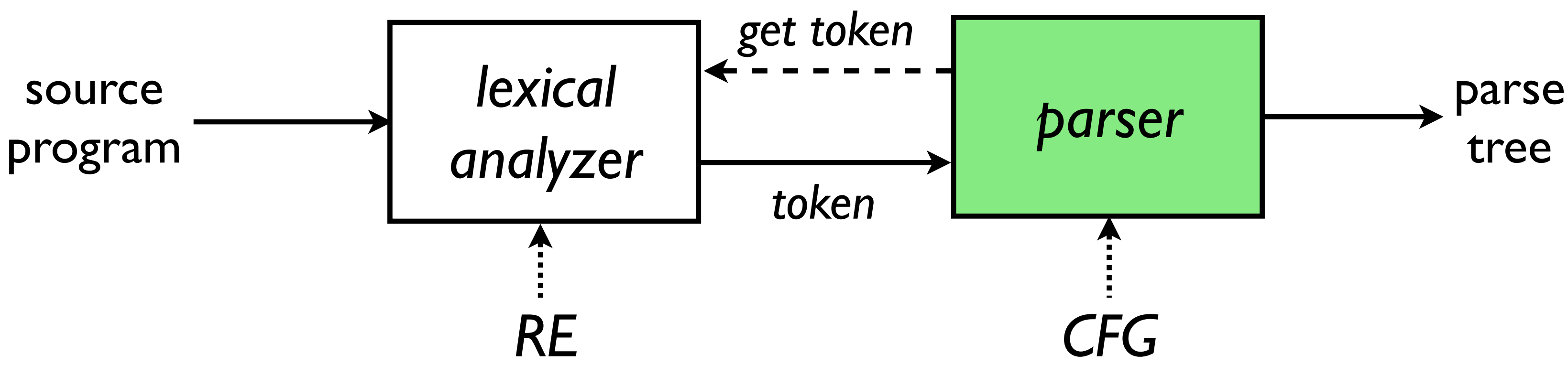
Lecture 3
Parsing

Syntax Analysis

- *Transform a sequence of* ***tokens*** *into a* ***parse tree****:*



- *Syntactic structure is specified using* ***contex-free grammars***
- *A parse tree is a representation of the hierarchical structure of a phrase in the language.*
- *Secondary tasks: syntax error detection and recovery*

