Lecture 7-8 — Context-free grammars and bottom-up parsing

- **Language syntax is described by context-free grammars, and the job of the parser is to transform a (syntactically correct) input to a tree. A cfg defines a set of syntax trees; abstract syntax trees are a simplified version of those.**
- **Topics for these two classes are:**
  - **Ambiguity and expression grammars**
  - **Bottom-up (shift/reduce) parsing**
  - `ocamlyacc`

Grammar for (almost) MiniJava

```
Program -> ClassDeclList
ClassDecl -> class id { VarDeclList MethodDeclList }
VarDecl -> Type id ;
MethodDecl -> Type id ( FormalList ) { VarDeclList StmtList return Exp ; }
Formal -> Type id
Type -> int [ ] | boolean | int | id
Stmt -> { StmtList } | if ( Exp ) Stmt else Stmt
      | while ( Exp ) Stmt | System.out.println ( Exp ) ;
      | id = Exp ; | id [ Exp ] = Exp ;
Exp -> Exp Op Exp | Exp [ Exp ] | Exp . length
      | Exp . id ( ExpList ) | integer | true | false | id
      | this | new int [ Exp ] | new id ( ) | ! Exp | ( Exp )
Op -> && | < | <= | == | + | - | *
ExpList -> Exp ExpRest |
ExpRest -> , Exp ExpRest |
FormalList -> Type id FormalRest |
FormalRest -> , Type id FormalRest |
ClassDeclList = ClassDeclList VarDecl |
MethodDeclList = MethodDeclList MethodDecl |
VarDeclList = VarDeclList VarDecl |
StmtList = StmtList Stmt |
```

Parse tree example

- **Grammar describes a set of *parse trees* (aka *syntax trees*, *concrete syntax trees*).**
- **E.g. program** `class C {int f () { return 0; }}` **has this parse tree:**

  (*left as an exercise*)

- **It is easy to see the relationship between this concrete syntax tree and the abstract syntax of MiniJava.**

Context-free grammar (cfg) notation

- **A CFG is a set of productions** $A \to X_1 X_2 \ldots X_n$ $(n \geq 0)$**. If** $n = 0$**, we may write either** $A \to$ **or** $A \to \epsilon$**.**
- $A, B, C.... \in N$**, the set of** ***non-terminals*****. One non-terminal in the grammar is the** ***start symbol*****.**
- T **is the set of** ***tokens*****, aka** ***terminals***
- $X, Y, Z \in S = N \cup T$**, the set of** ***grammar symbols***
- $u, v, w \in T^*$
- $\alpha, \beta, \gamma \in S^*$
- **Productions** $A \to \alpha$**,** $A \to \beta$**, ..., abbreviated as** $A \to \alpha \mid \beta \mid ...$
- **A** ***parse tree*** **is a tree whose root is labelled with the start**

symbol, and whose other nodes are labelled with grammar symbols or the special symbol "$\bullet$". Furthermore, if a node labelled with non-terminal B has children, there must be a production $B \rightarrow X_1 X_2 \dots X_n$ such that either: the node has exactly n children and they are labelled $X_1, X_2, \dots, X_n$; or $n = 0$ (i.e. an ϵ-production) and the node has exactly one child, which is labelled "$\bullet$".

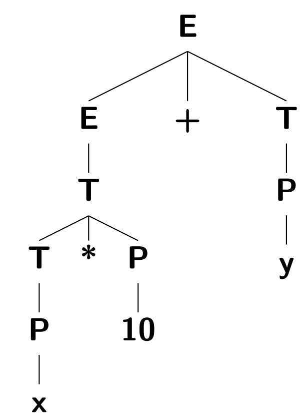
- *N.B. The above definition does not require that every non-terminal node have children.*

- A *sentential form* is any frontier of a parse tree (i.e. labels of the leaf nodes), with $\bullet$'s deleted. An *sentence* is a sentential form in T^*.

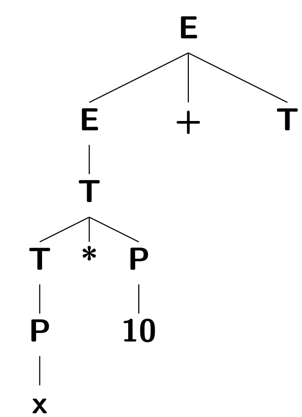
Exercises on cfg notation

$$E \longrightarrow E + T \mid T$$
$$T \longrightarrow T * P \mid P$$
$$P \longrightarrow id \mid int \mid (\ P\ )$$

- **Parse tree whose frontier is:** `x * 10 + y`**:**



- **Parse tree whose frontier is:** `x * 10 + T`**:**



More cfg notation

- A *parse tree from* A is defined the same as a parse tree, except the root is labelled with A.

- Similarly, an A-*form* is the frontier of a parse tree from A (with $\bullet$'s deleted), and an A-*sentence* is an A-form consisting only of tokens.

- $A \in N$ is *nullable* if ϵ is an A-sentence.

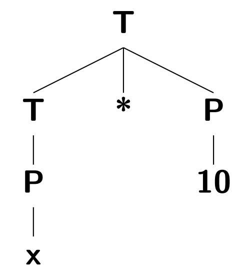
- An *ambiguous grammar* is one for which at least one sentence has more than one parse tree.

- A *parser* is a function of type string $\rightarrow$ AST $\cup$ $\{error\}$. (A *recognizer* is a function of type string $\rightarrow$ bool.)

More exercises on cfg notation

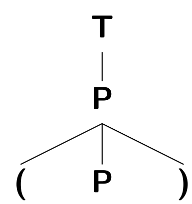
$E \longrightarrow E + T \mid T$
$T \longrightarrow T * P \mid P$
$P \longrightarrow id \mid int \mid (\ P\ )$

- **Parse tree from T whose frontier is:** `x * 10`**:**



- **A T-form that is not a T-sentence, and its parse tree:**

$$(P)$$



Extended cfg's

- An *extended cfg* is one whose right-hand sides can contain regular expression operations $*$, $+$, and $?$. These are used to abbreviate ordinary context-free rules:
  - β^* should be replaced by a new non-terminal, say B, and additional rules $B \rightarrow \epsilon \mid \beta B$ (or, equivalently, $B \rightarrow \epsilon \mid B\beta$)
  - β^+ should be replaced by a new non-terminal, say B, and additional rules $B \rightarrow \beta \mid \beta B$ (or, equivalently, $B \rightarrow \beta \mid B\beta$)
  - $\beta?$ should be replaced by a new non-terminal, say B, and additional rules $B \rightarrow \beta \mid$.

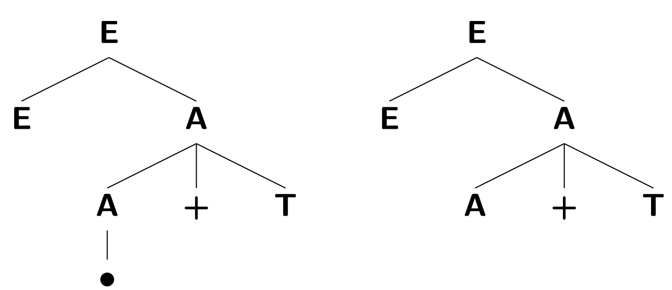
Exercises on extended cfg notation

$E \rightarrow EA$
$A \rightarrow \epsilon \mid A + T$
$T \rightarrow T(*P)^*$
$P \rightarrow id \mid int \mid (\ P\ )$

- **Transform the rule for T to remove the Kleene star.**

  $T \rightarrow TB$
  $B \rightarrow B * P \mid \epsilon$

- **Show that both `E+T` and `EA+T` are sentential forms:**



ASTs for expressions

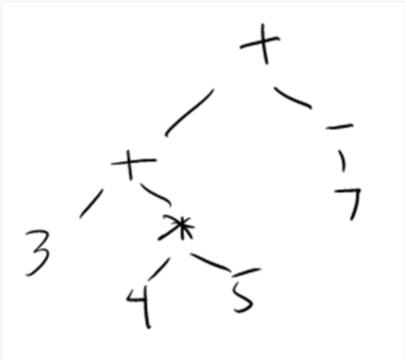
- **Recall from lecture 3 how we defined a simple abstract syntax for expressions and an `eval` function for it:**

```
type expr = Int of int | Plus of expr*expr
          | Times of expr*expr | Negate of expr

let rec eval e = match e with
      Int i -> i
    | Plus (e1, e2) -> eval e1 + eval e2
    | Times (e1, e2) -> eval e1 * eval e2
    | Negate e1 -> -(eval e1)
```

Expression grammars

- **The expression `3 + 4*5 + -7` has AST**



  The shape of this AST represents the precedence of multiplication and the left-associativity of addition. This ensures that `eval` would return the correct value.

- **Parsing produces a parse tree which is translated to an AST. It simplifies this translation greatly if the shape of the concrete syntax tree correctly represents precedences and associativities of operators.**

Some expression grammars

G_A: E $\longrightarrow$ id | E - E | E * E

G_B: E $\longrightarrow$ id | id - E | id * E

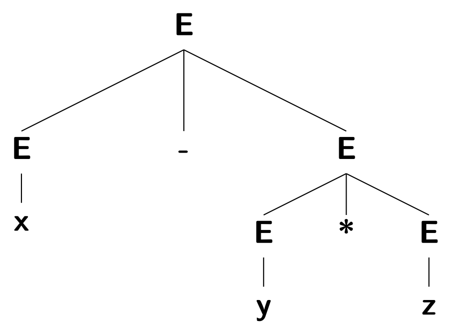
G_C: E $\longrightarrow$ id | E - id | E * id

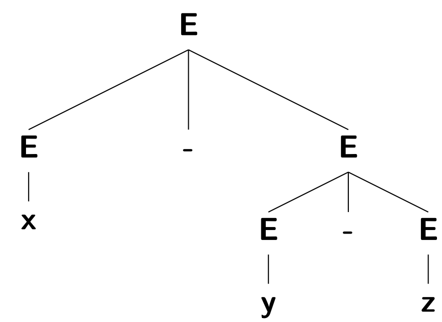
G_D: E $\longrightarrow$ T - E | T
T $\longrightarrow$ id | id * T

G_E: E $\longrightarrow$ E - T | T
T $\longrightarrow$ id | T * id

G_F: E $\longrightarrow$ T E′
E′ $\longrightarrow$ ϵ | - E
T $\longrightarrow$ id T′
T′ $\longrightarrow$ ϵ | * T

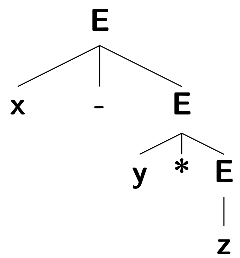
- G_A: **E** $\rightarrow$ **id** | **E - E** | **E * E**
  - **x - y * z** **x - y - z**

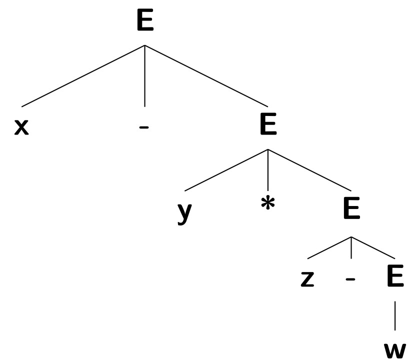




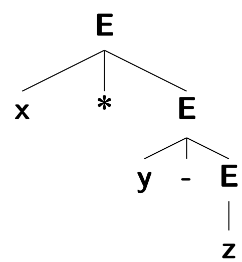
- **Ambiguous?** *Yes* **Precedence?** *No* **Associativity?** *None*

- G_B: **E** $\rightarrow$ **id | id - E | id * E**
  - **x - y * z** **x - y * z - w**



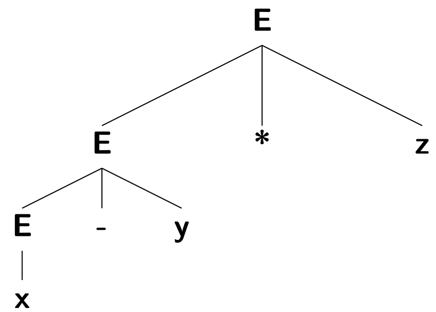


  - **x * y - z**

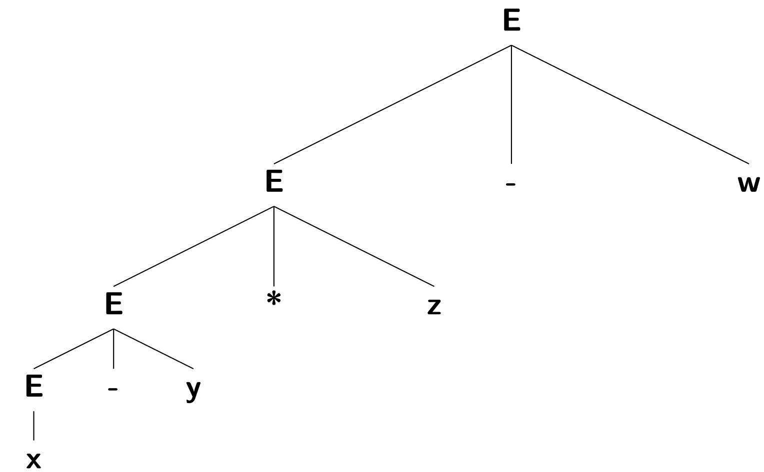


- **Ambiguous?** *No* **Precedence?** *No* **Associativity?** *Right*

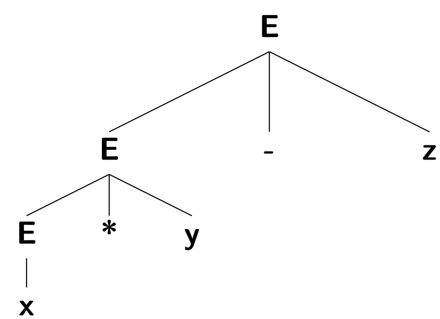
- G_C: **E $\rightarrow$ id | E - id | E * id**
  - **x - y * z**



x - y * z - w

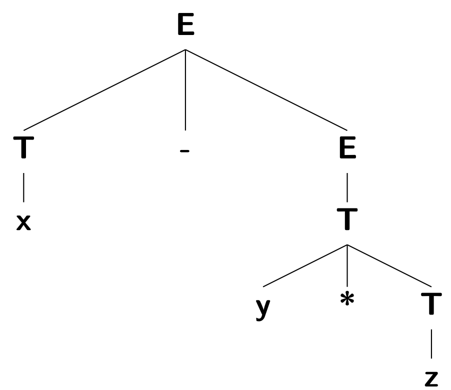


  - **x * y - z**

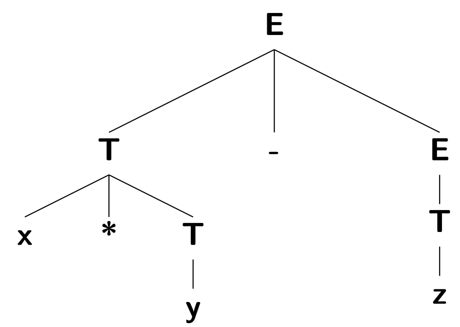


- **Ambiguous?** *No* **Precedence?** *No* **Associativity?** *Left*

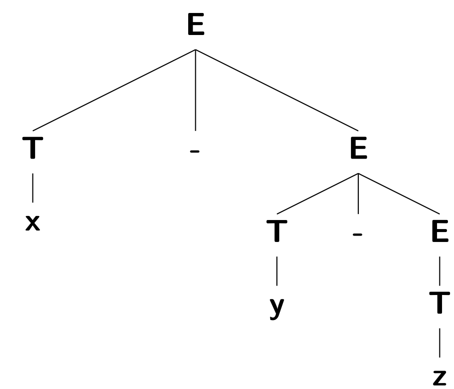
- G_D: **E** $\longrightarrow$ **T - E** $|$ **T**
  T $\longrightarrow$ **id** $|$ **id * T**
  - **x - y * z**



x * y - z



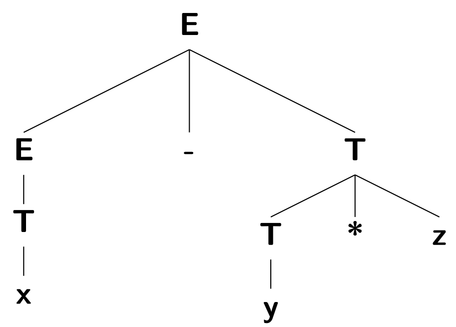
x - y - z

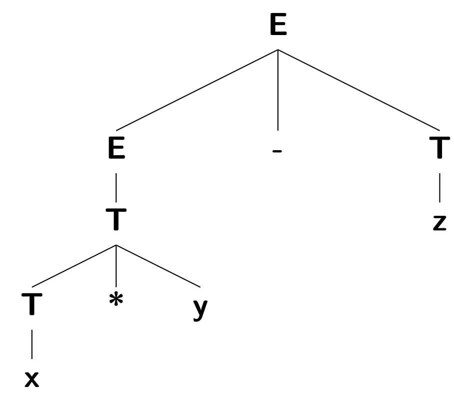


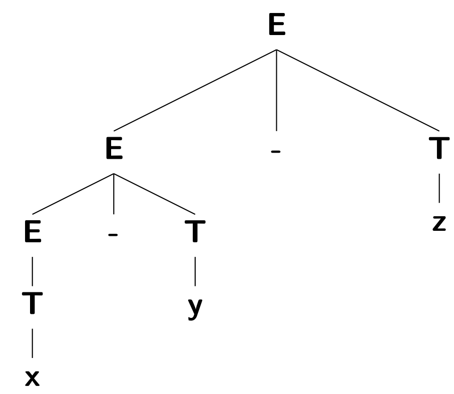
- **Ambiguous?** *No* **Precedence?** *Yes* **Associativity?** *Right*

- G_E: **E $\longrightarrow$ E - T | T**
  T $\longrightarrow$ id | T * id

  - **x - y * z** **x * y - z** **x - y - z**







- **Ambiguous?** *No* **Precedence?** *Yes* **Associativity?** *Left*

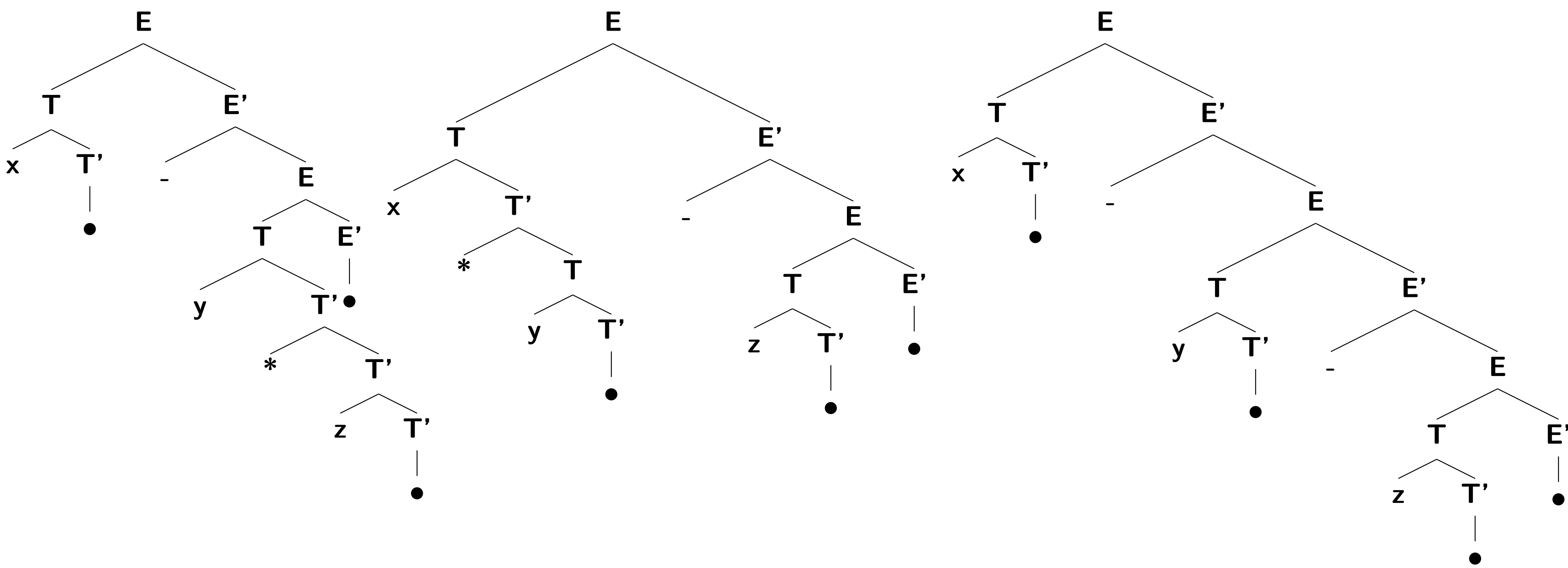
- G_F: $\textbf{E} \longrightarrow \textbf{T E}'$

  $\textbf{E}' \longrightarrow \epsilon \mid \textbf{- E}$

  $\textbf{T} \longrightarrow \textbf{id T}'$

  $\textbf{T}' \longrightarrow \epsilon \mid \textbf{* T}$

  - **x - y * z** **x * y - z** **x - y - z**



- **Ambiguous?** *No* **Precedence?** *Yes* **Associativity?** ***Right***

Parser generators

- **Like lexer generators, these are programs that input a specification — in the form of a context-free grammar, with an action associated with each production — and output a parser.**
- **The most famous of all parser generators is `yacc` — which, ironically, stands for "yet another compiler compiler." Originally written to generate parsers in C, it has been copied in many other languages. We will use `ocamlyacc`.**
- **We start with a small but complete example.**
- **Next week, we will discuss how to write a parser by hand, using the method of *recursive descent*.**

Example - expression grammar

- **In this example, we will use ocamlyacc to create a parser for this grammar:**

  $$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E} + \mathbf{T} \mid \mathbf{E} - \mathbf{T}$$
  $$\mathbf{T} \rightarrow \mathbf{P} \mid \mathbf{T} * \mathbf{P} \mid \mathbf{T} / \mathbf{P}$$
  $$\mathbf{P} \rightarrow \mathbf{id} \mid (\mathbf{E})$$

- **It will product ASTs of type** `exp`**:**

```
(* File: exp.ml *)
type exp =
    Plus of exp * exp
  | Minus of exp * exp
  | Mult of exp * exp
  | Div of exp * exp
  | Id of string
```

Example - exprlex.mll

```
{ type token = PlusT | MinusT | TimesT | DivideT
             | OParenT | CParenT | IdT of string | EOF }
let numeric = ['0' - '9']
let letter = ['a' - 'z' 'A' - 'Z']
rule tokenize = parse
  | "+"           {PlusT}
  | "-"           {MinusT}
  | "*"           {TimesT}
  | "/"           {DivideT}
  | "("           {OParenT}
  | ")"           {CParenT}
  | letter (letter | numeric | "_")* as id    {IdT id}
  | [' ' '\t' '\n']    {tokenize lexbuf}
  | eof           {EOF}
```

Example - exprparse.mly

```
%token <string> IdT
%token OParenT CParentT TimesT DivideT PlusT MinusT EOF
%start main
%type <exp> main
%%
expr:
    term                    {$1}
  | expr PlusT term         {Plus($1,$3)}
  | expr MinusT term        {Minus($1,$3)}

term:
    factor                  {$1}
  | term TimesT factor      {Mult($1,$3)}
  | term DivideT factor     {Div($1,$3)}

factor:
    IdT                     {Id $1}
  | OParenT expr CParenT {$2}

main:
  | expr EOF {$1}
```

Shift-reduce parsing

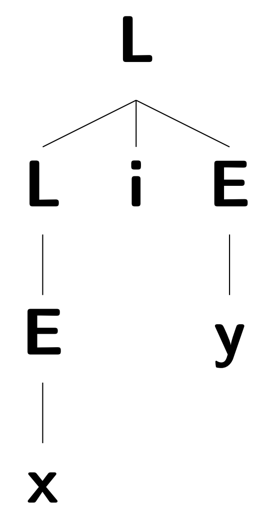
- `ocamlyacc` **uses a method of parsing known as** *shift/reduce*, **a.k.a.** *bottom-up parsing*. **Here's how it works:**
  - **Keep a stack of grammar symbols (initially empty). Based on this stack, and the next input token ("lookahead symbol"), take one of these actions:**
    - **Shift: Move lookahead symbol to stack**
    - **Reduce $A \rightarrow \alpha$: Symbols on top of stack are α; replace them by A.** *(If you create a tree node here, you can construct the parse tree while parsing.)*
    - **Accept: When stack consists of just the start symbol, and input is exhausted**
    - **Reject**

Shift-reduce example 1

- $L \rightarrow L\ ;\ E \mid E$
  $E \rightarrow id$

Input: x; y

Action	Stack	Input
Shift		x ; y
Reduce $E \rightarrow id$	x	; y
Reduce $L \rightarrow E$	E	; y
Shift	L	; y
Shift	L ;	y
Reduce $E \rightarrow id$	L ; y	
Reduce $L \rightarrow L$; E	L ; E	
Accept	E	

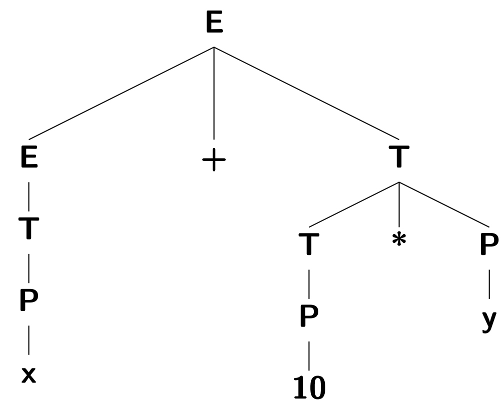


Shift-reduce example 2

- $$E \rightarrow E + T \mid T$$
  $$T \rightarrow T * P \mid P$$
  $$P \rightarrow id \mid int$$

Input: x + 10 * y

Action	Stack	Input
Shift		x + 10 * y
Reduce	x	+ 10 * y
Reduce	P	+ 10 * y
Reduce	T	+ 10 * y
Shift	E	+ 10 * y
Shift	E +	10 * y
Reduce	E + 10	* y
Reduce	$E + P$	* y
Shift	$E + T$	* y
Shift	$E + T$ *	y
Reduce	$E + T$ * y	
Reduce	$E + T$ * P	
Reduce	$E + T$	
Accept	E	



Parsing conflicts

- Parsers for programming languages must be very efficient, but no efficient method for parsing arbitrary cfg's is known. So all parser generators accept only certain grammars. The hardest part of using any parser generator is getting the grammar into a form the parser generator will accept.

- `yacc` (and `ocamlyacc`) accept a class of grammars known as *LALR(1) grammars*. We will not be able to describe exactly what distinguishes this class; that is done in CS 426.

- If you present a non-LALR(1) grammar to ocamlyacc, it will report an error message, called a *conflict*.

- To understand how to deal with conflicts, we need to look at shift-reduce parsing a little closer.

S/R parsing $\equiv$ parse tree

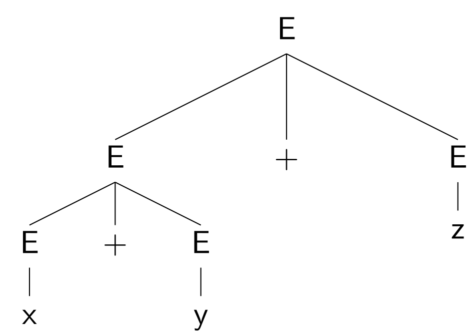
- **Each shift-reduce parse — i.e. each sequence of s/r actions — produces a unique parse tree.**
- **Every parse tree is built by a unique s/r parse:**
  - **Traverse the tree in post-order. A leaf node corresponds to a shift action, and an internal node to a reduce action.**
- **An LALR(1) parser generator will accept a grammar only if it can determine, based on the stack and lookahead symbol, the unique correct action.**
- **It follows that ambiguous grammars can never be LALR(1).**

Shift-reduce example 3

- **Grammar:** $E \rightarrow E + E \mid E * E \mid id$
  Input: x + y + z

Show a parse tree, and corresponding s/r parse, that represents left-associativity of addition.

Action	**Stack**	**Input**
Shift		x + y + z
Reduce	x	+ y + z
Shift	E	+ y + z
Shift	E +	y + z
Reduce	E + y	+ z
Reduce	E + E	+ z
Shift	E	+ z
Shift	E +	z
Reduce	E + z	
Reduce	E + E	
Accept	E	



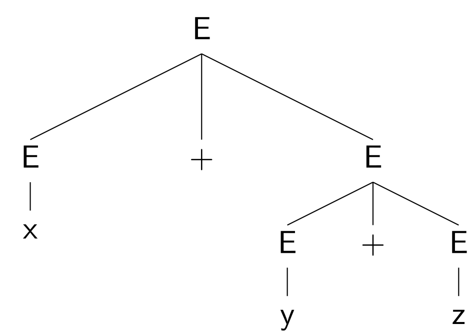
Shift-reduce example 3 (cont.)

- **Grammar:** $E \rightarrow E + E \mid E * E \mid id$

  Input: x + y + z

Show a parse tree, and corresponding s/r parse, that represents right-associativity of addition.

Action	Stack	Input
Shift		x + y + z
Reduce	x	+ y + z
Shift	E	+ y + z
Shift	$E +$	y + z
Reduce	$E +$ y	+ z
Shift	$E + E$	+ z
Shift	$E + E +$	z
Reduce	$E + E +$ z	
Reduce	$E + E + E$	
Reduce	$E + E$	
Accept	E	



Dealing with ambiguity

- `yacc` has a special trick for dealing with ambiguity: annotations telling the parser explicitly what to do in some cases.
- For the previous grammar, there are four interesting inputs: `x+y+z`, `x*y*z`, `x+y*z`, `x*y+z`.
- Consider `x+y+z`. It has two parse trees. For both, the stack looks the same until the second + is the lookahead symbol.
- What is the right decision?

Dealing with ambiguity (cont.)

- **For `x*y*z`, consider where the two stack configurations that can occur for the two parse trees differ. What is the correct decision?**

Stack	Input	Action	New Stack	Stack	Input	Action	New Stack
E * E	* z	Shift	E * E *	E * E	* z	Reduce	E

Reduce is the correct decision by left-associativity of multiplication.

- **Do the same for `x+y*z`:**

Stack	Input	Action	New Stack	Stack	Input	Action	New Stack
E + E	* z	Shift	E + E *	E + E	* z	Reduce	E

Shift is the correct decision by precedence of multiplication over addition.

- **and for `x*y+z`:**

Stack	Input	Action	New Stack	Stack	Input	Action	New Stack
E * E	+ z	Shift	E * E +	E * E	+ z	Reduce	E

Reduce is the correct decision by precedence of multiplication over addition.

Precedence declarations

- **Looking at these four cases, we can see that they will be parsed correctly if we follow these rules:**
  - **If the operator nearest the top of the stack and the lookahead symbol have the same precedence, then *shift* if the operator is right-associative, and *reduce* if it is left-associative.**
  - **If the operator nearest the top of the stack has *higher* precedence than the lookahead symbol, then *reduce*; otherwise, *shift*.**
- `ocamlyacc` **will follow these rules if you tell it which operators have higher precedence, and which are left- or right-associative. Do that using *precedence declarations*...**

Precedence declarations (cont.)

- **Precedence declarations are added to the `ocamlyacc` specification after the `%token` declarations. Syntax:**
  - `%left` ***symbol ... symbol***
  - `%right` ***symbol ... symbol***
  - `%nonassoc` ***symbol ... symbol***

- **Two symbols appearing in the same precedence declaration have the same precedence and the given associativity. (If they appear in a `%nonassoc` declaration, they cannot follow one another, e.g. `x < y < z`.)**

- **Two symbols appearing in different precedence declarations have difference precedences: the one that comes earlier has lower precedence.**

Precedence declaration example

- **Recall this `ocamlyacc` specification:**

```
expr:
    term                    {$1}
  | term PlusT expr         {Plus($1,$3)}
  | term MinusT expr        {Minus($1,$3)}

term:
    factor                  {$1}
  | factor TimesT term      {Mult($1,$3)}
  | factor DivideT term     {Div($1,$3)}

factor:
    IdT                     {Id $1}
  | OParenT expr CParenT {$2}

main:
  | expr EOF                {$1}
```

Precedence declarations (cont.)

- **This can be simplified with precedence declarations (after the `%token` declarations):**

```
%left PlusT MinusT
%left TimesT DivideT
%start main
%type <expr> main
%%
expr:
    IdT                    {Id $1}
  | expr PlusT expr        {Plus($1,$3)}
  | expr MinusT expr       {Minus($1,$3)}
  | expr TimesT expr       {Plus($1,$3)}
  | expr DivideT expr      {Minus($1,$3)}
  | OParenT expr CParenT   {$2}

main:
  | expr EOF               {$1}
```

Debugging ocamlyacc specifications

- **In doing MP4, the main question will be: what operators are causing conflicts? Once you've identified them, you can add precedence declarations.**
- **When you run `ocamlyacc`, it will report the number of conflicts. Running with the `-v` option produces a file with the extension `.output`, containing details.**
- **E.g. grammar `Expr` → `Expr` + `Expr` | `id` has a conflict. Search for "conflict" in the `.output` file:**

```
6: shift/reduce conflict (shift 5, reduce 1) on plus
state 6
Expr : Expr . plus Expr  (1)
Expr : Expr plus Expr .  (1)
```

This says that there is a problem with the `plus` token.

Constructing AST's

- **Precedence rules make it easier to construct AST's, because concrete syntax is closer to abstract syntax.**
- **Different non-terminals can produce different** *types* **of values. An important case is "list-like" syntax categories. E.g. consider this grammar:**

$$
\begin{aligned}
&\mathit{funcall} \rightarrow \mathit{id} \ (\ \mathit{arglist} \ ) \\
&\mathit{arglist} \rightarrow \mathit{funcall} \ \mathit{arglistrest} \mid \\
&\mathit{arglistrest} \rightarrow \ , \ \mathit{funcall} \ \mathit{arglistrest} \mid
\end{aligned}
$$

- **Suppose our abstract syntax is**

```
type funcall = Funcall of string * (funcall list)
```

- **Here is how to do this in `ocamlyacc`:**

```
funcall:
        IdT OParenT arglist CParenT   { Funcall($1, $3) }
arglist:
        funcall arglistrest           { $1 :: $2 }
      |                               { [] }
arglistrest:
        CommaT funcall arglistrest    { $2 :: $3 }
      |                               { [] }
```

Bonus topic: A little LR theory...

- **The shift-or-reduce decision seems very mysterious: We know what to do when we already have the parse tree, but how can we know, based only on the grammar, what will be the correct action in every case?**

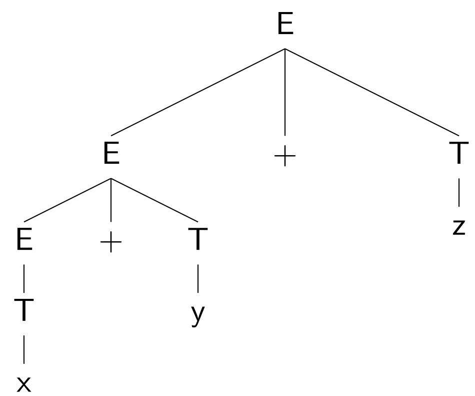
- **We can start by looking at the "stack configurations" of s/r parses. Define $SC(G) = \{ \alpha \in S^* \mid \alpha$ can be the stack in a shift-reduce parse for $G\}$.**

- **Consider this example: $A \rightarrow$ id $\mid$ (A)**

More examples of $SC(G)$

$E \rightarrow E + T \mid T$

$T \rightarrow id$

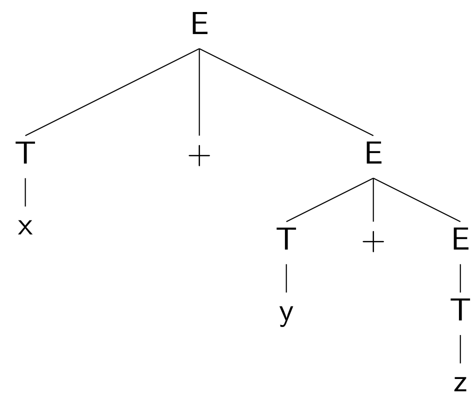


$$SC(G) = \{id, T, E, E+, E + id, E + T\}$$

More examples of $SC(G)$ (contd.)

E $\longrightarrow$ T + E | T

T $\longrightarrow$ id



$SC(G) = \{id, T, T, T+, T + id, T + T, T + T+, T + T + id, T + T + T, T + T + E, T + E, E\}$.

In general, $SC(G) = (T+)^*(id|T|E)$.

A little LR theory (cont.)

- *Theorem [Knuth]* **For any grammar G, $SC(G)$ is a finite-state language over S.**

- `yacc` **starts by constructing the "characteristic DFA" for the specified grammar. To parse a sentence, repeat the following:**

  - **Take the stack and concatenate the lookahead symbol. If the result is in $SC(G)$, then shift; o/w reduce.**

- **Actually, constructing the characteristic DFA is just the start. The simple parsing method just given does not always work. (E.g. even when it says to reduce, it may not be clear which production to reduce by.) The full construction of the parser is quite involved; see CS426, or a compiler textbook.**

Wrap-up

- **Today and Tuesday, we discussed:**
  - **Cfg's, parsing, expression grammars, and ambiguity**
  - **Shift/reduce parsing and ocamlyacc**
- **We discussed it because:**
  - **ocamlyacc-type parser generators are widely-used tools for generating parsers.**
- **Next week, we will:**
  - **Talk about *top-down* parsing, a relatively simple way to write parsers by hand.**
- **What to do now:**
  - ***HW4 (written homework)***