CS2403 Programming Languages

Syntax and Semantic

Chung-Ta King
Department of Computer Science
National Tsing Hua University

(Slides are adopted from *Concepts of Programming Languages*, R.W. Sebesta)

國立清華大學
National Tsing Hua University

Extended by api.

Roadmap

Ch. 1	Classification of languages What make a "good" language?
Ch. 2	Evolution of languages
Ch. 3	How to define languages?
Ch. 4	How to compile and translate programs?
Ch. 5	Variables in languages
Ch. 7	Statements and program constructs in languages
Ch. 15	Functional and logic languages

國立清華大學
National Tsing Hua University

Outline

- Introduction (Sec. 3.1)
- The General Problem of Describing Syntax (Sec. 3.2)
- Formal Methods of Describing Syntax (Sec. 3.3)
- Attribute Grammars (Sec. 3.4)
- Describing the Meanings of Programs: Dynamic Semantics (Sec. 3.5)

Description of a Language

- Syntax: the form or structure of the expressions, statements, and program units
- Semantics: the meaning of the expressions, statements, and program units
  - What programs do, their behavior and meaning
- So, when we say one's English grammar is wrong, we actually mean __________ error?

What Kind of Errors They Have?

กิน ข้าว คน (syntax error)

คน กิน ข้าว

ข้าว กิน คน (sematic error)

Syntax and Semantics in PL

Ex:

```
int x = 10;
int y = x + u;
```

Valid Syntax :

```
int x = 100;
int y = x + u;
```

Invalid semantics : u undefined

Terminology :

```cpp
#include <iostream>

using namespace std;

 int main( )
{
   char b = 'a';
   int X = 50;
   float Z=6.8;
   char s[20] = "int203";
   int main = 1;
    cout  << "  b  = " <<  b  << '\n';
    cout  << "  X  = " <<  X  << '\n';
    cout  << " Z  = " <<  Z  << '\n';
    cout  << " s  = " << s  << '\n';
    cout  << " main  = " << main  << '\n';
    return 0 ;
}
```

```
 b  = a
 X  = 50
Z  = 6.8
s  = int203
main  = 1
```

Terminology

```
#include <iostream>

using namespace std;

 int main( )
{
  char b = 'a';
  int X = 50;
  float Z=6.8;
  char s[20] = "int203";
  int main = 1;
   cout  << "  b  = " <<  b  << '\n';
   cout  << "  X  = " <<  X  << '\n';
   cout  << " Z  = " <<  Z  << '\n';
   cout  << " s  = " << s  << '\n';
   cout  << " main  = " << main  << '\n';
   return 0 ;
}
```

Reserved word : int,float,char,return
Operator (**Special character)** : (,) , = , ;
Character constant : ‘a’
Floating point constant : 6.8
Integer constant : 50
String constant : “int203”
Identifier : b,X,Z s , u ,cont
Keyword : main
Statement/sentence : char b = ‘a’;

Keywords are ‘predefined identifier that can be used as identifiers again

國立清華大學
National Tsing Hua University

Outline

- Introduction (Sec. 3.1)
- The General Problem of Describing Syntax (Sec. 3.2)
- Formal Methods of Describing Syntax (Sec. 3.3)
  - Issues in Grammar Definitions: Ambiguity, Precedence, Associativity, …
- Attribute Grammars (Sec. 3.4)
- Describing the Meanings of Programs: Dynamic Semantics (Sec. 3.5)

Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

```
X → ε
X → a | aY
Y → b
```

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form **A → γ**

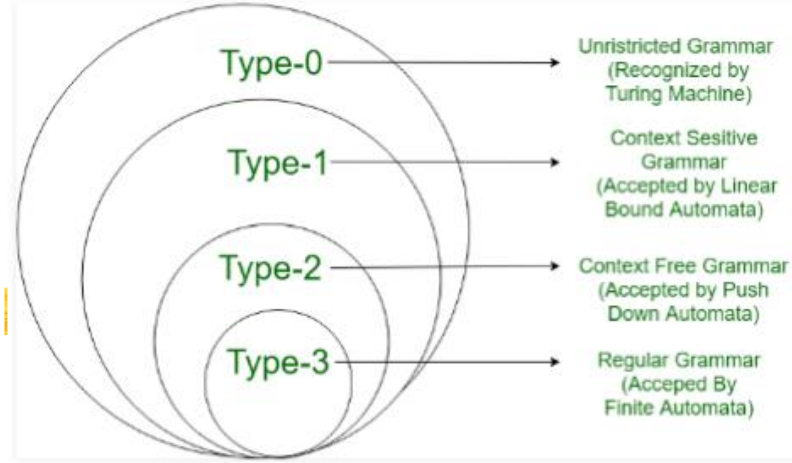
where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example

```
S → X a
X → a
X → aX
X → abc
X → ε
```



Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example

```
AB → AbBc
A → bcA
B → b
```

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α → β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

Example

```
S → ACaB
Bc → acB
CB → DB
aD → Db
```

ตัวอย่าง 1.1

ไวยากรณ์ G0 = ({S,A,B}, {a,b},S, { S→ ABB, AB→b, A→ε, bB→cA, B→a })

ไวยากรณ์ G1 = ({S,A,B}, {a,b},S, { S→ ABA,BA→bB , bB→ε, A→a ,B→b })

ไวยากรณ์ G2 = ({S,A,B}, {a,b},S, { S→ AB,A→bB , A→a ,B→ ε, B→b })

ไวยากรณ์ G3 = ({S,A,B}, {a,b},S, { S→ aA, A→bB , A→a , A→ε, B→b })

ไวยากรณ์ G4 = ({S,A,B}, {a,b},S, { S→ Aa, A→Bb , A→a , A→ε, B→b })

กำหนดให้ไวยากรณ์ G1_1 ประกอบด้วย

ไวยากรณ์ G1_1 : ({S, B, C},{ a, b , c }, S, P) Formal language.

โดย P = { S → aSBC , S → abC, bB → bb, bC → bc, CB → BC, cC → cc }

จงพิสูจน์ว่า $a^2b^2c^2$ เป็นประโยคที่อยู่ในภาษา L(G1_1) หรือไม่

S ⇒ aSBC แทนด้วย $S \rightarrow aSBC$

⇒ aabCBC แทนด้วย $S \rightarrow abC$

⇒ aabBCC แทนด้วย $CB \rightarrow BC$

⇒ aabbCC แทนด้วย $bB \rightarrow bb$

⇒ aabbcC แทนด้วย $bC \rightarrow bc$

⇒ aabbcc แทนด้วย $cC \rightarrow cc$

Formal Description of Syntax

Most widely known methods for describing syntax:

- Formal form (Context-Free Grammars)
  - Developed by Noam Chomsky in the mid-1950s
  - Define a class of languages: context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe ALGOL 58
  - Equivalent to context-free grammars

Lexeme & token

"if",a,c,f -> terminal(lexeme)

S -> aBcAf

("if " ,reserved word) ->token

S -> "if" B "{"
A
"}"

("{" ,operator) ->token

("}" ,operator) ->token

in PL , terminal symbols, "a" or "if" , ard called as *" lexeme"*.
and its category of lexemes is called "*token"*

國立清華大學
National Tsing Hua University

BNF (it 's equivalent to "CFG")

- A BNF grammar consists of four parts:
  - The set of *tokens* and *lexemes* (*terminals*)
  - The set of *non-terminals*, e.g., <sentence>, <verb>
  - The *start* symbol, e.g., <sentence>
  - The set of *production rules*, e.g.,

<sentence> → <noun> <verb> <preposition> <noun>
<noun> → *place*
<verb> → "is" | "belongs" <preposition> → "in" | "to"

The *start* symbol is the particular non-terminal that forms the starting point of generating a sentence of the language ie. <sentence> .

An Example Grammar

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

<program> is the start symbol

a, b, c, const, +, -, ;, = are the terminals (*lexemes)*

Derivation

- A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols), e.g.,

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Derivation Leftmost & rightmost

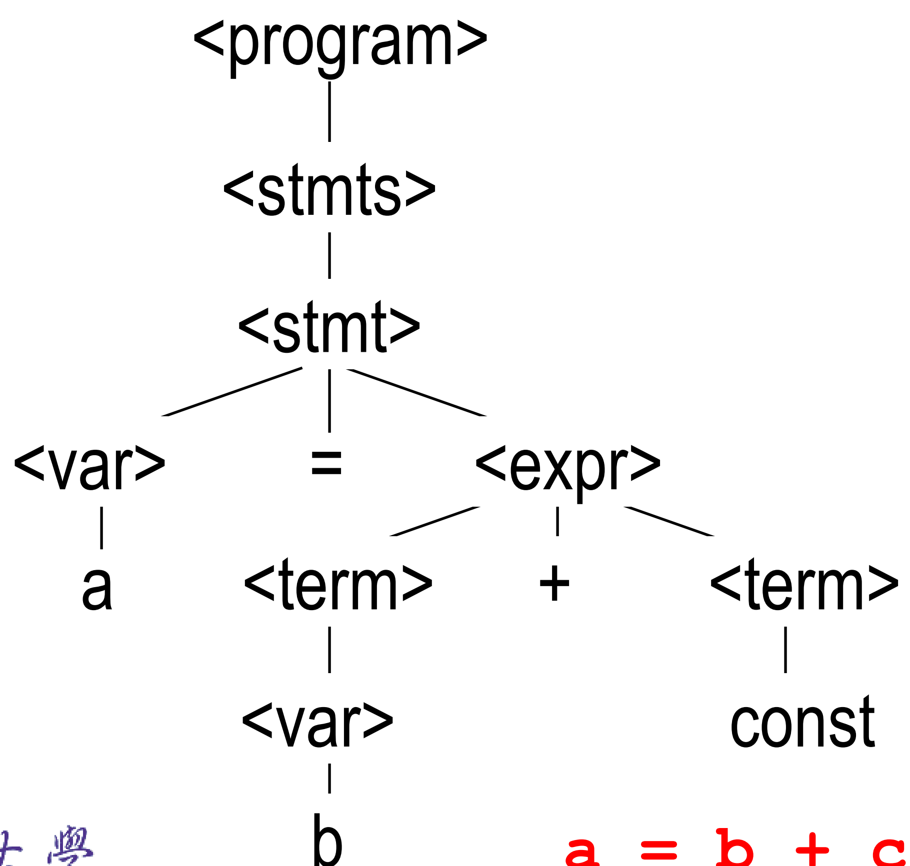
$S \to aABb$
$A \to aA \mid a$
$B \to bB \mid b$

Derivations for a string "aaabb".

- **Leftmost:** $S \Rightarrow a\underline{A}Bb \Rightarrow aa\underline{A}Bb \Rightarrow aaa\underline{B}b \Rightarrow aaabb$
- **Rightmost:** $S \Rightarrow aA\underline{B}b \Rightarrow a\underline{A}bb \Rightarrow aa\underline{A}bb \Rightarrow aaabb$

Parse Tree

- A hierarchical representation of a derivation



a = b + const

Ambiguity in Grammars

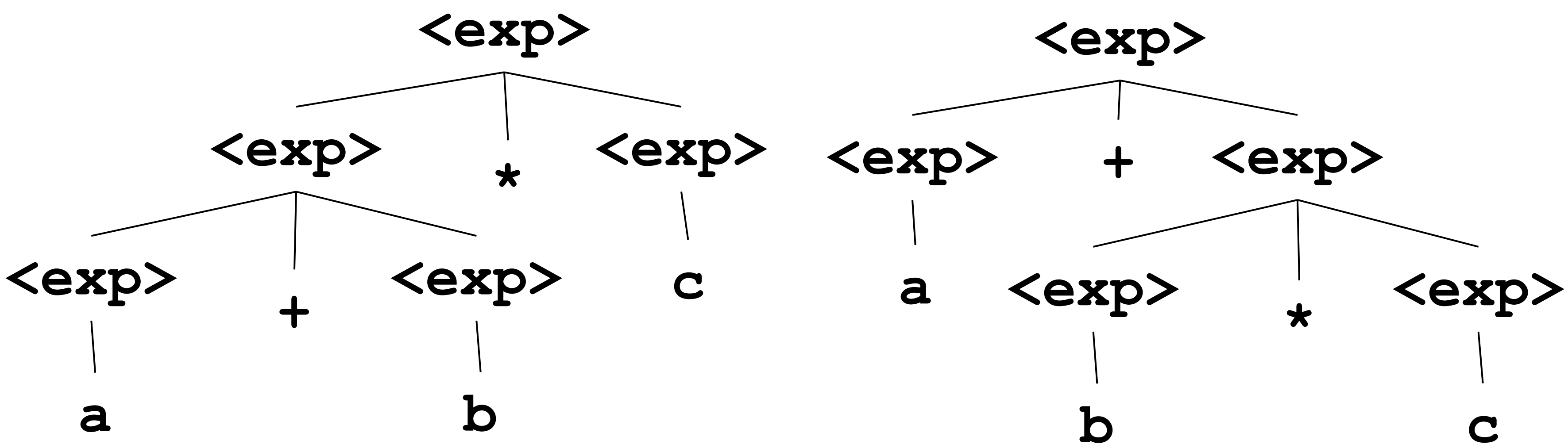
- If a sentential form can be generated by two or more distinct parse trees, the grammar is said to be *ambiguous*, because it has two or more different meanings
- Problem with ambiguity:
  - Consider the following grammar and the sentence **a+b*c**

```
<exp> → <exp> + <exp> |
        <exp> * <exp> |
        (<exp>) |
        a | b | c
```

An Ambiguous Grammar

```
<exp> → <exp> + <exp> |
        <exp> * <exp> |
        (<exp>) |
        a | b | c
```

- Two different parse trees for **a+b*c**



Means (a+b)*c   Means a+(b*c)

Three "Equivalent" Grammars

G1 : <*subexp*> → **a** | **b** | **c** | <*subexp*> **-** <*subexp*>

G2 : <*subexp*> → <*var*> **-** <*subexp*> | <*var*>
<*var*> → **a** | **b** | **c**

G3 : <*subexp*> → <*subexp*> **-** <*var*> | <*var*>
<*var*> → **a** | **b** | **c**

These grammars all define the same language: the language of strings that contain one or more **a**s, **b**s or **c**s separated by minus signs, e.g., **`a-b-c`**. But...

Extended BNF

- Optional parts are placed in brackets [ ]
  <proc_call> → ident [(<expr_list>)]
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  `<term> → <term> (+|-) const`
- Repetitions (0 or more) are placed inside braces { }
  `<ident> → letter {letter|digit}`

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```