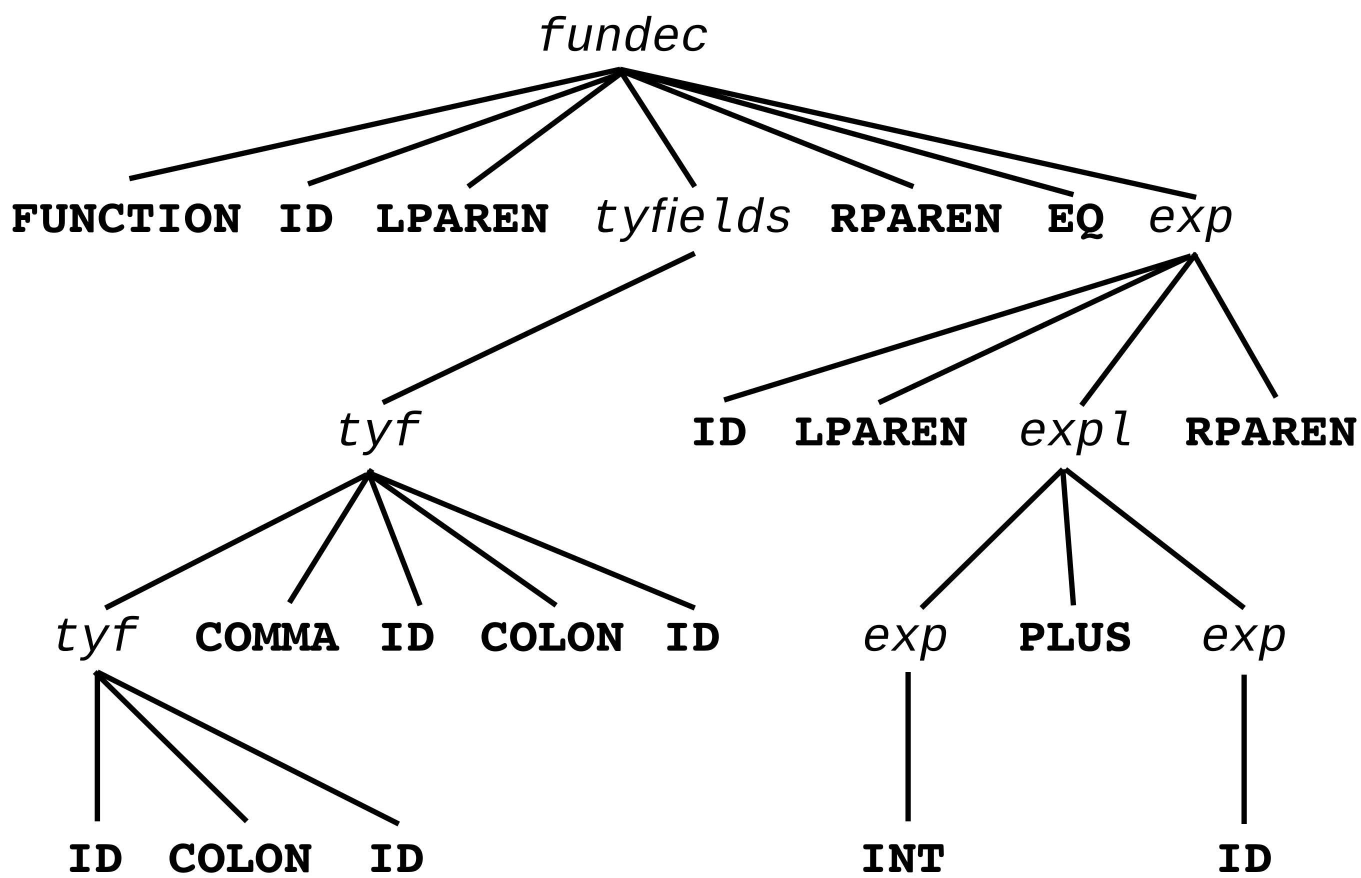
Syntax Analysis

`function f(a:int,b:string) = g(1+a)`

Tokens

FUNCTION
ID(f)
LPAREN
ID(a)
COLON
ID(int)
COMMA
ID(b)
COLON
ID(string)
RPAREN
EQ
ID(g)
LPAREN
INT(1)
PLUS
ID(a)
RPAREN

Parse Tree



Main Parsing Problems

- *How to specify the syntactic structure of a programming language?*

  use *Context-Free Grammars* (CFG)

- *How to parse: given CFG and token stream, how to build the parse tree?*
  - bottom up parsing
  - top down parsing
- *How to make sure parse tree is unique? (the ambiguity problem)*
- *Given a CFG, how to build a parser?*

  use ML-Yacc *parser generator*

- *How to detect, report, and recover from syntax errors*

Grammars

A *grammar* is a precise specification of a programming language *syntax*.

A grammar is normally specified using *Bachus-Naur Form* (BNF):

1. two sets of symbols

   terminal: **if**, **id**, **(**, **)** (the lexical tokens)

   nonterminal: *stmt*, *expr* (the phrase classes)

2. a set of *productions* or *rewriting rules*

```
stmt -> if expr then stmt else stmt
expr -> expr + expr | expr * expr
      | ( expr ) | id
```

The latter abbreviates the 4 rules:

```
expr -> expr + expr
expr -> expr * expr
expr -> ( expr )
expr -> id
```

Context-Free Grammars (CFG)

A *context-free grammar* is defined as a quadruple <T, N, P, S>, where

T is a finite set of terminal symbols
N is a finite set of nonterminal symbols
P is a finite set of *productions*:

$$N \rightarrow \sigma \quad \text{with} \quad N \in \mathrm{N} \text{ and } \sigma \in (\mathrm{N} \cup \mathrm{T})^*$$

S ∈ N is the *start symbol*

Example

T = { **+**, *****, **(**, **)**, **id** }

N = { *E* }

P = { *E* → *E* **+** *E*, *E* → *E* ***** *E*, *E* → **(***E***)**, *E* → **id** }
S = *E*

BNF: *E* → *E* **+** *E* | *E* ***** *E* | **(** *E* **)** | **id**

Derivations

A *sentence* is a string of terminal symbols (or tokens).

