Context Free Grammars

Derivations vs Parses

Grammar is used to derive string or construct parser

A **derivation** is a sequence of applications of rules
- Starting from the start symbol
- S ⇒ ... ⇒ ... ⇒ ... ⇒ (sentence)

Leftmost and rightmost derivations
- At each derivation step, a **leftmost** derivation always replaces the leftmost non-terminal symbol
- **Rightmost** derivation always replaces the rightmost one

Example

E → E * E | E + E | (E) | id

Leftmost derivation:
E ⇒ E + E ⇒ E * E + E ⇒ id * E + E ⇒ id * id + E ⇒ ...
⇒ id * id + id * id

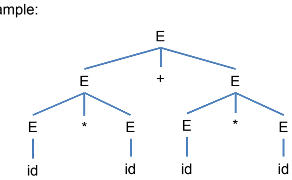
Rightmost derivation:
E ⇒ E + E ⇒ E + E * E ⇒ E + E * id ⇒ E + id * id ⇒ ...
⇒ id * id + id * id

Parse Tree

Parse tree:
- Internal nodes are non-terminals
- Leaves are terminals

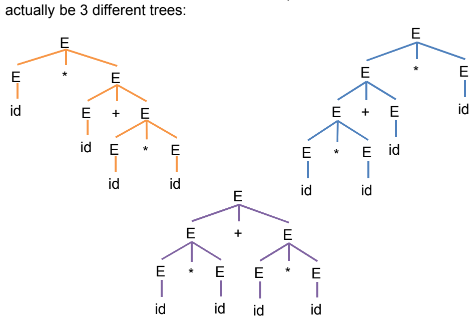
It filters out the order of replacement and describes the hierarchy

The same parse tree results from both the rightmost and leftmost derivations in the previous example:



Different Parse Trees

While the two derivations could have the same parse tree for id * id + id * id there can actually be 3 different trees:



Ambiguity

A grammar G is **ambiguous** if there exists a string *str* ∈ L(G) such that more than one parse trees derive *str*

We prefer unambiguous grammars.

Ambiguity is the property of a grammar and not the language

It is possible to rewrite the grammar to remove ambiguity

Removing Ambiguity

Method 1: Specify **precedence**.

You can build precedence into the grammar by having a different non-terminal for each precedence level:

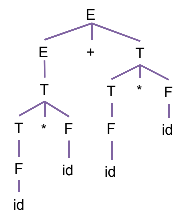
- Lowest level — highest in the tree (lowest precedence)
- Highest level — lowest in the tree
- Same level — same precedence

For the previous example,

E → E * E | E + E | (E) | id

rewrite it to:

E → E + T | T
T → T * F | F
F → id | (E)



Removing Ambiguity

Method 2: Specify **associativity**.

- When recursion is allowed, we need to specify associativity

For the previous example,

E → E * E

Allows both right and left associativity.

We can rewrite it to force it either way:

Left associative :

E → E * T

Right associative:

E → T * E

In a programming language, most operators are left associative.

Syntax Analysis

We've only discussed grammar from the point of view of derivation.

What is **syntax analysis**?

- To process an input string for a given grammar, and compose the derivation if the string is in the language
- Two subtasks:
  - to determine if string in the language or not
  - to construct the parse tree

Is it possible to construct such a parser?

Types of Parsers

Universal parser

- Can parse any CFG grammar. (Early's algorithm)
- Powerful but extremely inefficient

Top-down parser

- It is goal-directed, expands the start symbol to the given sentence
- Only works for certain class of grammars
- To start from the root of the parse tree and reach leaves
- Find leftmost derivation
- Can be implemented efficiently by hand

Types of Parsers

Bottom-up parser

- It tries to reduce input string to the start symbol
- Works for wider class of grammars
- Starts at leaves and build tree in bottom-up fashion
- Find reverse order of the rightmost derivation
- Automated tool generates it automatically

Parser Output

We have a choice of outputs from the parser:

- A parse tree (concrete syntax tree), or
- An abstract syntax tree

Example Grammar:

`E → int | ( E ) | E + E`

and an input:

`5 + ( 2 + 3 )`

After lexical analysis, we have a sequence of tokens

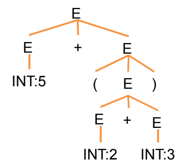
`INT:5 '+' '(' INT:2 '+' INT:3 ')'`

Parser Output

The **parse tree** traces the operation of the parser.

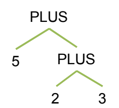
Captures the nested structure but contains too much information:

- Parentheses (precedence encoded in tree hierarchy)
- Single-successor nodes (could be collapsed/omitted)



We prefer an **Abstract Syntax Tree** (AST):

- AST also captures the nested structure.
- AST abstracts from the concrete syntax.
- AST is more compact and easier to use.



Summary

We specify the syntax structure using CFG even if the programming language itself is not context free.

A parser can:

- Answer if an input *str* ∊ L(G)
- and build a parse tree
- or build an AST instead
- and pass it to the rest of compiler.

Parsing

Parsing

We will study two approaches:

Top-down

- Easier to understand and implement manually

Bottom-up

- More powerful, can be implemented automatically

Top Down Parsers

Recursive descent

- Simple to implement, use backtracking

Predictive parser

- Predict the rule based on the 1st *m* symbols without backtracking
- Restrictions on the grammar to avoid backtracking

LL(k) — predictive parser for LL(k) grammar

- Non recursive and only k symbol look ahead
- Table driven — efficient

Parsing Using Backtracking

Approach: For a non-terminal in the derivation, productions are tried in some order until

- A production is found that generates a portion of the input,

or

- No production is found that generates a portion of the input, in which case backtrack to previous non-terminal.

Parsing fails if no production for the start symbol generates the entire input.

Terminals of the derivation are compared against input.

- Match — advance input, continue parsing
- Mismatch — backtrack, or fail

Parsing Using Backtracking

Grammar:

```
E → T + E | T
T → int * T | int | ( E )
```

Input string:

```
int * int
```

Start symbol: E

Assume:

- When there are alternative rules, try right rule first

Parsing Using Backtracking

Input	Derivation	Action
int * int	E	pick rightmost rule E → T
int * int	E ⇒ T	pick rightmost rule T → (E)
int * int	E ⇒ T ⇒ (E)	"(" does not match "int"
int * int	E ⇒ T	Failure, backtrack one level.
int * int	E ⇒ T ⇒ int	pick next rule T → int
int * int	E ⇒ T ⇒ int	"int" matches input "int"
int * int	E ⇒ T	We have more tokens, so this is failure too. Backtrack.
int * int	E ⇒ T ⇒ int * T	Match int * Expand T.
int * int	E ⇒ T ⇒ int * T ⇒ int * (E)	pick rightmost rule E → (E)
int * int	E ⇒ T ⇒ int * T ⇒ int * (E)	"(" does not match input "int"
int * int	E ⇒ T ⇒ int * T	Failure, backtrack one level.
int * int	E ⇒ T ⇒ int * T ⇒ int * int	pick next rule T → int
int * int	E ⇒ T ⇒ int * T ⇒ int * int	Match whole input. Accept.

Implementation

Create a procedure for each non-terminal:

1. Checks if input symbol matches a terminal symbol in the grammar rule
2. Calls other procedure when non-terminals are part of the rule
3. If end of procedure is reached, success is reported to the caller

```
E → int | ( E ) | E + E

void E() {
    switch(lexer.yylex()) {
        case INT:    eat(INT); break;
        case LPAREN: eat(LPAREN); E(); eat(RPAREN); break;
        case ???:    E(); eat(PLUS); E(); break;
    }
}
```

Problems

Unclear what to label the last case with.

What if we don't label it at all and make it the default?

Consider parsing 5 + 5:

We'd find INT and be done with the parse with more input to consume. We'd want to backtrack, but there's no prior function call to return to.

What if we put the call to E() prior to the switch/case?

Then E() would always make a recursive call to E() with no end case for the recursion.

Left Recursion

A production is left recursive if the same nonterminal that appears on the LHS appears first on the RHS of the production.

Recursive descent parsers cannot deal with left recursion.

However, we can rewrite the grammar to represent the same language without the need for left recursion.

Removing Left Recursion

In general, we can eliminate all **immediate** left recursion:

```
A → A x | y
```

By changing the grammar to:

```
A → y A'
A' → x A' | ε
```

Not all left recursion is immediate may be hidden in multiple production rules

```
A → BC | D
B → AE | F
```

There is a general approach for removing **indirect** left recursion, but we'll not worry about if for this course.

Recursive Descent Summary

Recursive descent is a simple and general parsing strategy
- Left-recursion must be eliminated first
- But this can be done automatically

It is not popular because of its inefficiency:
- Backtracking re-parses the string
- Undoing **semantic actions** (actions taken upon matching a production much like the actions from our lexer) may be difficult!

Techniques used in practice do no backtracking at the cost of restricting the class of grammar