parser

Project-1 discussion
Limitations of regular languages
Parser overview
Context-free grammars (CFG’s)
Derivations
Ambiguity
Error handling

The contents are copied from

- https://web.stanford.edu/class/cs143/lectures/lecture05.pdf
- https://web.stanford.edu/class/cs143/lectures/lecture06.pdf
- Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 1 and sec. 2.1-2.4
- https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Project-1: Specifying a PL (using flex)

Specify rules for identifiers, keywords, etc. for a programming language

Write regular expressions for each

Use flex to tokenize a given input file

Value simplicity!

“Nature is pleased with simplicity. And nature is no dummy”- Isaac Newton

“It's not easy to write good software. […] it has a lot to do with valuing simplicity over complexity.” - Barbara Liskov

“Debugging is twice as hard as writing the code in the first place.Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.” - Brian Kernighan

“Simplicity does not precede complexity, but follows it.” - Alan Perlis

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.” - Tony Hoare

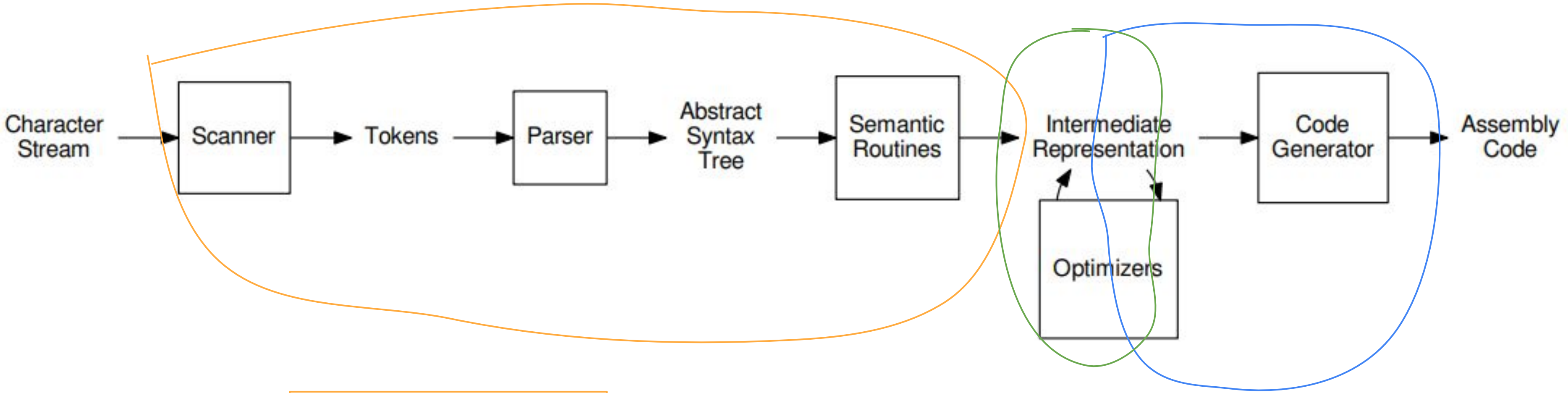
https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Tips on Building Large Systems

- KISS (Keep It Simple, Stupid!)
    - Simple in design, simple to use and modify etc.
- Don’t optimize prematurely
- Design systems that can be tested
- It is easier to modify a working system than to get a system working

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Review: compiler structure

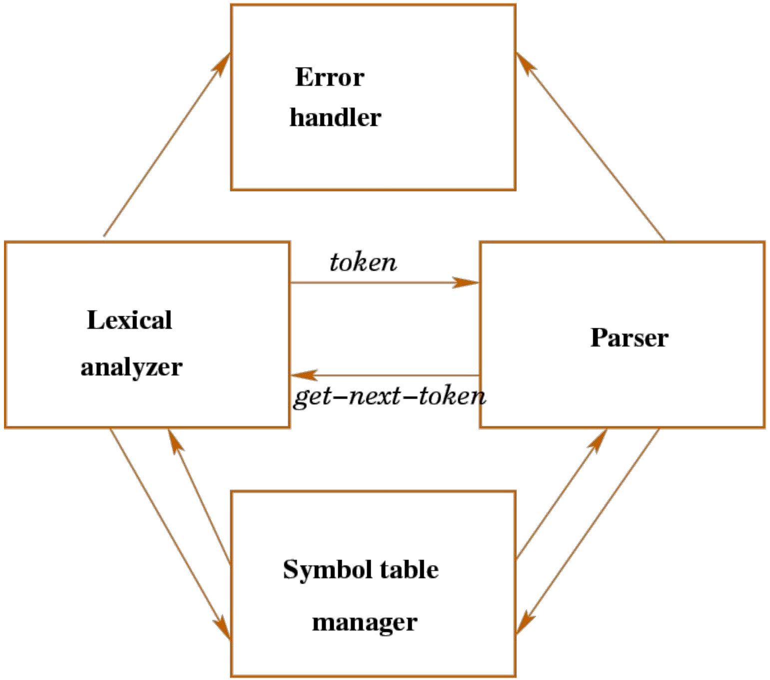


Front end

Back-end

➔ Deals with **target system**

Fig2.2 with some modifications from the book https://www3.nd.edu/~dthain/compilerbook/chapter2.pdf



ABOUT THE SEPARATION OF THE LEXER FROM THE PARSER.

