CMPSC 160
Translation of Programming Languages

Lecture 3: Lexical Analysis (Scanning)
+ Introduction to Parsing

Relationship between RE/NFA/DFA

RE→NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA →DFA (*subset construction*)

- Build the simulation

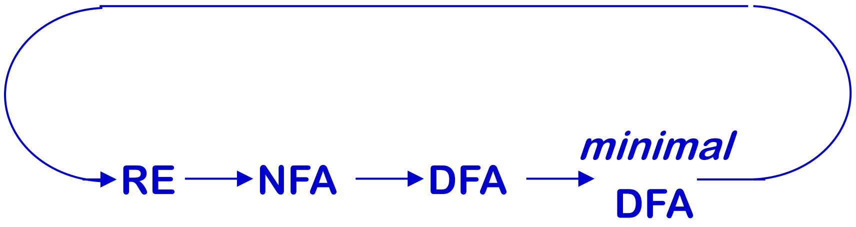
DFA → Minimal DFA

- Hopcroft's algorithm

DFA →RE

- Union together paths from s_0 to a final state

The Cycle of Constructions



RE → NFA → DFA → minimal DFA

You have learned all these algorithms in CMPSC138!

Automating Scanner Construction

To build a scanner:

1 Write down the RE that specifies the tokens

2 Translate the RE to an NFA

3 Build the DFA that simulates the NFA

4 Systematically shrink the DFA

5 Turn it into code

Scanner generators

- Lex , Flex, Jlex, and Jflex work along these lines
- Algorithms are well-known and well-understood

Scanner Implementations

- An automatically generated scanner.
- A hand-coded approach.

- Course: the use of generated scanners
- Most commercial compilers and open-source compilers use hand-crafted scanners.
  - performance gain VS. convenience
  - scanners are simple and they change infrequently

Building Faster Scanners from DFAs

```
state = $s_0$;
string = $\varepsilon$;
char  = get_next_char();
while (char != eof) {
   state = $\delta$(state,char);
   string = string + char;
   char = get_next_char();
}
if  (state in Final) then
   report acceptance;
else
    report failure;
```

Table-driven recognizers (which store the transition function in an array) waste a lot of effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Branch back to the top

We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code (it is OK, this is automatically generated code)
- Takes fewer operations per input character

Building Faster Scanners from the DFA

A direct-coded recognizer for Num

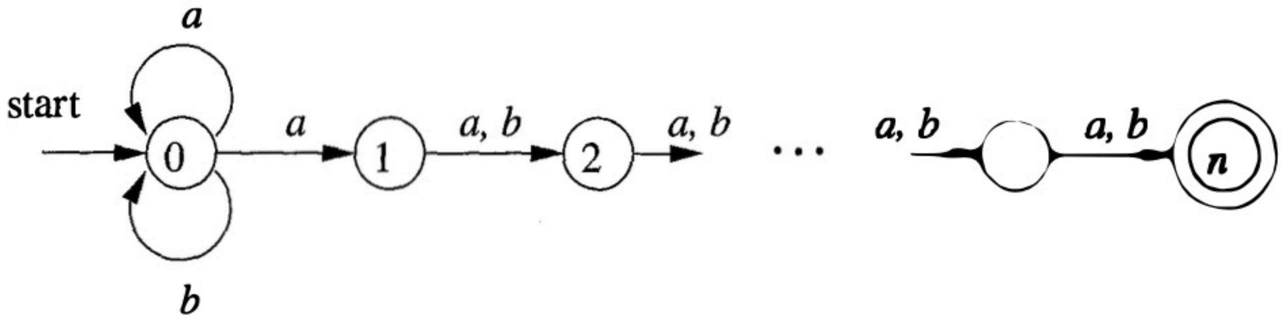
```
         goto $s_0$;
     $s_0$:  string $\leftarrow \varepsilon$;
         char $\leftarrow$ get_next_char();
         if (char = '+' || char= '–')
           then goto $s_1$;
           else goto $s_e$;
     $s_1$: string $\leftarrow$ string+ char;
         char $\leftarrow$ get_next_char();
         if ( '0' $\leq$ char $\leq$ '9')
           then goto $s_2$;
           else goto $s_e$;
     $s_2$: string $\leftarrow$ string+ char;
         char $\leftarrow$ get_next_char();
         if ( '0' $\leq$ char $\leq$ '9')
           then goto $s_2$;
           else if (char = '.')
             then goto $s_3$
             else goto $s_e$;
     $s_3$ : string $\leftarrow$ string+ char;
         char $\leftarrow$ get_next_char();
         if ( '0' $\leq$ char $\leq$ '9')
           then goto $s_3$;
           else if (char = eof)
              then report acceptance;
              else goto $s_e$;
     $s_e$: print error message;
         return failure;
```

- Many fewer operations per character
- State is encoded as the location in the code

Limitation of DFA/RE?

