# CS406: Compilers

Spring 2020

Week 4: Parsers

# Parsers - Overview

- Also called syntax analyzers
- Determine two things:

1. If a program is valid syntactically

   - Is an English sentence grammatically correct?

2. Structure of programming language constructs

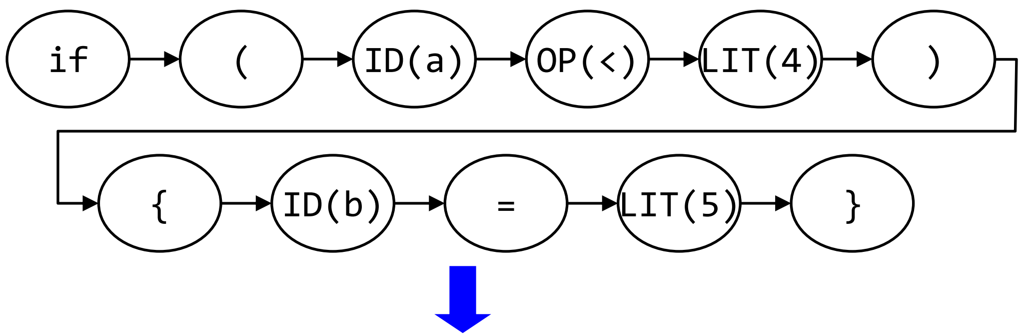
   - E.g. the sequence `IF, ID(a), OP(<), ID(b), {, ID(a), ASSIGN, LIT(5), }, ;, }` refers to `if-statement` ?

   - Diagramming English sentences

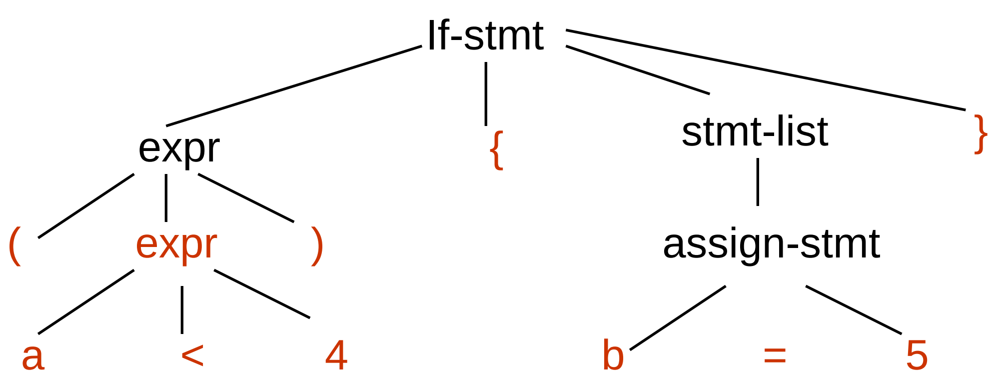
# Parsers - Overview

- Input: stream of tokens
- Output: Parse tree
  - sometimes implicit

Stream of tokens:




Parse tree:



# Parsers – what do we need to know?

1. How do we define language constructs?
   – Context-free grammars
2. How do we determine: 1) valid strings in the language? 2) structure of program?
   – LL Parsers, LR Parsers
3. How do we write Parsers?
   – E.g. use a parser generator tool such as Bison

# Languages

- A language is (possibly infinite) set of strings
- Regular expressions describe *regular languages*

weakness: can’t describe a string of the form:

$$\{\ (^{i}\ )^{i}\ \mid\ i>=1 \}$$

Parenthesized expressions: `((( int x; )))`

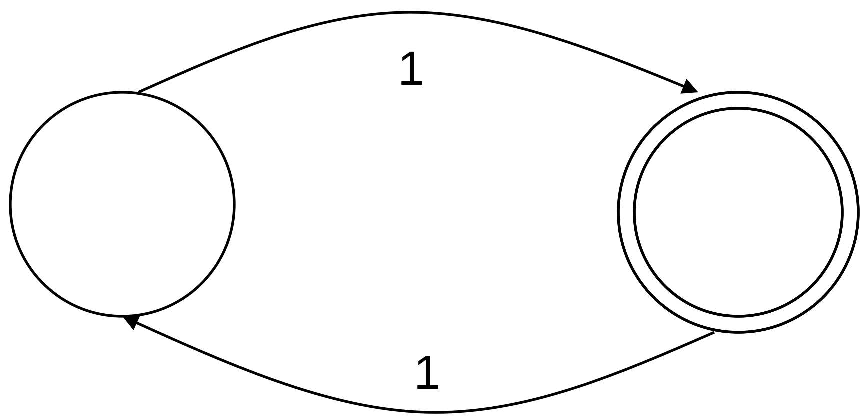
*Programming language syntax is i.e. recursive*

N es:

```
IF
  IF
  FI
IF
FI
```

# Trivia

- Regular expressions can describe strings:
  `{ mod k | k = # states in FA}`




“accept all strings having odd number of 1s”

# Context Free Grammar (CFG)

- Natural notation for describing recursive structure definitions. Hence, suitable for specifying language constructs.
- Consist of:
  - A set of *Terminals*
  - A set of *Non-terminals*
  - A *Start Symbol*
  - A *set of Productions*

# Context Free Grammar (CFG)

- Terminology:

  *Terminals* – T
  *Non-terminals* – N
  *Start Symbol* – S∈N
  *Productions* – P (also called rules sometimes)

  $X \longrightarrow Y_1Y_2Y_3..Y_N \mid X \in N,\ Y_i \in N \cup T \cup \epsilon/\lambda$

- Grammar G = (T, N, S, P)

  E.g. G = ({a,b}, {S, A, B}, S, {S → AB, A → Aa
  A → a, B → Bb, B → b})

- *G is context free. Why?*

# Terminology

- *Strings* are composed of symbols
  - A A a a B b b A a is a string
  - We will use Greek letters to represent strings composed of both terminals and non-terminals
- *L(G)* is the language produced by the grammar *G*
  - All strings consisting of only terminals that can be produced by G
  - In our example, L(G) = a+b+
  - The language of a context-free grammar is a context-free language
  - All regular languages are context-free, but not vice versa

# String Derivations

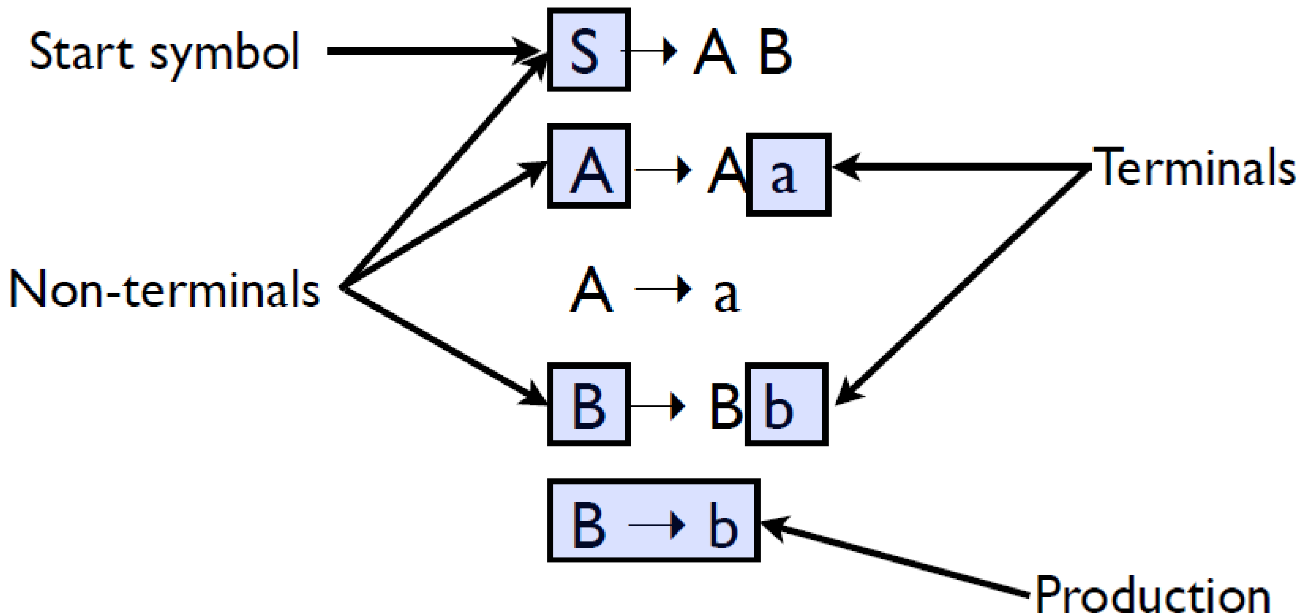
- How do we apply the grammar rules repeatedly to determine the validity of a string? (i.e. string belongs to the language specified by the grammar)

1. Always start with the Start Symbol

2. Replace any Non-terminal X in the string by the right-hand side of the production

3. Repeat Step 2 until there are no more non-terminals

# Simple grammar




*Backus Naur Form (BNF)*

# Generating strings

$S \rightarrow A\ B$

$A \rightarrow A\ a$

$A \rightarrow a$

$B \rightarrow B\ b$

$B \rightarrow b$

- Given a start rule, productions tell us how to rewrite a non-terminal into a different set of symbols
- Some productions may rewrite to $\lambda$. That just removes the non-terminal

To derive the string "a a b b b" we can do the following rewrites:

$$S \Rightarrow A\ B \Rightarrow A\ a\ B \Rightarrow a\ a\ B \Rightarrow a\ a\ B\ b \Rightarrow$$

$$a\ a\ B\ b\ b \Rightarrow a\ a\ b\ b\ b$$

# Exercise

Which of the below strings are accepted by the grammar:

$A \rightarrow aAa$
$A \rightarrow bBb$
$A \rightarrow \lambda$
$B \rightarrow cA$
$B \rightarrow \lambda$

1. abcba
2. abcbca
3. abba
4. abca

# Programming language syntax

- Programming language syntax is defined with CFGs
- Constructs in language become non-terminals
  - May use auxiliary non-terminals to make it easier to define constructs

$$\text{if\_stmt} \rightarrow \text{if ( cond\_expr ) then statement else\_part}$$

$$\text{else\_part} \rightarrow \text{else statement}$$

$$\text{else\_part} \rightarrow \lambda$$

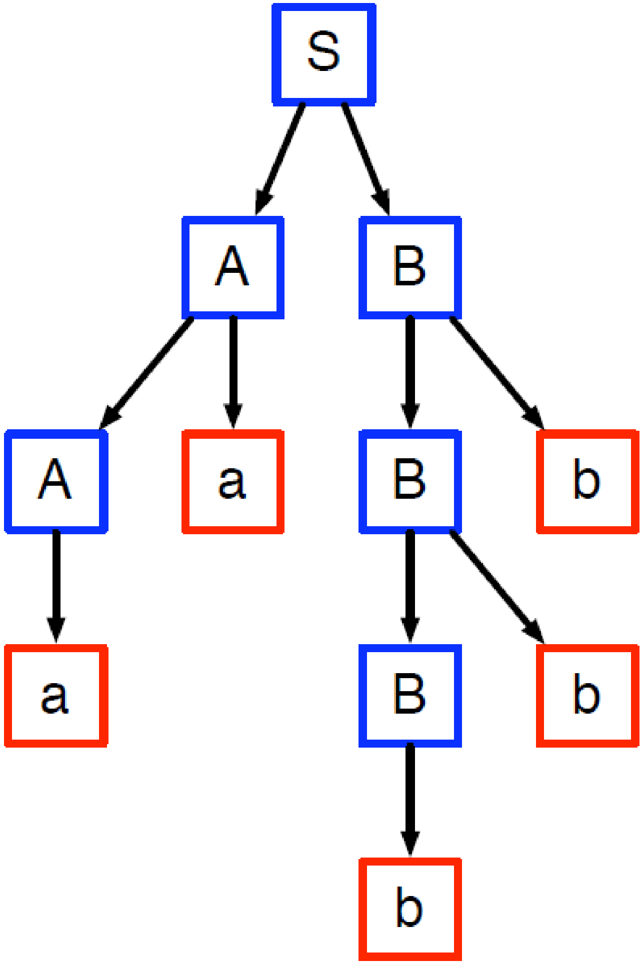
- Tokens in language become terminals

# CFG Contd..

- Is it enough if parsers answer “yes” or “no” to check if a string belongs to context-free language?
  - Also need a parse tree
- What if the answer is a “no”?
  - Handle errors
- How do we implement CFGs?
  - E.g. Bison

# Parse trees

- Tree which shows how a string was produced by a language
  - Interior nodes of tree: non-terminals
    - Children: the terminals and non-terminals generated by applying a production rule
  - Leaf nodes: terminals



# Parse Trees and String Derivations

- Recall: sequence of rules applied to produce a string is a derivation
- A derivation defines a parse tree
  - A parse tree may have many derivations

# Leftmost derivation

- Rewriting of a given string starts with the leftmost symbol
- Exercise: do a leftmost derivation of the input program

  *F(V + V)*

  using the following grammar:

| | | |
|---|---|---|
| E | → | Prefix (E) |
| E | → | V Tail |
| Prefix | → | F |
| Prefix | → | λ |
| Tail | → | + E |
| Tail | → | λ |

- What does the parse tree look like?

# Rightmost derivation

- Rewrite using the rightmost non-terminal, instead of the left
- What is the rightmost derivation of this string?

*F(V + V)*

| | | |
|---|---|---|
| E | → | Prefix (E) |
| E | → | V Tail |
| Prefix | → | F |
| Prefix | → | λ |
| Tail | → | + E |
| Tail | → | λ |

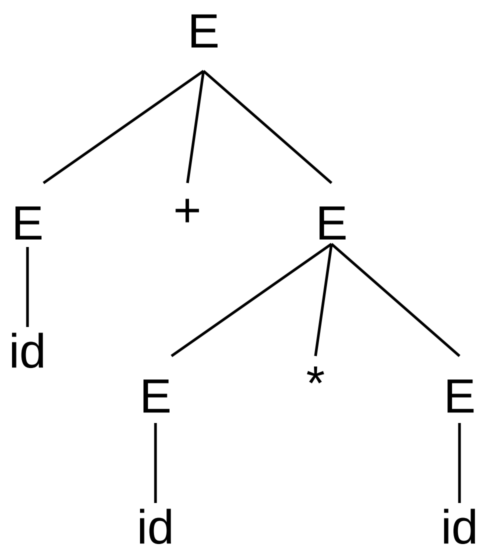
# Ambiguity

- Grammar that produces more than one parse tree for some string

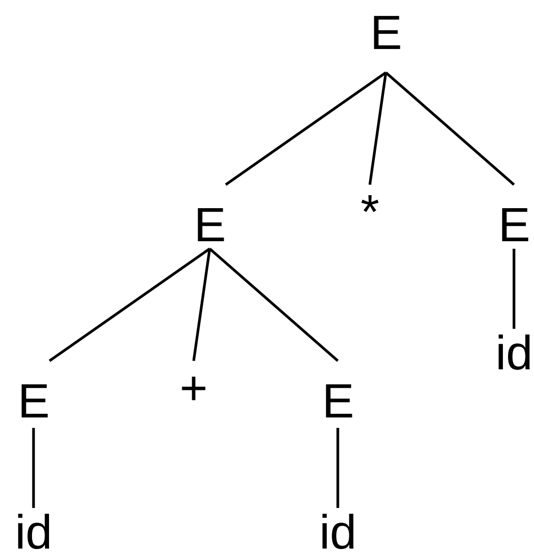
E.g. `E -> E + E | E * E | id`

String: **id+id*id**

```
E->E+E
E->id+E
E->id+E*E
E->id+id*E
E->id+id*id
```




```
E->E*E
E->E+E*E
E->id+E*E
E->id+id*E
E->id+id*id
```



# Ambiguity – what to do?

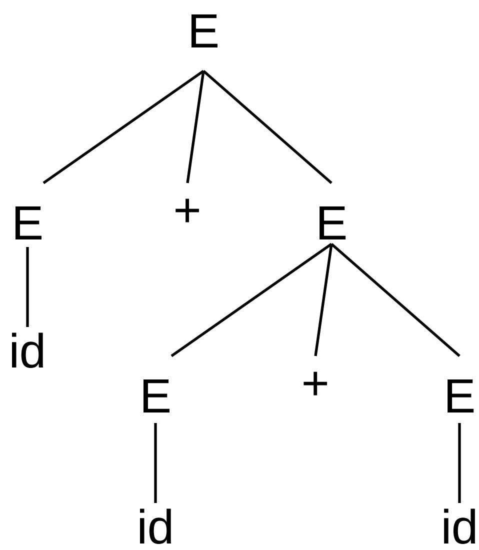
- Ignore it
  - Give hints to other components of the compiler on how to resolve it
- Fix it
  - Manually
  - May make the grammar complicated and difficult to maintain

