Compilers

6.1 Error Handling

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	- compilation error
Correctness	your favorite program	Tester/User	- logic error

Error Handling

- Error handler should
  - Report errors accurately and clearly
  - Recover from an error quickly
  - Not slow down compilation of valid code

Error Handling

- Panic mode 오류 무시(오류인지만 알려주고 계속 수행 / 현재 컴파일러)
- Error productions 오류가 많이 나는 것들을 문법에 집어넣는 방법
- Automatic local or global correction 오류 고쳐주기 / 안쓰임

Error Handling

- Panic mode is simplest, most popular method
- When an error is detected:
  - Discard tokens until one with a clear role is found
  - Continue from there
- Looking for ***synchronizing*** tokens
  - Typically the statement or expression terminators

Error Handling

- Consider the erroneous expression

  (1 + + 2) + 3

  discard

- 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)*** throw away all thins between '(' ')'

  throw away all the input to int and count as E

Error Handling

- Error productions
  - specify known common mistakes in the grammar
- Example:
  - Write 5 x instead of 5 * x
  - Add the production $E \rightarrow \ldots \mid E\ E$
- Disadvantage
  - Complicates the grammar

Error Handling

- Error correction
- Idea: find a correct "nearby" program

  PL/C always produce valid PL/1 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

Error Handling

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible

- Present
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling

Compilers

6.2 Abstract Syntax Trees

Abstract Syntax Trees

- A parser traces the derivation of a sequence of tokens
- But 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

Abstract Syntax Trees

- Consider the grammar

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

- And the string

  5 + (2 + 3)

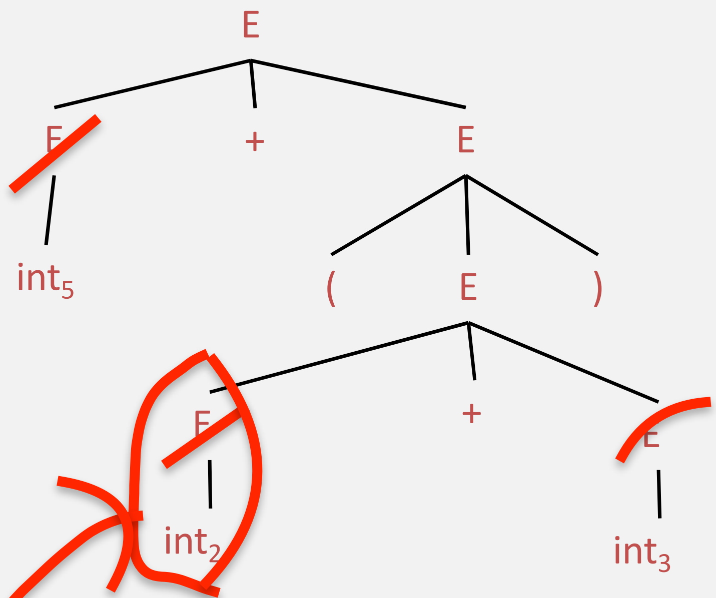
- After lexical analysis (a list of tokens)

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

- During parsing we build a parse tree …

Abstract Syntax Trees

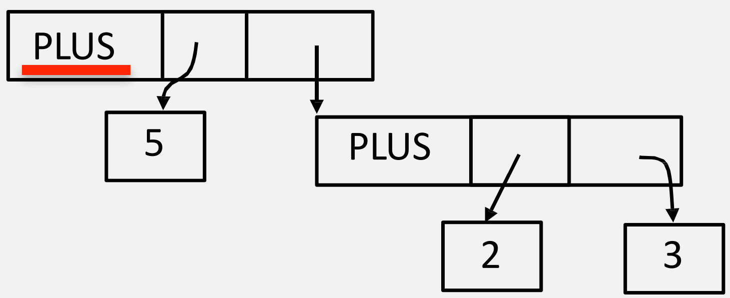
- A parse tree:
- Traces the operation of the parser
- Captures nesting structure
- But too much information
  - Parentheses
  - Single-successor nodes



Abstract Syntax Trees

- Also captures the nesting structure
- But ***abstracts*** from the concrete syntax

  => more compact and easier to use
- An important data structure in a compiler



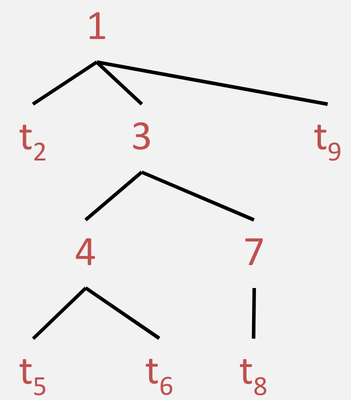
Compilers

6.3 Recursive Descent Parsing

Recursive Descent Parsing

- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream:

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



top down parsing

Recursive Descent Parsing

- Consider the grammar

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

- Token stream is: $(\; int_5 \;)$

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

Recursive Descent Parsing

E → **T** | T + E

T → int | int * T | (E)

E

T

~~int~~ backtracking

(int_5)

↑

backtracking을 계속해가면서 brust 방식으로 전부 다 하나씩 넣어봄

Recursive Descent Parsing

Choose the derivation that is a valid recursive descent parse for the string id + id in the given grammar. Moves that are followed by backtracking are given in red.

E → E' | E ' + E
E' → -E' | id | (E)

1.
E
E'
E' + E
id + E
id + E'
id + id

2.
E
E' + E
id + E
id + E'
id + id

3.
E
E'
-E'
Id
(E)
E' + E
-E' + E
id + E
id + E'
id + -E'
id + id

4.
E
E'
id
E' + E
id + E
id + E'
id + id

Compilers

6.4 Recursive Descent Algorithm

Recursive Descent Algorithm

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES
- Let the global next point to the next input token

Recursive Descent Algorithm

- Define boolean functions that check for a match of:
  - A given token terminal

    `bool term(TOKEN tok) { return *next++ == tok; }`
  - The nth production of S:

    bool S_n() { … }   true if match
  - Try all productions of S:

    `bool S() { … }`   true if any production of S match the input

Recursive Descent Algorithm

- 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());}
```

left-to-right order evaludation
"side-effect"
(왼쪽에서 false면 뒤에는 안한다)

form 맞출려고 (필요없는 코드)

restore

Recursive Descent Algorithm

- 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 Algorithm

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

- Easy to implement by hand

Recursive Descent Algorithm

$E \rightarrow T \mid T + E$

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

```
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)

int * int

RDA Limitation

- If a production for non-terminal X succeeds
  - Cannot backtrack to try a different production for X later

- General recursive-descent algorithms support such "full" backtracking
  - Can implement any grammar

RDA Limitation

- Presented recursive descent algorithm is not general
  - But is easy to implement by hand

- Sufficient for grammars where for any non-terminal at most one production can succeed

- The example grammar can be rewritten to work with the presented algorithm
  - By *left factoring*, the topic of a future lecture

T -> int | int * T | (E)

T -> int T’ | (E)

T -> * T | epsilon

Recursive Descent Algorithm

Which lines are incorrect in the recursive descent implementation of this grammar?

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

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

1. Line 3
2. Line 5
3. Line 6
4. Line 12
5. Line 13

```
1.  bool term(TOKEN tok) { return *next++ == tok; }
2.  bool E_1() { return E'(); }
3.  bool E_2() { return E'() && term(PLUS) && term(ID); }
4.  bool E()  {
5.          TOKEN *save = next;
6.          return (next = save, E_1()) && (next = save, E_2());
7.  }
8.  bool E'_1() { return term(MINUS) && E'(); }
9.  bool E'_2() { return term(ID); }
10. bool E'_3() { return term(OPEN) && E() && term(CLOSE); }
11. bool E'() {
12.         TOKEN *save = next;
13.         return   (next = save, T_1())
14.                || (next = save, T_2())
15.                || (next = save, T_3());
16. }
```

Compilers

6.5 Left Recursion

Left Recursion

- 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

Left Recursion

- Consider the left-recursive grammar

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

the very last thing that it produces
is the first thing that appears in the input

- S generates all strings starting with a β and followed by any number of α's

right-to-left derivation
but we need left-to-right

- Can rewrite using right-recursion

  $S \rightarrow \beta\, S'$

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

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

  $S \rightarrow \beta_1\,S' \mid \ldots \mid \beta_m\,S'$

  $S' \rightarrow \alpha_1\,S' \mid \ldots \mid \alpha_n\,S' \mid \varepsilon$

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

Left Recursion

Choose the grammar that correctly eliminates left recursion from the given grammar:

$E \to E + T \mid T$
$T \to id \mid (E)$

1.
$E \to E + id \mid E + (E) \mid id \mid (E)$

2.
$E \to TE'$
$E' \to + TE' \mid \varepsilon$
$T \to id \mid (E)$

3.
$E \to E' + T \mid T$
$E' \to id \mid (E)$
$T \to id \mid (E)$

4.
$E \to id + E \mid E + T \mid T$
$T \to id \mid (E)$

Left Recursion

- Recursive descent
  - Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Used in production compilers
  - E.g., gcc