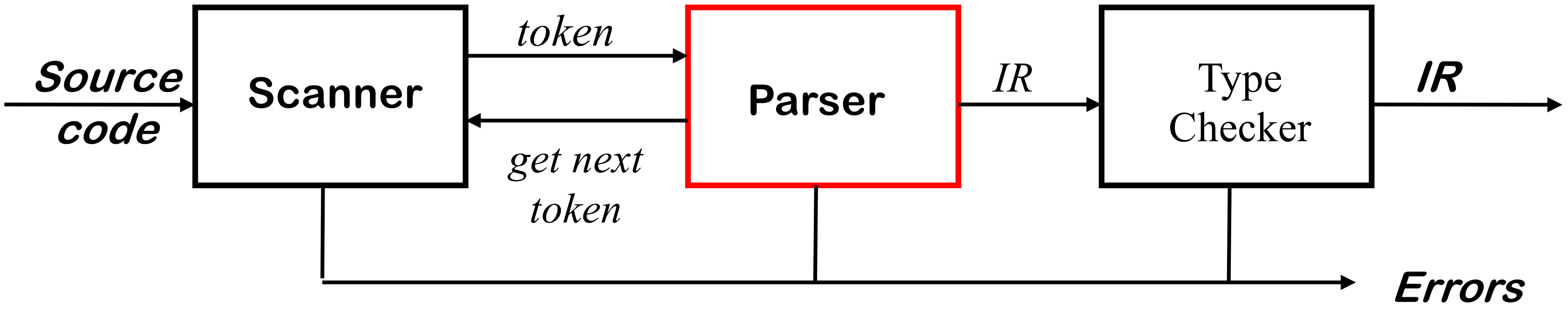
- Combining subset construction (NFA→DFA) and hopcroft's algorithm (DFA→ minimal DFA), at worst time, could still lead to a state exponential blowup, but when?
- Understanding the worst case would let you know the limitations of regular expressions.

$$L_n = (\mathbf{a}|\mathbf{b})^*\mathbf{a}(\mathbf{a}|\mathbf{b})^{n-1}$$



- Could you prove that any DFA for the language Ln must have at least 2^n states?
- Solution: Use a more powerful formalism: ***context-free grammars (pushdown automata)***.

The Front End: Parser



Parser

- Input: A sequence of words (along with its syntactic category, i.e., token)
- Output: **A parse tree, a syntactic structure for the program**, which fits the words into a grammatical model of the source programming language.

Parser VS Scanner

- Unlike scanners, where hand-coding is common, tool-generated parsers are more common than hand-coded parsers.

- About Five Lectures:
    - 1 Lectures: Introduction to parsing
    - 1 lectures: Top-down parsing in the form of recursive-descent parsers and LL(1) parsers
    - 1- 2 Lectures: Bottom-up parsing as exemplified by LR(1) parsers
    - 1- 2 Lecture: Several practical issues that arise in parser construction

Why not RE?

- Case analysis: recognizing algebraic expressions over variables and the operators +, -, × , and ÷
  - variable: $[a\ldots z]\,([a\ldots z]\,|\,[0\ldots 9])^*$,
  - Expression $[a\ldots z]\,([a\ldots z]\,|\,[0\ldots 9])^* \quad ((+\,|\,-\,|\,\times\,|\,\div)\;\; [a\ldots z]\,([a\ldots z]\,|\,[0\ldots 9])^*\,)^*$

- Any problem? a + b × c
  - Missing Precedence; no idea of evaluation order 😭
  - Adding parentheses?
    - we cannot find a RE that will match all expressions with balanced parentheses
    - Paired constructs, such as “begin” and “end”, “{“ and “}”, play an important role in most programming languages
    - The language $(^m\,)^n$ where $m = n$ is not regular. In principle, DFAs cannot count.

RE lack the power to describe the full syntax of most programming languages

Context-free Grammar

❖ **Key: powerful than RE, but still could lead to efficient recognizer.**

❖ Context-free Grammar (CFGs):

- A CFG is precise and understandable
- large subclasses of the CFGs have the property that they lead to efficient recognizer

- It is so effective because it **embraces the recursive nature** of most programming languages
  - ❖ Example sentence: if(x){ if(y){ if(z) { } } }
  - ❖ Example grammar: B -> if(id) { B }

- The collection of sentences that can be **derived** from *G* is called the *language defined by G*, denoted *G*

- The set of languages defined by context-free grammars is called the set of context-free languages.

An Example Context-free Grammar (CFG) Grammar

❖ Suppose we want to describe all legal **arithmetic expressions** using addition, subtraction, multiplication, and division.

```
1  Start    →     Expr
2  Expr     →     Expr Op Expr
3           |     num
4           |     id
5  Op       →     +
6           |     -
7           |     *
8           |     /
```

Start symbol:	*S* = *Start*	**abstraction for all sentences for the CFG**
Non-terminal symbols:	*N* = { *Start*, *Expr*, *Op* }	**syntactic structure abstraction**
Terminal symbols:	*T* = { num, id, +, -, *, / }	**from our scanner**
Productions:	**substrings (key structure) for our language**	

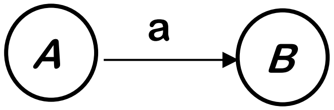
Context-free Grammar

Formally, a grammar is a four-tuple, $G = (S,N,T,P)$

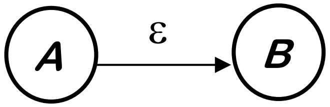
- T is a set of *terminal symbols*
  - These correspond to tokens returned by the scanner
  - For the parser, tokens are indivisible units of syntax
- N is a set of *non-terminal symbols*
  - These are syntactic variables that can be substituted during a derivation
  - Variables that denote sets of substrings occurring in the language
- S is the *start symbol* : $S \in N$
  - All the strings in *L(G)* are derived from the start symbol
- P is a set of *productions* or *rewrite rules* : $\mathrm{P} : \mathbf{N} \rightarrow (\mathrm{N} \cup \mathrm{T})^*$

RE can be turned into CFG

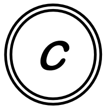
❖ NFA → CFG



A -> aB



A -> B



A -> ε

- Lexical rules can be described with CFG.
- CFGs are most useful in describing the nested chain structure or syntactic structure such as balanced parenthesis, if else etc.
  and these can’t be defined by Regular Expression.
- By using the CFG, it is very difficult to construct the recognizer.
- **Only use powerful tool when needed, as it comes with overhead.**
- Key: powerful than RE, but still could lead to efficient recognizer.

Vocabulary from the Derivation process

- *Sentence* of G: String of terminals in L(G)
- *Derivation*: A sequence of rewrites according to productions
- The process of discovering a derivation for a sentence is called ***parsing***
- *Sentential Form* of G: String of non-terminals and terminals from which a sentence of G can be derived.

Key steps in Derivations

- At each step, we make two choices
  1. Choose a non-terminal to replace
  2. Choose a production to apply
- Different choices lead to different derivations

Two types of derivation are of interest

- Leftmost derivation — replace leftmost non-terminal at each step
- Rightmost derivation — replace rightmost non-terminal at each step

These are the two *systematic* derivations (the first choice is fixed)

Leftmost vs. Rightmost derivations

Q1) For every leftmost derivation, there is a rightmost derivation, and vice versa. True or False?

