Syntax

Context free grammar – parsing

2.3

Introduction

- What is the problem
  - Given a context free grammar, is a user (programmer) input (a program) acceptable by the grammar, i.e., does there exist a *parse tree* for the input with respect to the grammar?
  - Review of CFG
    - Symbols: terminal and non-terminals
    - Productions
    - Parse tree (derivation) of a user input (program)

Objective

- Develop a program, called *parser,* to recognize a user input string.
- A parser is a *language recognizer.*

Overview of parsers

- For any CFG we can create a parser that runs in $O(n^3)$ time. Examples:
  - Early's algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- $O(n^3)$ time is too slow

- There exists *special classes* of CFG that can be parsed *faster,* e.g., two important classes are
  - *LL grammar*:
    - Left to right, produce left-most derivation.
  - *LR grammar*:
    - Left to right, produce right-most derivation.
    - Subclasses:
      - SLR, LALR

- Property of LL/LR grammars
  - Every LL(1) grammar is also LR(1) grammar
  - Some LR(1) grammar is not LL(1) grammar
  - LR(1) is a “larger” grammar than LL(1)

- Parsers
  - *LL parsers* for LL grammar:
    - “top down” or “predictive” approach
  - *LR parsers* for LR grammar
    - “bottom up” or “shift reduce” approach
- Parameterized parsers: LL(n) or LR(n) where
  - n is the number of tokens to look ahead when “scanning” the tokens of an input program

LL parsing: top down

- Top dow: start from the start symbol, contruct a parse tree for an input

LL parsing: top down

- Top down
  - Initially, the tree (to construct) contains only the start symbol (non-terminal)
  - *Peek* the first token *A*, so, we *guess* the first production and use it to produce the children of the left most non-terminal in the *fringe of the tree*.
  - *Match* the terminals in the fringe with the input, in a depth first manner, until a non-terminal is met.
  - Repeat the second and third step

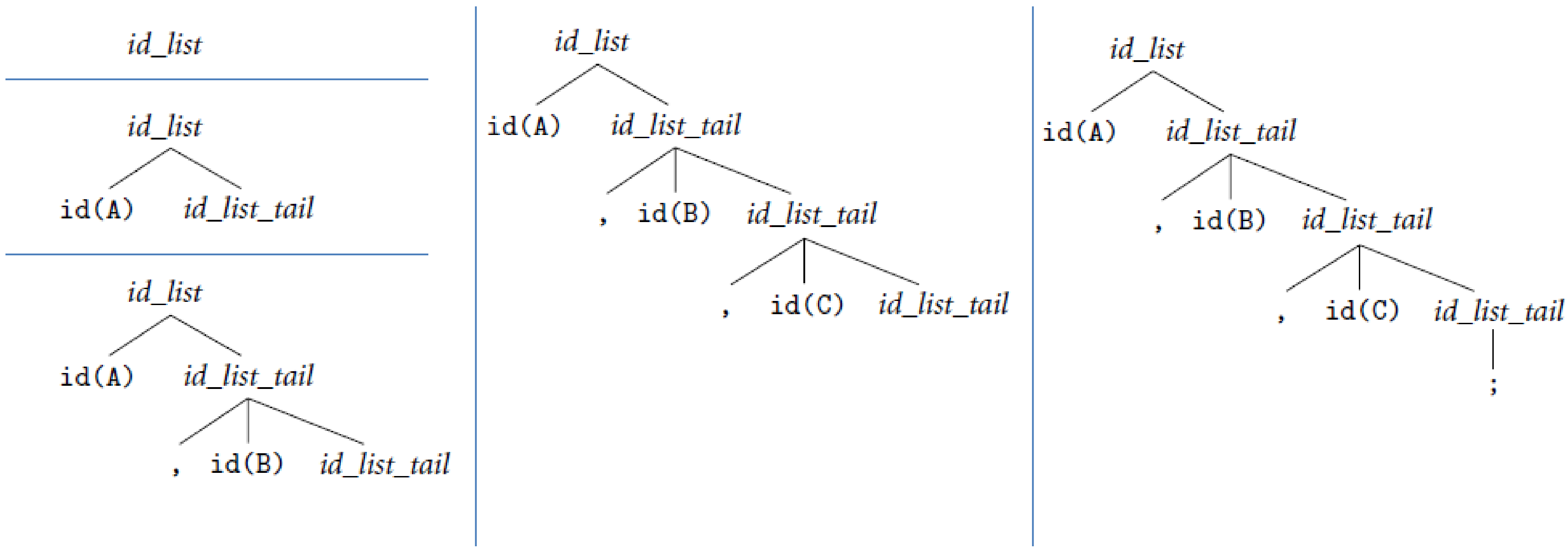
- Example
  - Grammar

```
<idList>      -> id <idListTail>
<idListTail> -> , id <idListTail>
<idListTail> -> ;
```

  - User (programmer) input

```
A, B, C;
```

$$
\begin{aligned}
&\mathit{id\_list} \longrightarrow \texttt{id}\ \mathit{id\_list\_tail} \\
&\mathit{id\_list\_tail} \longrightarrow \texttt{,}\ \texttt{id}\ \mathit{id\_list\_tail} \\
&\mathit{id\_list\_tail} \longrightarrow \texttt{;}
\end{aligned}
$$



- Problem with top down approach: it doesn’t work on this grammar:

```
<id_list>          -> <id_list_prefix>;
<id_list_prefix> -> <id_list_prefix> , id | id
```

With a current non-terminal <id_list_prefix>, When we see an id, we don’t know which production to use: the first of <id_list_prefix> or the second (after |)

LR Parsing

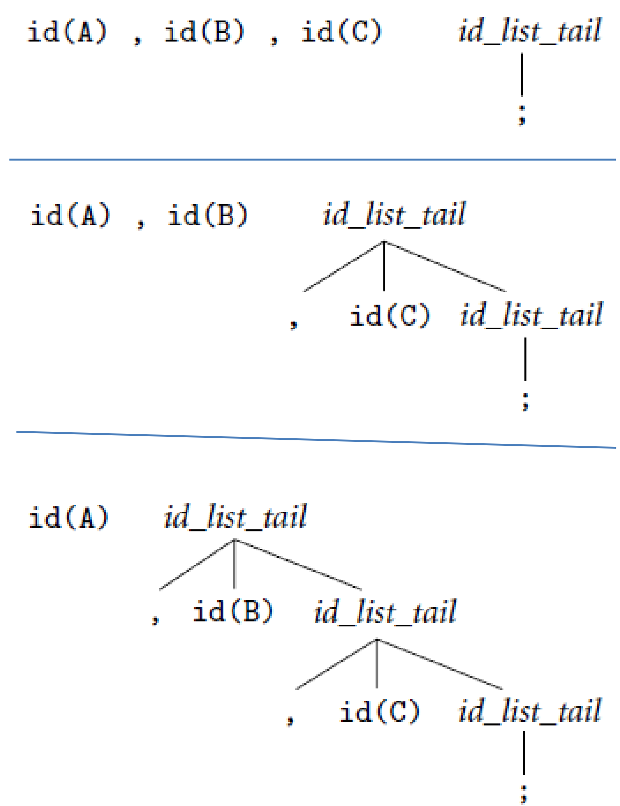
- Bottom up
  - Scan the tokens of the input until finding that the most recent tokens form the right hand side of a production. Create a new node (with the nonterminal at the left side of the production) and set these tokens as its children.
  - Replace those tokens by the new node, and repeat the step above
  - As a result, the parse tree built formed a right-most derivation tree.

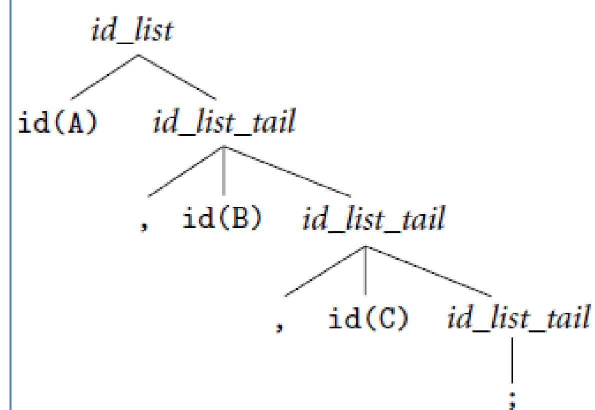
$id\_list \longrightarrow$ `id` id_list_tail

$id\_list\_tail \longrightarrow$ `,` `id` id_list_tail

$id\_list\_tail \longrightarrow$ `;`

```
id(A)
id(A) ,
id(A) , id(B)
id(A) , id(B) ,
id(A) , id(B) , id(C)
id(A) , id(B) , id(C) ;
```





Parsing Algorithms

- Problem: how to write a parser for this grammar? There are two approaches:
  - Recursive descent
  - Table driven

LL(1): example grammar

```
1. <program>   → <stmt_list> $$
2. <stmt_list> → <stmt> <stmt_list> | ε
3. <stmt>      → id := <expr> | read id  | write <expr>
4. <expr>      → <term> <term_tail>
5. <term_tail> → <add_op> <term> <term_tail> | ε
6. <term>      → <factor> <fact_tail>
7. <fact_tail> → <mult_op> <factor> <fact_tail> | ε
8. <factor>    → ( <expr> ) | id | number
9. <add_op>    → + | -
10. <mult_op>  → * | /
```

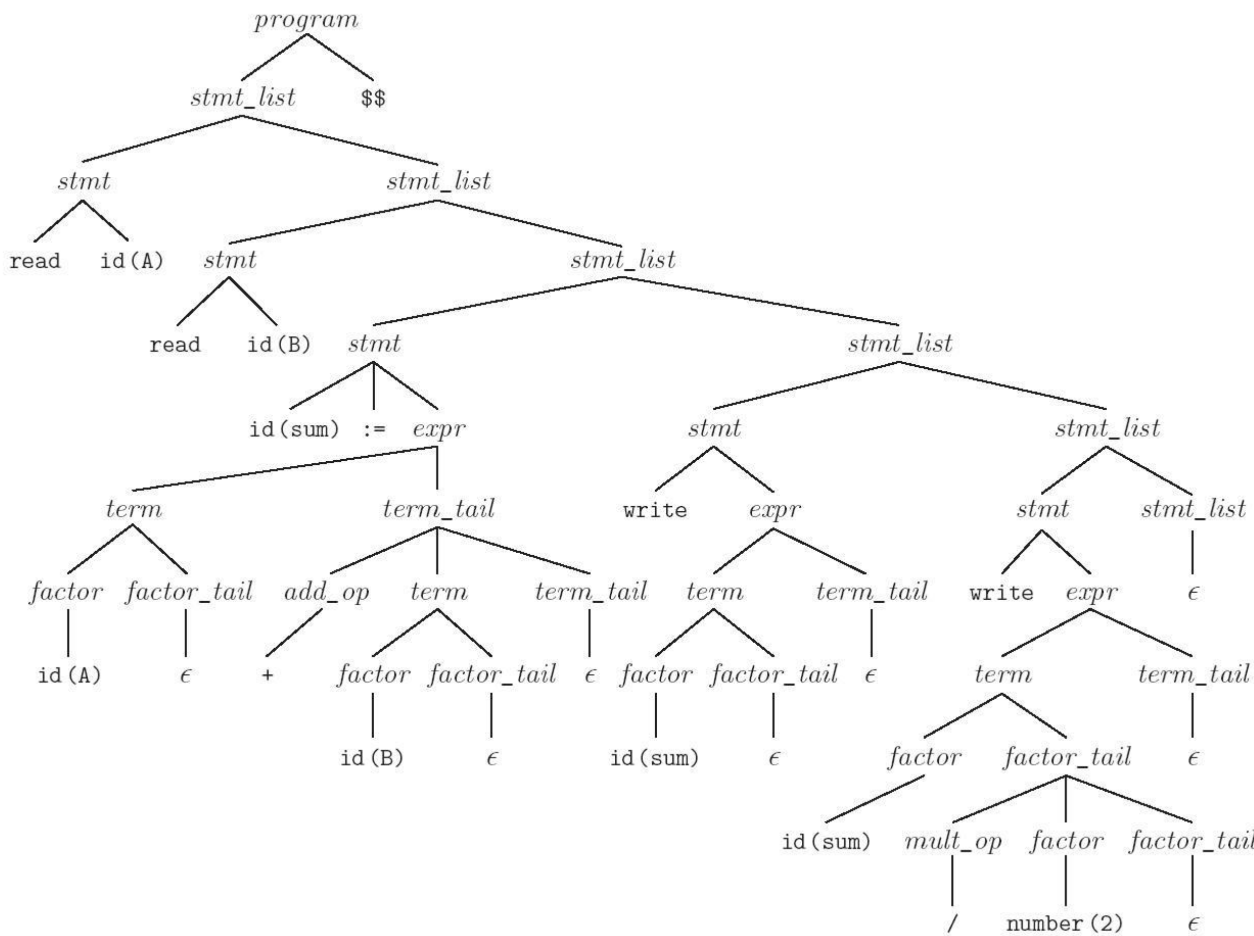
- Example program (compute average)

```
read A
read B
sum := A + B
write sum
write sum / 2
```

LL Parser

- Parsing the input – top down: start from the *start symbol* of the grammar (the top) and predict needed production on the basis of the current left-most non-terminal in the tree and the current input token

- Parsing tree of the example program



LL Parsers

- Table driven parsers
- Recursive descent

Table driven parser

- Data structures
  - Prediction table: tells what production is used given non-terminal symbol and a token
  - Parse stack for the parse tree
  - Input sequence of tokens

Data structure

- Predication table: given a non-terminal and a token, which production is to use (number in the table is defined in the next page)

Top-of-stack nonterminal	Current input token											
	`id`	`number`	`read`	`write`	`:=`	`(`	`)`	`+`	`-`	`*`	`/`	`$$`
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

1. $program \longrightarrow stmt\_list$ `$$` {`id`, `read`, `write`, `$$`}
2. $stmt\_list \longrightarrow stmt\ \ stmt\_list$ {`id`, `read`, `write`}
3. $stmt\_list \longrightarrow \epsilon$ {`$$`}
4. $stmt \longrightarrow$ `id` `:=` $expr$ {`id`}
5. $stmt \longrightarrow$ `read` `id` {`read`}
6. $stmt \longrightarrow$ `write` $expr$ {`write`}
7. $expr \longrightarrow term\ \ term\_tail$ {`(`, `id`, `number`}
8. $term\_tail \longrightarrow add\_op\ \ term\ \ term\_tail$ {`+`, `-`}
9. $term\_tail \longrightarrow \epsilon$ {`)`, `id`, `read`, `write`, `$$`}
10. $term \longrightarrow factor\ \ factor\_tail$ {`(`, `id`, `number`}
11. $factor\_tail \longrightarrow mult\_op\ \ factor\ \ factor\_tail$ {`*`, `/`}
12. $factor\_tail \longrightarrow \epsilon$ {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
13. $factor \longrightarrow$ `(` $expr$ `)` {`(`}
14. $factor \longrightarrow$ `id` {`id`}
15. $factor \longrightarrow$ `number` {`number`}
16. $add\_op \longrightarrow$ `+` {`+`}
17. $add\_op \longrightarrow$ `-` {`-`}
18. $mult\_op \longrightarrow$ `*` {`*`}
19. $mult\_op \longrightarrow$ `/` {`/`}

Table driven parser

- The idea of table driven parser is to match the *current* symbol of the parse tree (so far) with the (tokens in the) input program
  - Initially, the start symbol is in the parse stack
  - If the current (top) symbol N in parse stack is terminal
    - matches the current token in the input, throw away N
    - If not matching the current token, report error
  - Otherwise, select a production of N in terms of the current token, replace the current symbol by the body of the production (in an reversed order) in the parse stack.
  - Repeat process above until there is no symbol left

- Example
  - Program:

```
read A
```

  - As a sequence of tokens: `read id(A) $$`

Parse stack	Input stream	Comment
<program>	read A $$	`<program> -> <stmt_list> $$`
<stmt_list> $$	read A $$	`<stmt_list> → <stmt> <stmt_list>`
<stmt> <stmt_list> $$	read A $$	<stmt> -> read id
read id <stmt_list> $$	read A $$	match! Move forward.
id <stmt_list> $$	A $$	Match!
<stmt_list> $$	$$	<stmt_list> -> ε
$$	$$	done

Recursive descent parser

- The idea for writing the parser
  - Peek a token in the input program, guess (predict) a production rule, and reduce a non-terminal symbol
  - Discuss sample pieces of this parser in the textbook
    - For each non-terminal, there is one routine (procedure/function)
    - Routines are written by hand (so, hard to maintain)
    - Good when the parsing is simple
    - Help generate higher quality error message / maybe better performance

- Parser
  - Input:
    - a program
    - The context free grammar
  - Output: yes if the program follows the grammar; otherwise report parsing error

– Main function

`input_token`: current token of the input program. A global variable

```
main(){
   // get the first token
   input_token = scan();
   return (program());
}
```

- Each other functions are written based on the corresponding productions.
- E.g., `program()` function
  - It is obtained from the production

    `<program>    → <stmt_list> $$`
  - `program()` returns ok if the input program follows the production on `<program>`; and returns parse_error otherwise
    - Idea: if part of the input program is stmt_list and the rest is $$, then the input program follows the grammar of <program>, otherwise, it is parse_error.

- `program()`

```
program
Input: no input
Output: ok if the input program follows the
production on <program>; and parse_error otherwise.
{  // <program> -> <stmt_list> $$
   // $$ is the end of the program token
   case input_token of
     id, read, write, $$: // why these tokens?
        // if part of the input program is stmt_list
        if (stmt_list() == ok)
           //rest of the program must be "end of the program"
           return match($$);
        else return parse_error
     otherwise: return parse_error
}
```

- `Match(expected):`

```
match
Input: expectedToken: the expected token
Output: ok and (as side effect) gets the next token
into input_token if expectedToken is the same as
the input_token; otherwise, parse_error
{
   if (expectedToken == input_token)
        // get next token from the input program
       input_token = scan();
       return ok;
  else return parse_error
}
```

– E.g., `smt_list()` function

- From production

```
<stmt_list>    → <stmt> <stmt_list> | ε
```

- `smt_list()` returns ok if a sequence of tokens [after (including) the current token] of the input program follow the production; returns parse_error otherwise
- Idea: (part of) the input program follows the production <stmt_list> if it is empty or first part of the *current* input program follows <stmt> and the rest follows <stmt_list>

- stmt_list()

```
//<stmt_list>  → <stmt> <stmt_list> | ε

stmt_list
Input: no input
Output: ……
{
   case input_token of
     id, read, write:
       if (stmt() == ok)
         return (stmt_list())
       else return parse_error
     $$: return ok //
     otherwise return parse_error
}
```

- `stmt()`
  - `obtained from`

    `<stmt> → id := <expr> | read id | write <expr>`
  - `smt_list()` returns ok if a sequence of tokens [after (including) the current token] of the input program follows the production; returns parse_error otherwise

- `stmt()` function

```
<stmt> → id := <expr> | read id | write <expr>
```

```
stmt()
Input:
Output: ……
{  case input_token of
      id:
         match(id);
         if match(:=) return expr()
         else return parse_error
      read
         match(read);
         return match(id)
      write
         match(write);
         return expr()
      otherwise return parse_error
}
```

Find the predict tokens*(not required)

- Problem: find the tokens that predict the applicability of a production rule (i.e., how do we label the case satements?)
  - For <program>, they are {`id, read, write, $$`}

- Two functions allow us to find the predict tokens
  - FIRST(a): The terminals (and ε) that can be the first tokens of the non-terminal symbol a.
  - FOLLOW(A): The terminals that can follow the terminal or nonterminal symbol A
- The predict tokens, denoted by PREDICT(A → a)
  - FIRST(a) U FOLLOW(A) if EPS(a) (a can be empty)
  - FIRST(a) otherwise

- Compute FIRST(X) for all symbol X
  - Obvious facts
    - For any terminal c
      - EPS(c) :=false; FIRST(c) :=c
    - For any non-terminal X
      - ESP(X) := true, if exists X -> ε; false, otherwise
      - FIRST(X) := {}

- Revise FIRST and EPS, building on the facts and production
  1. For every production X->Y1, …, Yn
     - Add FIRST(Y1) to FIRST(X)
     - If EPS(Y1), add FIRST(Y2) to FIRST(X)
     - Repeat until the end or not EPS(Yi)
  2. Repeat step 1 until there is no changes to EPS(X) and FIRST(X) for any X

- Compute FOLLOW(X) for all symbol X
  - For all symbol X, initialize FOLLOW(X) to be {}
  - Revise FOLLOW(X) (for all X) according to the productions
    1. For all production A -> a B b
       a) Add FIRST(b) to FOLLOW(B)
    2. For all production A->aB or A->a B b (EPS(b)=true)
       a) Add FOLLOW(A) to FOLLOW(B)
    3. Repeat 1 and 2 until there is no change to FOLLOW(X) for any X.

- Compute predict(A->a):
  - FIRST(a) U FOLLOW(A) if EPS(a) (a can be empty)
  - FIRST(a) otherwise

- Example
  - A simiplified version of an earlier grammar

```
1. <program>   → <stmt_list> $$
2. <stmt_list> → <stmt> <stmt_list> | ε
3. <stmt>      → id:=<expr> | read id | write <expr>
4. <expr>      → <term> <term_tail>
```

  - Comput predict()
    - Obvious facts
      - Terminals: EPS(id) = false …
      - Non-terminals: EPS(<program>):=false, EPS(<stmt_list>:)=true … FIRST(<program>):={}, …
    - Revise EPS() and FIRST()

Theoretical Foundations

- Automata theory
  - Formal language: a set of strings over a finite alphabet. Specified by
    - Generator: regular expression, context free grammar
    - Recognizer: DFA, push-down automata
- Connection
  - DFA – scanner
  - Push-down automata – parser