Parser Overall Picture

First overall parser

top down parser in more details

The Formal Language Hierarchy (Chomsky Hierarchy)

Type 3: Regular Languages → Finite Automata, Regular Expressions

- Lexical Analysis (Flex/Lex)

Type 2: Context-Free Languages → Pushdown Automata, CFG

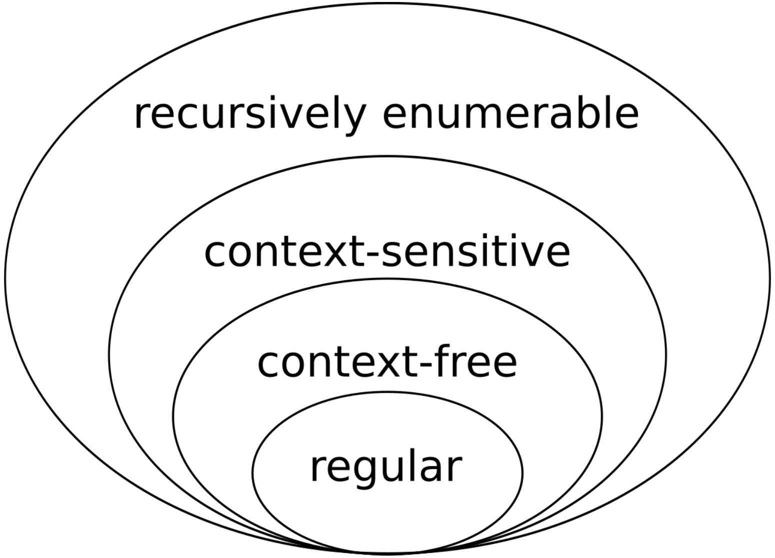
- Syntax Analysis (Bison/Yacc, Recursive Descent)

Type 1: Context-Sensitive Languages → Linear Bounded Automata

- Some semantic checks

Type 0: Recursively Enumerable Languages → Turing Machines

- General computation



https://en.wikipedia.org/wiki/Chomsky_hierarchy

Context free grammars (CFG)

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

Generating strings:

identify the start symbol S (a non-terminal)

- This represent the set of all strings in L(S)

- Choose a grammar $\alpha \rightarrow \beta$
- rewrite α with β

repeat this until no more nonterminals

Why Context-Free Grammars (CFG) for Parsing?

- Regular expressions can't handle nested structures
- CFGs naturally express programming language syntax
- Efficient parsing algorithms exist for CFGs

CFG example

$S \to aS$

$S \to Sb$

$S \to cS$

$S \to \varepsilon$

Regular expression equal to this?

➔ $(a + b + c)^*$

Derivation of string **acb**

S (use $S \to aS$)	
$\to aS$	(use $S \to Sb$)
$\to aSb$	(use $S \to cS$)
$\to acSb$	(use $S \to \varepsilon$)
$\to acb$	Final String

Examples CFG for L =

- $\{a^n b^n : n > 0\}$?
  - Cfg exists
- $\{a^n b^m c^n : n > 0, m > 0\}$?
  - Cfg exists
- $\{a^n b^n c^n : n > 0\}$?
  - No cfg exists (violates the pumping lemma)

Why Context-Free Grammars (CFG) for Parsing?

Regular expressions can't handle nested structures

CFGs naturally express programming language syntax

Efficient parsing algorithms exist for CFGs

Grammar Normal Forms-Chomsky Normal Form (CNF)

Every production is either:

A → BC (two non-terminals)

A → a (single terminal)

S → ε (only start symbol can derive empty)

Original:

E → E + E | E * E | (E) | id

CNF:

E → IE' | LY | LX | id
Y →XE'
X →ER
E' →PE | ME | PC | MC
C →EE'
L →(R→)   P → +   M→ *   I →id

Simplified Grammar for Parsing

- **Simplify parsing algorithms**
- **Enable theoretical proofs**
- **Standardize grammar analysis**

Real-World Grammar Examples

```
# Python Simple statement grammar
stmt: simple_stmt | compound_stmt
simple_stmt: expr_stmt | return_stmt
compound_stmt: if_stmt | while_stmt | def_stmt
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

```
# C Simple statement grammar
statement: expression ';'
          | IF '(' expression ')' statement
          | IF '(' expression ')' statement ELSE statement
          | WHILE '(' expression ')' statement
          | '{' statement* '}';
```

```
json: object | array
object: '{' pairs '}'
pairs: pair | pair ',' pairs
pair: STRING ':' value
array: '[' elements ']'
elements: value | value ',' elements
value: STRING | NUMBER | object | array | 'true' | 'false' | 'null'
```

Parser Landscape Overview

Top-Down Parsers (LL Family)

↗ Recursive Descent (hand-written)

↗ LL(1), LL(k) (predictive)

↗ Tools: ANTLR, JavaCC

Top-Down: Start from root, build parse tree downward

- Leftmost derivation

LL: Often hand-written, educational

Bottom-Up Parsers (LR Family)

↗ LR(0), SLR(1), LR(1), LALR(1)

↗ Tools: Bison/Yacc, CUP

Bottom-Up: Start from leaves, build parse tree upward

- Rightmost derivation in reverse

LR: Most parser generators (more powerful)

LL vs LR - Concrete Example

Goal: Parse "2 + 3 * 4"

LL Process:

- E → E + T (predict)
- → T + T (expand E → T)
- → F + T (expand T → F)
- → 2 + T (match '2')
- → 2 + T * F (expand T → T * F)
- → 2 + F * F (expand T → F)
- → 2 + 3 * F (match '3')
- → 2 + 3 * 4 (match '4')

Goal: Parse "2 + 3 * 4"

LR Process (shift/reduce):

- 2 + 3 * 4 (shift '2')
- F + 3 * 4 (reduce: 2 → F)
- T + 3 * 4 (reduce: F → T)
- E + 3 * 4 (reduce: T → E)
- E + 3 * 4 (shift '+')
- E + 3 * 4 (shift '3')
- E + F * 4 (reduce: 3 → F)
- E + T * 4 (reduce: F → T)
- E + T * 4 (shift '*')
- E + T * 4 (shift '4')
- E + T * F (reduce: 4 → F)
- E + T (reduce: T * F → T)
- E (reduce: E + T → E)

Works like a Pattern Recognition

Goal: Parse "2 + 3 * 4"

LL Process:

- E → E + T (predict)
- → T + T (expand E → T)
- → F + T (expand T → F)
- → 2 + T (match '2')
- → 2 + T * F (expand T → T * F)
- → 2 + F * F (expand T → F)
- → 2 + 3 * F (match '3')
- → 2 + 3 * 4 (match '4')

Goal: Parse "2 + 3 * 4"

LR Process (shift/reduce):

- 2 + 3 * 4 (shift '2')
- F + 3 * 4 (reduce: 2 → F)
- T + 3 * 4 (reduce: F → T)
- E + 3 * 4 (reduce: T → E)
- E + 3 * 4 (shift '+')
- E + 3 * 4 (shift '3')
- E + F * 4 (reduce: 3 → F)
- E + T * 4 (reduce: F → T)
- E + T * 4 (shift '*')
- E + T * 4 (shift '4')
- E + T * F (reduce: 4 → F)
- E + T (reduce: T * F → T)
- E (reduce: E + T → E)

Works like a Pattern Recognition

Both approaches typically build ASTs bottom-up:

LL: Build subtrees during recursive returns

LR: Build nodes during reduce actions

Example AST construction in both

```
// LL Recursive Descent
Expr* parse_expression() {
    Expr* left = parse_term();
    if (current_token == PLUS) {
        get_token();
        Expr* right = parse_expression();
        return new AddExpr(left, right);  // Build on return
    }
    return left;
}

// LR Bison
expression: term PLUS term { $$ = new AddExpr($1, $3); }
          | term           { $$ = $1; }
          ;
```

LL(1) vs LR(1) Comparison

Predictive Top-Down

Top-down, Leftmost derivation

1 token lookahead

Implemented via Recursive procedures

- Must eliminate left recursion

Shift-Reduce Bottom-Up

1 token lookahead

Implemented with State machines + stack

How to simply implement parser

Recursive descent

```python
def parse_expression(self):
    left = self.parse_term()
    while self.current_token in ('+', '-'):
        op = self.current_token
        self.advance()
        right = self.parse_term()
        left = BinaryOp(left, op, right)
      return left
```

Table driven LL(1)

```c
// Parse table driving recursive calls
void parse() {
    stack.push(START_SYMBOL);
    while (!stack.empty()) {
        symbol = stack.top();
        if (is_terminal(symbol)) {
            match(symbol);
            stack.pop();
        } else {
            rule = parsing_table[symbol][lookahead];
            stack.pop();
            push_rule_on_stack(rule);  // in reverse
order
        }
    }
  }
```

Introduction to Top-Down Parsing

Characteristics:

- Parse tree constructed from the top
- From left to right
- Terminals seen in order of appearance

Today's Focus:

- Syntax-Directed Translation
- Recursive Descent Parsing
- LL(1) predictive top-down parsing
- Parser tools (Bison examples

Mostly copied from https://web.stanford.edu/class/cs143/lectures/lecture06.pdf
Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3
https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Parse tree

a graphical representation of a derivation.

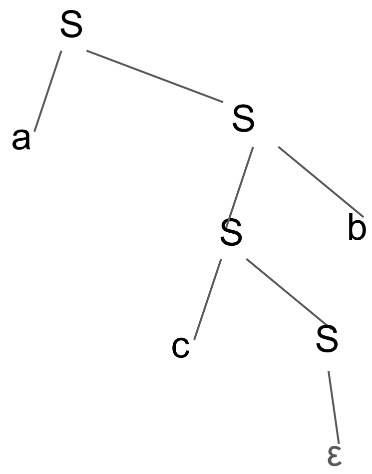
S (use S → aS)

→ aS (use S → Sb)

→ aSb (use S → cS)

→ acSb (use S →ε)

→ acb Final String



exercises

CFG for all strings over {a,b} which contain the substring “**baba”**

Draw parse tree for **ababab**

CFG for all strings over {a,b} which starts with **ba** ends with **ba**

Draw parse tree for **baaabba**

Ambiguity

A CFG is ambiguous:

- If for some string s,
    - the grammar has > 1 different parse tree

Arithmetics

$E \to E + E \mid E - E \mid E * E \mid E / E \mid id$

Dangling else

$E \to$ if E then E | if E then E else E | OTHER

$E \to E + E \mid E - E \mid E * E \mid E / E \mid id$

Rewrite grammar (right associative):

- $E \to id \mid id + E \mid id - E \mid id * E$

Or declare **precedence and associativity(in yacc):**

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

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

Abstract syntax tree

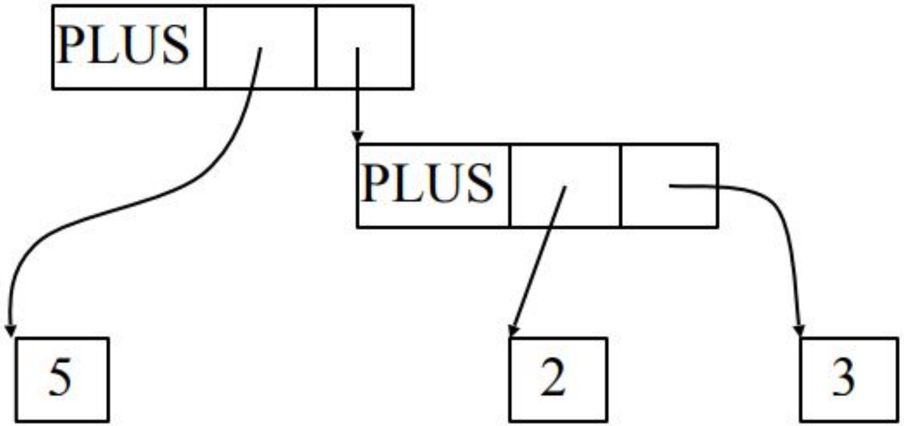
So far a parser traces the derivation of a sequence of tokens

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

• Abstract syntax trees

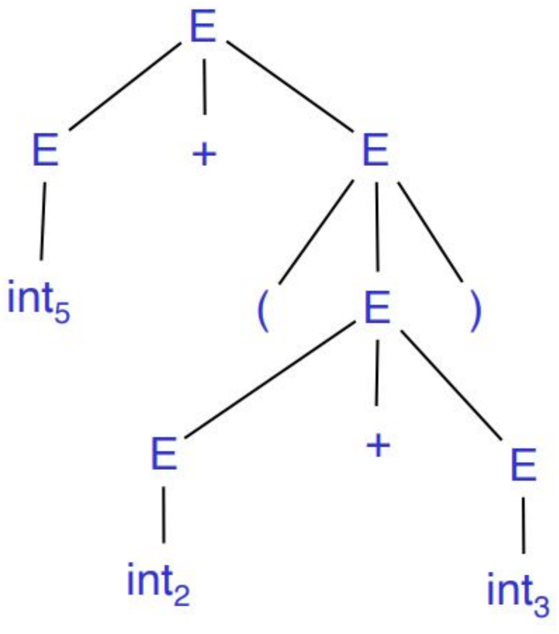
– Like parse trees but ignore some details

– Abbreviated as AST



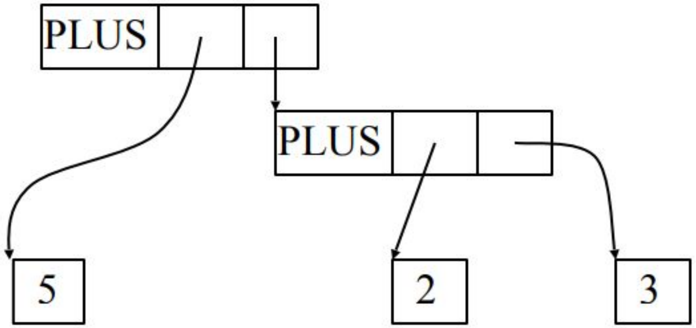
https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

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

Semantic actions

This is what we'll use to construct ASTs

Each grammar symbol may have **attributes**

- For terminal symbols (lexical tokens) attributes can be calculated by the lexer

Each production may have an **action**

- **X → Y1 … Yn { action }**
  - **action can refer to/compute symbol attributes**

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

Semantic Actions: Adding Behavior to Grammar Rules

What are semantic actions?

- Code that runs when a grammar rule is applied
- Used to build ASTs, compute values, check types, generate code

Bison- Simple Calculator Example:

E → E1 + E2 { $$ = $1 + $3; } // Compute value

E → number { $$ = $1; } // Pass number value up

E → (E1) { $$ = $1; } // Pass through parentheses

- $$ = result for the left-hand side (what we're building)
- $1, $2, $3 = values from the 1st, 2nd, 3rd symbols on right-hand side

expr ::= expr + term

| term

```
// parse expr ::= ...
void expr() {
  expr();
  if (current token is PLUS) {
    getNextToken();
    term();
  }
}
```

expr ::= term { + term }*

```
// parse
void expr() {
  term();
  while (next symbol is PLUS) {
    getNextToken();
    term();
  }
}
```

https://courses.cs.washington.edu/courses/cse401/22au/lectures/F-ll-parsing.pdf

```
// parse
// factor ::= int | id | ( expr )
void factor() {
   switch(nextToken) {
   case INT:
       process int constant;
       getNextToken();
       break;
       …
   case ID:
       process identifier;
       getNextToken();
       break;
   case LPAREN:
       getNextToken();
       expr();
       getNextToken();
   }
}
```

➔ Next: How to process terminals, non-terminals

➔ How to eliminate recursion and left factoring problem

E.g. left factoring

ifStmt ::= if (expr) stmt

| if (expr) stmt else stmt

https://courses.cs.washington.edu/courses/cse401/22au/lectures/F-ll-parsing.pdf

Annotating grammar with actions

Consider the grammar

$E \rightarrow int \mid E + E \mid (E)$

For each symbol X define an attribute X.val

- For terminals,
  - val is the associated lexeme
- For non-terminals,
  - val is the expression's value (and is computed from values of subexpressions)

We annotate the grammar with actions:

E → int { E.val = int.val }

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

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

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

String: 5 + (2 + 3)

Tokens: int5 ‘+’ ‘(‘ int2 ‘+’ int3 ‘)’

Productions	Equations
E → E1 + E2	E.val = E1.val + E2.val
E1 → int5	E1.val = int5.val = 5
E2 → (E3)	E2.val = E3.val
E3 → E4 + E5	E3.val = E4.val + E5.val
E4 → int2	E4.val = int2.val = 2
E5 → int3	E5.val = int3.val = 3

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

Building a Calculator with Semantic Actions