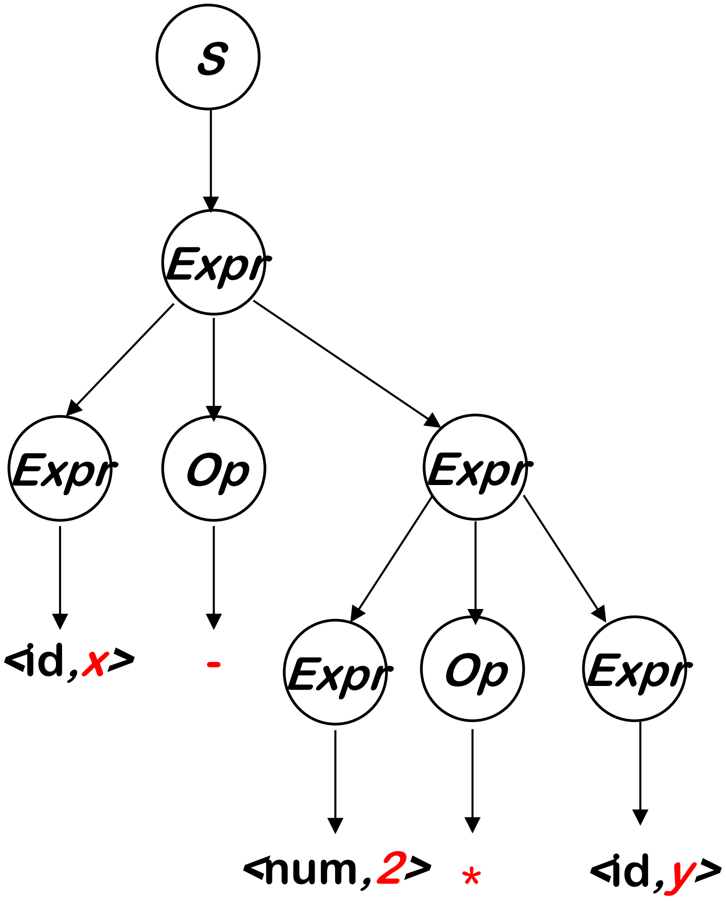
Q2) Does every word generated by a CFG have a leftmost and a rightmost derivation?

Q3) Could there be sentences which have more than leftmost and a rightmost derivation?

A Leftmost Derivation and the Corresponding Parse Tree

A leftmost derivation for x – 2 * y

Rule	Sentential Form
—	*S*
1	*Expr*
2	*Expr Op Exp*
4	<id,*x*> *Op Expr*
6	<id,*x*> - *Expr*
2	<id,*x*> - *Expr Op Exp*
3	<id,*x*> - <num,*2*> *Op Expr*
7	<id,*x*> - <num,*2*> * *Expr*
4	<id,*x*> - <num,*2*> * <id,*y*>



One natural way to evaluate the expression is with a simple postorder tree-walk.

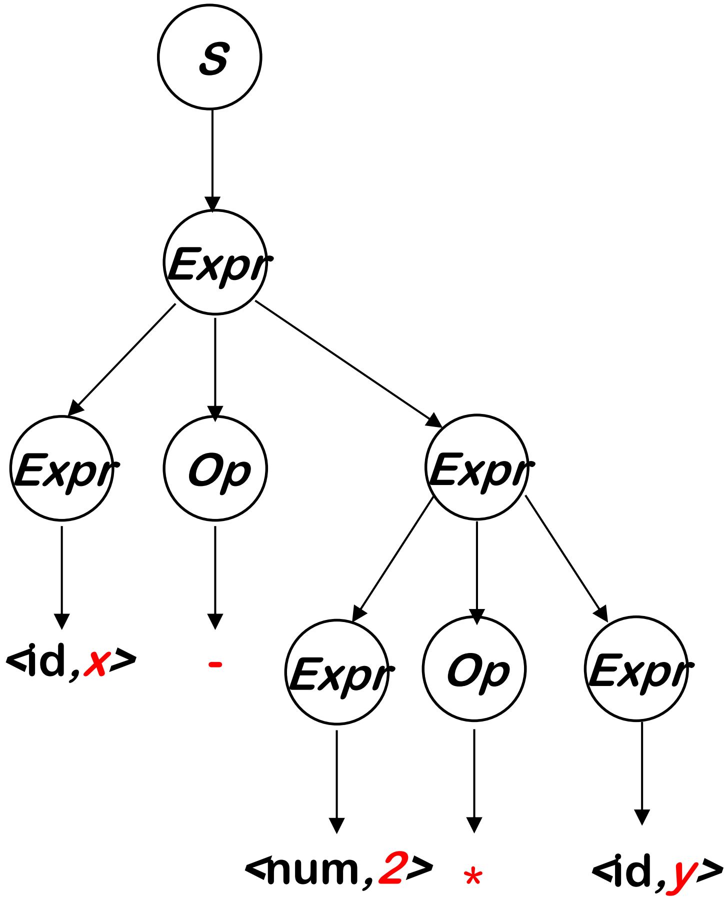
This evaluates as x - (2 * y)

A Rightmost Derivation and the Corresponding Parse Tree

A rightmost derivation for x – 2 * y

Rule	Sentential Form
—	S
1	Expr
2	Expr Op Expr
2	Expr Op Expr Op Expr
4	Expr Op Expr Op <id,y>
7	Expr Op Expr * <id,y>
3	Expr Op <num,2> * <id,y>
6	Expr - <num,2> * Expr
4	<id,x> - <num,2> * <id,y>

This evaluates as x - (2 * y) too



Production Rule Order VS Final Parse Tree

Rule	Sentential Form
—	S
1	Expr
2	Expr Op Exp
4	<id,x> Op Expr
6	<id,x> - Expr
2	<id,x> - Expr Op Exp
3	<id,x> - <num,2> Op Expr
7	<id,x> - <num,2> * Expr
4	<id,x> - <num,2> * <id,y>

A leftmost derivation

Rule	Sentential Form
—	S
1	Expr
2	Expr Op Expr
2	Expr Op Expr Op Expr
4	Expr Op Expr Op <id,y>
7	Expr Op Expr * <id,y>
3	Expr Op <num,2> * <id,y>
6	Expr - <num,2> * Expr
4	<id,x> - <num,2> * <id,y>

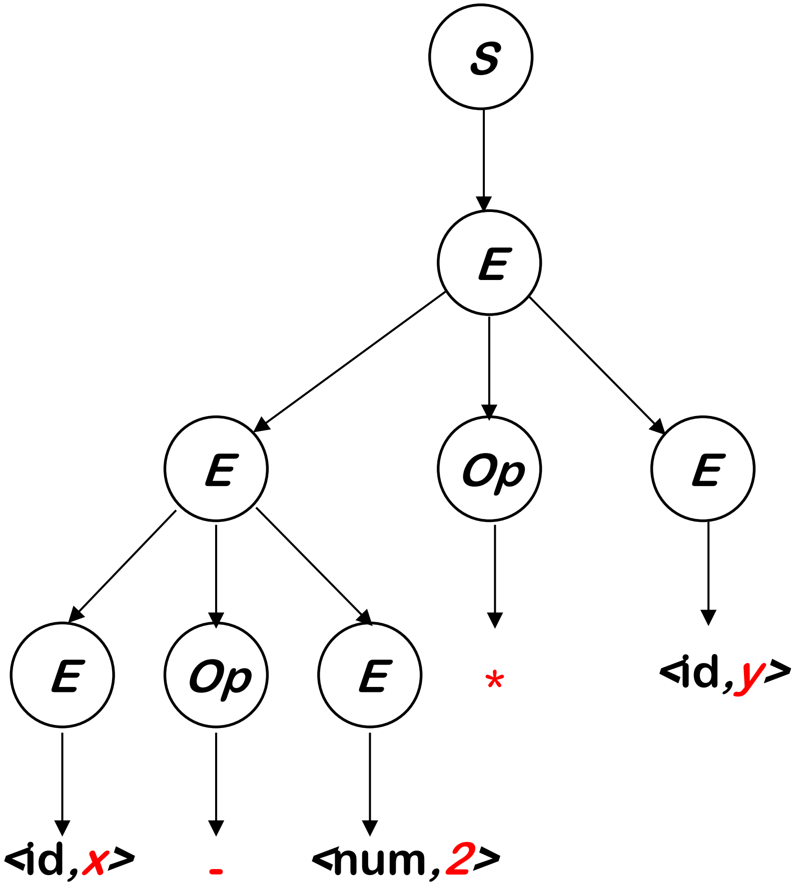
A rightmost derivation

- The leftmost and rightmost derivations use the same set of rules; they apply those rules in a different order.
- Because a parse tree represents the rules applied, but not the order of their application, the parse trees for the two derivations are identical.
- What will lead to different parse tree? → The evaluation order.

