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 syntactical 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 ∈ 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

- \( T = \{\text{ONE, TWO, THREE, FOUR}\} \)
- \( NT = \{\langle\text{number}\rangle, \langle\text{single}\rangle, \langle\text{double}\rangle, \langle\text{triple}\rangle\} \)
- \( S = \langle\text{number}\rangle \)
- \( P = \{ \)
  - \( \langle\text{number}\rangle ::= \langle\text{single}\rangle \langle\text{double}\rangle \langle\text{triple}\rangle \)
  - \( \langle\text{single}\rangle ::= \text{ONE} | \text{TWO} | \text{THREE} | \text{FOUR} \)
  - \( \langle\text{double}\rangle ::= \langle\text{single}\rangle \langle\text{single}\rangle \)
  - \( \langle\text{triple}\rangle ::= \langle\text{single}\rangle \langle\text{double}\rangle \)
  - \( \langle\text{triple}\rangle ::= \langle\text{double}\rangle \langle\text{single}\rangle \)
- \( \} \)
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
              | incontroversibly
Each team

- Use the grammar to generate 3 taunts
  - \( \leq 5 \) words
  - \( > 5 \) and \( \leq 10 \) 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)

- $T = \{ =, A, B, C, +, *, (, ) \}$
- $N = \{ <assign>, <id>, <expr> \}$
- $S = <assign>$
- $P = \{$
  
  $<assign> \rightarrow <id> = <expr>$
  
  $<id> \rightarrow A | B | C$
  
  $<expr> \rightarrow <id> + <expr>$
  
  $| <id> * <expr>$
  
  $| ( <expr> )$
  
  $| <id>$
  
$\}$
Derivation of \( A = B + (C \times A) \)

\[
\begin{align*}
  &<\text{assign}> \\
  &\Rightarrow <\text{id}> = <\text{expr}> \\
  &\Rightarrow A = <\text{expr}> \\
  &\Rightarrow A = <\text{id}> + <\text{expr}> \\
  &\Rightarrow A = B + <\text{expr}> \\
  &\Rightarrow A = B + ( <\text{expr}> ) \\
  &\Rightarrow A = B + ( <\text{id}> \times <\text{expr}> ) \\
  &\Rightarrow A = B + ( C \times <\text{expr}> ) \\
  &\Rightarrow A = B + ( C \times <\text{id}> ) \\
  &\Rightarrow A = B + ( C \times A )
\end{align*}
\]
Parse tree for $A = B + (C * A)$
BNF Grammar (Example 2)

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

- Derivation for \( A = B + C \ast A \)?
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 $A = B + C \times A$

- Ambiguous
BNF Grammar (Example 3)

- $T = \{=, A, B, C, +, *, (, )\}$
- $N = \{<\text{assign}>, <\text{id}>, <\text{expr}>, <\text{term}>, <\text{factor}>\}$
- $S = <\text{assign}>$
- $P = \{$
  $<\text{assign}> \rightarrow <\text{id}> = <\text{expr}>
  <\text{id}> \rightarrow A \mid B \mid C
  <\text{expr}> \rightarrow <\text{expr}> + <\text{term}> \mid <\text{term}>
  <\text{term}> \rightarrow <\text{term}> * <\text{factor}> \mid <\text{factor}>
  <\text{factor}> \rightarrow ( <\text{expr}> ) \mid <\text{id}>
\}$
- Parse tree for $A = B + C * A$ ?
Example 3
Parse Tree for $A = B + C \times A$

- 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 = \{\text{begin, end, ;, =, } A, B, C, +, *\}
N = \{\langle\text{program}\rangle, \langle\text{stmt_list}\rangle, \langle\text{stmt}\rangle, \langle\text{var}\rangle,
\langle\text{stmt_tail}\rangle, \langle\text{expression}\rangle, \langle\text{expr_tail}\rangle\}
Start = \langle\text{program}\rangle

P =
\{ 
1. \langle\text{program}\rangle \rightarrow \text{begin} \langle\text{stmt_list}\rangle \text{end}
2. \langle\text{stmt_list}\rangle \rightarrow \langle\text{stmt}\rangle \langle\text{stmt_tail}\rangle
3. \langle\text{stmt_tail}\rangle \rightarrow ; \langle\text{stmt_list}\rangle
4. \langle\text{stmt_tail}\rangle \rightarrow \lambda
5. \langle\text{stmt}\rangle \rightarrow \langle\text{var}\rangle = \langle\text{expression}\rangle
6. \langle\text{var}\rangle \rightarrow A
7. \langle\text{var}\rangle \rightarrow B
8. \langle\text{var}\rangle \rightarrow C
9. \langle\text{expression}\rangle \rightarrow \langle\text{var}\rangle \langle\text{expr_tail}\rangle
10. \langle\text{expr_tail}\rangle \rightarrow + \langle\text{var}\rangle \langle\text{expr_tail}\rangle
11. \langle\text{expr_tail}\rangle \rightarrow \ast \langle\text{var}\rangle \langle\text{expr_tail}\rangle
12. \langle\text{expr_tail}\rangle \rightarrow \lambda
\}

Example 1:
begin
A = B + C;
C = A \ast B
end

Example 2:
begin
A = B + A \ast C;
C = A \ast 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
  ...
Recursive Descent Parser for a Small Programming Language

1) \( <\text{program}> \rightarrow \text{begin} <\text{stmt_list}> \text{end} \)

```c
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) \(<\text{stmt\_list}> \rightarrow <\text{stmt}> <\text{stmt\_tail}>\)

\begin{verbatim}
stmt_list ()
{
    // rule 2
    call stmt;
    call stmt_tail;
    return;
}
\end{verbatim}
Recursive Descent Parser for a Small Programming Language

3) $\langle \text{stmt\_tail} \rangle \rightarrow ; \langle \text{stmt\_list} \rangle$
4) $\langle \text{stmt\_tail} \rangle \rightarrow \lambda$

```
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) $<\text{stmt}> \rightarrow <\text{var}> = <\text{expression}>$

\begin{verbatim}
stmt ()
{
    // rule 5
    call var;
    if (current_token == =)
    {
        get_next_token;
        call expression;
    }
    else
    {
        call error_routine;
        return;
    }
}
\end{verbatim}
Recursive Descent Parser for a Small Programming Language

6) \( \text{<var>} \rightarrow \text{A} \)
7) \( \text{<var>} \rightarrow \text{B} \)
8) \( \text{<var>} \rightarrow \text{C} \)

```c
var ()
{
    // rule 5
    if (current_token == \text{A})
    {
        // rule 6
        get_next_token;
    }
    else if (current_token == \text{B})
    {
        // rule 7
        get_next_token;
    }
    else if (current_token == \text{C})
    {
        // rule 8
        get_next_token;
    }
    else
    {
        call error_routine;
    }
    return;
}
```
Recursive Descent Parser for a Small Programming Language

9) \(<expression> \rightarrow <var><expr_tail>\)

```c
expression ()
{
    // rule 9
    call var;
    call expr_tail;
    return;
}
```
Recursive Descent Parser for a Small Programming Language

10) \( <\text{expr}_{\text{tail}}> \rightarrow + <\text{var}><\text{expr}_{\text{tail}}> \)

11) \( <\text{expr}_{\text{tail}}> \rightarrow * <\text{var}><\text{expr}_{\text{tail}}> \)

12) \( <\text{expr}_{\text{tail}}> \rightarrow \lambda \)

\( \text{expr}_{\text{tail}} () \)

\{ 
  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
    \(<nt_i> \rightarrow \text{string of } T \text{ and } <nt> \text{ symbols}\)
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 $<nt_j>$. 