A derivation is a sequence of strings in $(N \cup T)^*$, starting with the start symbol S, where each string is produced by replacing a nonterminal with the rhs of one of its productions.

```
E
E + E
E + E * E
E + id * E
(E) + id * E
(E) + id * id
(E * E) + id * id
(id * E) + id * id
(id * id) + id * id
```

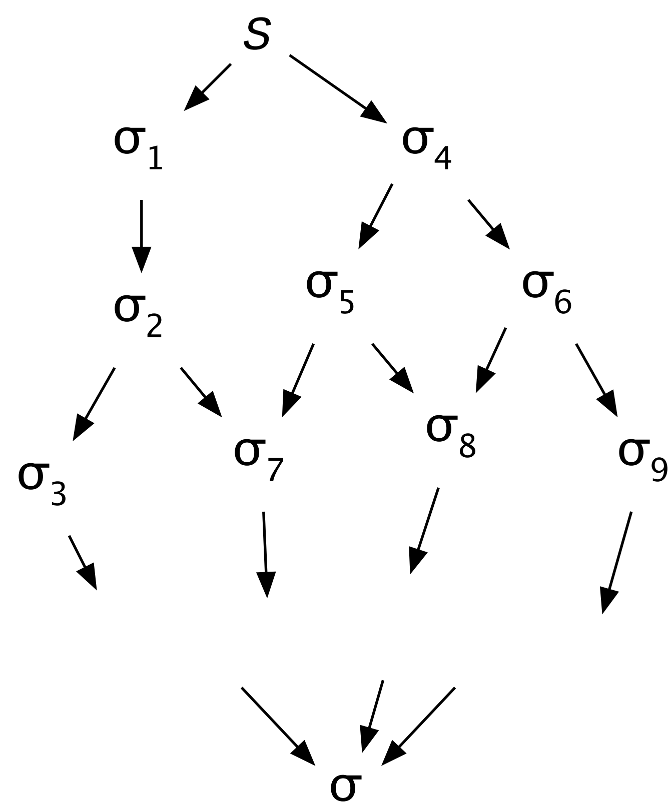
a sentence (no nonterminals)

```
E
E + E
E + E * E
E + E * id
E + id * id
(E) + id * id
(E * E) + id * id
(E * id) + id * id
(id * id) + id * id
```

a leftmost derivation

Multiple Derivations

$S \rightarrow^* \sigma$



There will be multiple derivations taking the start symbol S to a terminal sentence σ, depending on order in which productions are applied. Each path determines a parse tree.

Language of a CFG

A derivation is a sequence

$$S \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \ . \ . \ . \ \rightarrow \sigma_n \quad (\text{or } S \rightarrow^* \sigma_n)$$

where σ_n consists only of terminal symbols ($\sigma_n \in T^*$).

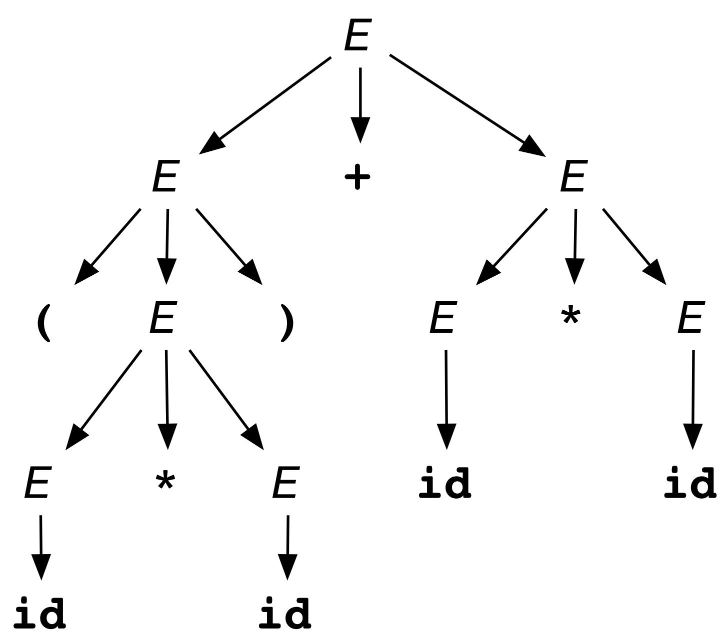
The language L(G) defined by grammar G = <T, N, P, *S*> is the set of strings of terminals that are derivable from *S*:

$$L(G) = \{ \sigma \in T^* \mid S \rightarrow^* \sigma \}$$

Parse Trees

A *parse tree* is a graphical representation of a derivation, but the order of nonterminal replacements is not indicated.

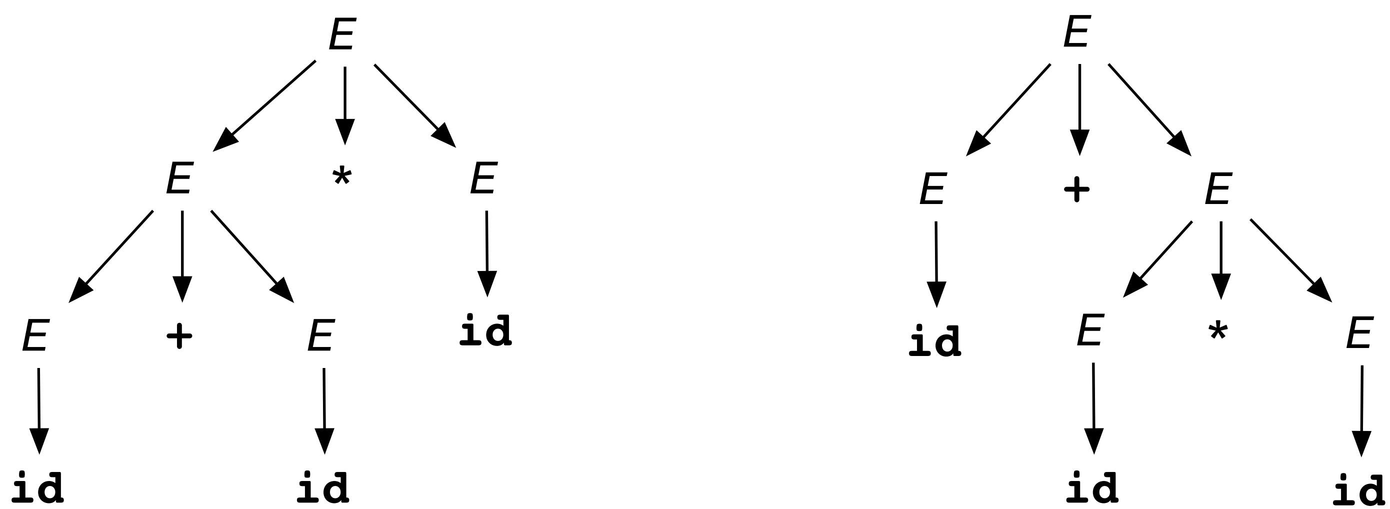
(id * id) + id * id



Ambiguity

A single sentence can have multiple parse trees, meaning that its structure in ambiguous. We say the CFG is *ambiguous*.

`id + id * id`



Removing Ambiguity

There are techniques for transforming a grammar to remove certain kinds of ambiguities.

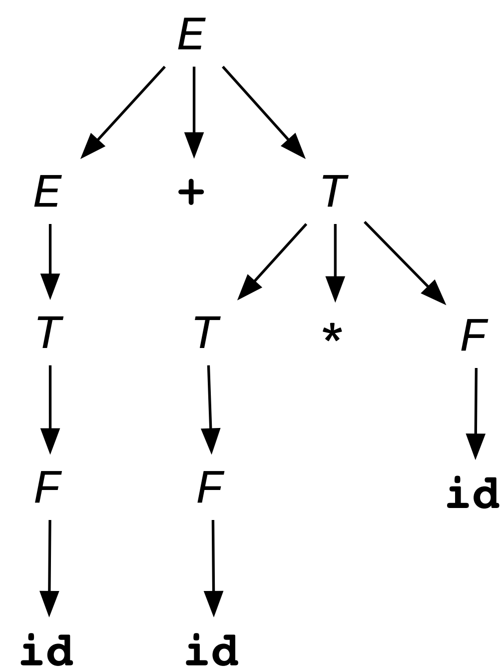
Ambiguous

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

Unambiguous

$E \rightarrow E$ **+** T
$T \rightarrow T$ ***** F
$F \rightarrow$ **(**E**)**
$F \rightarrow$ **id**

id + id * id



Idea: Express precedence through new nonterminals.

Top-Down parsers

A top-down parser tries to construct a parse tree top-down as it scans the token stream from left to right.

This can require *backtracking*, but most programming languages can be parsed without backtracking.

Recursive descent or *predictive parsing* is a type of top-down parsing that can be used when:

1) production rules can be distinguished based on the possible first tokens of sentences derived from their rhs (FIRST sets)

2) there are no left-recursive productions (e.g. $E \rightarrow E + E$ *)*

Recursive Descent

$S \to$ **if** E **then** S **else** S
$S \to$ **begin** $S\ L$
$S \to$ **print** E
$E \to$ **num = num**

$L \to$ **end**
$L \to$ **;** $S\ L$

First sets:

FIRST(**if** E **then** S **else** S **) = {if}**
FIRST(**begin** $S\ L$ **) = {begin}**
FIRST(**print** E **) = {print}**

FIRST(**end** **) = {end}**
FIRST(**;** $S\ L$ **) = {;}**

FIRST(**num = num** **) = {num}**

Note that each rule is uniquely determined by the first symbol of the sentences it generates.

Recursive Descent Parser

```
datatype token = IF | THEN | ELSE | BEGIN | END |
               | PRINT | SEMI | NUM | EQ

val nexttok = ref(getToken())
fun match t = if !nexttok = t
              then nexttok := getToken()
              else error()

fun S() = case !nexttok
            of IF => (match IF; E(); match THEN; S();
                      match ELSE; S())
             | BEGIN => (match BEGIN; S(); L())
             | PRINT => (match PRINT; E())

and L() = case !nexttok
            of END => match END
             | SEMI => (match SEMI; S(); L())

and E() = (match NUM; match EQ; match NUM)
```

Recursive Descent Parser

Notes:

There is a set of recursive functions, one representing each nonterminal symbol.

For each of these functions there is a case rule for each production for that nonterminal, guarded by the first symbol generated by the rhs of the production.

Each production has a unique first symbol that can be used to distinguish it from other possible productions. In general there might be a set of possible first symbols, but these sets would need to be disjoint for the different productions so they could be used to "predict" the proper production.

Left Recursive Productions

A production like

$E \rightarrow E + T$

is bad because it would lead to a function definition for E of the form:

```
fun E() = (E(); match PLUS; T())
```

which would clearly not terminate -- it would not even look at the next token.

There is a systematic way to transform productions to eliminate left recursion. This results in:

$E \rightarrow T\ R$
$R \rightarrow +\ T\ R$
$R \rightarrow \epsilon$

Eliminating Left Recursion

Transform a left recursive production of the form

$$A \rightarrow A \ \alpha$$
$$A \rightarrow \beta$$

by introducing a new nonterminal R with productions

$$A \rightarrow \beta \ R$$
$$R \rightarrow \alpha \ R$$
$$R \rightarrow \epsilon$$

This can be generalized if there are several left recursive productions:

$$\begin{array}{l} A \rightarrow A \ \alpha \\ A \rightarrow A \ \gamma \\ A \rightarrow \beta \end{array} \quad \Rightarrow \quad \begin{array}{l} A \rightarrow \beta \ R \\ A \rightarrow \gamma \ R \\ R \rightarrow \alpha \ R \\ R \rightarrow \epsilon \end{array}$$

Top-Down parsers

We need to formally define the set of possible first tokens generated from a sentence $\alpha \in (N \cup T)^*$ (a production rhs).

FIRST(α) is the set of possible first tokens that can occur in sentences generated from α. If the string α starts with a nonterminal, then that nonterminal constitutes the FIRST set.
If α starts with a terminal, we may have to resort to the more complicated algorithm described in Algorithm 3.13 (Appel, p. 49).