Programming Language Syntax

Chapter 2 Part 1

Programming Language Syntax

- Grammar of the language
- Rules for forming the statements, expressions, etc
- Languages have different grammars:
  - "a + b * c" vs. "(+ a (* b c)) "
  - if(x < 10) y = 20;
    vs.
  - if (x < 10) then y := 20

Chapter 2 Part 1 2

C vs. Pascal Grammars

C

```
sum = 0;
prod = 1;
for( int j = 1; j<= 10 ; j++)
{
        sum += j;
        prod *= j;
}
```

Pascal

```
sum := 0;
prod := 1;
for j := 1 to 10 do
begin
  sum   := sum + j;
   prod  := prod * j
end
```


Backus-Naur Form (BNF)

- BNF is a formal way to describe the syntax of a programming language.
- BNF was developed by John Backus and was first used to describe ALGOL 60
- BNF uses context-free grammars (CFG) to describe language syntax. You will see many more of these in Comp 310.
- CFGs were invented by Noam Chomsky (1959) to describe natural languages.

Syntax Rule– assign statement

<assign_stmt> → <var> = <expr>

- Rule describing the form of an assignment statement
- Nonterminals are in brackets.
- Nonterminals need to be expanded
- Metasymbol: →
- Terminal: =

Syntax Rule– assign statement

Scott textbook notation uses *italics*

assign_stmt → *var* = *expr*

- Nonterminals are in *italics*.
- Nonterminals need to be expanded
- Metasymbol: →
- Terminal: =

Small grammar for assignment

<assign_stmt> → <var> = <expr>

<var> → a| b| c

<expr> → <var> | <expr> + <var>

Show that a = b + c is a legal assignment statement.

<assign_stmt>
=> <var> = <expr>
=> a = <expr>
=> a = <expr> + <var>
=> a = <var> + <var>
=> a = b + <var>
=> a = b + c

At each step apply exactly one substitution rule from grammar!

Revise to include other operators variables and numbers

<assign_stmt> → <var> = <expr>

<var> → a| b| c

<expr> → <var> | <expr> + <var>

Syntax rules for a list

<id> → a|b|c|d
<list> → <id> | <list>, <id>

Expanding <list> to derive c, a, b
<list> => <list>, <id>
=> <list>, <id>, <id>
=> <id>, <id>, <id>
=> c, <id> , <id>
=> c, a, <id>
=> c, a, b

This computation is called a left-most derivation.

Syntax Rules – signed integer

<digit> → 0|1|2|3|4|5|6|7|8|9
<integer> → <digit> |
<digit> <integer>

<signed_int> → <sign><integer> |
<integer>
<sign> → + | –

Show that 546 and –7892 are signed integers.

Grammar for a Small Language

<prog> → begin <stmt_list> end
<stmt_list> → <stmt>
| <stmt>; <stmt-list>
<stmt>→ <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
|<var> | <integer>

Leftmost derivation of the program

begin A = B + C end

<prog>
→ begin <stmt_list> end
→ begin <stmt> end
→ begin <var> = <expression> end
→ begin A= <expression> end
→ begin A= <var> + <var> end
→ begin A= B+ <var> end
→ begin A= B+ C end

Successful derivation !

Exercises

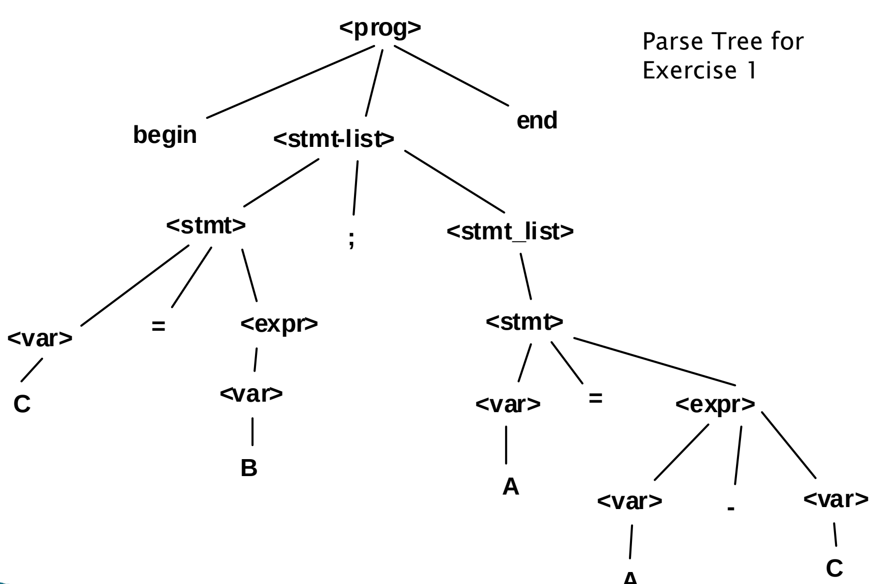
1. Find a leftmost derivation for the following program. Show all steps.

```
begin
    C = B;
    A = A - C
end
```