- SIMPLIFICATION OF DESIGN. If the lexical analyzer has removed comments and white space, then the parser's computations will be simpler.
- EFFICIENCY. It is easier to optimize and maintain simple little components than a large sophisticated one.
- PORTABILITY. If the lexical analyzer is the only part of the compiler which has to worry about the input file (alphabetic peculiarities) and its support then the compiler portability will be enhanced.

Other possible tasks of the lexical analyzer:

- Correlating error messages from the compiler to the source program
- Macro expansion.

https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Lexical.html/node14.html

Parser vs scanner

scanning

- is like **constructing words** out of letters,
- w-h-i-l-e - > while

parsing

- is like **constructing sentences** out of words in a natural language

A parser has the primary responsibility for recognizing syntax:

- the program being compiled is a **valid sentence**.

What is a valid sentence?

To parse a computer program,

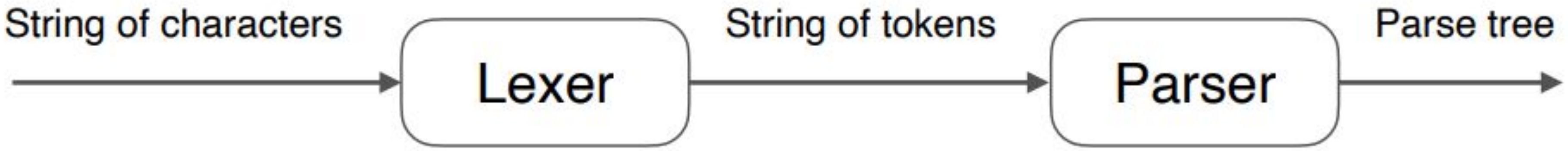
➔ we must first describe the form of valid sentences in a language.

The functionality of parser

Input: sequence of tokens from lexer

Output: parse tree of the program

(Conceptually, but in practice parsers return an AST)



https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

example

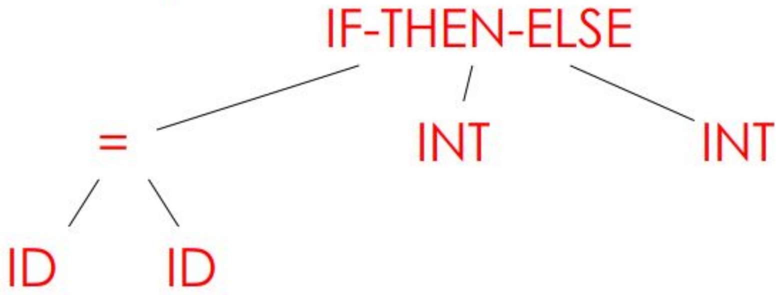
Cool

if x = y then 1 else 2 fi

Parser input

IF ID = ID THEN INT ELSE INT FI

Parser output



https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

The main task of parser:

Not all strings of tokens are programs . . .

If while fi try 4

. . . parser must distinguish the strings of tokens between valid and invalid

- **If i = 4 then 1;**
  - **valid**
- **If while fi try 4**
  - **invalid**

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

We need

➔ A language for describing valid strings of tokens

➔ A method for distinguishing valid from invalid strings of tokens

Formal languages

Today: context-free languages (CNF)

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Formal languages are very important in CS

– Especially in programming languages

Regular languages

– The weakest formal languages widely used

– Many applications

We will today study **context-free languages (CNF)**

Beyond Regular Languages

Many languages are not regular

- You cannot describe them by REs

Strings of balanced parentheses are not regular:

$$\{(^i\ )^i \mid i \geq 0\}$$

requiring counting modulo a fixed integer

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Limitations of Regular Languages Express?

A fundamental limitation of REs;

- The corresponding FA cannot count because they have only a finite set of states.

- Finite automaton can't remember # of times it has visited a particular state

CFGs are more powerful than regular expressions and can express a richer set of structures.

- Because they allow recursions

Context-Free Grammars (G)

a set of rules that describe how to form sentences.

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

SN:

$$\begin{array}{rcl} \mathit{SheepNoise} & \rightarrow & \texttt{baa}\ \mathit{SheepNoise} \\ & | & \texttt{baa} \end{array}$$

. aa followed by more SheepNoise."

- **SheepNoise** is a syntactic variable representing the set of strings that can be derived from the grammar.
    - It is **a nonterminal symbol.**

The second rule

- "SheepNoise can also derive the string baa."

how to apply rules in SN to derive sentences in L(SN)

First identify

- the start symbol of SN
  - This represent the set of all strings in L(SN)
  - It must be one of non-terminals
  - ShipNoise

Start with ShipNoise

- **A nonterminal**

Choose a grammar α → β

- **Rewrite α with β**

Repeat this rewriting process

- **until** the prototype string contains **no more nonterminals**

Formal notations

A CFG consists of

– A set of terminals T

– A set of non-terminals N

– A start symbol S (a non-terminal)

– A set of productions

$$X \rightarrow Y_1 Y_2 \ldots Y_n$$

$$\text{where } X \in N \text{ and } Y_i \in T \cup N \cup \{\varepsilon\}$$

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Notational Conventions

– Non-terminals are written upper-case

- ShipNoise

– Terminals are written lower-case

- baa

– The start symbol is the left-hand side of the first production

- ShipNoise

Example: simple arithmetics

$$
\begin{aligned}
E \;&\rightarrow\; E * E \\
&\mid\; E + E \\
&\mid\; (E) \\
&\mid\; \mathrm{id}
\end{aligned}
$$

1	*Expr*	$\rightarrow$	(*Expr*)
2		|	*Expr* *Op* name
3		|	name
4	*Op*	$\rightarrow$	+
5		|	-
6		|	×
7		|	÷

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon

Notes

The idea of a CFG is a big step.

