivation, **1** symbol look ahead

- Basic Challenge: Given current token, which production does the parser select if next item in sentential form is a nonterminal?

  E.g., if S is on the stack and input is +, then parser must select production S → +SS

- In general: for input **a** and sentential form A, either
  - A □ α □ X **a**β
  - A □ α □ X ε and derivation of A is succeeded by **a**.
- Intuitively, **a** is in the *predictor set* of A → α

  if Aβ □ αβ □ X aγ, for β, γ E ΣX

LL(1) Grammars

- **Definition**: A grammar is LL(1) if the predictor sets of all productions with the same LHS are disjoint.
- E.g. S-Grammars are LL(1)

Grammar

1. $S \rightarrow + SS$
2. $S \rightarrow - SS$

Production	Predictor Set
$S \rightarrow + S\ S$	{+}
$S \rightarrow - S\ S$	{-}
$S \rightarrow * S\ S$	{*}
$S \rightarrow / S\ S$	{/}
$S \rightarrow$ neg S	{neg}
$S \rightarrow$ integer	{integer}

Note: Sets s1 and s2 are disjoint
if $s1 \cap s2 = \{\}$
i.e., they have nothing in common