Error Handling
Syntax-Directed Translation
Recursive Descent Parsing

Lecture 6



## Announcements

- PA1
  - Due today at midnight
  - README, test case
  - Your name(s)!
- WA1
  - Due today at 5pm
- PA2
  - Assigned today
- WA2
  - Assigned Tuesday



## Outline

- Extensions of CFG for parsing
  - Precedence declarations
  - Error handling
  - Semantic actions
- Constructing a parse tree
- Recursive descent



## Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g. in C)

| Error kind | Example | 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



## 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

## 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



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



## Syntax 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 → … | 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 all tools support it



## 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

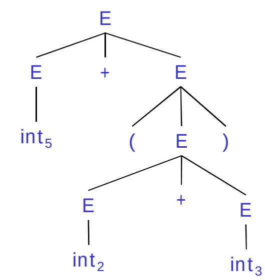
- So far a parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program
- <u>Abstract syntax trees</u>
  - Like parse trees but ignore some details
  - Abbreviated as AST

## Abstract Syntax Tree. (Cont.)

- Consider the grammar
  $E \rightarrow int \mid ( E ) \mid E + E$
- 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 …



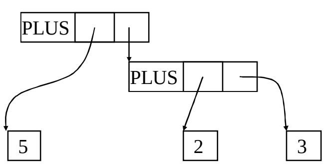
## Example of Parse Tree




- 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 <u>abstracts</u> from the concrete syntax
  => more compact and easier to use
- An important data structure in a compiler



## Semantic Actions

- This is what we'll use to construct ASTs
- Each grammar symbol may have <u>attributes</u>
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer
- Each production may have an <u>action</u>
  - Written as: $X \rightarrow Y_1 \ldots Y_n$ { action }
  - That can refer to or compute symbol attributes



## Semantic Actions: An Example

- 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:

| | |
|---|---|
| $E \rightarrow int$ | { E.val = int.val } |
| $\mid E_1 + E_2$ | { $E.val = E_1.val + E_2.val$ } |
| $\mid ( E_1 )$ | { $E.val = E_1.val$ } |



## Semantic Actions: An Example (Cont.)

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

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

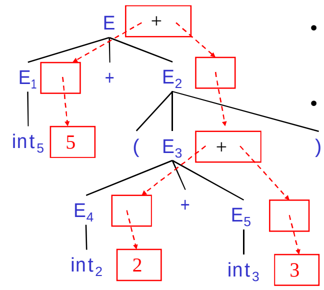
## Semantic Actions: Notes

- Semantic actions specify a system of equations
  - Order of resolution is not specified
- 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 $E_5.val$
- The parser must find the order of evaluation



## Dependency Graph




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

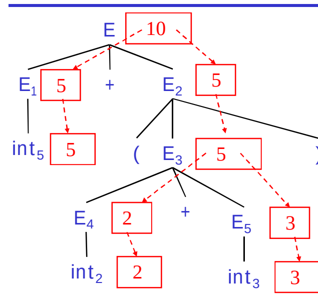


## Evaluating Attributes

- 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.)

- <u>Synthesized</u> 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 <u>S-attributed</u> 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

## A Line Calculator

- Each line contains an expression
  $E \rightarrow int \mid E + E$
- Each line is terminated with the = sign
  $L \rightarrow E = \mid + E =$
- In second form the value of previous line is used as starting value
- A program is a sequence of lines
  $P \rightarrow \varepsilon \mid P\ L$



## Attributes for the Line Calculator

- Each E has a synthesized attribute val
  - Calculated as before
- Each L has an attribute val
  $L \rightarrow E =$ { L.val = E.val }
  $\mid + E =$ { L.val = E.val + L.prev }
- We need the value of the previous line
- We use an inherited attribute L.prev

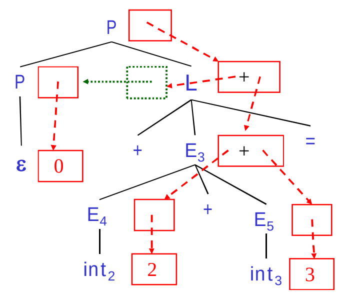


## Attributes for the Line Calculator (Cont.)

- Each P has a synthesized attribute val
  - The value of its last line
    $P \rightarrow \varepsilon$ { P.val = 0 }
    $\mid P_1\ L$ { P.val = L.val;
    L.prev = $P_1$.val }
  - Each L has an inherited attribute prev
  - L.prev is inherited from sibling $P_1$.val
- Example …