But:

- **Membership** in a language is **“yes” or “no”**
  - We will need a parse tree of the input
- Must handle **errors** gracefully
- Need an **implementation** of CFG’s (e.g., bison)

Form of the grammar is important

- Many grammars generate the same language
- Tools are sensitive to the grammar
- Note: Tools for regular languages (e.g., flex) are sensitive to the form of the regular expression, but this is rarely a problem in practice

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Derivations

Grammar

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

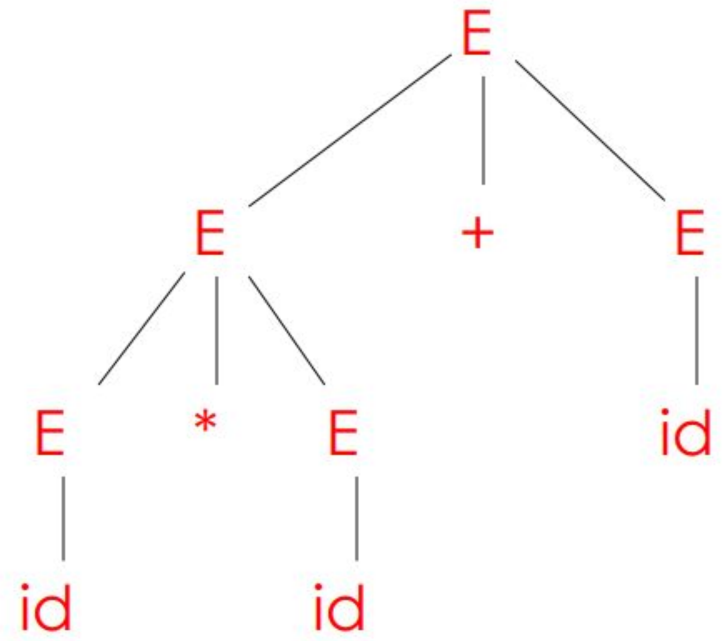
String

$id * id + id$

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Left most derivation

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Notes on Parse tree

Parse tree has

– Terminals at the leaves

– Non-terminals at the interior nodes

• An **in-order traversal of the leaves** is the original input

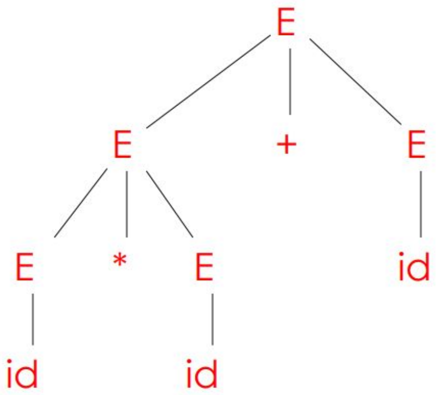
• The parse tree shows the association of operations, the input string does not

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Left-most and Right-most Derivations

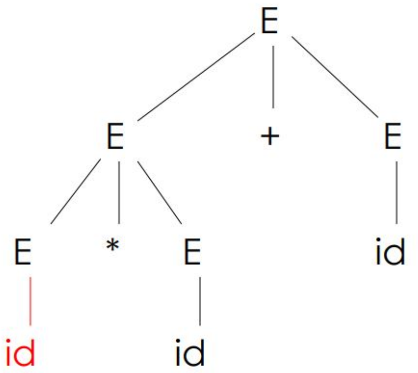
Leftmost derivation
a derivation that rewrites, at each step, the leftmost nonterminal

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Rightmost derivation
a derivation that rewrites, at each step, the rightmost nonterminal

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$



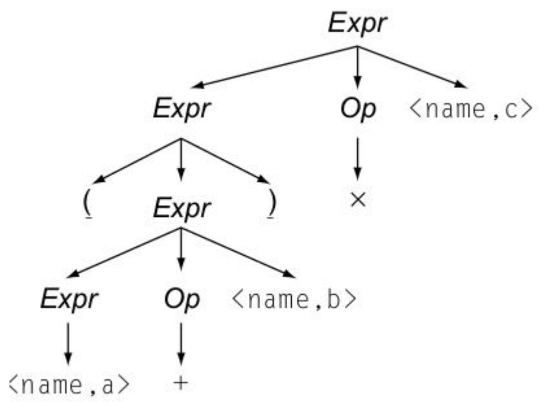
https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Rightmost derivation to generate the sentence “(a + b) × c”?

Rule	Sentential Form
	Expr
2	*Expr* *Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr* *Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

Rightmost Derivation of (a + b) × c



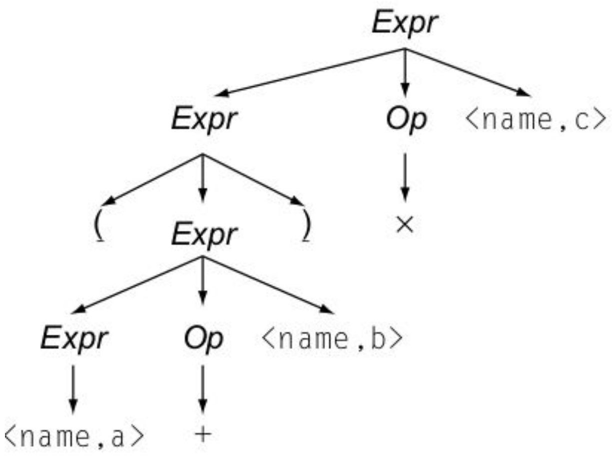
Corresponding Parse Tree

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Leftmost derivation to generate the sentence “(a + b) × c”?

Rule	Sentential Form
	Expr
2	*Expr* *Op* name
1	(*Expr*) *Op* name
2	(*Expr* *Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name

Leftmost Derivation of (a + b) x c



Corresponding Parse Tree

Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon

Notes on Derivations

- We are not just interested in whether **s ∈ L(G)**
  - We need a parse tree for **s**
- A derivation defines **a parse tree**
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation
  - right-most and left-most derivations have the same parse tree
  - The difference is the order in which branches are added

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Ambiguity

Grammar

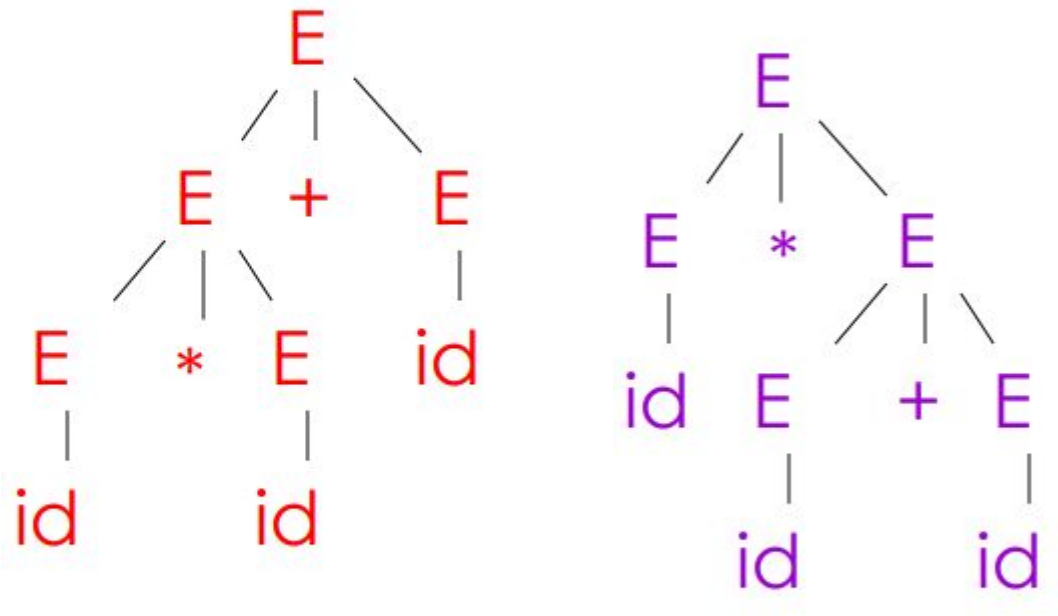
$E \rightarrow E+E \mid E*E \mid (E) \mid id$

String

$id * id + id$

Rightmost, leftmost

derivations?



A grammar is **ambiguous** if it has more than one parse tree for the same string

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Ambiguity

A grammar is ambiguous if it has more than one parse tree for some string

– Equivalently, there is more than one right-most or leftmost derivation for some string

• Ambiguity is BAD

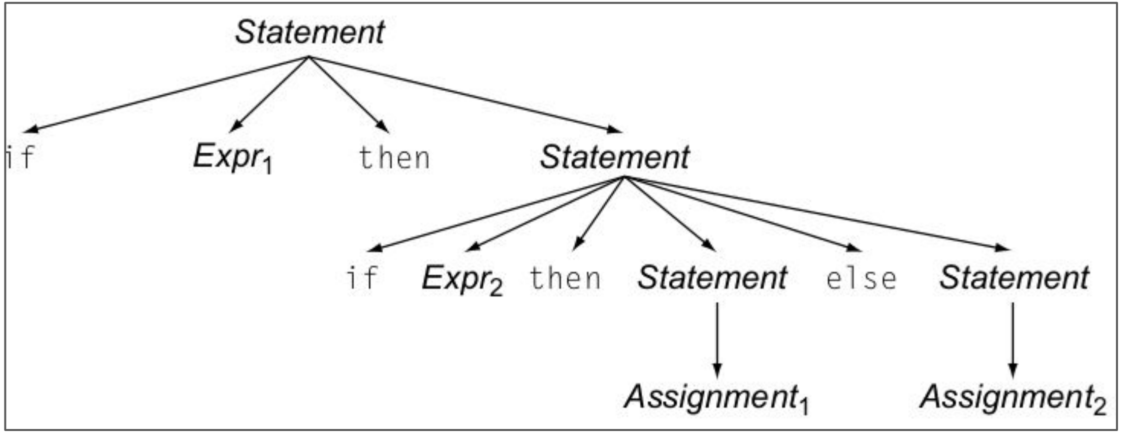
– Leaves meaning of some programs ill-defined

