Error Handling
Syntax-Directed Translation
Recursive Descent Parsing

CS143

Lecture 6

Instructor: Fredrik Kjolstad

Slide design by Prof. Alex Aiken, with modifications

Announcements

- PA1 & WA1
  - Due today at midnight

- PA2 & WA2
  - Assigned today

Outline

- Extensions of CFG for parsing
  - Precedence declarations
  - Error handling
  - Semantic actions
- Constructing an abstract syntax tree (AST)
- Recursive descent parsing

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
Correctness	your favorite program	Tester/User

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

Syntax Error Recovery

- Approaches from simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction

- Not all are supported by all parser generators

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 <u>synchronizing</u> tokens
  - Typically the statement or expression terminators

Error Recovery: Panic Mode (Cont.)

- Consider 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

  $E \rightarrow$ int | E + E | (E) | error int | (error)

Error Recovery: Error Productions

- Idea: specify in the grammar known common mistakes
- Essentially promotes common errors to alternative syntax
- Example:
  - Write 5 x instead of 5 * x
  - Add the production $E \rightarrow \ldots \mid E\ E$
- Disadvantage
  - Complicates the grammar

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 supported by most tools

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

Abstract Syntax Trees

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

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

- <u>Abstract syntax trees</u>
  – Like parse trees but ignore some details
  – Abbreviated as AST

Abstract Syntax Trees (Cont.)

- Consider the grammar

  $E \rightarrow \text{int} \mid (E) \mid E + E$

- And the string

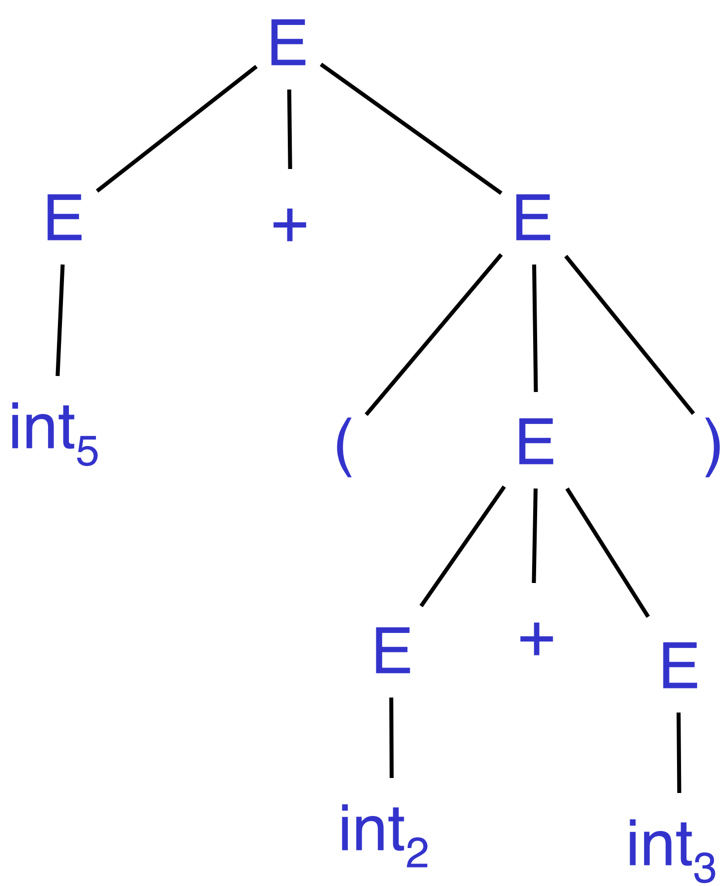
  5 + (2 + 3)

- After lexical analysis (a list of tokens)

  int_5 '+' '(' int_2 '+' int_3 ')'