```
// Each rule now computes a value
expr : expr '+' term   { $$ = $1 + $3; }    // Add
     | expr '-' term   { $$ = $1 - $3; }    // Subtract
     | term            { $$ = $1; }         // Single term
     ;

term : term '*' factor { $$ = $1 * $3; }    // Multiply
     | term '/' factor { $$ = $1 / $3; }    // Divide
     | factor          { $$ = $1; }         // Single factor
     ;

factor : NUMBER        { $$ = $1; }         // Use number value
       | '(' expr ')'  { $$ = $2; }         // Parentheses
       ;
```

See an Example with bison: infix notation calculator [Infix Calc (Bison 3.8.1)]

- **yyparse()** returns a value of `0` if the input it parses is valid
- **yyparse()** calls a routine **yylex()** everytime it wants to obtain a **token** from the input.

See https://github.com/adaskin/compiler-course/tree/main/lecture_notes/5-bison-example

Notes on Semantic actions

- Semantic actions specify a system of equations
- Declarative Style

– Order of resolution is not specified

– The parser figures it out

- Imperative Style

– The order of evaluation is fixed

– Important if the actions manipulate global state

- We'll explore actions as pure equations

– Style 1

– But note bison has a fixed order of evaluation for actions

- Example:

E3.val = E4.val + E5.val

– Must compute E4.val and E5.val before E3.val

– We say that E3.val depends on E4.val and E5.val

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

Dependency graph

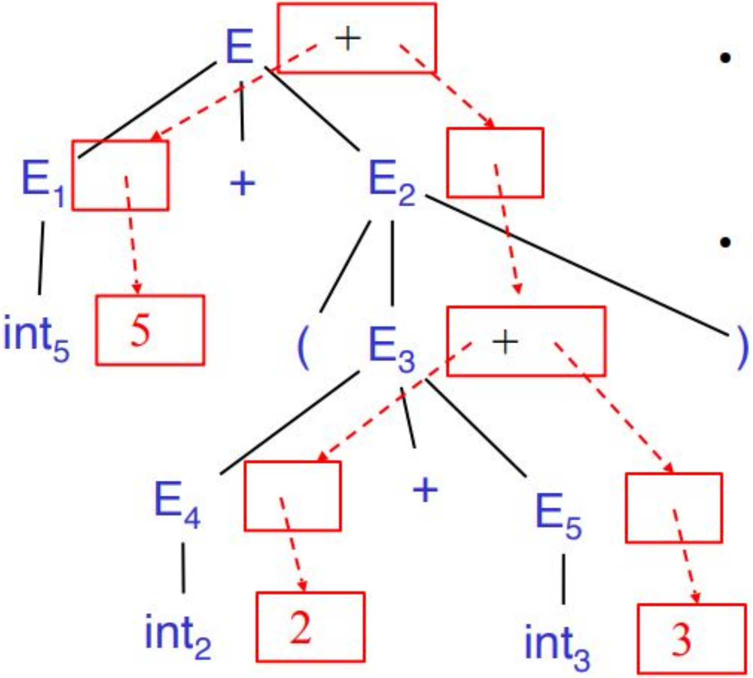
Each node labeled E has one slot for the val attribute

An attribute must be computed after all its successors in the dependency graph have been computed

– In this example attributes can be computed bottom-up

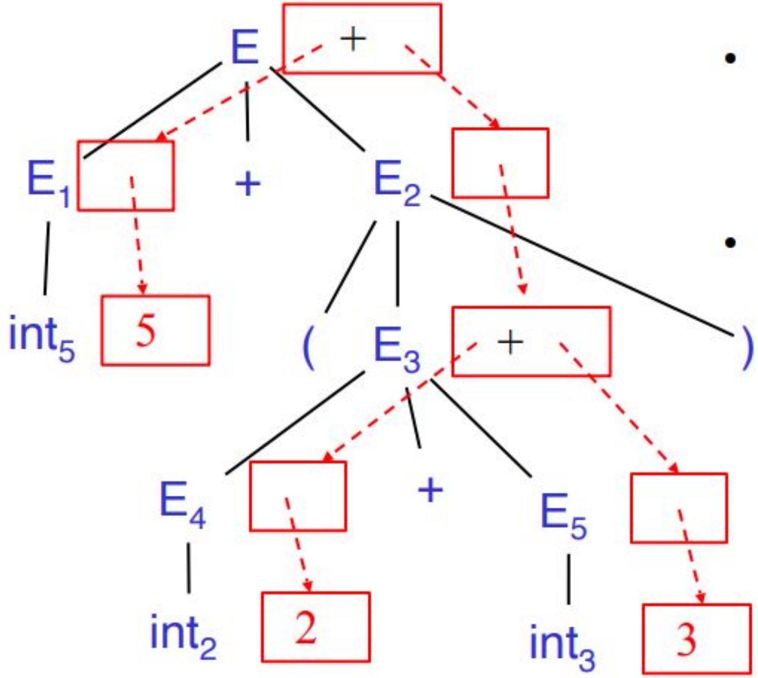
Such an order exists when there are no cycles

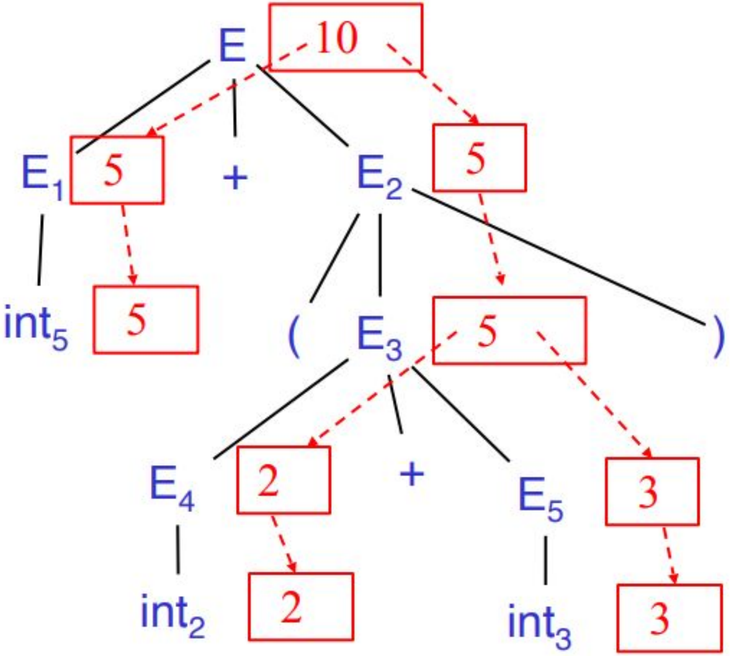
– Cyclically defined attributes are not legal



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

Dependency graph





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

Two Types of Attribute Flow

Synthesized Attributes (Bottom-Up)

Flow in parse tree: Children → Parent

Similar to: Function return values

Use for: Expression values, building AST nodes

```
// Children compute, parent uses results
  expr : expr '+' term { $$ = $1 + $3; }

  // $1 and $3 from children
```

90% of the time, you'll use synthesized attributes!

Inherited Attributes (Top-Down)

Flow in parse tree:

Parent → Children, or Sibling → Sibling

Similar to: Function parameters

Use for: Scope information, type context

```
// Parent passes info down to children
decl : TYPE IDENTIFIER {
    $$ = new_declaration($1, $2);
    // Pass type info down for variable usage
  }
```

Example: A line calculator

Each line contains an expression

E → int | E + E

Each line is terminated with the = sign

L → E = | + E =

• In second form the value of previous line is used

as starting value

• A program is a sequence of lines

P → ε | P L

Each E has a synthesized attribute val

– Calculated as before

Each L has an attribute val

L → E = { L.val = E.val }

| + E = { L.val = E.val + L.prev }

- We need the value of the previous line
- We use an inherited attribute **L.prev**

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

Real Example: Line Calculator with Memory

Problem: Build a calculator that remembers previous result

```
Input:    Output:
5 =       5
+ 3 =     8
+ 2 =     10
```

Solution using Inherited Attributes

```
// Grammar with both synthesized and inherited attributes
program : /* empty */       { $$ = 0; }             // Start with 0
        | program line      { $$ = $2; }            // Last line's value
        ;

line : expr '='             { $$ = $1; }            // New expression
     | '+' expr '='         { $$ = previous + $2; } // Continue from previous
     ;

  // The magic: 'previous' is inherited from the program
```

- `previous` flows down from program to line (inherited)
- Line result flows up to program (synthesized)

Example: A line calculator

Each line contains an expression

$E \rightarrow$ int | E + E

Each line is terminated with the = sign

$L \rightarrow$ E = | + E =

• In second form the value of previous line is used

as starting value

• A program is a sequence of lines

$P \rightarrow \varepsilon$ | P L

Each P has a synthesized attribute val

– The value of its last line

$P \rightarrow \varepsilon$ { P.val = 0 }

| P1 L { L.prev = P1.val;

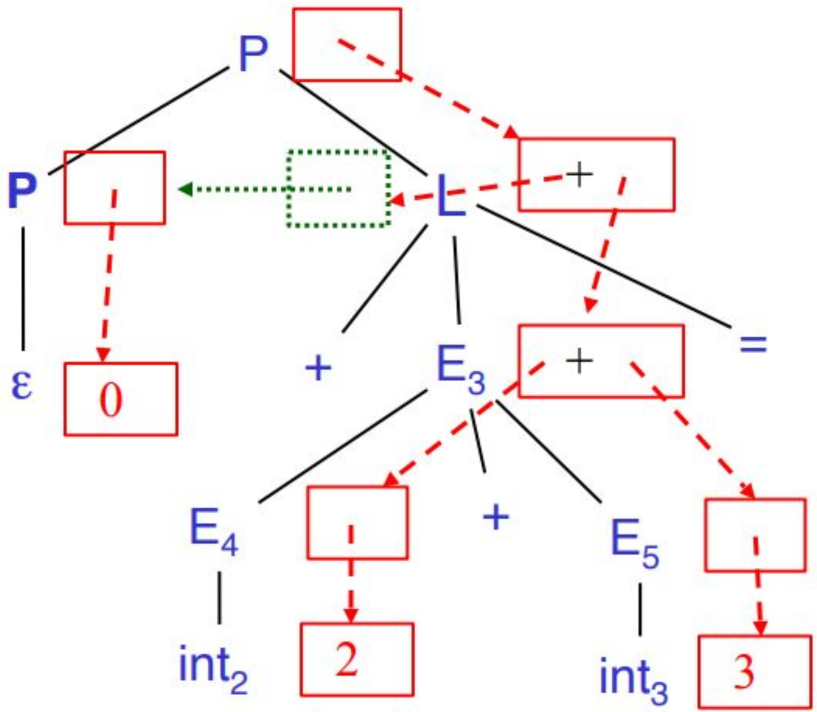
P.val = L.val }

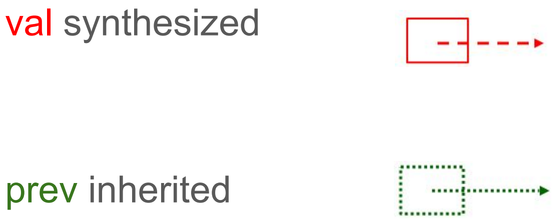
– Each L has an inherited attribute prev

– L.prev is inherited from sibling P1.val

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

Example of Inherited Attributes

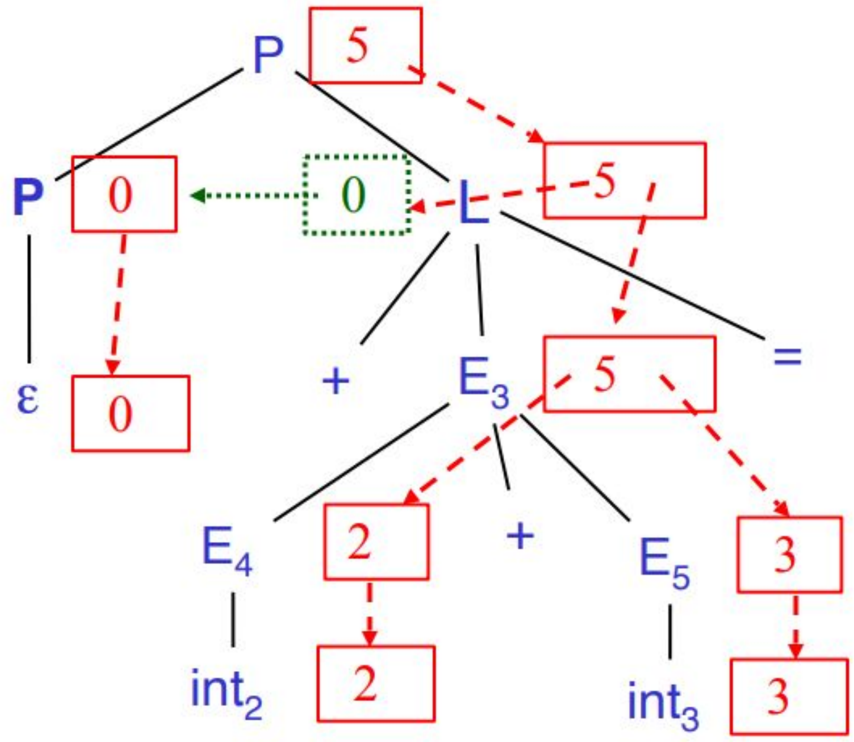


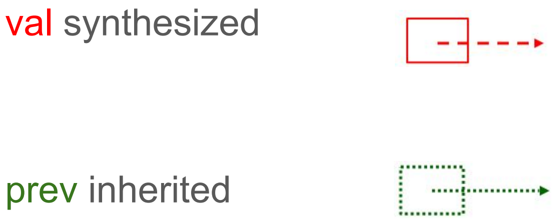


All can be computed in bottom-up order

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

Example of Inherited Attributes





All can be computed in bottom-up order

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

Semantic actions (syntax-directed translation)

Semantic actions can be used to build ASTs

• And many other things as well

– Also used for type checking, code generation,computation, …

• Process is called **syntax-directed translation (SDT)**

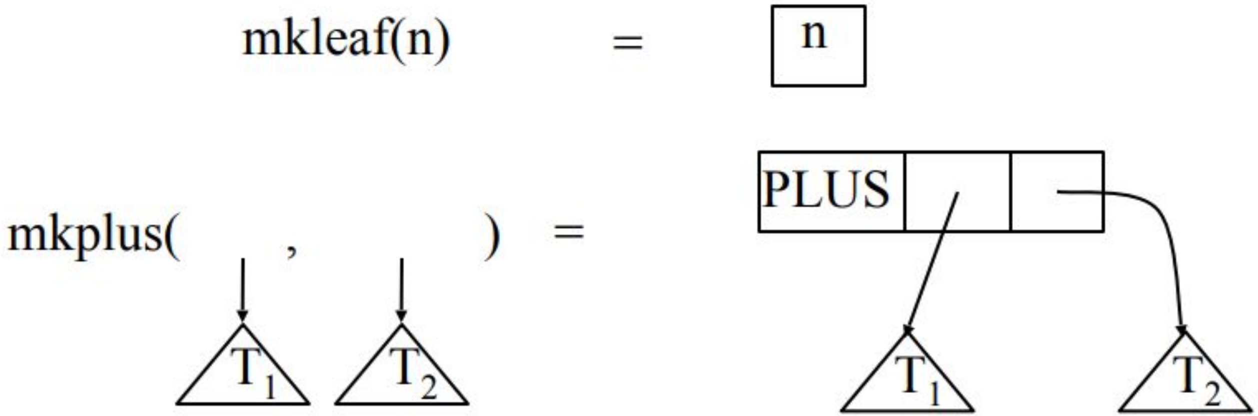
– Substantial generalization over CFGs

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

SDT: constructing an AST

We first define the AST data type

• Consider an abstract tree type with two constructors:



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

We define a synthesized attribute **ast**

– Values of ast values are ASTs

– We assume that int.lexval is the value of the integer lexeme

– Computed using semantic actions

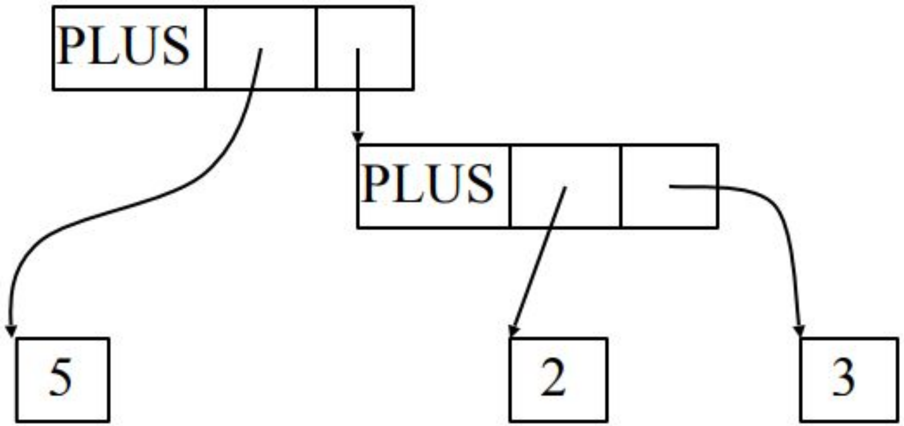
```
E → int E.ast = mkleaf(int.lexval)
  | E1 + E2 E.ast = mkplus(E1.ast, E2.ast)
  | ( E1 ) E.ast = E1.ast
```

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

Consider the string int5 ‘+’ ‘(‘ int2 ‘+’ int3 ‘)’

- A bottom-up evaluation of the ast attribute:

```
E.ast = mkplus(mkleaf(5), mkplus(mkleaf(2), mkleaf(3))
```



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

A simple intro to Bison examples(we will see more after LR(1))

YACC (Yet Another Compiler Compiler) was a widely used parser generator in the Unix environment, recently supplanted by the GNU Bison parser which is generally compatible.

Bison is designed to automatically invoke Flex as needed,

so it is easy to combine the two into a complete program

Bison is optimized for LR(1)

The file structure similar to lex:

```
%{
 /*C preamble code, #include etc*/

%}
/* declarations*/
%%
/*grammar rules*/
%%
/*C epilogue code*/
```

```
%{
#include <stdio.h>
%}
%token TOKEN_INT
%token TOKEN_PLUS
%token TOKEN_MINUS
%token TOKEN_MUL
%token TOKEN_DIV
%token TOKEN_LPAREN
%token TOKEN_RPAREN
%token TOKEN_SEMI
%token TOKEN_ERROR
```

```
%%
program : expr TOKEN_SEMI;
expr : expr TOKEN_PLUS term
| expr TOKEN_MINUS term
| term
;
term : term TOKEN_MUL factor
| term TOKEN_DIV factor
| factor
;
factor: TOKEN_MINUS factor
| TOKEN_LPAREN expr TOKEN_RPAREN
| TOKEN_INT
;
%%
int yywrap() { return 0; }
```

https://www3.nd.edu/~dthain/compilerbook/chapter5.pdf

```c
#include <stdio.h>
extern int yyparse();
int main()
{
    if (yyparse() == 0)
    {
        printf("Parse successful!\n");
    }
    else
    {
        printf("Parse failed.\n");
    }
}
```

https://www3.nd.edu/~dthain/compilerbook/chapter5.pdf

The resulting code creates a single function **yyparse()** that returns an integer:

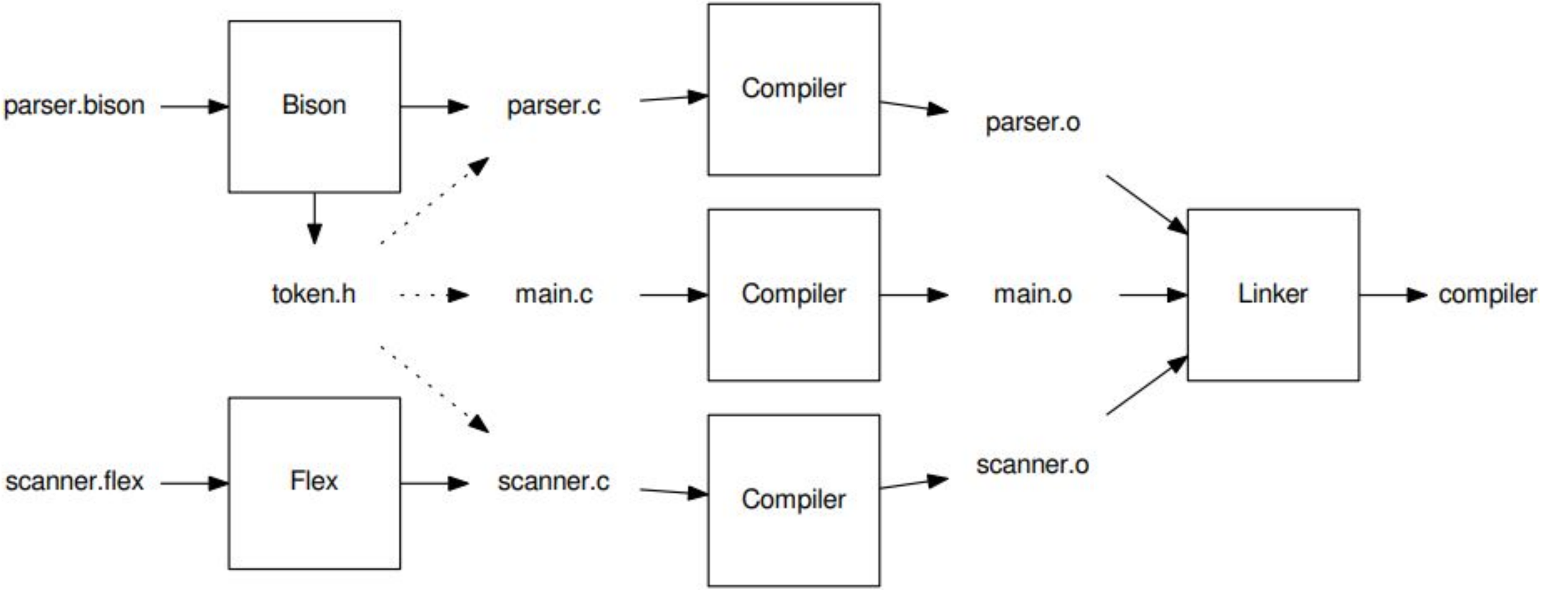
zero indicates a successful parse, one indicates a parse error, and two indicates an internal problem such as memory exhaustion

See examples: https://www.gnu.org/software/bison/manual/html_node/Examples.html

Python ply example:

https://www.dabeaz.com/ply/example.html

Build Procedure for Bison and Flex Together



https://www3.nd.edu/~dthain/compilerbook/chapter5.pdf

Building AST by using parser tools

```c
// Define what our AST nodes look like
typedef struct ASTNode {
    enum { NUMBER, BINOP } type;
    union {
        int value;                      // For numbers
        struct {                        // For binary operations
            struct ASTNode* left;
            char operator;
            struct ASTNode* right;
        } binop;
    };
  } ASTNode;

  // function prototypes

  ASTNode* create_number(int value);

  ASTNode* create_binop(ASTNode* left, char op, ASTNode* right);

  ASTNode* create_variable(char* name);

  void free_ast(ASTNode* node);

  void print_ast(ASTNode* node, int indent);
```

```c
#include "ast.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

ASTNode* create_number(int value) {
    ASTNode* node = malloc(sizeof(ASTNode));
    node->type = AST_NUMBER;
    node->number_value = value;
    return node;
}

ASTNode* create_binop(ASTNode* left, char op, ASTNode* right) {
    ASTNode* node = malloc(sizeof(ASTNode));
    node->type = AST_BINOP;
    node->binop.left = left;
    node->binop.operator = op;
    node->binop.right = right;
    return node;
}

ASTNode* create_variable(char* name) {
    ASTNode* node = malloc(sizeof(ASTNode));
    node->type = AST_VARIABLE;
    node->variable_name = strdup(name);  // Copy the string
    return node;
}
```

```c
void print_ast(ASTNode* node, int indent) {
    if (node == NULL) return;

    for (int i = 0; i < indent; i++) printf("  ");

    switch (node->type) {
        case AST_NUMBER:
            printf("Number: %d\n", node->number_value);
            break;
        case AST_BINOP:
            printf("BinOp: %c\n", node->binop.operator);
            print_ast(node->binop.left, indent + 1);
            print_ast(node->binop.right, indent + 1);
            break;
        case AST_VARIABLE:
            printf("Variable: %s\n", node->variable_name);
            break;
    }
  }
```

Usage in Bison

```
%{
#include "ast.h"
%}

%union {
    int int_value;
    char* string_value;
    ASTNode* node_value;
}

%token <int_value> NUMBER
%token <string_value> IDENTIFIER
%type <node_value> expr term factor

%%

expr : expr '+' term { $$ = create_binop($1, '+', $3); }
     | expr '-' term { $$ = create_binop($1, '-', $3); }
     | term          { $$ = $1; }
     ;

term : term '*' factor { $$ = create_binop($1, '*', $3); }
     | term '/' factor { $$ = create_binop($1, '/', $3); }
     | factor          { $$ = $1; }
     ;

factor : NUMBER        { $$ = create_number($1); }
       | IDENTIFIER    { $$ = create_variable($1); }
       | '(' expr ')' { $$ = $2; }
       ;
```

```python
class ASTNode:
    """Base class for all AST nodes"""
    def __repr__(self):
        return self.__str__()

class Number(ASTNode):
    def __init__(self, value):
        self.type = "NUMBER"
        self.value = value

    def __str__(self):
        return f"Number({self.value})"
class BinOp(ASTNode):
    def __init__(self, left, op, right):
        self.type = "BINOP"
        self.left = left
        self.op = op
        self.right = right

    def __str__(self):
        return f"BinOp({self.op}, {self.left}, {self.right})"
```

```python
class Variable(ASTNode):
    def __init__(self, name):
        self.type = "VARIABLE"
        self.name = name

    def __str__(self):
        return f"Variable({self.name})"


# Constructor functions (for consistency with other
implementations)
def create_number(value):
    return Number(value)

def create_binop(left, op, right):
    return BinOp(left, op, right)

def create_variable(name):
    return Variable(name)
```

```python
# Pretty printer
def print_ast(node, indent=0):
    if node is None:
        return

    spaces = "  " * indent

    if node.type == "NUMBER":
        print(f"{spaces}Number: {node.value}")
    elif node.type == "BINOP":
        print(f"{spaces}BinOp: {node.op}")
        print_ast(node.left, indent + 1)
        print_ast(node.right, indent + 1)
    elif node.type == "VARIABLE":
         print(f"{spaces}Variable: {node.name}")
```

Use in PLY

```python
import ply.yacc as yacc
from ast import *

# Parser rules with AST construction
def p_expr_plus(p):
    'expr : expr PLUS term'
    p[0] = create_binop(p[1], '+', p[3])

def p_expr_minus(p):
    'expr : expr MINUS term'
    p[0] = create_binop(p[1], '-', p[3])

def p_expr_term(p):
    'expr : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = create_binop(p[1], '*', p[3])
```

```python
def p_term_divide(p):
    'term : term DIVIDE factor'
    p[0] = create_binop(p[1], '/', p[3])

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_number(p):
    'factor : NUMBER'
    p[0] = create_number(p[1])

def p_factor_variable(p):
    'factor : IDENTIFIER'
    p[0] = create_variable(p[1])

def p_factor_parens(p):
    'factor : LPAREN expr RPAREN'
    p[0] = p[2]
```

Ocaml

```ocaml
(* Algebraic data type for AST nodes *)
type ast_node =
  | Number of int
  | BinOp of ast_node * char * ast_node
  | Variable of string

(* Constructor functions *)
let create_number value = Number value

let create_binop left op right = BinOp (left, op, right)

let create_variable name = Variable name

...
```

```ocaml
(* In parser.mly *)
%{
open Ast
%}

%token <int> NUMBER
%token <string> IDENTIFIER
%token PLUS MINUS TIMES DIVIDE LPAREN RPAREN
%start main
%type <Ast.ast_node> main expr term factor

%%

main:
    expr { $1 }

expr:
    expr PLUS term  { create_binop $1 '+' $3 }
  | expr MINUS term { create_binop $1 '-' $3 }
  | term            { $1 }

term:
    term TIMES factor  { create_binop $1 '*' $3 }
  | term DIVIDE factor { create_binop $1 '/' $3 }
  | factor             { $1 }

factor:
    NUMBER            { create_number $1 }
  | IDENTIFIER        { create_variable $1 }
  | LPAREN expr RPAREN { $2 }
```

```c
// Define what our AST nodes look like
typedef struct ASTNode {
    enum { NUMBER, BINOP } type;
    union {
        int value;                      // For numbers
        struct {                        // For binary operations
            struct ASTNode* left;
            char operator;
            struct ASTNode* right;
        } binop;
    };
  } ASTNode;

  // function prototypes

expr : expr '+' term   { $$ = create_binop($1, '+', $3); }
     | expr '-' term   { $$ = create_binop($1, '-', $3); }
     | term            { $$ = $1; }
     ;

term : term '*' factor { $$ = create_binop($1, '*', $3); }
     | factor          { $$ = $1; }
     ;

factor : NUMBER        { $$ = create_number($1); }
       | '(' expr ')'  { $$ = $2; }
         ;
```

Result for "2 + 3 * 4"

```
    +
   / \
  2   *
     / \
    3   4

==============

BinOp: +
  Number: 2
  BinOp: *
    Number: 3
    Number: 4
```

Semantic Actions: Key Takeaways

```
When to Use Semantic Actions:

 1.   Building ASTs (most common)

      expr : expr '+' term { $$ = new AddNode($1, $3); }

 2.   Type Checking

      assignment : ID '=' expr { check_types($1.type, $3.type); }

 3.   Symbol Table Management

      declaration : TYPE ID { add_to_symbol_table($2, $1); }

 4.   Code Generation

      expr : expr '+' term { $$ = generate_add($1, $3); }
```

In Project, you need to write actions for smth like this

```
Goal     ::= MainClass ( ClassDeclaration )* <EOF>
MainClass   ::= "class" Identifier "{" "public" "static" "void" "main" "(" "String" "[" "]" Identifier ")"
"{" Statement "}" "}"
ClassDeclaration    ::= "class" Identifier ( "extends" Identifier )? "{" ( VarDeclaration )* (
MethodDeclaration )* "}"
VarDeclaration  ::= Type Identifier ";"
MethodDeclaration   ::= "public" Type Identifier "(" ( Type Identifier ( "," Type Identifier )* )? ")" "{"
( VarDeclaration )* ( Statement )* "return" Expression ";" "}"
Type     ::= "int" "[" "]"
|    "boolean"
|    "int"
|    Identifier
Statement   ::= "{" ( Statement )* "}"
|    "if" "(" Expression ")" Statement "else" Statement
|    "while" "(" Expression ")" Statement
|    "System.out.println" "(" Expression ")" ";"
|    Identifier "=" Expression ";"
|    Identifier "[" Expression "]" "=" Expression ";"
```

Minijava grammar (written in BNF)
[The MiniJava Language Specification Grammar]()

```
Expression  ::= Expression ( "&&" | "<" | "+" | "-" | "*" ) Expression
|   Expression "[" Expression "]"
|   Expression "." "length"
|   Expression "." Identifier "(" ( Expression ( "," Expression )* )? ")"
|   <INTEGER_LITERAL>
|   "true"
|   "false"
|   Identifier
|   "this"
|   "new" "int" "[" Expression "]"
|   "new" Identifier "(" ")"
|   "!" Expression
|   "(" Expression ")"
Identifier  ::= <IDENTIFIER>
```

Minijava grammar (written in BNF)
https://www.cambridge.org/resources/052182060X/
https://www.cs.purdue.edu/homes/hosking/502/project/grammar.html

Summary of SDT

We can specify language syntax using CFG

• A parser will answer whether $s \in L(G)$

– … and will trace a parse tree

– … in whose productions we build an AST

– … that we pass on to the rest of the compiler

Intro to top down parsing

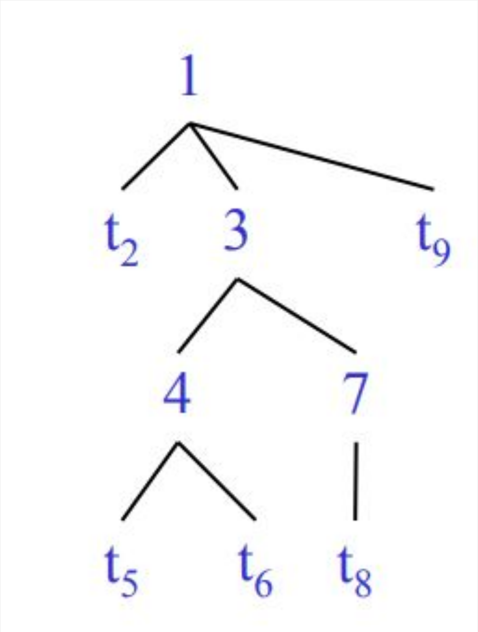
The parse tree is constructed

– From the top

– From left to right

Terminals are seen in order of appearance in the token stream:

t2 t5 t6 t8 t9



Recursive descent

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

Token stream is: (int5)

Start with **top-level non-terminal E**

– Try the rules for E in order

If there is a mismatch backtrack

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

Recursive descent

E → T |T + E

T → int | int * T | (E)

E → T |T + E

T → int | int * T | (E)

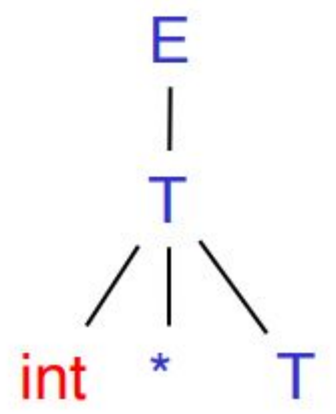
- Mismatch: int is not (
- backtrack

Token stream is: (int5)

Start with **top-level non-terminal E**

– Try the rules for E in order

If there is a mismatch backtrack



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

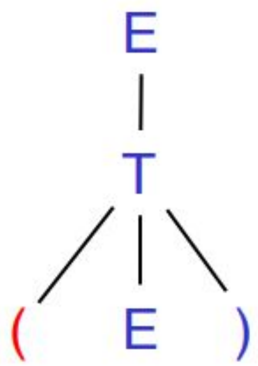
Recursive descent

E → T |T + E

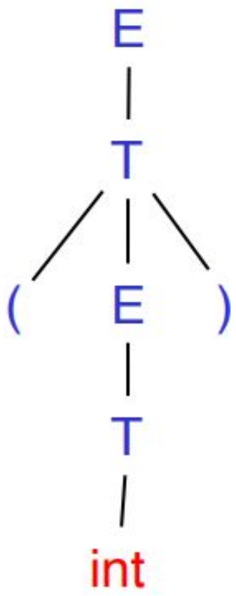
T → int | int * T | (E)

Token stream is: (int5)

- Match: (
- Advance input
  - Move to int5



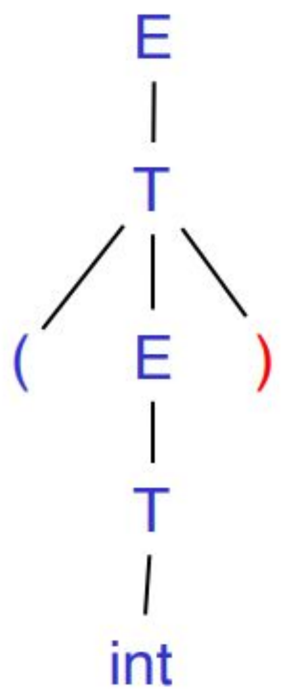
- Match: int5
- Advance input
  - Move to)



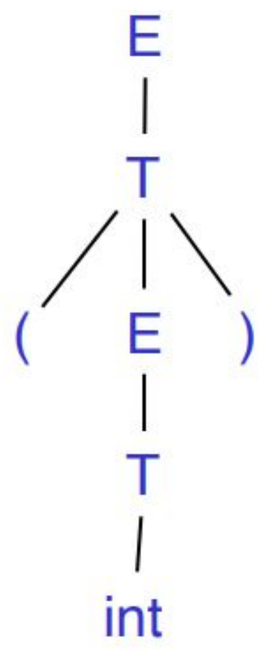
https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Recursive descent

- Match:)
- Advance input
  - Move



- End of input, accept!



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

Writing a limited recursive descent parser

Let TOKEN be the type of tokens

– Special tokens INT, OPEN, CLOSE, PLUS, TIMES

Let the global **next** point to the next token

Define boolean functions that check the token string for a match of

- A given token terminal

```
bool term(TOKEN tok) {
        return *next++ == tok;
        }
```

- The nth production of S:

```
bool Sn() { … }
```

- Try all productions of S:

```
bool S() { … }
```

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

Writing a limited recursive descent parser

- For production **E → T**

```
bool E1() {
   return T();
}
```

- For production **E → T + E**

```
bool E2() {
   return T() && term(PLUS) && E();
}
```

- For all productions of E (with backtracking)

```
bool E(){
   TOKEN *save = next;
   return (next = save, E1()) || (next = save, E2());
}
```

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

Functions for non-terminal T

```
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T(){
   TOKEN *save = next;
   return (next = save, T1()
        || (next = save, T2())
        || (next = save, T3());
}
```

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

Notes on writing recursive descent parser

To start the parser

– Initialize next to point to first token

– Invoke **E()**

• Notice how this simulates the example parse

• Easy to implement by hand

– But not completely general

– Cannot backtrack once a production is successful

– Works for grammars where at most one production can succeed for a non-terminal

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

Example

```
E → T |T + E
T → int | int * T | ( E )
bool term(TOKEN tok) { return *next++ == tok; }
bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {TOKEN *save = next;
     return (next = save, E1()) || (next = save, E2()); }
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next;
     return (next = save, T1())
     || (next = save, T2())
     || (next = save, T3()); }
```

(int)

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

Programming examples

```
// Direct translation of grammar rules
ASTNode* parse_expression() {
    ASTNode* left = parse_term();
    if (current_token == PLUS) {
        get_next_token();
        ASTNode* right = parse_expression();
        return create_binop_node('+', left, right);
    }
    return left;
}

ASTNode* parse_term() {
    ASTNode* left = parse_factor();
    if (current_token == TIMES) {
        get_next_token();
        ASTNode* right = parse_term();
        return create_binop_node('*', left, right);
    }
    return left;
}
```

Programming examples

```python
# Object-oriented with exception handling
def parse_expression(self):
    left = self.parse_term()
    while self.current_token.type in ('PLUS', 'MINUS'):
        op = self.current_token
        self.advance()
        right = self.parse_term()
        left = BinOp(left, op, right)
    return left

def parse_term(self):
    left = self.parse_factor()
    while self.current_token.type in ('TIMES', 'DIVIDE'):
        op = self.current_token
        self.advance()
        right = self.parse_factor()
        left = BinOp(left, op, right)
    return left
```

Programming examples

```ocaml
(* Pattern matching and recursive structure *)
let rec parse_expression tokens =
  let left, tokens = parse_term tokens in
  match tokens with
  | (PLUS, _) :: rest ->
      let right, rest = parse_expression rest in
      Binop(Plus, left, right), rest
  | (MINUS, _) :: rest ->
      let right, rest = parse_expression rest in
      Binop(Minus, left, right), rest
  | _ -> left, tokens

and parse_term tokens =
  let left, tokens = parse_factor tokens in
  match tokens with
  | (TIMES, _) :: rest ->
      let right, rest = parse_term rest in
      Binop(Times, left, right), rest
  | (DIVIDE, _) :: rest ->
      let right, rest = parse_term rest in
      Binop(Divide, left, right), rest
  | _ -> left, tokens
```

When Recursive Descent Does Not Work

Consider a production `S → S a`

```
bool S1() { return S() && term(a); }
bool S() { return S1(); }
```

- `S()` goes into an infinite loop

- A left-recursive grammar has a non-terminal `S`

  `S →+ Sα` for some `α`
- Recursive descent does not work in such cases

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

Elimination of Left Recursion

Consider the left-recursive grammar

$$S \rightarrow S\ \alpha \mid \beta$$

S generates all strings starting with a **β** and followed by a number of **α**

with **right-recursion**

$$S \rightarrow \beta\ S'$$

$$S' \rightarrow \alpha\ S' \mid \varepsilon$$

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

More on elimination rule for left recursion

In general

$\mathbf{S \to S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m}$

All strings derived from **S** start with one of **β1,…, βm** and continue with several instances of

α1,…,αn

Rewrite as

$S \to \beta_1\ S' \mid \ldots \mid \beta_m\ S'$

$S' \to \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$

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

Example

1. $P \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow id$
5. $T \rightarrow$ int

1. $P \rightarrow E$
2. $E \rightarrow T\ E'$
3. $E' \rightarrow +\ T\ E'$
4. $E' \rightarrow \varepsilon$
5. $T \rightarrow id$
6. $T \rightarrow$ int

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

Indirect left recursion

The grammar

$$S \rightarrow A\ \alpha \mid \delta$$

$$A \rightarrow S\ \beta$$

is also left-recursive because $\mathbf{S \rightarrow^{+} S\ \beta\ \alpha}$

- This left-recursion can also be eliminated

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

A general algorithm (skipped)

```
impose an order on the nonterminals, A1, A2, ..., An

for i ← 1 to n do;
    for j ← 1 to i - 1 do;
        if ∃ a production Ai→Ajγ
            then replace Ai→Ajγ with one or more
                 productions that expand Aj
    end;
    rewrite the productions to eliminate
        any direct left recursion on Ai
end;
```

■ **FIGURE 3.6** Removal of Indirect Left Recursion.

General algorithm for top down parsing

```
root ← node for the start symbol, S;
focus ← root;
push(null);
word ← NextWord( );

while (true) do;
    if (focus is a nonterminal) then begin;
        pick next rule to expand focus (A → β1,β2,...,βn);
        build nodes for β1,β2...βn as children of focus;
        push(βn,βn−1,...,β2);
        focus ← β1;
    end;
    else if (word matches focus) then begin;
        word ← NextWord( );
        focus ← pop( )
    end;
    else if (word = eof and focus = null)
        then accept the input and return root;
        else backtrack;
end;
```

FIGURE 3.2 A Leftmost, Top-Down Parsing Algorithm.

- on the mismatch, it must undo the actions

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Backtrack free parsing

The major source of inefficiency in the leftmost, top-down parser arises from its need to backtrack

- on the mismatch, it must undo the actions

0	*Goal*	$\rightarrow$	*Expr*	6	*Term'*	$\rightarrow$	$\times$ *Factor Term'*
1	*Expr*	$\rightarrow$	*Term Expr'*	7		|	$\div$ *Factor Term'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	$\rightarrow$	*Factor Term'*	11		|	name

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Using a lookahead symbol

- For this grammar, the parser can avoid backtracking by
  - considering both the focus symbol and the next input symbol(lookahead symbol).
- Using a one symbol lookahead, the parser can disambiguate all of the choices that arise in parsing the right-recursive expression grammar.
  - the grammar is backtrack free with a lookahead of one symbol

0	*Goal*	$\rightarrow$	*Expr*	6	*Term'*	$\rightarrow$	x *Factor Term'*
1	*Expr*	$\rightarrow$	*Term Expr'*	7		|	÷ *Factor Term'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	$\rightarrow$	*Factor Term'*	11		|	name

For each grammar symbol α,

define the set **first(α)** as the set of terminal symbols that can appear as the first word in some string derived from **α**

- If **α** is either a terminal, **ε**, or **eof**,
  - then **first(α)** is **{α}**.
- For a nonterminal A,
  - **first(A)** contains the complete set of terminal symbols that can appear as the leading symbol in a sentential form derived from A.

example

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	x *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	***Expr'***	***Term***	***Term'***	***Factor***
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Computing First sets

```
For Terminals:
   For each terminal a ∈ Σ: FIRST(a) = {a}

For Non-Terminals:
Repeat:
   For each rule X → Y1Y2...Yk in a grammar G:
        if a is in FIRST(Y1) OR a is in FIRST(Yn) and Y1...Yn-1 ⇒ ε
          then add a to FIRST(X)
        if Y1...Yk ⇒ ε
          then add ε to FIRST(X).
until no more changes occur
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

first(ε) = {ε}

- matches no word returned by the scanner.

parser needs to know which words can appear as **the leading symbol after a valid application of rule 4**

- the set of symbols that can follow an **Expr’**

0	*Goal*	$\rightarrow$	*Expr*	6	*Term′*	$\rightarrow$	x *Factor Term′*
1	*Expr*	$\rightarrow$	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	$\rightarrow$	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	$\rightarrow$	*Factor Term′*	11		|	name

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Follow sets of nonterminals

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr′*
2	*Expr′*	$\rightarrow$	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term′*
6	*Term′*	$\rightarrow$	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

	Expr	***Expr′***	***Term***	***Term′***	***Factor***
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Computing Follow Sets for a Grammar G

```
FOLLOW(A) is the set of terminals that can come after non-terminal A
FOLLOW(S) = {$} where S is the start symbol.

Repeat:
   if A → αBβ then:
       add FIRST(β) (excepting ε) to FOLLOW(B).
   if A → αB or FIRST(β) contains ε then:
       add FOLLOW(A) to FOLLOW(B).
until no more changes occur.
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Using first and follow

For productions

$$\text{FIRST}^+(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \textit{otherwise} \end{cases}$$

In backtrack free grammar, any nonterminal A $A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset, \ \forall \ 1 \le i,j \le n, \ i \ne j.$$

0	$Goal$	$\rightarrow$	$Expr$	6	$Term'$	$\rightarrow$	$\times$ $Factor\ Term'$
1	$Expr$	$\rightarrow$	$Term\ Expr'$	7		$\mid$	$\div$ $Factor\ Term'$
2	$Expr'$	$\rightarrow$	$+$ $Term\ Expr'$	8		$\mid$	ϵ
3		$\mid$	$-$ $Term\ Expr'$	9	$Factor$	$\rightarrow$	($Expr$)
4		$\mid$	ϵ	10		$\mid$	num
5	$Term$	$\rightarrow$	$Factor\ Term'$	11		$\mid$	name

only productions 4 and 8 have different **first+** and **first** sets

	Production	FIRST set	FIRST$^+$ set
4	$Expr' \rightarrow \epsilon$	$\{\epsilon\}$	$\{\epsilon$, eof,)$\}$
8	$Term' \rightarrow \epsilon$	$\{\epsilon\}$	$\{\epsilon$, eof, +, -,)$\}$

The common prefix

11	*Factor*	$\rightarrow$	`name`
12		$\mid$	`name` [*ArgList*]
13		$\mid$	`name` (*ArgList*)
15	*ArgList*	$\rightarrow$	*Expr MoreArgs*
16	*MoreArgs*	$\rightarrow$	, *Expr MoreArgs*
17		$\mid$	ϵ

Because productions 11, 12, and 13 all begin with name, they have identical first+ sets. When the parser tries to expand an instance of Factor with a lookahead of name, it has no basis to choose among 11, 12, and 13.

Eliminating common prefixes (left factoring)

transform these productions to create disjoint first+ sets.

11	*Factor*	$\rightarrow$	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)
15	*ArgList*	$\rightarrow$	*Expr MoreArgs*
16	*MoreArgs*	$\rightarrow$	, *Expr MoreArgs*
17		|	ϵ

11	*Factor*	$\rightarrow$	name *Arguments*
12	*Arguments*	$\rightarrow$	[*ArgList*]
13		|	(*ArgList*)
14		|	ϵ

Take a non terminal

$$A \rightarrow \alpha\beta 1 \mid \alpha\beta 2 \mid \cdot\cdot\cdot \mid \alpha\beta n$$
$$\mid \gamma 1 \mid \gamma 2 \mid \cdot\cdot\cdot \mid \gamma j$$

Rewrite the original as

$$A \rightarrow \alpha\ B \mid \gamma 1 \mid \gamma 2 \mid \cdot\cdot\cdot \mid \gamma j$$
$$B \rightarrow \beta 1 \mid \beta 2 \mid \cdot\cdot\cdot \mid \beta n$$

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

Summary

Backtrack-free grammars lend themselves to simple and efficient parsing with a recursive descent

By using first, first+ and follow sets we can generate **predictive top down-parser(LL(1) parser)**

An example predictive LL(1) with recursive descent parser

Three helper functions are needed:

scan_token()

- returns the next token on the input stream.

putback_token(t)

- puts an unexpected token back on the input stream.

expect_token(t)

- calls scan token to retrieve the next token.

```
1. P → E
2. E → T E'
3. E' → + T E'
4. E' → ϵ
5. T → F T'
6. T' → * F T'
7. T' → ϵ
8. F → ( E )
9. F → int
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

1. P → E
2. E → T E'
3. E' → + T E'
4. E' → ϵ
5. T → F T'
6. T' → * F T'
7. T' → ϵ
8. F → (E)
9. F → int

```
int parse_P(){
   return parse_E() && expect_token(TOKEN_EOF);
}
int parse_E(){
   return parse_T() && parse_E_prime();
}
int parse_E_prime(){
   token_t t = scan_token();
   if (t == TOKEN_PLUS) {
       return parse_T() && parse_E_prime();
   }
   else{
       putback_token(t);
       return 1;
   }
}
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

1. P → E
2. E → T E’
3. E’ → + T E’
4. E’ → ϵ
5. T → F T’
6. T’ → * F T’
7. T’ → ϵ
8. F → (E)
9. F → int

```
int parse_T_prime(){
    token_t t = scan_token();
    if (t == TOKEN_MULTIPLY)    {
        return parse_F() && parse_T_prime();
    }
    else    {
        putback_token(t);
        return 1;
    }
}
```

```
int parse_F(){
    token_t t = scan_token();
    if (t == TOKEN_LPAREN)    {
        return parse_E() && expect_token(TOKEN_RPAREN);
    }
    else if (t == TOKEN_INT)    {
        return 1;
    }
    else    {
        printf("parse error: unexpected token %s\n",
                token_string(t));
        return 0;
    }
}
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Next week table driven LL(1) parser & bottom-up parser