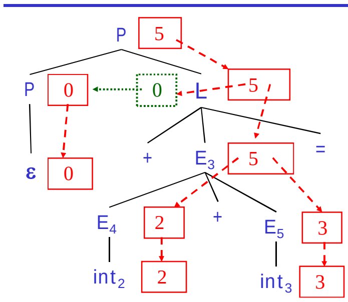
## Example of Inherited Attributes




- val synthesized
- prev inherited
- All can be computed in depth-first order



## Example of Inherited Attributes




- val synthesized
- prev inherited
- All can be computed in depth-first order

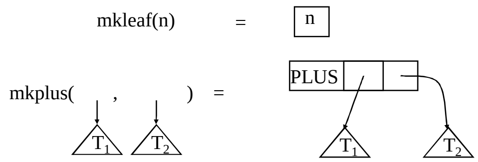


## Semantic Actions: Notes (Cont.)

- Semantic actions can be used to build ASTs
- And many other things as well
  - Also used for type checking, code generation, …
- 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 a Parse Tree

- 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

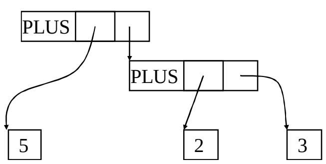
| | |
|---|---|
| $E \rightarrow$ int | E.ast = mkleaf(int.lexval) |
| $\mid E_1 + E_2$ | E.ast = mkplus($E_1$.ast, $E_2$.ast) |
| $\mid ( E_1 )$ | E.ast = $E_1$.ast |



## Parse Tree Example

- Consider the string $\text{int}_5$ '+' '(' $\text{int}_2$ '+' $\text{int}_3$ ')'
- A bottom-up evaluation of the ast attribute:

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






## Summary

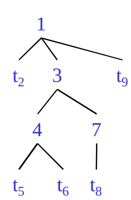
- We can specify language syntax using CFG
- A parser will answer whether $s \in L(G)$
  - … and will build a parse tree
  - … which we convert to an AST
  - … and pass on to the rest of the compiler



## 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

  $E \rightarrow T + E \mid T$

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

- Token stream is: $\text{int}_5 * \text{int}_2$
- Start with top-level non-terminal E
  - Try the rules for E in order

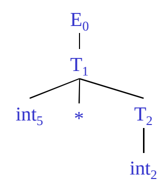
## Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \to T_1 + E_2$
  - Then try a rule for $T_1 \to ( E_3 )$
    - But ( does not match input token $int_5$
  - Try $T_1 \to$ int . Token matches.
    - But + after $T_1$ does not match input token *
  - Try $T_1 \to$ int * $T_2$
    - This will match but + after $T_1$ will be unmatched
- We've exhausted the choices for $T_1$
  - Failure!
  - Backtrack, try another choice for $E_0$



## Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \to T_1$
- Follow same steps as before for $T_1$
  - And succeed with $T_1 \to$ int * $T_2$ and $T_2 \to$ int
  - With the following parse tree






## 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 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; }`
  - A given production of S (the $n^{th}$)
    `bool S_n() { … }`
  - Any production of S:
    `bool S() { … }`



## A Recursive Descent Parser (3)

- For production $E \to T$
  `bool E_1() { return T(); }`
- For production $E \to T + E$
  `bool E_2() { return T() && term(PLUS) && E(); }`
- For all productions of E (with backtracking)

```
bool E() {
  TOKEN *save = next;
  return   (next = save, E_1())
        || (next = save, E_2()); }
```



## A Recursive Descent Parser (4)

- Functions for non-terminal T

```
bool T_1() { return term(OPEN) && E() && term(CLOSE); }
bool T_2() { return term(INT) && term(TIMES) && T(); }
bool T_3() { return term(INT); }

bool T() {
  TOKEN *save = next;
  return   (next = save, T_1())
        || (next = save, T_2())
        || (next = save, T_3()); }
```

## Recursive Descent Parsing. Notes.

- 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 does not always work …



## Example

$E \to T + E \mid T$ int * int

$T \to$ 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(OPEN) && E() && term(CLOSE); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(INT); }

bool T() { TOKEN *save = next;  return   (next = save, T1())
                                      || (next = save,  T2())
                                      || (next = save,  T3()); }
```



## When Recursive Descent Does Not Work

- Consider a production $S \to S\ a$

  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

  $S \to^+ S\alpha$ for some $\alpha$
- Recursive descent does not work in such cases



## Elimination of Left Recursion

- Consider the left-recursive grammar

  $S \to S\ \alpha \mid \beta$
- S generates all strings starting with a $\beta$ and followed by a number of $\alpha$
- Can rewrite using right-recursion

  $S \to \beta\ S'$

  $S' \to \alpha\ S' \mid \varepsilon$



## More Elimination of Left-Recursion

- In general

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

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

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



## General Left Recursion

- The grammar

  $S \to A\ \alpha \mid \delta$

  $A \to S\ \beta$

  is also left-recursive because

  $S \to^+ S\ \beta\ \alpha$
- 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
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar