

Parsing

Part I

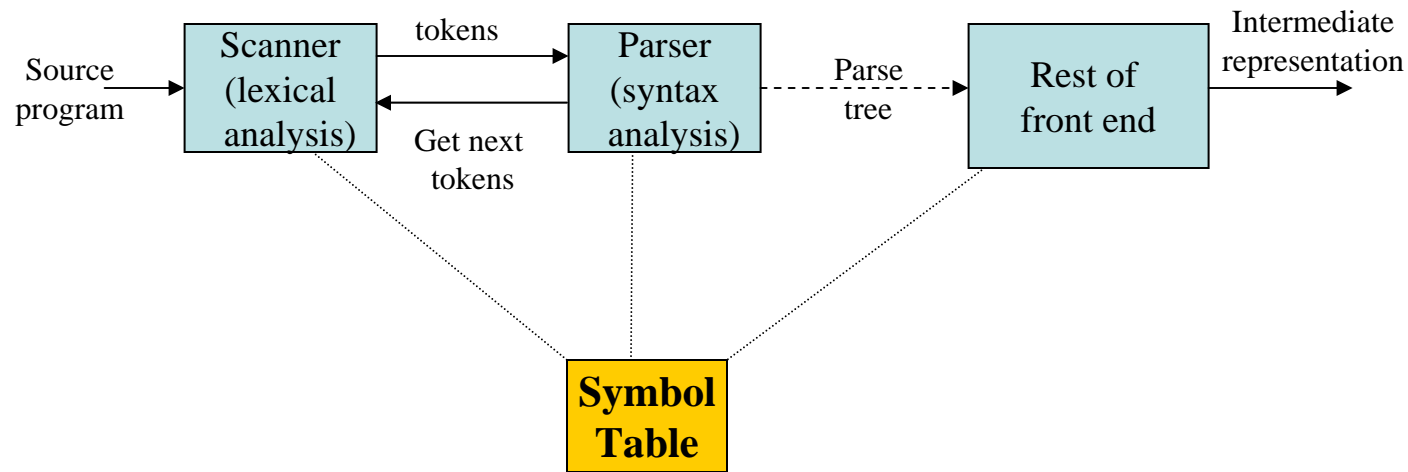
Language and Grammars

- Every (programming) language has precise rules
 - In English:
 - Subject Verb Object
 - In C
 - programs are made of functions
 - » Functions are made of statements etc.

Parsing

- A.K.A. Syntax Analysis
 - Recognize sentences in a language.
 - Discover the structure of a document/program.
 - Construct (implicitly or explicitly) a tree (called as a parse tree) to represent the structure.
 - The above tree is used later to guide translation.

Role of the parser



- Verifies if the string of token can be generated from the grammar
- Error?
 - Report with a good descriptive, helpful message
 - Recover and continue parsing!
- Build a parse tree !!

Rest of Front End

- Collecting token information
- Type checking
- Intermediate code generation

Errors in Programs

- **Lexical**

if x<1 then n y = 5:

“Typos”

- **Syntactic**

if ((x<1) & (y>5))) ...

{ ... { ... _ ... }

- **Semantic**

if (x+5) then ...

Type Errors

Undefined IDs, etc.

- **Logical Errors**

if (i<9) then ...

Should be <= not <

Bugs

Compiler cannot detect Logical Errors

Error Detection

- Much responsibility on Parser
 - Many errors are syntactic in nature
 - Precision/ efficiency of modern parsing method
 - Detect the error as soon as possible
- Challenges for error handler in Parser
 - Report error clearly and accurately
 - Recover from error and continue..
 - Should be efficient in processing
- Good news is
 - Simple mechanism can catch most common errors
- Errors don't occur that frequently!!
 - 60% programs are syntactically and semantically correct
 - 80% erroneous statements have only 1 error, 13% have 2
 - Most error are trivial : 90% single token error
 - 60% punctuation, 20% operator, 15% keyword, 5% other error

Adequate Error Reporting is Not a Trivial Task

- Difficult to generate clear and accurate error messages.