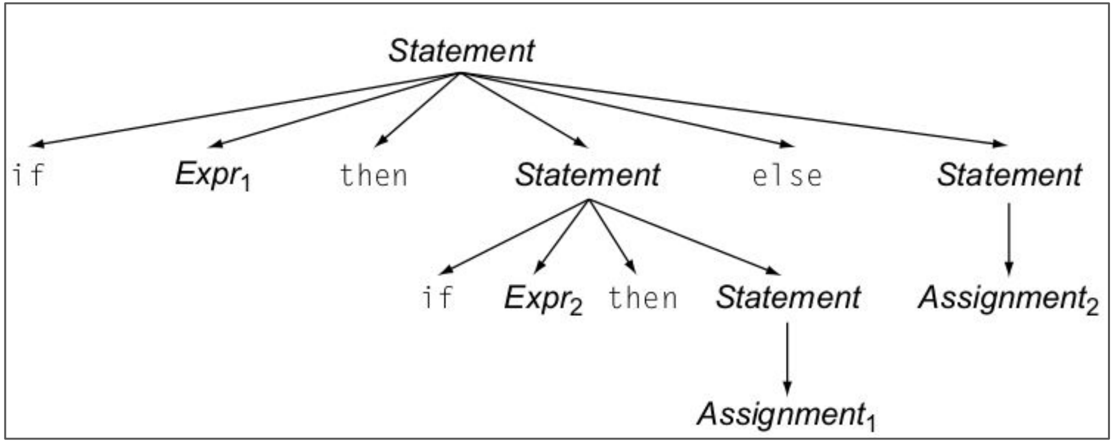
https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

1	*Statement*	$\rightarrow$	`if` *Expr* `then` *Statement* `else` *Statement*
2		|	`if` *Expr* `then` *Statement*
3		|	*Assignment*
4		|	... *other statements* ...

`if` $Expr_1$ `then` `if` $Expr_2$ `then` $Assignment_1$ `else` $Assignment_2$

The else part could belong to the outer if or to the inner if.





Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon

Handling ambiguity by rewriting grammar

Most direct method

- **rewrite grammar unambiguously (this is not always possible(inherently ambiguous context-free languages!)**

1	*Statement*	$\rightarrow$	`if` *Expr* `then` *Statement*
2		|	`if` *Expr* `then` *WithElse* `else` *Statement*
3		|	*Assignment*
4	*WithElse*	$\rightarrow$	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Assignment*

Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon

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

To enforce precedence of * over +

$E \rightarrow E' + E \mid E'$

$E' \rightarrow id * E' \mid id \mid (E)* E' \mid (E)$

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

No general techniques for handling ambiguity

• Impossible to convert automatically an ambiguous grammar to an unambiguous one

• Used with care, ambiguity can simplify the grammar

– Sometimes allows more natural definitions

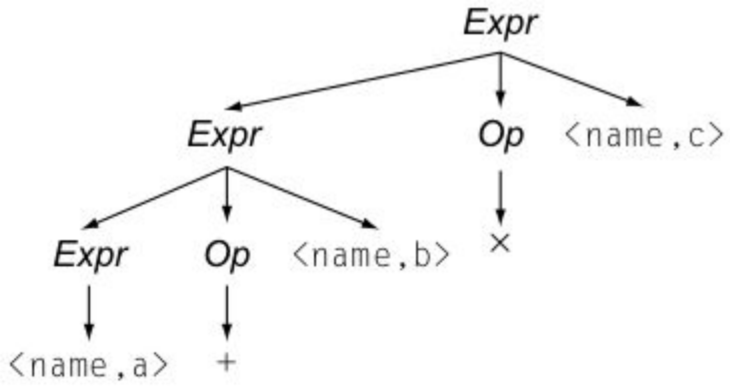
– **We need disambiguation mechanisms**

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Need a mechanism to add precedence

Rule	Sentential Form
	Expr
2	*Expr* *Op* name
6	*Expr* x name
2	*Expr* *Op* name x name
4	*Expr* + name x name
3	name + name x name

Derivation of a + b x c



Corresponding Parse Tree

With a simple postorder tree-walk.
first compute a + b
and then multiply by c to produce the result **(a + b) x c**.
How to add precedence?

Precedence and Associativity Declarations

Instead of rewriting the grammar

– Use the more natural (ambiguous) grammar

– Along with disambiguating declarations

• Most tools allow **precedence and associativity declarations** to disambiguate grammars

• Examples …

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Associativity

a property of operators that determines the order in which operations are evaluated when multiple operators of the same precedence appear in an expression.