2. Show that you cannot derive the program if A= A – C is changed to A = A – C;

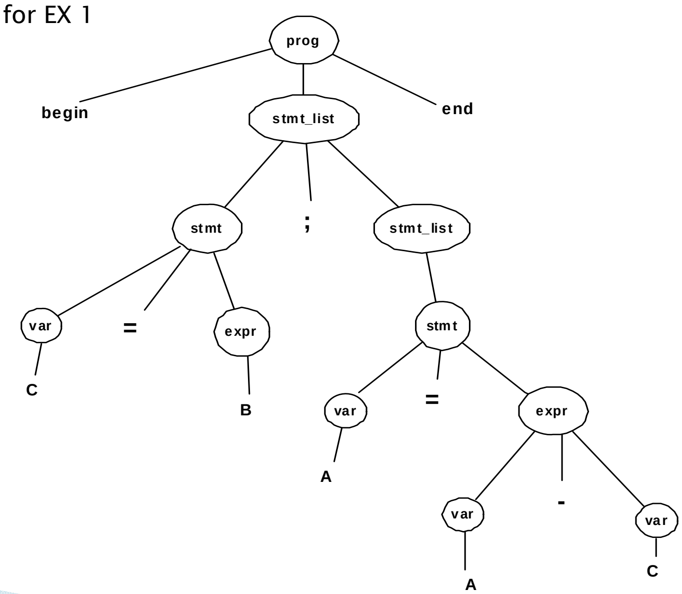
Parse Trees

- Used to represent grammar derivations hierarchically as a tree
- The nonterminals are the nodes
- The start nonterminal is the root of the tree
- The terminals are the leaves of the tree



Parse Tree for Exercise 1

Parse tree for EX 1



G1: expression grammar

<expr> → <expr> + <expr>
| <expr> – <expr>
| <expr> * <expr>
| <expr> / <expr>
| < (<expr>)
| <var> |<int>

<var> → x | y | z | a | b | c
<digit> → 0|1|2|3|4|5|6|7|8|9|
<int> → <digit> | <digit> <int>

What are the terminals? nonterminals? metasymbols?

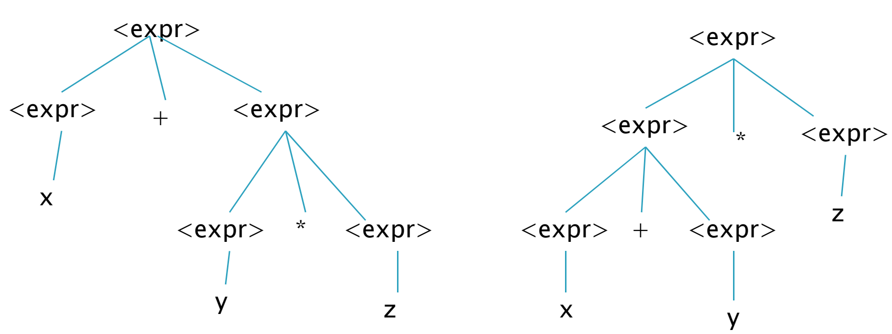
Exercises

1. Create a parse tree for the expression
   x + y * z using grammar G1

2. Create a parse tree for the expression
   x * (y + 5)/ 100 using Grammar G1

3. Show that there is more than one parse tree
   for x + y * z using grammar G1?

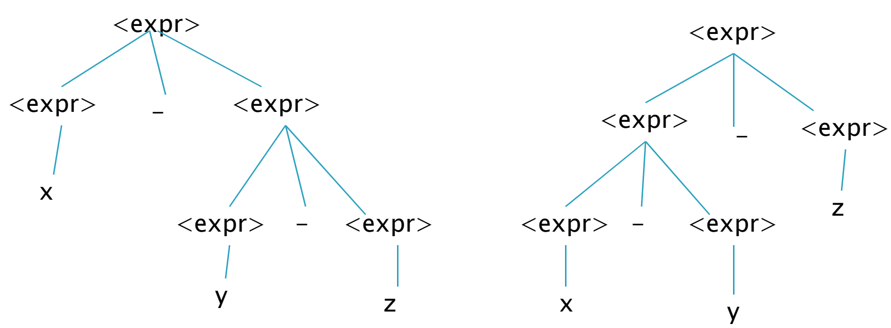
Two parse trees for x + y * z



What is the value of x + y * z
if x = 2, y =4, z = 3 ?

G1 is an ambiguous grammar!

Two parse trees for x – y – z



What is the value of x – y – z
when x = 10, y = 4 , z = 3 ?

Syntax Rule– if statement

<if_stmt> →
 if (<bool_expr>) then <stmt> |
 if (<bool_expr>) then <stmt> else <stmt>

- Metasymbols: → |
- Terminals: if ( ) then else

To complete this, we need BNF definition of <stmt> and <bool_expr>.

<stmt>

<stmt> → <if_stmt> | <assign_stmt> | …

<assign_stmt> → <var> = <expr>

<bool_expr>

- Our grammar should allow for the following kinds of boolean expressions

- x <= 10
- x > 0 or y < 10
- (b > 0) and (a/b >= y)
- not (x == (y + 7))

<bool_expr> Definition

<rop> → < | <= | > | >= | != | ==

<rel_expr> → <expr> <rop> <expr>

Use <expr> from G1

<bop> → and | or

<bool_expr> → <rel_expr>
| <bool_expr> <bop> <bool_expr>
| not <bool_expr>
| (<bool_expr>)

Exercise

- Find parse trees for the following boolean expressions:

- a + b < 10
- (a > 0) or (b < 20)
- not (a == b + 1)

if-then-else statement

<if_stmt> →

if (<bool_expr>) then <stmt>

| if(<bool_expr>) then <stmt> else <stmt>

Consider :

```
if ( x <  10) then
if ( x == 5) then
y = 2
else
y = 3
```


if – then– else statement is ambiguous.

Two parses of previous if statement

```
if ( x <  10) then
      if ( x == 5) then
              y = 2
else
      y = 3
```

Input x = 4 and y = 0
Output y = *initial value*

Parse A

```
if ( x <  10) then
    if ( x == 5) then
       y = 2
    else
       y = 3
```

Input x = 4 and y = 0
Output y = 3

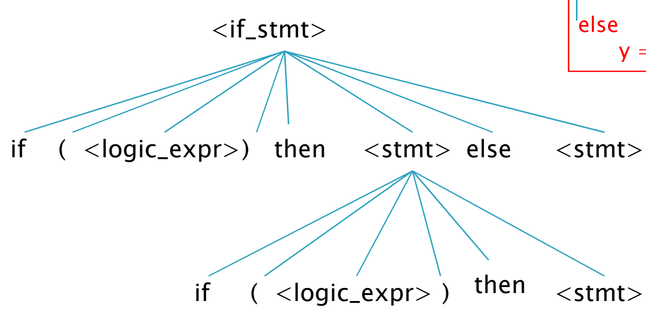
Parse B

Exercises

4. Draw a parse tree corresponding to parse A on previous
5. Draw a parse tree corresponding parse B on the previous slide.

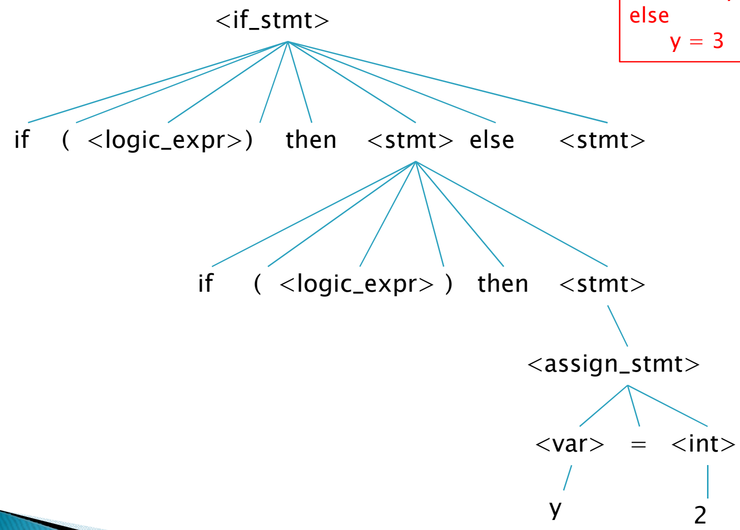
First Parse

```
if ( x <  10) then
   if ( x == 5) then
        y = 2
else
    y = 3
```



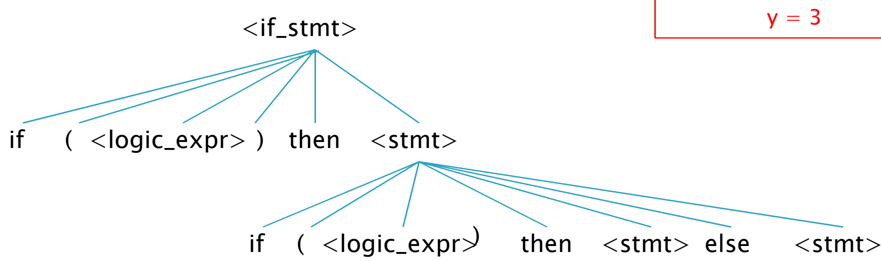
First Parse Continued

```
if ( x ==  10) then
    if ( x == 5) then
          y = 2
else
     y = 3
```



Second Parse

```
if ( x <  10)
    if ( x == 5)
          y = 2
    else
           y = 3
```



Dangling Else Problem

- Refers to the ambiguity of the if – then – else statement.
- Problem: Matching the “elses” with the “ifs”
- Handle ambiguity with special compiler rules
  - Match ‘else’ with closest previous unmatched ‘if’

Grammar for a small language (SML)

<prog> → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt>; <stmt-list>
<stmt> → <assign_stmt> | <if_stmt>
<assign_stmt>→ <var> = <expression>
<if_stmt> → if (<bool_expr>) then <stmt>
| if(<bool_expr>) then <stmt> else <stmt>

<expr> → <expr> + <expr>| <expr> - <expr>
| <expr> * <expr>| <expr> / <expr>
| (<expr>)| <var> |<int>
<rop> → < | <= | > | >= | != | ==
<rel_expr> → <expr> <rop> <expr>
<bop> → and | or
<bool_expr> → <rel_expr>
|<bool_expr> <bop> <bool_expr> | not <bool_expr> | (<bool_expr>)

<var> → x | y | z | a | b | c
<digit> → 0|1|2|3|4|5|6|7|8|9|
<int> → <digit> | <digit> <int>

Ambiguous Grammars

- A grammar is ambiguous if there are at least two different parse trees for some sentence.

- G1 is ambiguous (*next slide*)
- G2 is not ambiguous (*next slide + 1*)
- Since a parse tree corresponds to the meaning of a program statement, ambiguity is not a good feature of a programming language grammar.

G1: ambiguous expression grammar

```
<expr> → <expr> + <expr>
       | <expr> - <expr>
       | <expr> * <expr>
       | <expr> / <expr>
       | < ( <expr>)
       | <var>  |<int>

<var> → x  | y | z | a | b | c
<digit> → 0|1|2|3|4|5|6|7|8|9|
<int>  → <digit>  |  <digit> <int>
```

G2: Unambiguous Expression Grammar with Precedence and Assoc. Rules

E → E + T | E –T |T
T → T *F | T / F | F
F→ (E) | <var> | <int>

Notation
E is <expression>
T is <term>
F is <factor>

Exercises (Use G2)
6. Find a parse tree for x + y * z
7. Find a parse tree for x – y – z
8. Find a parse tree for (x + y) *z

G3: Unambiguous Boolean Expression Grammar with Precedence and Assoc. Rules

B→ B or R | R
R→ R and V | V
V→ (B) | not B| <rel_expr>

Notation
B is <bool_expr>
R is <term>
V is <factor>

Exercise (Use G3)
Find a parse tree for
R1 and R2 or R3 and not R4 where
R1,R2, R3, R4 are relational expressions

Extended BNF (EBNF)

More metasymbols for BNF

[…] optional
{… }* may be repeated 0 or more times
{ … }$^+$ must be repeated at least once
(… | …) choice

If metasymbol is also a terminal, underline it when used as a terminal.

Example using EBNF G4

- <expr> → <term> { (+ | -) <term>}*
- <term> → <factor> { (* | /) <factor> }*
- <factor> → <LP><expr> <RP>| <var> | <int>
- <LP> → (
- <RP> →)
- <var> → <letter> { (<letter> | <digit>) }*
- <letter> → a|b|c| ….|z
- <digit> → 0|1|2|3|4| …|9
- <int> → {<digit>}$^+$

- Derive: a + 4 * b - 23

Expand <expr> to get *a+ 4 * b -23*. (Leftmost expansion)

```
<expr> => <term> + <term> - <term>
       =>  <factor> + <term> - <term>
       =>   <expr>  + <term> - <term>
       =>   <var>  + <term> - <term>
       =>     a     +  <term> - <term>
       =>     a     +  <factor> * <factor> - <term>
       =>     a     +  <expr> * <factor> - <term>
       =>     a     +  <int> * <factor> - <term>
       =>     a     +  4* <factor> - <term>
       =>     a     +  4* <expr> - <term>
       =>     a     +  4* <var> - <term>
       =>     a     +  4* b- <term>
       =>     a     +  4* b- <factor>
       =>     a     +  4* b- <expr>
       =>     a     +  4* b- <int>
       =>     a     +  4* b- <digit><digit>
       =>     a     +  4* b-  2<digit>
       =>     a     +  4* b- 23
```


Exercises using EBNF G3

9. Draw the parse tree for derivation of
   *a + 4 * b - 23*

10. Derive *sum = sum + (x - 1)*