CS 460

Programming Languages

Fall 2021

Dr. Watts

(23 October 2023)

Assignments

- Project 1
  - Submission script closed
  - New results sent this morning
- Exercise 2
  - Script running so that your groups can improve their testing techniques
- Exercise 3
  - Posted – let me know if you see typos
  - Part 1 due next week
- Project 2
  - Coming soon

What is Syntactic Analysis?

- Also known as Parsing
- Determining if the order of the tokens generated for the lexemes of the input are in a legal order according to some grammar
- Creation of a parse tree
  - Explicit or implicit
- Error recovery
  - When an error is detected, the parser must get back to a normal state and continue analysis of the input
- Basis for translation

VS Semantic Analysis

- The meaning of the expressions, statements and program units.
- Static semantics
  - At compile time
- Dynamic semantics
  - At run time
- Attribute grammars
- Denotational semantics

Describing Syntax

- Language is a set of strings of characters from some alphabet.
- The strings of a language are called sentences or statements.
- Smallest units of the statements are words or lexemes.
- Syntax rules of a language describe what words are in the language and how they should be ordered.
- Natural languages (such as English) have a complex and extensive set of syntactcal rules.
- Programming languages have a relatively simple set of syntactical rules.

BNF

- Set of terminal symbols (T)
- Set of non-terminal symbols (N)
- Start symbol (S Є N)
- Set of production rules (P)
  - <non-terminal> → string of terminal and <non-terminal> symbols
  - <non-terminal> ::= string of terminal and <non-terminal> symbols

Simple BNF Grammar

- T = {ONE, TWO, THREE, FOUR}
- NT = {<number>, <single>, <double>, <triple>}
- S = <number>
- P = {
  - <number> ::= <single> <double> <triple>
  - <single> ::= ONE | TWO | THREE | FOUR
  - <double> ::= <single> <single>
  - <triple> ::= <single> <double>
  - <triple> ::= <double> <single>

  }

Taunt Generator Grammar

(with thanks to Monty Python and the Insulting Frenchman)

```
< taunt > ::= < sentence > | < taunt > < sentence > | < noun >!
< sentence > ::= < past-rel > < noun-phrase >
                 | < present-rel > < noun-phrase >
                 | < past-rel > < article > < noun >
< noun-phrase > ::= < article > < modified-noun >
< modified-noun > ::= < noun > | < modifier > < noun >
< modifier > ::= < adjective > | < adverb > < adjective >
< present-rel > ::= your < present-person > < present-verb >
< past-rel > ::= your < past-person > < past-verb >
< present-person > ::= steed | king | first-born
< past-person > ::= mother | father | grandmother | grandfather | godfather
< noun > ::= hamster | coconut | duck | newt | peril | chicken | vole | parrot
                 | mouse | twit | elderberry
< present-verb > ::= is | “masquerades as”
< past-verb > ::= was | personified | “smelt of”
< article > ::= a
< adjective > ::= silly | wicked | sordid | naughty | repulsive | malodorous
                 | ill-tempered
< adverb > ::= conspicuously | categorically | positively | cruelly
                 | incontrovertibly
```

Each team

- Use the grammar to generate 3 taunts
  - <= 5 words
  - > 5 and <= 10 words
  - > 10 words
- Submit the taunts in a file called Team#.txt

Context Free Grammars and Backus-Naur Form

- Context Free Grammar (CFG) – order of syntactical elements is important – meaning is not.
- Meaning is determined by context semantics.
- Backus-Naur Form (BNF)
  - ALGOL 58
  - John Backus – 1959
  - Peter Naur – 1960
- BNF is a natural notation for describing syntax

BNF Grammar (Example 1)

- T = {=, A, B, C, +, *, (,)}
- N = {<assign>, <id>, <expr>}
- S = <assign>
- P = {

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

}

Derivation of A = B + (C * A)

<assign>

=> <id> = <expr>

=> A = <expr>

=> A = <id> + <expr>

=> A = B + <expr>

=> A = B + (<expr>)

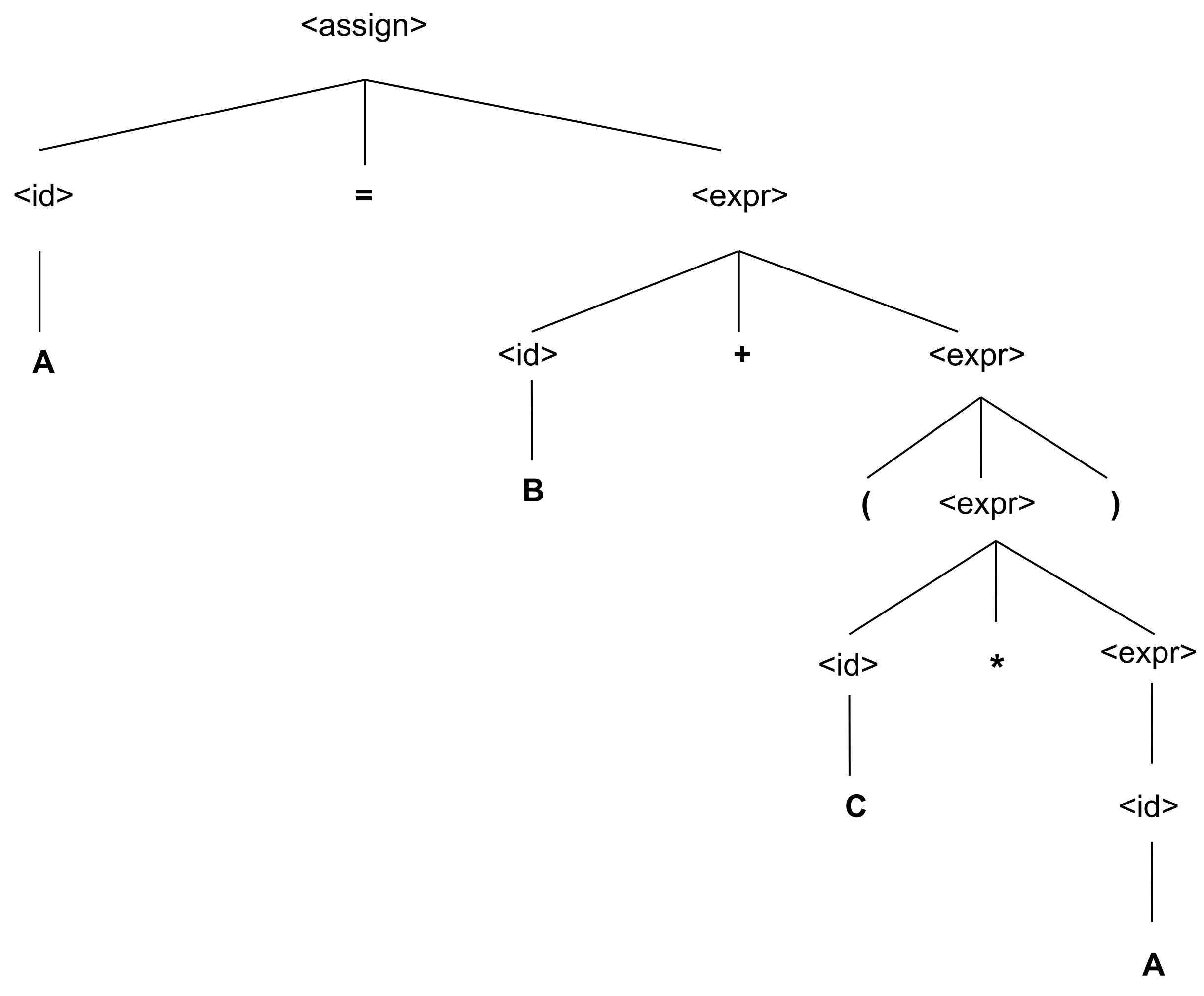
=> A = B + (<id> * <expr>)

=> A = B + (C * <expr>)

=> A = B + (C * <id>)

=> A = B + (C * A)

Parse tree for A = B + (C * A)



BNF Grammar (Example 2)

- T = {=, A, B, C, +, *, (,)}
- N = {<assign>, <id>, <expr>}
- S = <assign>
- P = {

  <assign> → <id> = <expr>
  <id> → A | B | C
  <expr> → <expr> + <expr>
  | <expr> * <expr>
  | (<expr>)
  | <id>

  }
- Derivation for A = B + C * A ?

One Possible Derivation

<assign>

=> <id> = <expr>

=> A = <expr> + <expr>

=> A = <id> + <expr>

=> A = B + <expr>

=> A = B + <expr> * <expr>

=> A = B + <id> * <expr>

=> A = B + C * <expr>

=> A = B + C * <id>

=> A = B + C * A

The Other Possible Derivation

<assign>

==> <id> = <expr>

==> A = <expr>

==> A = <expr> * <expr>

==> A = <expr> + <expr> * <expr>

==> A = <id> + <expr> * <expr>

==> A = B + <expr> * <expr>

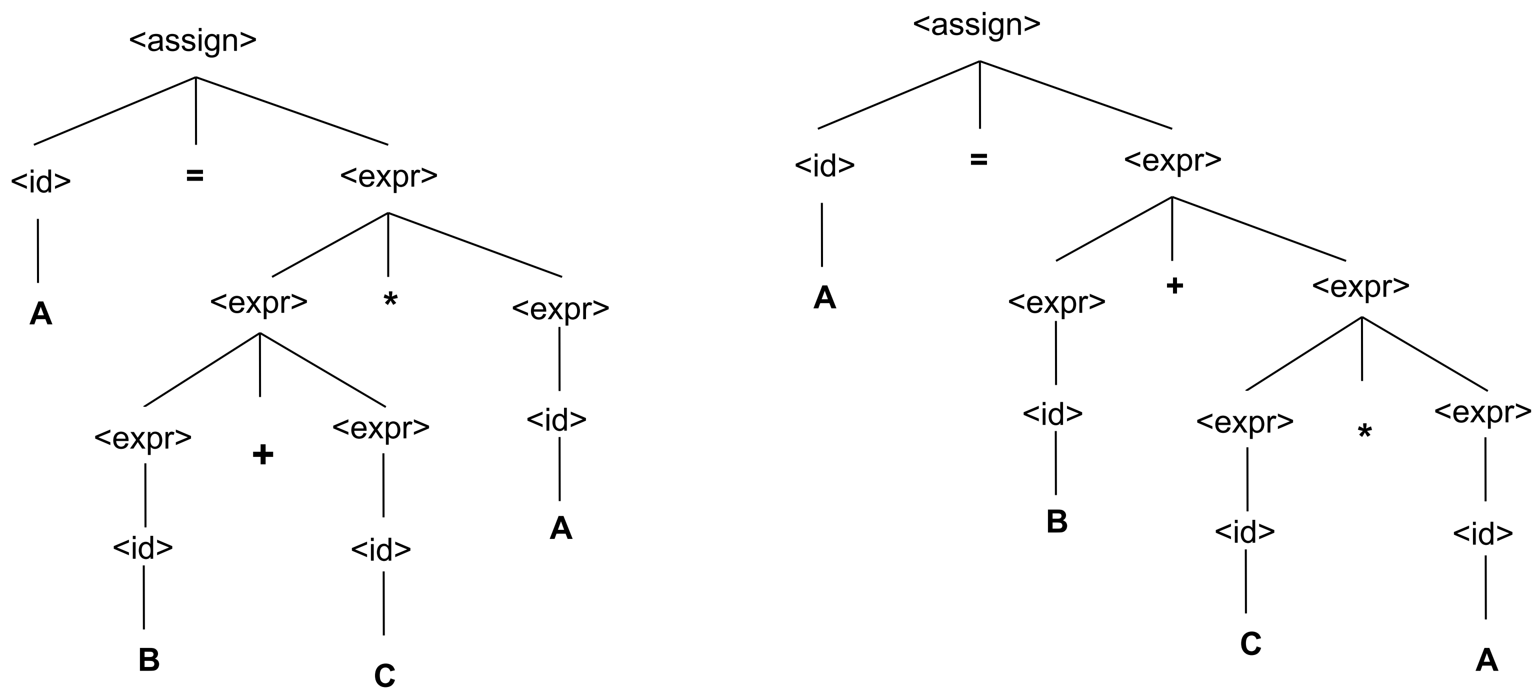
==> A = B + <id> * <expr>

==> A = B + C * <expr>

==> A = B + C * <id>

==> A = B + C * A

Example 2
Parse Trees for A = B + C * A



- Ambiguous

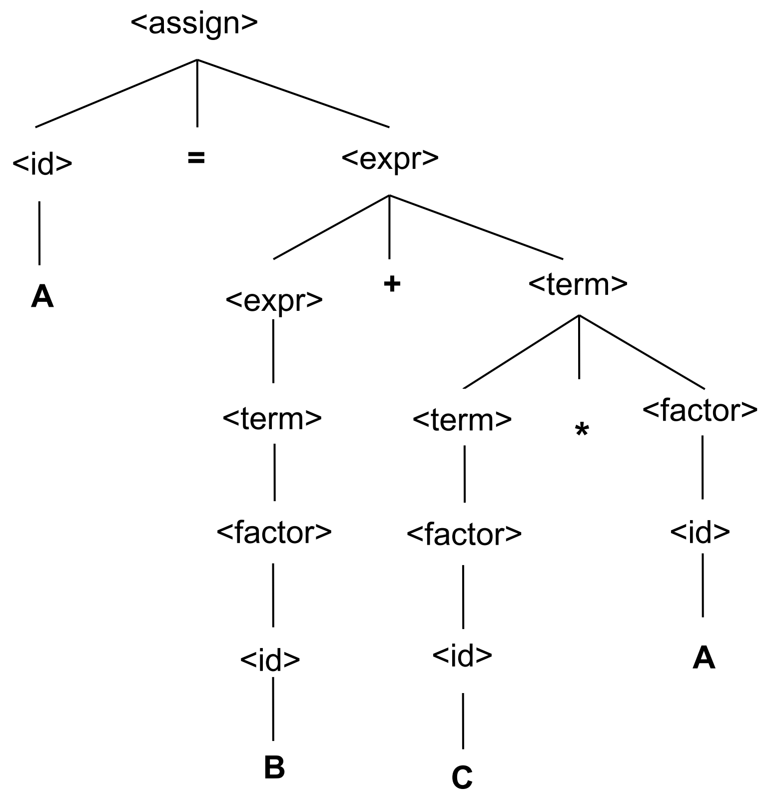
BNF Grammar (Example 3)

- T = {=,A, B, C, +, *, (,)}
- N = {<assign>, <id>, <expr>, <term>, <factor>}
- S = <assign>
- P = {

  <assign> → <id> = <expr>
  <id> → A | B | C
  <expr> → <expr> + <term> | <term>
  <term> → <term> * <factor> | <factor>
  <factor> → (<expr>) | <id>

  }
- Parse tree for A = B + C * A ?

Example 3
Parse Tree for A = B + C * A



- Operator precedence

Parsing

- Top Down
  - Recursive Descent
  - LL Parsing – Left to right scan of the input; Leftmost derivation
- Bottom Up
  - Shift Reduce
  - LR Parsing – Left to right scan of the input; Rightmost derivation

LL(1) Grammar for a Small Programming Language

T = {begin, end, ;, =, A, B, C, +, *}
N = {<program>, <stmt_list>, <stmt>, <var>, <stmt_tail>, <expression>, <expr_tail>}
Start = <program>

P =
{
1. <program> → **begin** <stmt_list> **end**
2. <stmt_list> → <stmt> <stmt_tail>
3. <stmt_tail> →**;** <stmt_list>
4. <stmt_tail> → λ
5. <stmt> → <var> **=** <expression>
6. <var> → **A**
7. <var> → **B**
8. <var> → **C**
9. <expression> → <var><expr_tail>
10. <expr_tail> → + <var><expr_tail>
11. <expr_tail> → * <var><expr_tail>
12. <expr_tail> → λ