Another Rightmost Derivation and the Corresponding Parse Tree

Another rightmost derivation and corresponding parse tree

Rule	Sentential Form
—	*S*
1	*Expr*
2	*Expr Op Expr*
4	*Expr Op* <id,*y*>
7	*Expr* * <id,*y*>
2	*Expr Op Expr* * <id,*y*>
3	*Expr Op* <num,*2*> * <id,*y*>
6	*Expr* - <num,*2*> * <id,*y*>
4	<id,*x*> - <num,*2*> * <id,*y*>



This evaluates as (x - 2) * y

This parse tree is different than the parse tree for the previous rightmost derivation, and gets the precedence wrong

Ambiguity

- One grammar can produce *two different parse trees* for the same sentence.
  - From a theoretical standpoint, this is fine. The sentence can be derived from the grammar and everyone is happy
  - The problem is that the way the program is interpreted stems from the parse tree

- We need to ensure that for each sentence in G, there is only one parse tree for that sentence
  - If there is *more than one parse tree* for a given sentence, our grammar is **ambiguous**
  - *To show a grammar G is ambiguous, find a sentence in G with two parse trees*

Ambiguous Grammars

Classic example — the dangling-else problem

$$\begin{array}{llll}
1 & \mathit{Stmt} & \rightarrow & \text{if } \mathit{Expr} \text{ then } \mathit{Stmt} \\
2 & & | & \text{if } \mathit{Expr} \text{ then } \mathit{Stmt} \text{ else } \mathit{Stmt} \\
 & & | & \mathit{more}
\end{array}$$

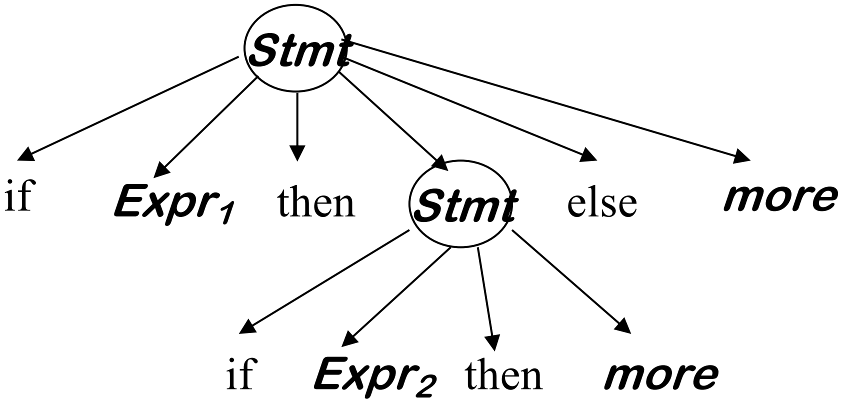
Ambiguity

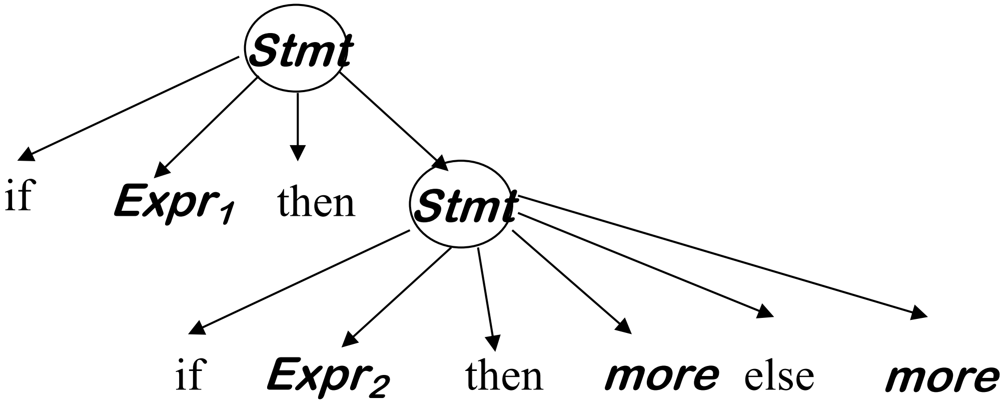
1 *Stmt* → if *Expr* then *Stmt*

2 | if *Expr* then *Stmt* else *Stmt*

| *more*

Two parse trees for “**if *Expr*$_1$ then if *Expr*$_2$ then *more* else *more***”.





production 2, then production 1

production 1, then production 2

Any idea to Remove the Ambiguity？

Removing the Ambiguity

- Rewrite the grammar to avoid generating the problem
- It accepts the same set of sentences as the original grammar, but ensures that each else has an unambiguous match to a specific if.
- Match each else to innermost unmatched if *(common sense rule)*

Removing the Ambiguity

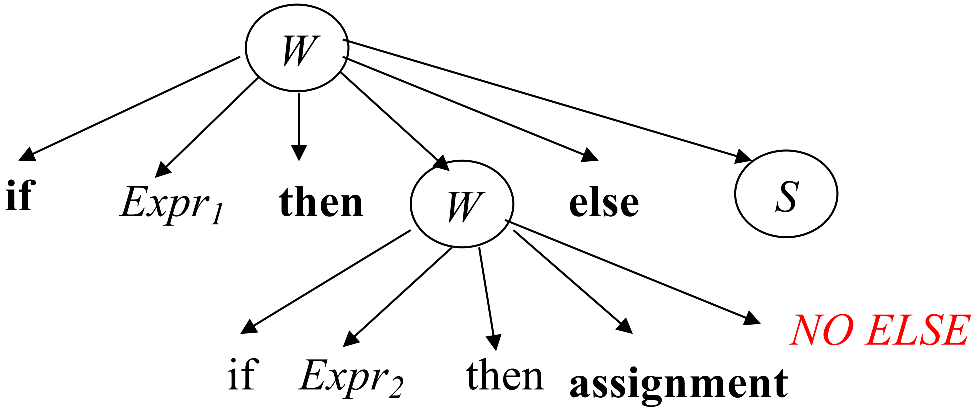
- Rewrite the grammar to avoid generating the problem
- It accepts the same set of sentences as the original grammar, but ensures that each else has an unambiguous match to a specific if.
- Match each <u>else</u> to innermost unmatched <u>if</u> *(common sense rule)*

- New rules enforce that only a "matched" statement (an if statements with an else part) can come before an else

```
Stmt     →      If Expr then Stmt
         |      If Expr then WithElse else Stmt
         |      Assignment
Withelse →      If Expr then ??? else ???
         |      Assignment
```

Ambiguity

Try the dangling-else derivations:



Can’t make a parse tree where the “else” associates with the first “if”

Parse Trees and Precedence

No notion of precedence → multiple order of evaluation between different operators → ambiguity.

For algebraic expressions

- Multiplication and division, first
- Subtraction and addition, next

To add precedence

- **Create a non-terminal for each *level of precedence***
- Isolate the corresponding part of the grammar
- Force parser to recognize high precedence sub-expressions first

Precedence

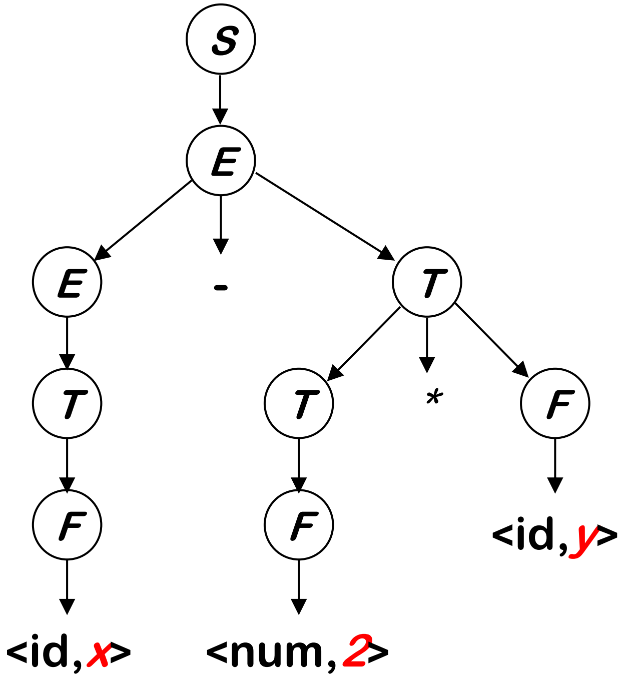
Adding the standard algebraic precedence :

1	*S*	→	*Expr*
2	*Expr*	→	*Expr + Term*
3		|	*Expr - Term*
4		|	*Term*
5	*Term*	→	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	→	num
9		|	id

Precedence

Rule	Sentential Form
	S
1	*Expr*
3	*Epr* - *Term*
7	*Term* - *Term*
8	*Factor* - *Term*
3	<id,*x*> - *Term*
7	<id,*x*> - *Term* * *Factor*
8	<id,*x*> - *Factor* * *Factor*
4	<id,*x*> - <num,*2*> * *Factor*
7	<id,*x*> - <num,*2*> * <id,*y*>

The leftmost derivation



Its parse tree

This produces x - (2 * y) , along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same parse tree and the same evaluation order, because the grammar directly encodes the desired precedence.

Associativity

1	*S*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		$\mid$	*Expr* - *Term*
4		$\mid$	*Term*
5	*Term*	→	*Term* * *Factor*
6		$\mid$	*Term* / *Factor*
7		$\mid$	*Factor*
8	*Factor*	→	num
9		$\mid$	id

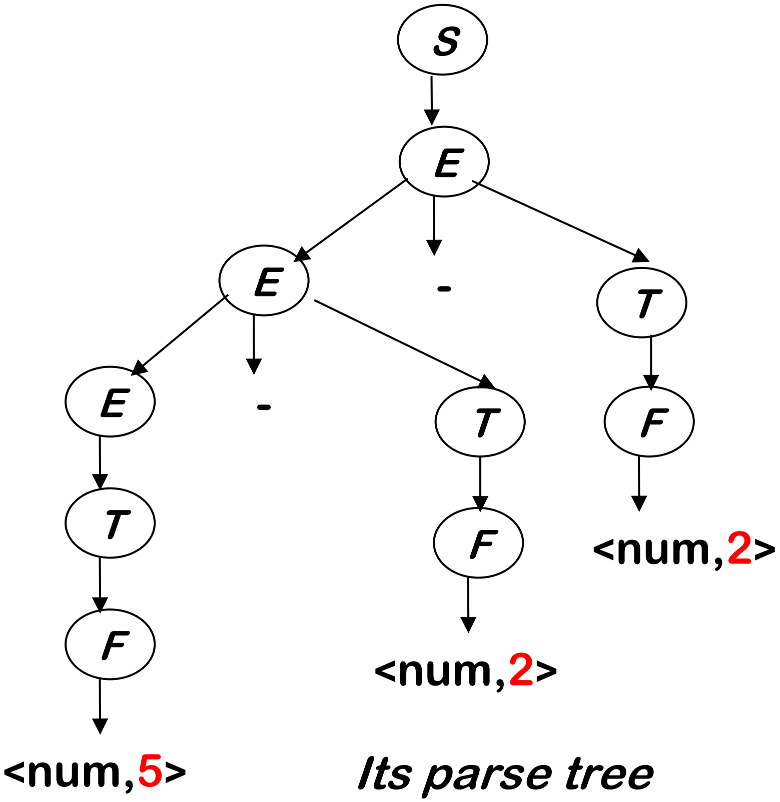
V.S

1	*S*	→	*Expr*
2	*Expr*	→	*Term* + *Expr*
3		$\mid$	*Term* - *Expr*
4		$\mid$	*Term*
5	*Term*	→	*Factor* * *Term*
6		$\mid$	*Factor* / *Term*
7		$\mid$	*Factor*
8	*Factor*	→	num
9		$\mid$	id

Associativity

Rule	Sentential Form
	S
1	*Expr*
3	*Epr* - *Term*
7	*Expr* - *Factor*
8	*Expr* - <num,*2*>
3	*Expr* - *Term* - <num,*2*>
7	*Expr* - *Factor* - <num,*2*>
8	*Expr* - <*num*,*2*> - <num,*2*>
4	*Term* - <num,*2*> - <num,*2*>
7	*Factor* - <num,*2*> - <num,*2*>
8	<num,*5*> - <num,*2*> - <num,*2*>

The rightmost derivation



Its parse tree

This produces (5 - 2) - 2 , along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same parse tree and the same evaluation order

Parsing Techniques

Top-down parsers *(LL(1), recursive descent parsers)*

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), shift-reduce parsers)*

- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step, a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars

Top-down Parsing Algorithm

Construct the root node of the parse tree, label it with the start symbol, and set the current-node to root node

Repeat until all the input is consumed (i.e., until the frontier of the parse tree matches the input string)

1 If the label of the current node is a non-terminal node A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child

2 If the current node is a terminal symbol:

If it matches the input string, consume it (advance the input pointer)

If it does not match the input string, **backtrack**

3 Set the current node to the next node in the frontier of the parse tree

If there is no node left in the frontier of the parse tree and input is not consumed, then backtrack

The key is picking the correct production in step 1

- That choice should be guided by the input string