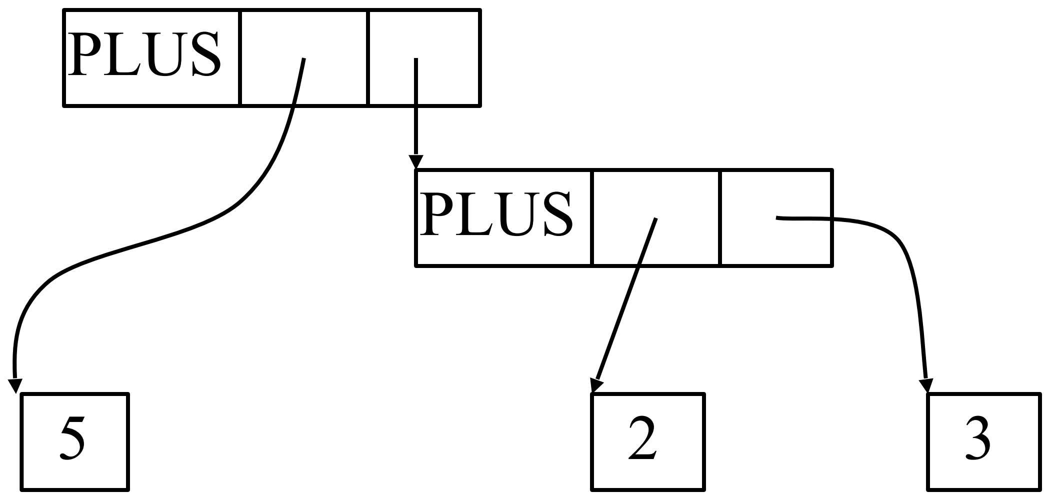
- During parsing we build a parse tree …

Example of Parse Tree



- Traces the operation of the parser
- Does capture the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes

Example of 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

Semantic Actions Extension to CFGs

- 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
  – Written as $X \to Y_1 \ldots Y_n$ { action }
  – That can refer to or compute symbol attributes

Semantic Actions: Example

- Consider the grammar

$$E \rightarrow \text{int} \mid E + E \mid (E)$$

- 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:

$$
\begin{array}{lll}
E \rightarrow \text{int} & & \{ E.val = \text{int}.val \} \\
\quad \mid E_1 + E_2 & & \{ E.val = E_1.val + E_2.val \} \\
\quad \mid (E_1) & & \{ E.val = E_1.val \}
\end{array}
$$

Semantic Actions: Example (Cont.)

- String: 5 + (2 + 3)
- Tokens: int_5 '+' '(' int_2 '+' int_3 ')'

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E_1 \rightarrow int_5$	$E_1.val = int_5.val = 5$
$E_2 \rightarrow (E_3)$	$E_2.val = E_3.val$
$E_3 \rightarrow E_4 + E_5$	$E_3.val = E_4.val + E_5.val$
$E_4 \rightarrow int_2$	$E_4.val = int_2.val = 2$
$E_5 \rightarrow int_3$	$E_5.val = int_3.val = 3$

Semantic Actions: Notes

- 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

Semantic Actions: Notes

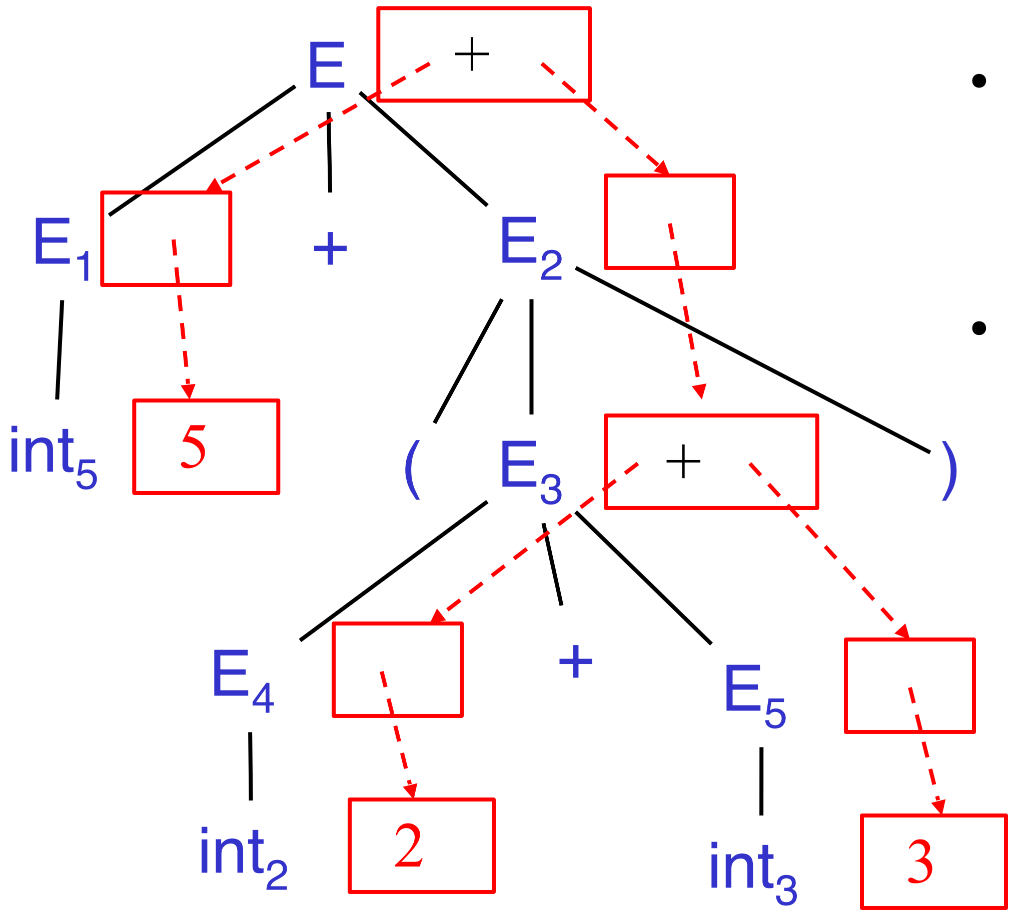
- We'll explore actions as pure equations
  - But note bison has a fixed order of evaluation for actions

- Example:

$$E_3.val = E_4.val + E_5.val$$

  - Must compute $E_4.val$ and $E_5.val$ before $E_3.val$
  - We say that $E_3.val$ depends on $E_4.val$ and $E_5.val$

Dependency Graph

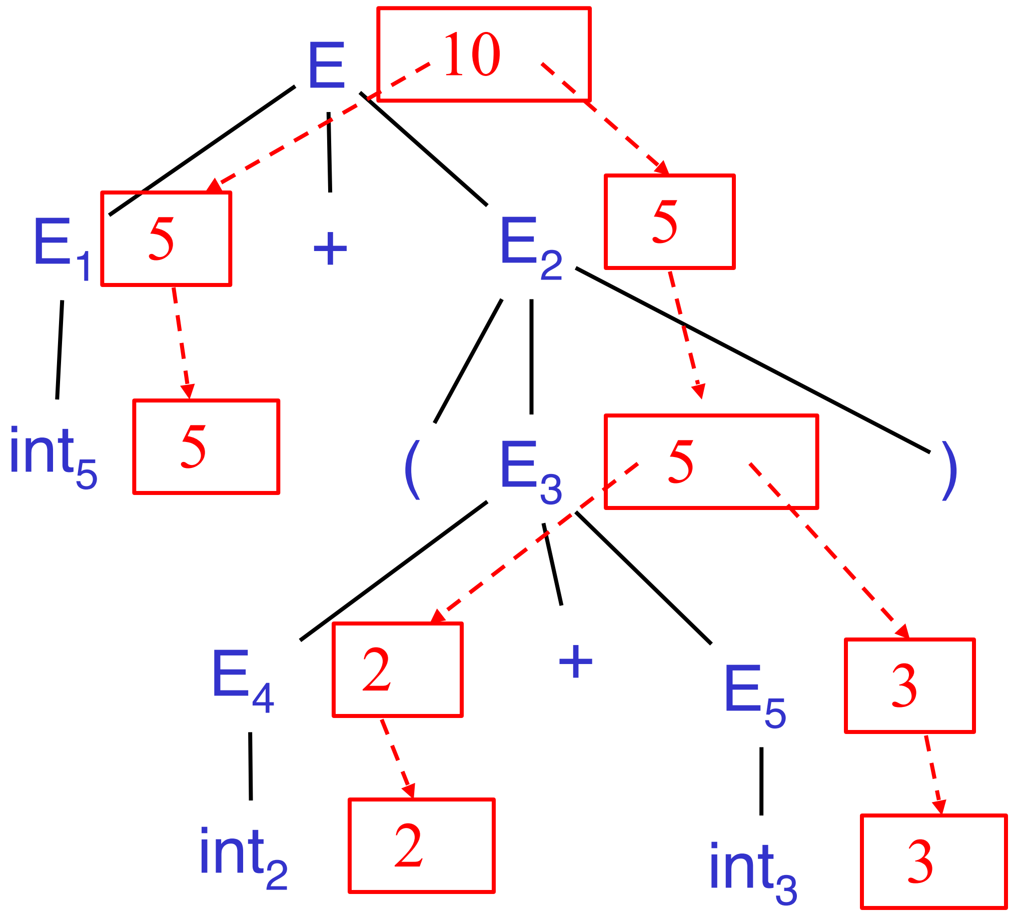


- Each node labeled E has one slot for the val attribute
- Note the dependencies

Evaluating Attributes

- An attribute must be computed after all its successors in the dependency graph have been computed
  - In previous example attributes can be computed bottom-up
- Such an order exists when there are no cycles
  - Cyclically defined attributes are not legal

Dependency Graph



Semantic Actions: Notes (Cont.)

- Synthesized attributes
  - Calculated from attributes of descendents in the parse tree
  - E.val is a synthesized attribute
  - Can always be calculated in a bottom-up order

- Grammars with only synthesized attributes are called S-attributed grammars
  - Most common case

Inherited Attributes

- Another kind of attribute
- Calculated from attributes of parent and/or siblings in the parse tree
- Example: a line calculator

Semantic Actions: Notes (Cont.)

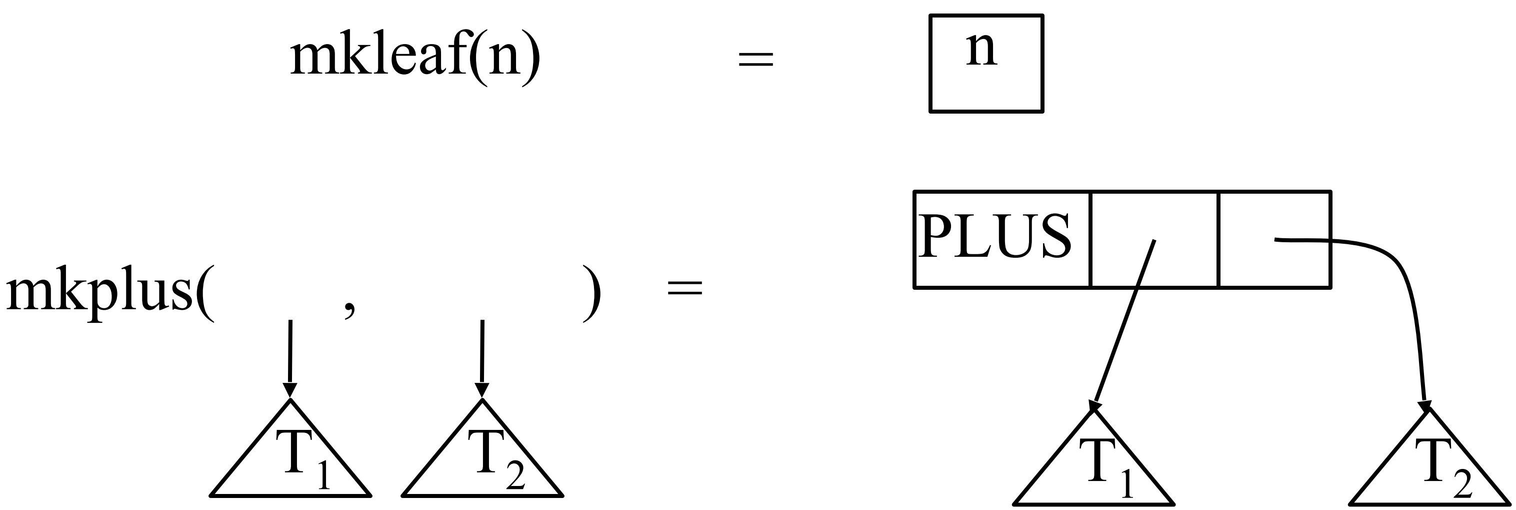
- Semantic actions can be used to build ASTs

- And many other things as well
  - Also used for type checking, code generation, computation, …

- Process is called <u>syntax-directed translation</u>
  - Substantial generalization over CFGs

Constructing an AST

- We first define the AST data type
  - Supplied by us for the project
- Consider an abstract tree type with two constructors:



Constructing an AST

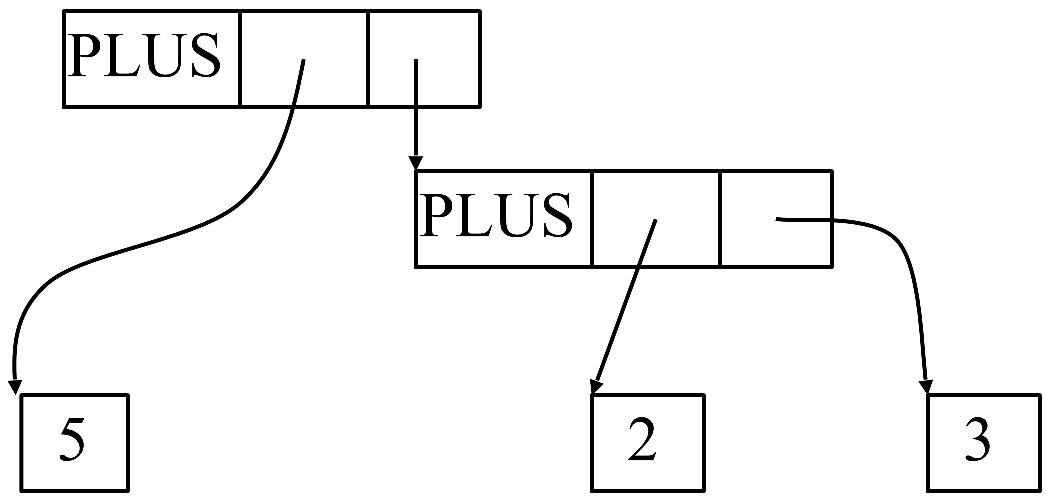
- We define a synthesized attribute ast
  - Values of ast values are ASTs
  - We assume that int.lexval is the value of the integer lexeme
  - Computed using semantic actions

$E \rightarrow \text{int}$	$E.\text{ast} = \text{mkleaf}(\text{int.lexval})$
$\mid E_1 + E_2$	$E.\text{ast} = \text{mkplus}(E_1.\text{ast}, E_2.\text{ast})$
$\mid (E_1)$	$E.\text{ast} = E_1.\text{ast}$

Abstract Syntax Tree Example

- Consider the string int_5 ‘+’ ‘(‘ int_2 ‘+’ int_3 ‘)’
- A bottom-up evaluation of the ast attribute:

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



Summary

- We can specify language syntax using CFG

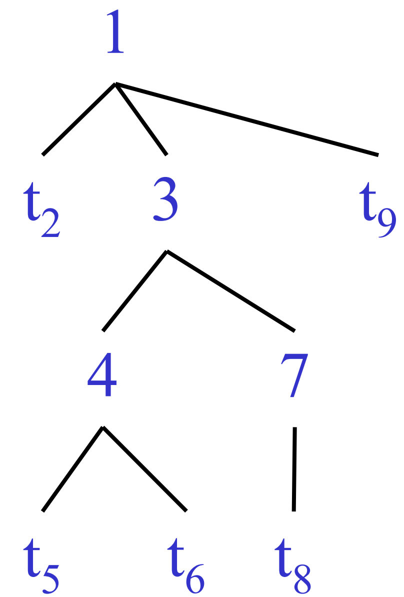
- A parser will answer whether $s \in L(G)$
  - … and will trace a parse tree
  - … in whose productions we build an AST
  - … that we pass on to the rest of the compiler

Intro to Top-Down Parsing: The Idea

- The parse tree is constructed
  - From the top
  - From left to right

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

  $t_2 \;\; t_5 \;\; t_6 \;\; t_8 \;\; t_9$



Recursive Descent Parsing

- Consider the grammar

  $E \rightarrow T \mid T + E$

  $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

- Token stream is: (int_5)

- Start with top-level non-terminal E
  – Try the rules for E in order

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

E

$(\; int_5 \;)$

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

E

T

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

E

T

int

Mismatch: int is not (!
Backtrack …

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

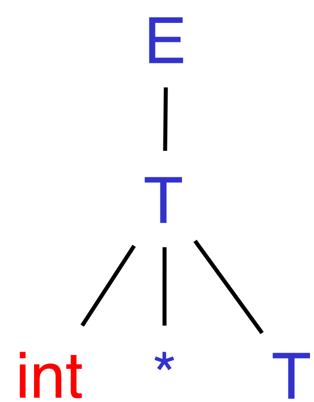
$$\begin{array}{c} E \\ | \\ T \end{array}$$

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



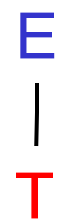
Mismatch: int is not (!
Backtrack …

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

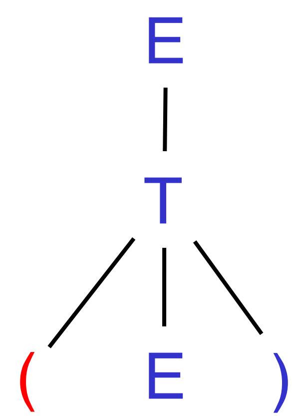


(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$



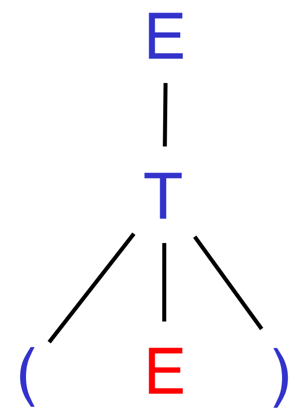
Match! Advance input.

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

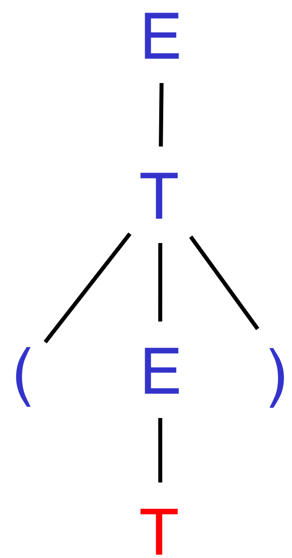


(int$_5$)

Recursive Descent Parsing

E → T |T + E

T → int | int * T | (E)

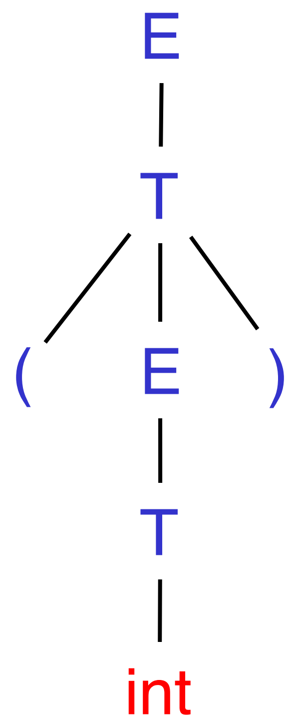


(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



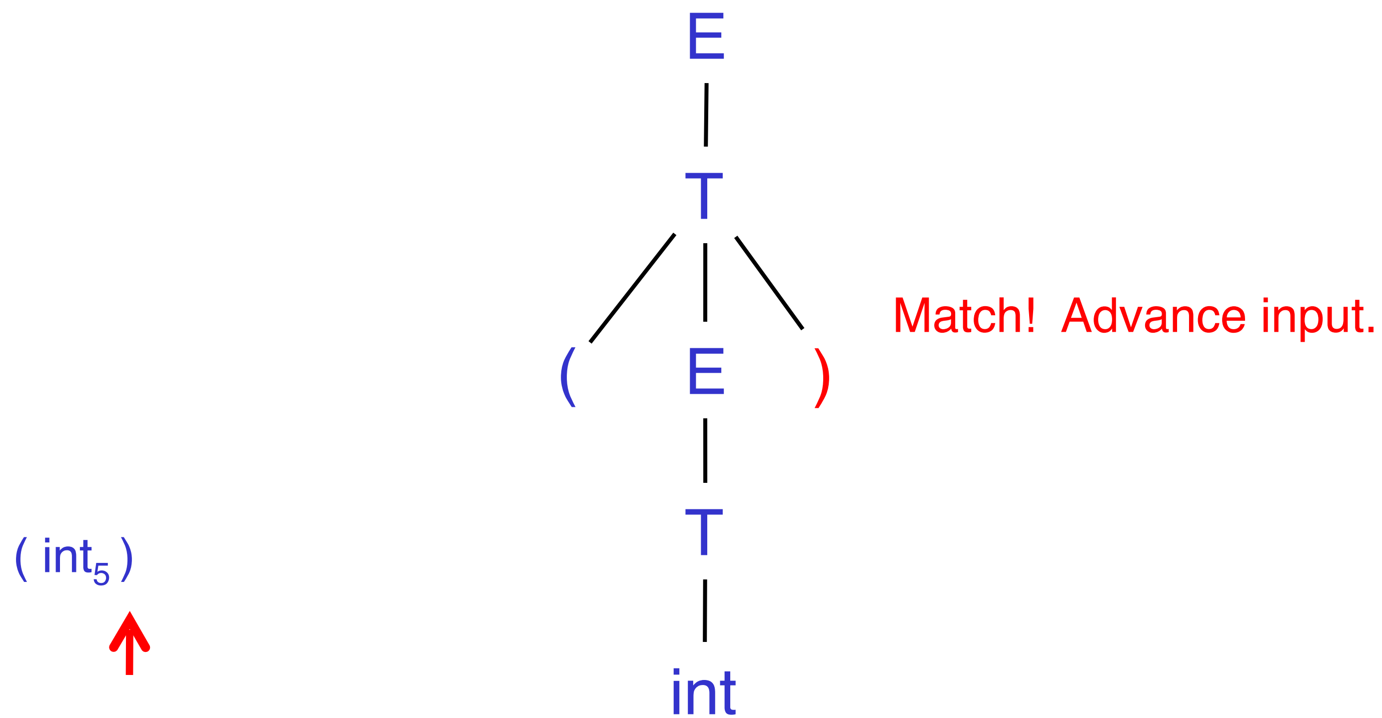
Match! Advance input.

(int_5)

Recursive Descent Parsing

$E \rightarrow T \mid T + E$

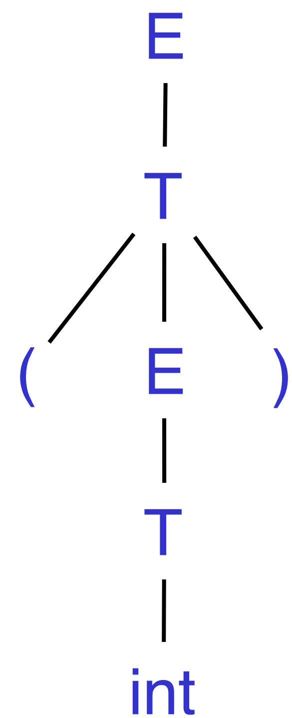
$T \rightarrow int \mid int * T \mid (E)$



Recursive Descent Parsing

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



End of input, accept.

(int_5)

A Recursive Descent Parser: Preliminaries

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES

- Let the global next point to the next token

A (Limited) Recursive Descent Parser (2)

- 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 S_n() { … }
  - Try all productions of S:
    ```
    bool S() { … }
    ```

A (Limited) Recursive Descent Parser (3)

- For production $E \rightarrow T$

  bool E_1() { return T(); }

- For production $E \rightarrow T + E$

  bool E_2() { return T() && term(PLUS) && E(); }

- For all productions of E (with backtracking)

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

A (Limited) Recursive Descent Parser (4)

- 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()); }
```

Recursive Descent Parsing. Notes.

- To start the parser
  - Initialize next to point to first token
  - Invoke E()
- 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

Example

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

(int)

```
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()); }
```

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S\ a$

  bool S_1() { return S() && term(a); }

  bool S() { return S_1(); }

- S() goes into an infinite loop

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

  $S \rightarrow^+ S\alpha$ for some α

- Recursive descent does not work in such cases

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

- Can rewrite using right-recursion

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

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

More Elimination of Left-Recursion

- In general

$$S \rightarrow S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

$$S \rightarrow \beta_1\ S' \mid \ldots \mid \beta_m\ S'$$
$$S' \rightarrow \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$$

General Left Recursion

- The grammar

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

  $A \rightarrow S\,\beta$

  is also left-recursive because

  $$S \rightarrow^{+} S\,\beta\,\alpha$$

- This left-recursion can also be eliminated

- See Dragon Book for general algorithm
  - Section 4.3

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Historically unpopular because of backtracking
  - Was thought to be too inefficient
  - In practice, with some tweaks, fast and simple on modern machines
- Backtracking can be controlled by restricting the grammar