}

Example 1:
```
begin
    A = B + C;
    C = A * B
end
```

Example 2:
```
begin
    A = B + A * C;
    C = A * B;
end
```

Parsing of Example 1

From main parsing routine, call <program> function

  From <program>, match **begin**; call <stmt_list> function

    From <stmt_list>, see **A** call <stmt> function

      From <stmt>, see **A** call <var> function

        From <var>, match **A**; return

      From <stmt>, match =; see **B** call <expression> function

        From <expression>, see **B** call <var> function

          From <var>, match **B**; return

        From <expression>, see **+** call <expr_tail> function

          From <expr_tail>, match **+**; see **C** call <var> function

            From <var>, match **C**; return

          From <expr_tail>, see **;** call <expr_tail>

            From <expr_tail>, see **;** return

          From <expr_tail>, return

        From <expression>, return

…

  From <program>, match end; return

From main parsing routine, print error count and terminate

Recursive Descent Parser for a Small Programming Language

1) <program> → **begin** <stmt_list> **end**

```
program ()
{
   if (current_token == begin)
   { // rule 1
        get_next_token;
        call stmt_list;
        if (current_token == end)
               get_next_token;
        else
               call error_routine;
   }
   else
        call error_routine;
   return;
}
```

Recursive Descent Parser for a Small Programming Language

2) <stmt_list> → <stmt> <stmt_tail>

```
stmt_list ()
{ // rule 2
      call stmt;
      call stmt_tail;
      return;
}
```

Recursive Descent Parser for a Small Programming Language

3) <stmt_tail> → ; <stmt_list>
4) <stmt_tail> → λ

```
stmt_tail ()
{
   if (current_token == ;)
   { // rule 3
       get_next_token;
       call stmt_list;
   }
   else if (current_token == end)
   { // rule 4
   }
   else
       call error_routine;
   return;
}
```

Recursive Descent Parser for a Small Programming Language

5) <stmt> → <var> **=** <expression>

```
stmt ()
{ // rule 5
   call var;
   if (current_token == =)
   {
       get_next_token;
       call expression;
   }
   else
       call error_routine;
   return;
}
```

Recursive Descent Parser for a Small Programming Language

6) <var> → **A**
7) <var> → **B**
8) <var> → **C**

```
var ()
{ // rule 5
    if (current_token == A)
    { // rule 6
          get_next_token;
    }
    else if (current_token == B)
    { // rule 7
          get_next_token;
    }
    else if (current_token == C)
    { // rule 8
          get_next_token;
    }
    else
          call error_routine;
    return;
}
```

Recursive Descent Parser for a Small Programming Language

9) <expression> → <var><expr_tail>

```
expression ()
{ // rule 9
      call var;
      call expr_tail;
      return;
}
```

Recursive Descent Parser for a Small Programming Language

10) <expr_tail> → + <var><expr_tail>
11) <expr_tail> → * <var><expr_tail>
12) <expr_tail> → λ

```
expr_tail ()
{
    if (current_token == +)
    { // rule 10
          get_next_token;
          call var;
          call expr_tail;
    }
    else if (current_token == *)
    { // rule 11
          get_next_token;
          call var;
          call expr_tail;
    }
    else if (current_token == ; or current_token == end)
    { // rule 12
    }
    else
          call error_routine;
    return;
}
```

Context Free Grammar Definition

- Given a Context Free Grammar of the form:
  - Terminals = $\{T_1, T_2, T_3, \ldots\}$
  - Non-terminals = $\{<nt_1>, <nt_2>, <nt_3>, \ldots\}$
  - A Start symbol from the set of non-terminals
  - A set of Production rules of the form

    $<nt_i> \rightarrow$ string of T and <nt> symbols

First and Follow Sets

- Firsts
  - A terminal symbol T_i is a member of the First Set of non-terminal symbol <nt_j> if T_i can become the first terminal symbol in a complete expansion of <nt_j>.
- Follows
  - A terminal symbol T_i is a member of the Follow Set of non-terminal symbol <nt_j> if T_i can become the first terminal symbol immediately following a complete expansion of <nt_j>.