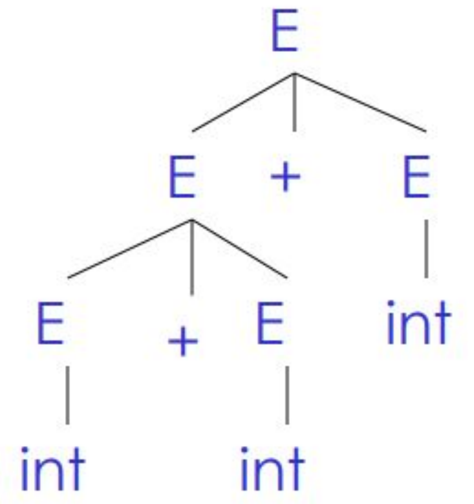
Left-associative `OP`

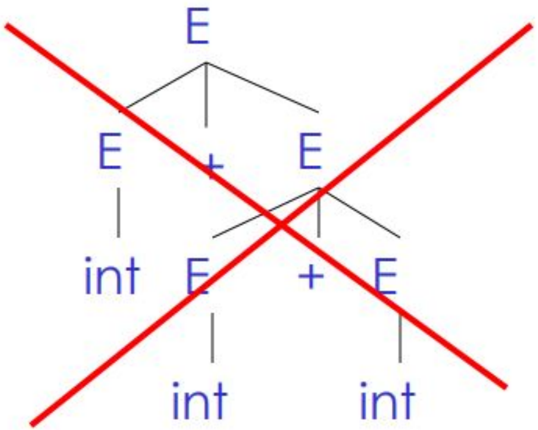
- the expression is evaluated from left to right.
  - **the operations are grouped from the left**
- `a OP b OP c`
  - `(a OP b) OP c`

Associativity Declarations

Consider the grammar **E → E + E | int**

- Ambiguous: two parse trees of **int + int + int**





Left associativity declaration: **%left +**

https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Precedence declarations

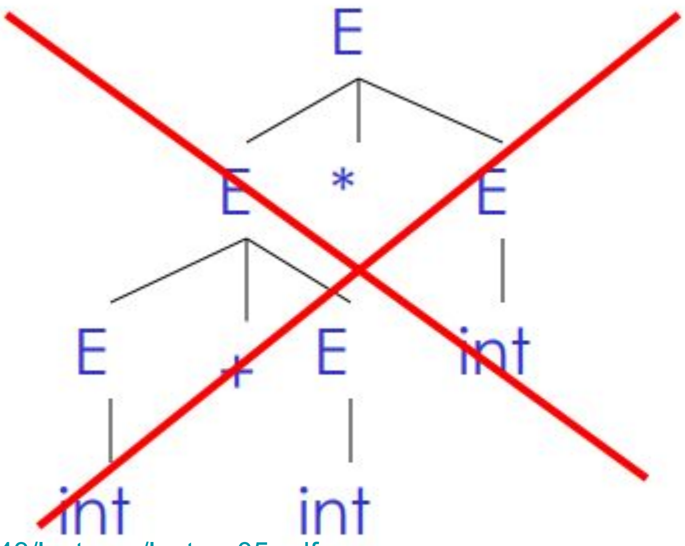
Grammar: $E \rightarrow E+E \mid E*E \mid int$

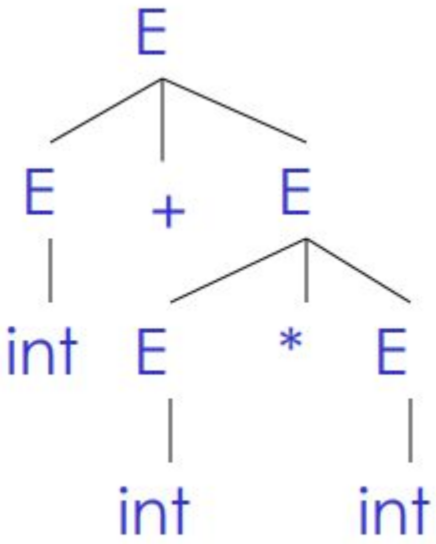
String: int + int * int

Precedence declarations:

%left +

%left *





https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

In Yacc

The precedences and associativities are attached to tokens in the declarations section.

This is done by a series of lines beginning with the yacc keywords **%left**, **%right**, or **%nonassoc**, followed by a list of tokens.

- All of the tokens on the same line are assumed to have the same precedence level and associativity;
- **the lines are listed in order of increasing precedence or binding strength.**

```
expr  : expr '=' expr
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | NAME
      ;
```

```
%right '='
%left '+' '-
%left '*' '/'
```

https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dh1/index.html

Changing precedence in yacc

```
%left '+' '-'
%left '*' '/'
%right not
%right '^'

%%

expr   : expr '+' expr
       | expr '-' expr
       | expr '*' expr
       | expr '/' expr
       | '-' expr %prec '*'
       | NAME
       ;
```

gives unary minus (negation) the same precedence as multiplication

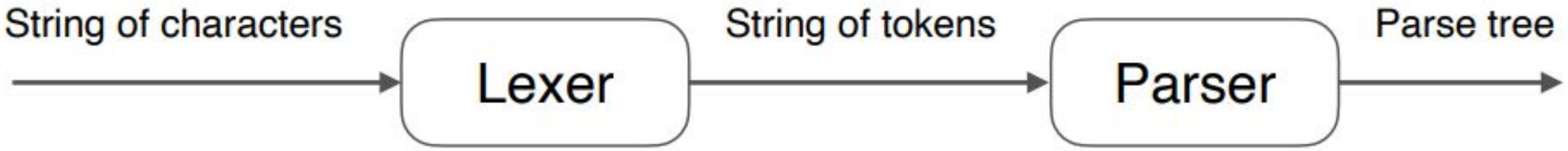
https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dh1/index.html