Example

```
function foo () {  
  ...  
  if (...) {  
    ...  
  } else {  
    ...  
    ...  
  }  
  ...  
}  
<eof>
```

Missing } here

Not detected until here

Example

```
int myVarr;  
...  
x = myVar;  
...
```

Misspelled ID here

Not detected until here

Error Recovery

- After first error recovered
 - Compiler must go on!
 - Restore to some state and process the rest of the input
- **Error-Correcting Compilers**
 - Issue an error message
 - Fix the problem
 - Produce an executable

Example

```
Error on line 23: "myVarr" undefined.  
"myVar" was used.
```

May not be a good Idea!!

- Guessing the programmers intention is not easy!

Error Recovery May Trigger More Errors!

- Inadequate recovery may introduce more errors
 - Those were not programmers errors

- Example:

```
int myVar flag ;
```

```
...
```

```
x := flag;
```

```
...
```

```
...
```

```
while (flag==0)
```

```
...
```

Declaration of flag is discarded

Variable flag is undefined

Variable falg is undefined

Too many Error message may be obscuring

- May bury the real message
- Remedy:
 - allow 1 message per token or per statement
 - Quit after a maximum (e.g. 100) number of errors

Error Recovery Approaches: Panic Mode

- Discard tokens until we see a “synchronizing” token.

Example

Skip to next occurrence of
`} end ;`
Resume by parsing the next statement

- The key...
 - Good set of synchronizing tokens
 - Knowing what to do then
- Advantage
 - Simple to implement
 - Does not go into infinite loop
 - Commonly used
- Disadvantage
 - May skip over large sections of source with some errors

Error Recovery Approaches: Phrase-Level Recovery

- Compiler corrects the program
by deleting or inserting tokens
...so it can proceed to parse from where it was.

Example

while (x==4) y:= a + b

Insert **do** to fix the statement



- The key...
Don't get into an infinite loop
...constantly inserting tokens and never scanning the actual source
- Generally used for **error-repairing** compilers
 - Difficulty: Point of error detection might be much later the point of error occurrence

Error Recovery Approaches: Error Productions

- Augment the CFG with “Error Productions”
- Now the CFG accepts anything!
- If “error productions” are used...

 Their actions:

{ print (“Error...”) }

- Used with...
 - LR (Bottom-up) parsing
 - Parser Generators

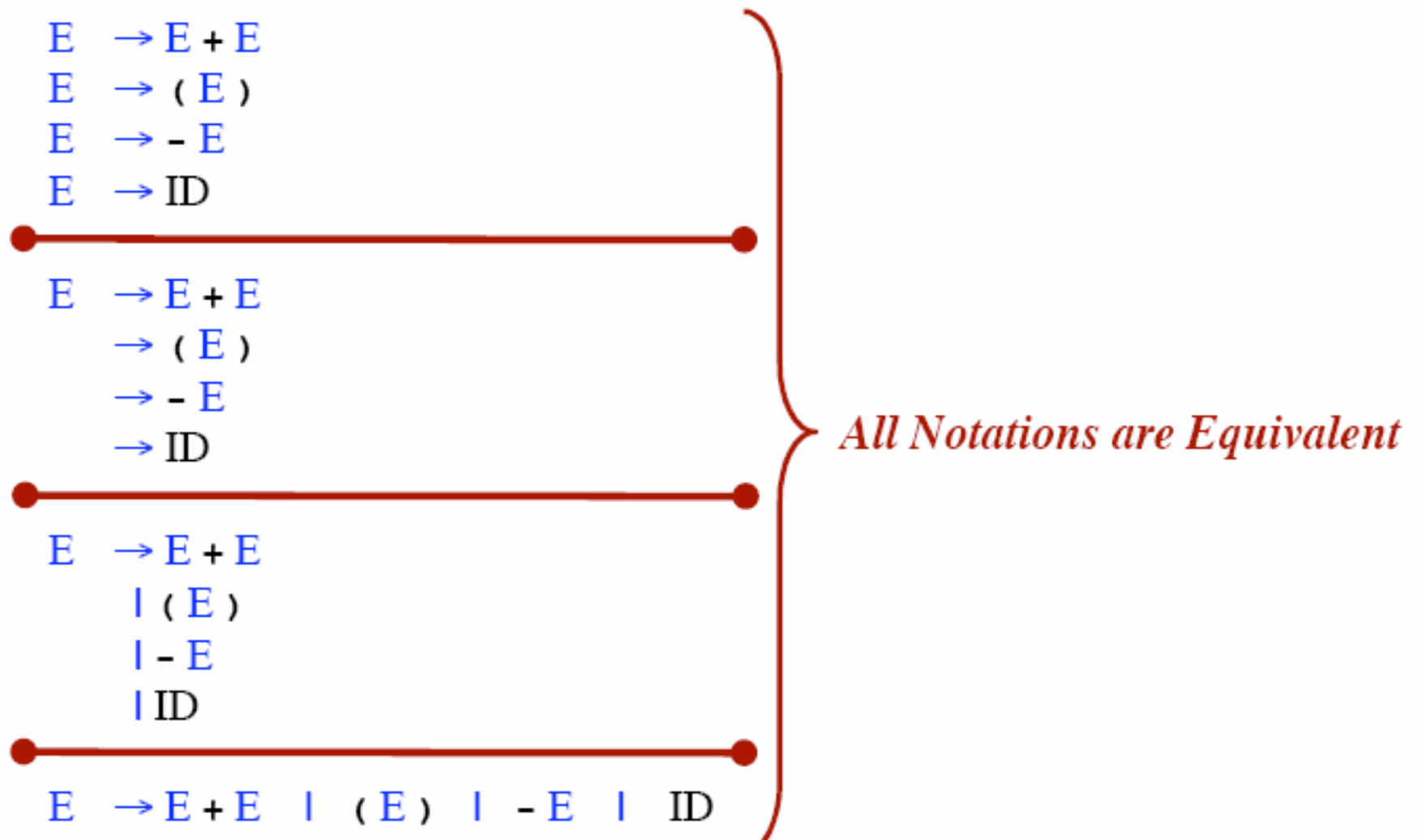
Error Recovery Approaches: Global Correction

- Theoretical Approach
- Find the minimum change to the source to yield a valid program
 - Insert tokens, delete tokens, swap adjacent tokens
- Global Correction Algorithm
 - Input:** grammatically incorrect input string x ; grammar G
 - Output:** grammatically correct string y
 - Algorithm:** converts $x \rightarrow y$ using minimum number changes (insertion, deletion etc.)
- Impractical algorithms - too time consuming

Context Free Grammars (CFG)

- A **context free grammar** is a formal model that consists of:
- **Terminals**
 - Keywords
 - Token Classes
 - Punctuation
- **Non-terminals**
 - Any symbol appearing on the lefthand side of any rule
- **Start Symbol**
 - Usually the non-terminal on the lefthand side of the first rule
- **Rules (or “Productions”)**
 - BNF: Backus-Naur Form / Backus-Normal Form
 - Stmt ::= if Expr then Stmt else Stmt

Rule Alternative Notations



Notational Conventions

Terminals

a b c ...

Nonterminals

A B C ...

S

Expr

Grammar Symbols (Terminals or Nonterminals)

X Y Z U V W ...

Strings of Symbols

α β γ ...

A sequence of zero
Or more terminals
And nonterminals

Strings of Terminals

x y z u v w ...

Including ϵ

Examples

$A \rightarrow \alpha B$

A rule whose righthand side ends with a nonterminal

$A \rightarrow x \alpha$

A rule whose righthand side begins with a string of terminals (call it “x”)

Derivations

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

A “Derivation” of “ $(\underline{id} * \underline{id})$ ”

$$E \Rightarrow (E) \Rightarrow (E * E) \Rightarrow (\underline{id} * E) \Rightarrow (\underline{id} * \underline{id})$$

“Sentential Forms”

A sequence of terminals and nonterminals in a derivation

$(\underline{id} * E)$

Derivation

If $A \rightarrow \beta$ is a rule, then we can write

$$\underbrace{\alpha A \gamma}_{\uparrow} \Rightarrow \alpha \beta \gamma$$

*Any sentential form containing a nonterminal (call it A)
... such that A matches the nonterminal in some rule.*

Derives in zero-or-more steps \Rightarrow^*

$$E \Rightarrow^* (\underline{id} * \underline{id})$$

If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$

Derives in one-or-more steps \Rightarrow^+

Given

G A grammar
S The Start Symbol

Define

$L(G)$ The language generated
 $L(G) = \{ w \mid S \Rightarrow^+ w \}$

“Equivalence” of CFG’s

If two CFG’s generate the same language, we say they are “**equivalent**.”

$$G_1 \approx G_2 \text{ whenever } L(G_1) = L(G_2)$$

In making a derivation...

Choose which nonterminal to expand

Choose which rule to apply

Leftmost Derivation

In a derivation... always expand the leftmost nonterminal.

E
 $\Rightarrow E + E$
 $\Rightarrow (E) + E$
 $\Rightarrow (E * E) + E$
 $\Rightarrow (\underline{id} * E) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

- | | |
|----|-----------------------|
| 1. | $E \rightarrow E + E$ |
| 2. | $\rightarrow E * E$ |
| 3. | $\rightarrow (E)$ |
| 4. | $\rightarrow - E$ |
| 5. | $\rightarrow ID$ |

Let \Rightarrow_{LM} denote a step in a leftmost derivation (\Rightarrow_{LM}^* means zero-or-more steps)

At each step in a leftmost derivation, we have

$$wA\gamma \Rightarrow_{LM} w\beta\gamma \quad \text{where } A \rightarrow \beta \text{ is a rule}$$

(Recall that w is a string of terminals.)

Each sentential form in a leftmost derivation is called a “**left-sentential form.**”

If $S \Rightarrow_{LM}^* \alpha$ then we say α is a “**left-sentential form.**”

Rightmost Derivation

In a derivation... always expand the rightmost nonterminal.

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

- | | |
|----|-----------------------|
| 1. | $E \rightarrow E + E$ |
| 2. | $\rightarrow E * E$ |
| 3. | $\rightarrow (E)$ |
| 4. | $\rightarrow - E$ |
| 5. | $\rightarrow ID$ |

Let \Rightarrow_{RM} denote a step in a rightmost derivation (\Rightarrow_{RM}^* means zero-or-more steps)

At each step in a rightmost derivation, we have

$$\alpha A w \Rightarrow_{RM} \alpha \beta w \quad \text{where } A \rightarrow \beta \text{ is a rule}$$

(Recall that w is a string of terminals.)

Each sentential form in a rightmost derivation is called a “**right-sentential form.**”

If $S \Rightarrow_{RM}^* \alpha$ then we say α is a “**right-sentential form.**”

Parse Tree

Two choices at each step in a derivation...

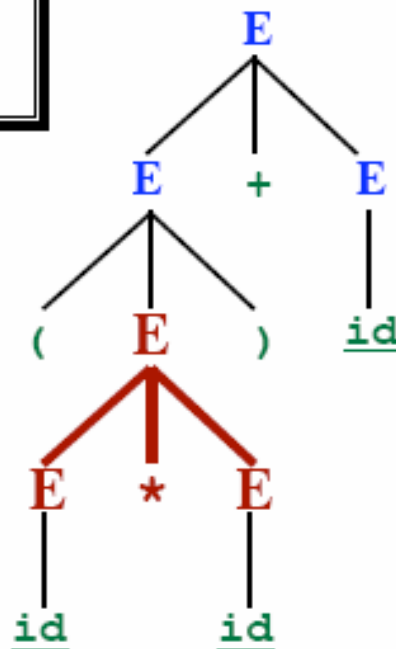
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Leftmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow (E) + E$
 $\Rightarrow (E * E) + E$
 $\Rightarrow (id * E) + E$
 $\Rightarrow (id * id) + E$
 $\Rightarrow (id * id) + id$

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$



Parse Tree

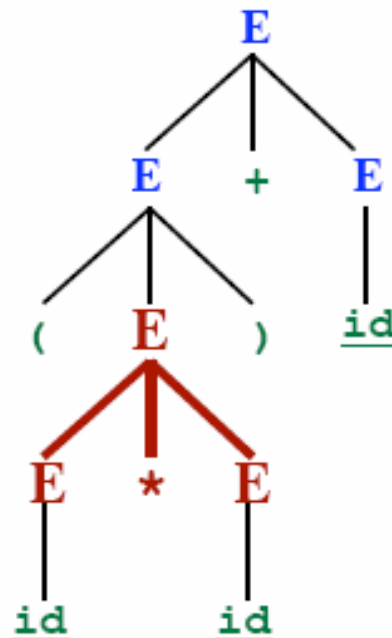
Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Rightmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$