# Ambiguity – ignore

- `E -> E + E | id`

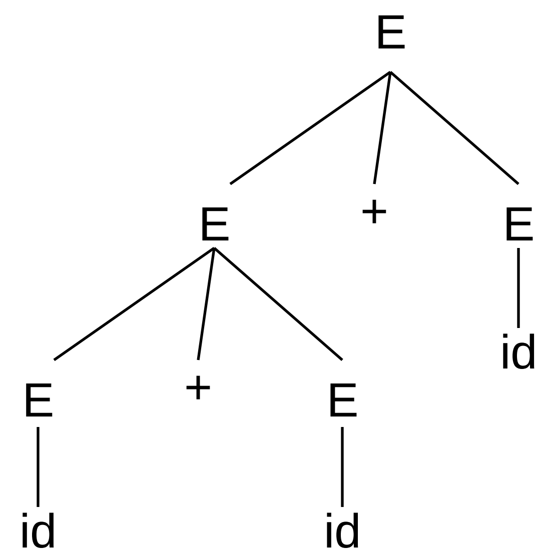
```
E->E+E
E->id+E
E->id+E+E
E->id+id+E
E->id+id+id
```

Produces:
`id+(id+id)`




```
E->E+E
E->E+E+E
E->id+E+E
E->id+id+E
E->id+id+id
```

Produces:
`(id+id)+id`




- Associativity declaration in Bison:

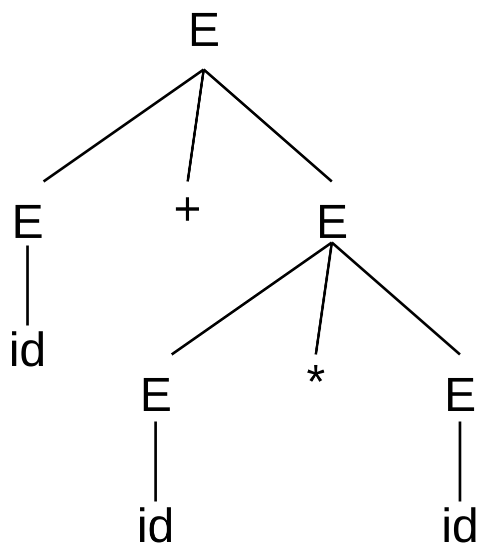
```
%left +
```

Picks the parse tree on the right

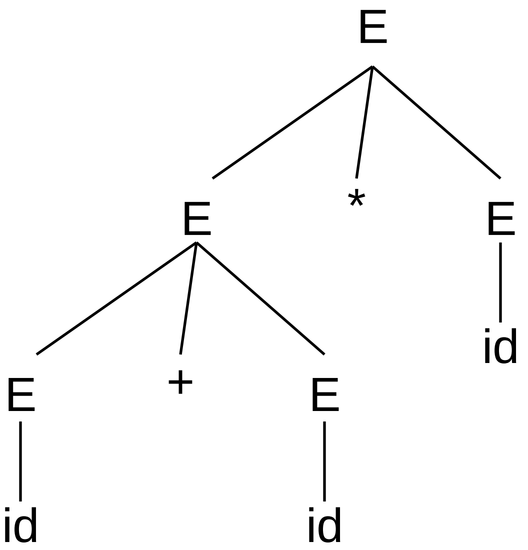
# Ambiguity - ignore

- `E -> E + E | E * E |` **`id`**

```
E->E+E
E->id+E
E->id+E*E
E->id+id*E
E->id+id*id
```




```
E->E*E
E->E+E*E
E->id+E*E
E->id+id*E
E->id+id*id
```