Recall: The functionality of parser

Input: sequence of tokens from lexer

Output: parse tree of the program

(Conceptually, but in practice parsers return an AST)



https://web.stanford.edu/class/cs143/lectures/lecture05.pdf

Input: the output of scanner: a stream of words annotated with their syntactic categories.

For a + b x c

- The input is: <name,a> + <name,b> x <name,c>

As output, the parser needs to produce either a derivation for the input program or an error message for an invalid program.

Error handling

Purpose of the compiler is

– To detect non-valid programs

– To translate the valid ones

• Many kinds of possible errors

Error kind	Example (C)	Detected by …
Lexical …….	$ …	Lexer
Syntax ……..	x *% …	Parser
Semantic ……	int x; y = x(3);	Type checker

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Syntax Error Handling

- Error handler should

– Report errors accurately and clearly

– Recover from an error quickly

– Not slow down compilation of valid code

- Good error handling is not easy to achieve

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Approaches to Syntax Error Recovery

From simple to complex

– Panic mode

– Error productions

– Automatic local or global correction

• Not all are supported by all parser generators

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Error Recovery: Panic Mode

Simplest, most popular method

• When an error is detected:

– Discard tokens until one with a clear role is found

– Continue from there

• Such tokens are called synchronizing tokens

– Typically the statement or expression terminators

example the erroneous expression

(1 + + 2) + 3

• Panic-mode recovery:

– Skip ahead to next integer and then continue

• Bison: use the special terminal **error** to describe how much input to skip

E → int | E + E | (E) | error int | (error)

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Syntax Error Recovery: Error Productions

Idea: specify known common mistakes in the grammar

• Essentially promotes common errors to alternative syntax

• Example:

– Write **5 x** instead of **5 * x**

– Add the production **E** → **… | E E**

• Disadvantage

– Complicates the grammar

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Error Recovery: Local and Global Correction

Idea: find a correct **“nearby”** program

– Try token insertions and deletions

– Exhaustive search

• Disadvantages:

– Hard to implement

– Slows down parsing of correct programs

– “Nearby” is not necessarily “the intended” program

– Not all tools support it

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Syntax Error Recovery: Past and Present

Past

– Slow recompilation cycle (even once a day)

– Find as many errors in one cycle as possible

– Researchers could not let go of the topic

Present

– Quick recompilation cycle

– Users tend to correct one error/cycle

– Complex error recovery is less compelling

– Panic-mode seems enough

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Abstract syntax tree

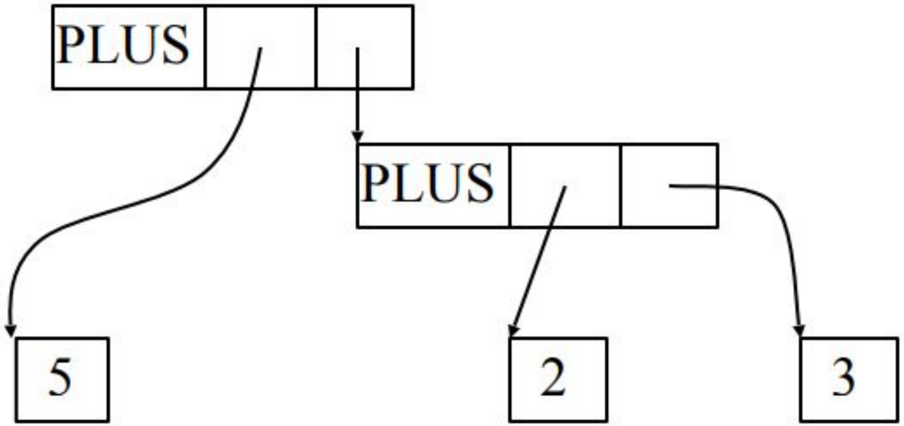
So far a parser traces the derivation of a sequence of tokens

• The rest of the compiler needs a structural representation of the program

• Abstract syntax trees

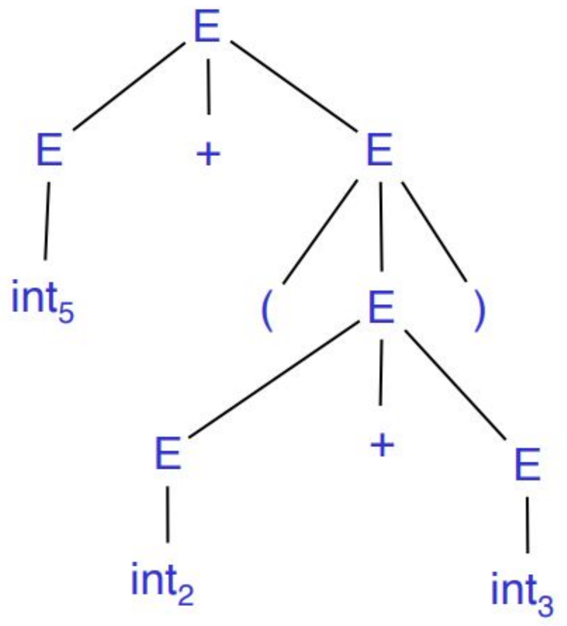
– Like parse trees but ignore some details