1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Parse Tree

Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

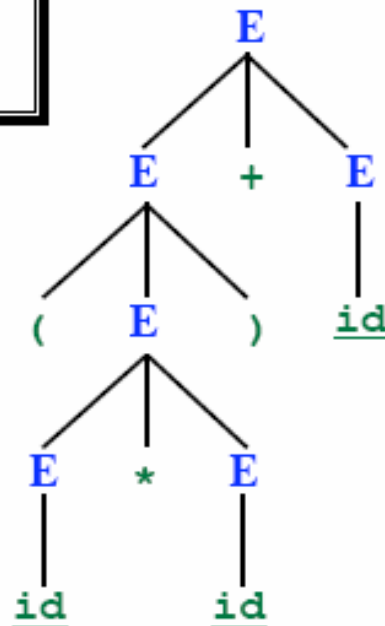
Leftmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow (E) + E$
 $\Rightarrow (E * E) + E$
 $\Rightarrow (\underline{id} * E) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + E$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

Rightmost Derivation:

E
 $\Rightarrow E + E$
 $\Rightarrow E + \underline{id}$
 $\Rightarrow (E) + \underline{id}$
 $\Rightarrow (E * E) + \underline{id}$
 $\Rightarrow (E * \underline{id}) + \underline{id}$
 $\Rightarrow (\underline{id} * \underline{id}) + \underline{id}$

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$



Parse Tree

Given a leftmost derivation, we can build a parse tree.

Given a rightmost derivation, we can build a parse tree.

Leftmost Derivation of

(id*id)+id

Rightmost Derivation of

(id*id)+id

Same Parse Tree



Every parse tree corresponds to...

- A single, unique leftmost derivation
- A single, unique rightmost derivation

Ambiguity:

However, one input string may have several parse trees!!!

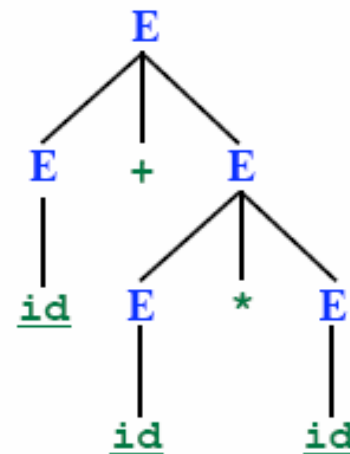
Therefore:

- Several leftmost derivations
- Several rightmost derivations

Ambiguous Grammar

Leftmost Derivation #1

E
 $\Rightarrow E + E$
 $\Rightarrow \underline{id} + E$
 $\Rightarrow \underline{id} + E * E$
 $\Rightarrow \underline{id} + \underline{id} * E$
 $\Rightarrow \underline{id} + \underline{id} * \underline{id}$

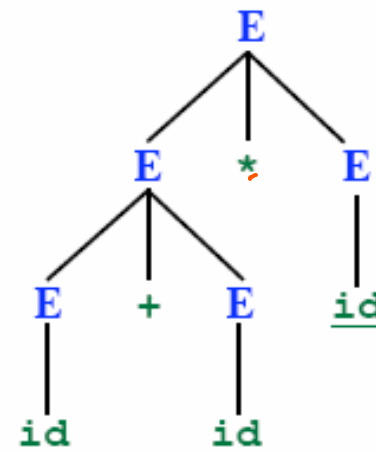


1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

Input: $id + id * id$

Leftmost Derivation #2

E
 $\Rightarrow E * E$
 $\Rightarrow E + E * E$
 $\Rightarrow \underline{id} + E * E$
 $\Rightarrow \underline{id} + \underline{id} * E$
 $\Rightarrow \underline{id} + \underline{id} * \underline{id}$



Ambiguous Grammar

- More than one Parse Tree for some sentence.
 - The grammar for a programming language may be ambiguous
 - Need to modify it for parsing.
- Also: Grammar may be left recursive.
- Need to modify it for parsing.