**`%left +`**
**`%left *`**

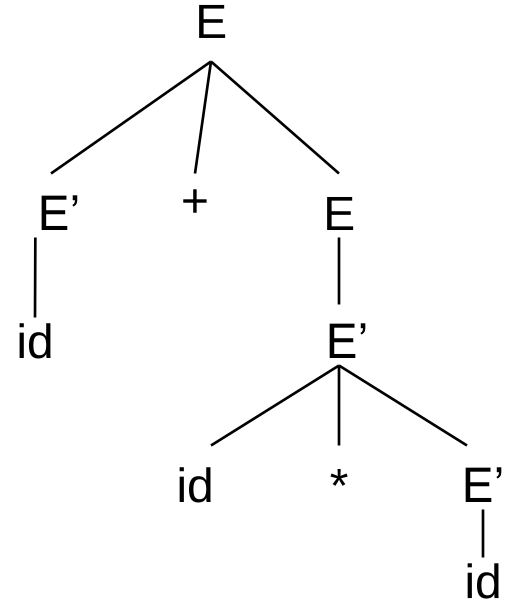
*Tells that * has higher precedence over + and both are left associative. So we get the tree on left.*

# Ambiguity – fixing

- Rewrite `E -> E + E | E * E | id` as

```
E -> E' + E | E'
E' -> id * E' | id | (E) * E' | (E)
```

```
E->E'+E
E->id+E
E->id+E'
E->id+id*E'
E->id+id*id
```




E controls generation of +

E' controls generation of *. *'s are nested deeper in the parse tree.

# Ambiguity - fixing

```
stmt -> if expr then stmt  |
        if expr then stmt else stmt |
        other
```

String: **if E1 then if E2 then S1 else S2**

**Exercise:** *verify if the above grammar is ambiguous. If so, rewrite the grammar to make it unambiguous.*

```
stmt -> matched | open
matched -> if expr then matched else matched |
      other
open ->  if expr then stmt |
      if expr then matched else open
```

# Error Handling

- Objective: detect invalid programs and provide meaningful feedback to programmer
  - Report errors accurately
  - Recover from errors quickly
  - Don't slow down compilation

# Error Types

- Many types of errors:
  - Lexical – use `Size` instead of `size`
  - Syntactic – extra brace
  - Semantic – `float sqr; sqr(2);`
  - Logical – use `=` instead of `==`

# Error Handling - Types

1. Panic mode
2. Error production
3. Automatic local or global correction

# Panic Mode Error Handling

- Simplest, most popular
- Discards tokens until one from a set of *synchronizing tokens* is found
- Synchronizing tokens have a clear role

  e.g. semicolons, braces
- E.g. i=i++j

  *policy:* while parsing an expression, discard all tokens until an integer is found. *This policy skips the additional +*
- Specifying policy in bison: `error` keyword

  `E -> E + E | (E) | id | error int | error`

# Error Productions

- Anticipate common errors
  - 2x instead of 2 *
- Augment the grammar
  - E -> EE | …
- Disadvantages:
  - Complicates the grammar

# Error Corrections

- Rewrite the program – find a “nearby” correct program
  - Local corrections – insert a semicolon, replace a comma with semicolon etc.
  - Global corrections – modify the parse tree with “edit distance” metric in mind
- Disadvantages?
  - Implementation difficulty
  - Slows down compilation
  - Not sure if “nearby” program is intended

# Top-down Parsing

- Also called recursive-descent parsing
- Equivalent to finding the left-derivation for an input string
  - Recall: expand the leftmost non-terminal in a parse tree
  - Expand the parse tree in pre-order i.e. identify parent nodes before children

# Top-down Parsing

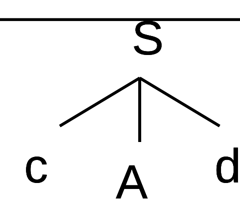
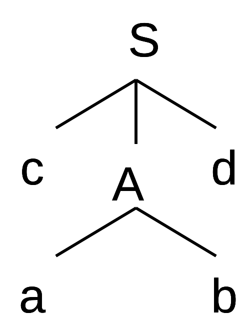
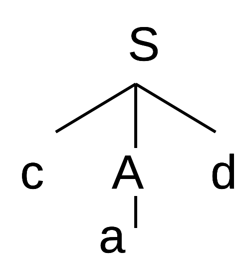
S -> cAd
A -> ab | a

String: cad

↑: next symbol to be read

*We need to backtrack after step 3 and reset input pointer*

*Can we do better ?*

| Step | Input string | Parse tree |
|---|---|---|
| 1 | cad | S |
| 2 | cad |  |
| 3 | cad |  |
| 4 | cad |  |

# Top-down Parsing

```
1) S -> F
2) S -> (S + F)
3) F -> a
```

string: (a+a)

string': (a+a)$

| | ( | ) | a | + | $ |
|---|---|---|---|---|---|
| S | 2 | - | 1 | - | - |
| F | - | - | 3 | - | - |

*Assume that the table is given.*

- Table-driven (Parse Table) approach doesn't require backtracking

*But how do we construct such a table?*

# First and follow sets

- First($\alpha$): the set of terminals (and/or $\lambda$) that begin all strings that can be derived from $\alpha$
  - First(A) = {x, y, $\lambda$}
  - First(xaA) = {x}
  - First (AB) = {x, y, b}
- Follow(A): the set of terminals (and/or \$, but no $\lambda$s) that can appear immediately after A in some partial derivation
  - Follow(A) = {b}

$S \rightarrow A\ B\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow \lambda$

$B \rightarrow b$

# First and follow sets

- $\text{First}(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$
- $\text{Follow}(A) = \{a \in V_t \mid S \Rightarrow^+ \ldots Aa \ldots\} \cup \{\$ \mid \text{if } S \Rightarrow^+ \ldots A\ \$\}$

| | |
|---|---|
| S: | start symbol |
| a: | a terminal symbol |
| A: | a non-terminal symbol |
| $\alpha,\beta$: | a string composed of terminals and non-terminals (typically, $\alpha$ is the RHS of a production |

| | |
|---|---|
| $\Rightarrow$: | derived in 1 step |
| $\Rightarrow^*$: | derived in 0 or more steps |
| $\Rightarrow^+$: | derived in 1 or more steps |

# Computing first sets

- Terminal: First(a) = {a}
- Non-terminal: First(A)
  - Look at all productions for A

    $A \rightarrow X_1 X_2 \ldots X_k$
  - $First(A) \supseteq (First(X_1) - \lambda)$
  - If $\lambda \in First(X_1)$, $First(A) \supseteq (First(X_2) - \lambda)$
  - If $\lambda$ is in $First(X_i)$ for all i, then $\lambda \in First(A)$
- Computing First($\alpha$): similar procedure to computing First(A)

# Top-down Parsing – predictive parsers

- Idea: we know sentence has to start with initial symbol
- Build up partial derivations by *predicting* what rules are used to expand non-terminals
  - Often called *predictive parsers*
- If partial derivation has terminal characters, *match* them from the input stream

Suggested reading: https://en.wikipedia.org/wiki/LL_parser

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

special "end of input" symbol

- A sentence in the grammar:

  x a c c $

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: S

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: A B c $

Predict rule

# A simple example

S → A B c $

Choose based on *first set* of rules

| |
|---|
| A → x a A |
| A → y a A |
| A → c |

B → b

B → λ

- A sentence in the grammar:

  x a c c $

Current derivation: x a A B c $

Predict rule *based on next token*

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a A B c $

Match token

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a A B c $

Match token

# A simple example

$S \rightarrow A\ B\ c\ \$$

Choose based on *first set* of rules

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a c B c $

Predict rule *based on next token*

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a c B c $

Match token

# A simple example

S → A B c $

A → x a A

A → y a A

A → c

B → b

B → λ

Choose based on *follow set*

- A sentence in the grammar:

  x a c c $

Current derivation: x a c λ c $

Predict rule *based on next token*

# A simple example

S → A B c $

A → x a A

A → y a A

A → c

B → b

B → λ

- A sentence in the grammar:

  x a c c $

Current derivation: x a c c $

Match token

# A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a c c $

Match token

# Top-down vs. Bottom-up parsers

- Top-down parsers expand the parse tree in *pre-order*
  - Identify parent nodes before the children
- Bottom-up parsers expand the parse tree in *post-order*
  - Identify children before the parents
- Notation:
  - LL(1): Top-down derivation with 1 symbol lookahead
  - LL(k): Top-down derivation with k symbols lookahead
  - LR(1): Bottom-up derivation with 1 symbol lookahead

# Suggested Reading

- Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D.Ullman: Compilers: Principles, Techniques, and Tools, 2/E, AddisonWesley 2007
  - Chapter 4 (Sections: 4.1 to 4.4)
- Fisher and LeBlanc: Crafting a Compiler with C
  - Chapter 4, Chapter 5(Sections 5.1 to 5.5, 5.9)