– Abbreviated as AST



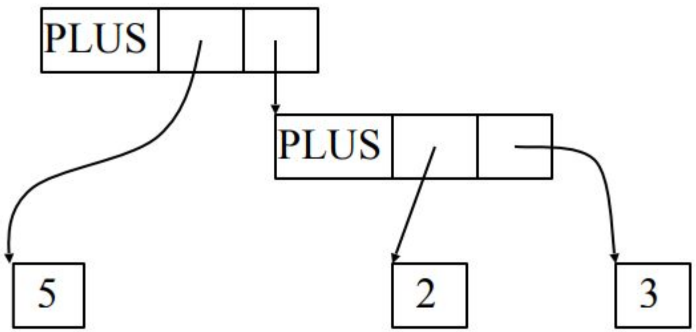
https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Grammar: E → int | (E) | E + E
String: 5 + (2 + 3)
After lex: int5 ‘+’ ‘(‘ int2 ‘+’ int3 ‘)’
During parsing we build a parse tree …



captures the nesting structure
• But too much info – Parentheses – Single-successor nodes

Abstract syntax tree



Also captures the nesting structure
• But abstracts from the concrete syntax
=> more compact and easier to use
• An important data structure in a compiler

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Parser Approaches for Your Project

Approach	Tools	Best For	Project Fit
Top-Down Recursive Descent	Handwritten (any language)	Educational understanding	All paths
LL(1) Predictive	ANTLR, JavaCC	Readable grammars	Java/Python/OCaml paths
LR(1) Bottom-Up	Bison, Yacc, PLY	Powerful, handles left-recursion	C/Flex Python-Ply, path
Parser Combinators	Haskell, OCaml, Python	Functional approach	OCaml path

How Parser Tools Differ

```
# PLY parser (bottom-up LALR)
def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression TIMES expression'''
    if p[2] == '+': p[0] = p[1] + p[3]
      else: p[0] = p[1] * p[3]
```

```
/* Bison grammar */
%token INT ID
%left '+'
%left '*'

%%
expr: expr '+' expr   { $$ = $1 + $3; }
    | expr '*' expr   { $$ = $1 * $3; }
    | INT             { $$ = $1; }
    | ID              { $$ = lookup($1); }
  %%
```

```
/* OCaml/Menhir (LR with FP)*/
%token <int> INT
%token PLUS TIMES
%left PLUS
%left TIMES

%start <Ast.expr> expr
%%
expr:
    | e1=expr PLUS e2=expr { Binop(Plus, e1, e2) }
    | e1=expr TIMES e2=expr { Binop(Times, e1, e2) }
   | INT { Int($1) }
```

AST Implementations

```c
#C Procedural

typedef enum { BINOP, NUMBER, VARIABLE } node_type;

struct ASTNode {
    node_type type;
    union {
        struct {
                char op;
                struct ASTNode *left, *right;
        } binop;
        int value;
        char* varname;
    };
  };
```

```python
#Python OOP

class ASTNode: pass

class BinOp(ASTNode):
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right

class Number(ASTNode):
    def __init__(self, value):
          self.value = value
```

```ocaml
#OCaml Functional

type expr =
  | Binop of string * expr * expr
  | Number of int
  | Variable of string
```

Error Handling

```
/* Bison - skip to next semicolon */
  stmt: expr ';' | error ';' { yyerror("Expected expression"); }
```

```
def p_statement(p):
    'statement : expression SEMI'
    try:
        p[0] = p[1]
    except ParseError as e:
        # Recover and continue
          self.parser.errok()
```

```
let parse_expression tokens =
  match parse_addition tokens with
  | Some expr, rest -> expr, rest
  | None, _ ->
      print_error "Expected expression";
        recover_expression token
```

Error Recovery

```
let rec parse_until_semicolon = function
  | SEMICOLON :: rest -> rest
  | _ :: tokens -> parse_until_semicolon tokens
  | [] -> []
```

```
expression:
    expression '+' expression
  | expression '*' expression
  | INT
  | error { yyerror("Invalid expression"); yyerrok; }
```

```
def p_error(p):
    if p:
        print(f"Syntax error at '{p.value}', line {p.lineno}")
        # Panic mode: discard until semicolon
        parser.errok()
        parser.restart()
```

Next lecture

- Semantic actions to build AST

  Each production may have an action

  – Written as: **X → Y1 … Yn { action }**

  – That can refer to or compute symbol attributes

  E → int { E.val = int.val }

  | E1 + E2 { E.val = E1.val + E2.val }

  | (E1) { E.val = E1.val }

- **Top-down parsers** begin with the root and grow the tree toward the leaves.
- **Bottom-up parsers** begin with the leaves and grow the tree toward the root.

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf