COP 3402 Systems Software

Syntax Analysis
(Parser)

Outline

1. Definition of Parsing
2. Context Free Grammars
3. Ambiguous/Unambiguous Grammars

Lexical and Syntax Analysis

	Lexical Analysis	Syntax Analysis
	regular languages	context-free languages
described by	regular expressions	context-free grammars
recognized by	deterministic finite automata	push-down automata
implement	lexer/scanner	parser
		tokens extracted by lexer are the non-terminal symbols of the CFG
tool	flex (lex)	bison (yacc)

Parsing Problem

Given a grammar for a language and a string, a parser

- determines if the string is syntactically correct, that is, if can be generated using the rules of the grammar, and
- constructs the parse tree if the string is correct.

A syntactically correct string is called sentence of the language.

For a compiler, a sentence is correct program.

We will focus on a recursive decent parser of a LL(1) grammar for PL/0.

Nested Structures

Languages containing nested structures such as correctly parenthesized programs cannot be described by regular expressions.

Even the much simpler language

```
{ ab, aabb, aaabbb, aaaabbbb, ... }
```

is not regular.

We need to add recursion to describe nested structures.

Recursion

The following rules suffice to define regular sets/languages:

Concatenation	`r s`
Alternation	`r	s`
Kleene closure (repetition)	`r*`

Regular sets are generated by regular expressions and recognized by finite state automata (FSA).

We obtain context free languages by adding recursion as a new rule.

Context Free Grammars

Context Free Languages are

- generated by Context Free Grammars (CFG) and
- recognized by Pushdown Automata.

The strings of a language are called sentences or statements.

Parsing is the task of determining whether the syntax (structure) of a given string conforms to the rules a given grammar, or equivalently, whether it is a sentence in the language

Context Free Grammars

Consider the following three rules (grammar):

```
<sentence>  -> <subject> <predicate>
<subject>   -> John | Mary
<predicate> -> eats | talks
```

where `->` means "is defined as"

The four possible sentences that can be derived from `<sentence>` are

```
John eats  John talks
Mary eats  Mary talks
```

Context Free Grammars

We refer to the rules used in the example grammar as

syntax rules, productions, syntactic equations, or rewriting rules.

`<subject>` and `<predicate>` are syntactic classes or categories.

Using a shorthand notation, the syntax rules of the example grammar are

```
S -> A B
A -> a | b
B -> c | d
```

The language (the set of sentences) of this grammar is

```
L = { ac, ad, bc, bd }
```

Context Free Grammars

A context free language is defined by a 4-tuple `(T,N,R,S)` as:

(1) The set of **terminal symbols T**

* They cannot be substituted by any other symbol

* This set is also called the **vocabulary**

```
S -> A B
A -> a | b
B -> c | d
```

terminal symbols (tokens)

Context Free Grammars

A context free language is defined by a 4-tuple `(T,N,R,S)` as:

(2) The set of non-terminal symbols (`N`)

* They denote syntactic classes

* They can be substituted by other symbols

non-terminal symbols

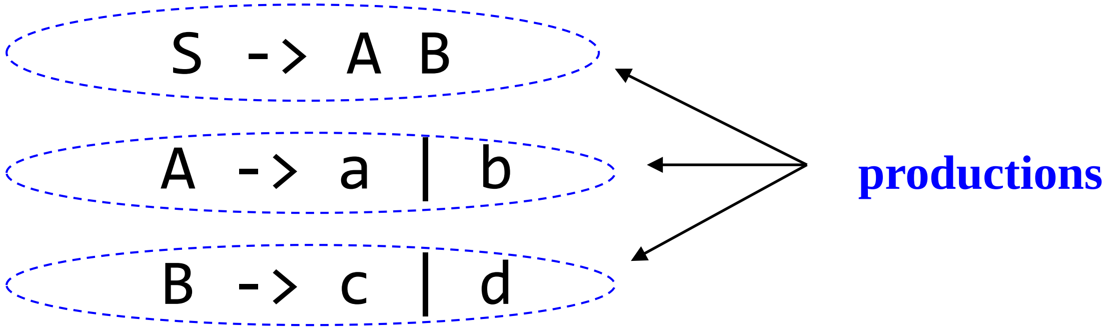
```
S  -> A B
A  -> a | b
B  -> c | d
```

Context Free Grammars

A context free language is defined by a 4-tuple `(T,N,R,S)` as:

(3) The set of syntactic equations or productions (the grammar).

* An equation or rewriting rule is specified for each non-terminal symbol (R)


```
S -> A B
A -> a | b
B -> c | d
```

Context Free Grammars

A context free language is defined by a 4-tuple `(T,N,R,S)` as:

(4) The start symbol (S)

```
S  -> A B
A  ->  a | b
B  ->  c | d
```

Context Free Grammars

A very well known meta-language used to specify context free grammars is Backus Naur Form (BNF)

It was developed by John Backus and Peter Naur in the late 50s to describe programming languages.

Noam Chomsky developed context free grammars in the early 50s.

Context Free Grammars

Example of a grammar for a simple assignment statement:

```
R1      <assgn> ::= <id> := <expr>
R2      <id>    ::= a | b | c
R3      <expr>  ::=   <id> + <expr>
R4                  | <id> * <expr>
R5                  | ( <expr> )
R6                  | <id>
```

Context Free Grammars

Grammar for a simple assignment statement:

```
R1  <assgn> ::= <id> := <expr>
R2  <id>    ::= a | b | c
R3  <expr>  ::=   <id> + <expr>
R4              | <id> * <expr>
R5              | ( <expr> )
R6              |  <id>
```

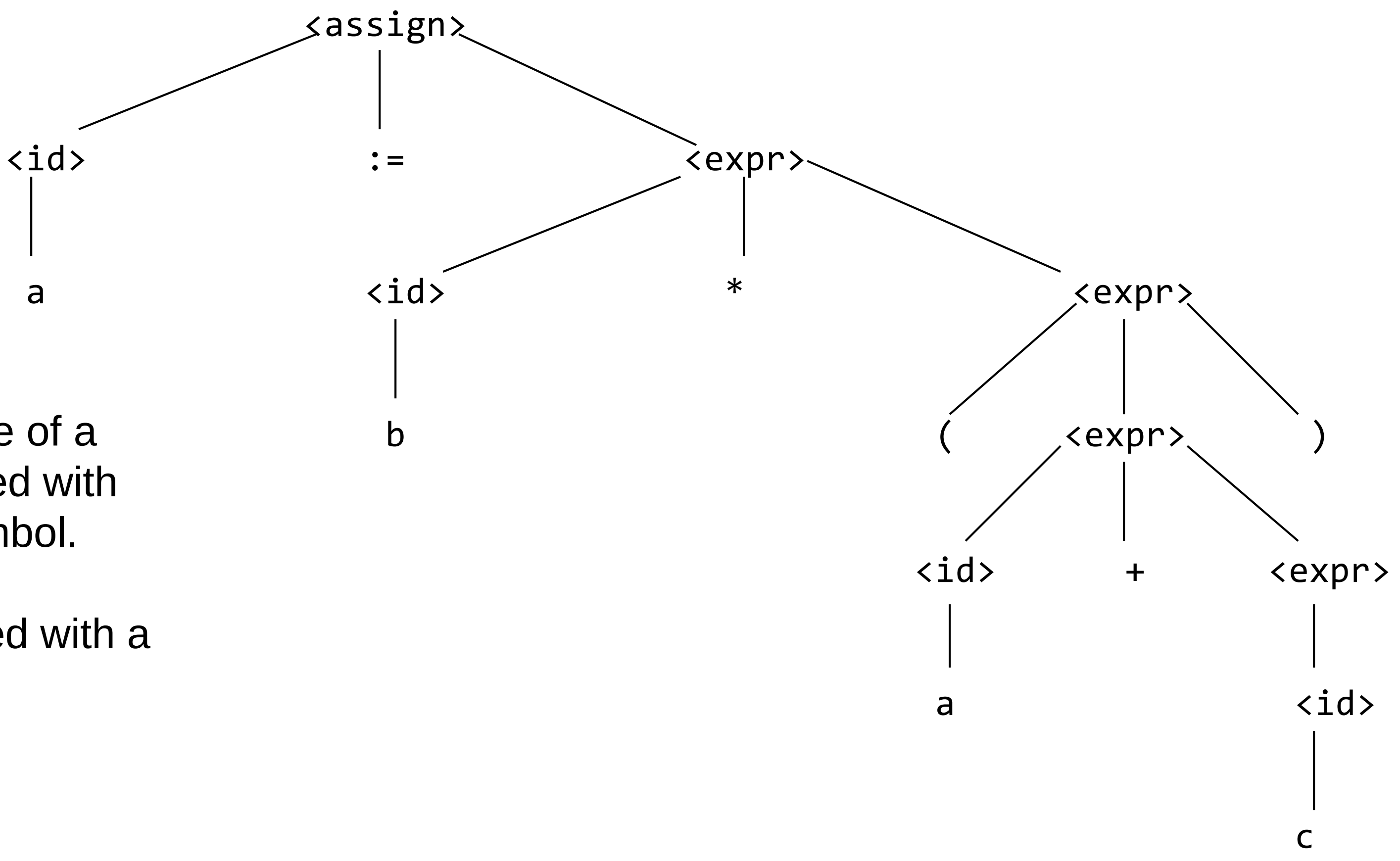
The statement **`a := b * ( a + c )`** is generated by this left most derivation:

```
   <assgn>
-> <id> := <expr>                  R1
-> a := <expr>                     R2
-> a := <id> * <expr>              R4
-> a := b * <expr>                 R2
-> a := b * ( <expr> )             R5
-> a := b * ( <id> + <expr> )      R3
-> a := b * ( a + <expr> )         R2
-> a := b * ( a + <id> )           R6
-> a := b * ( a + c )              R2
```

In a **left most derivation** only the left most non-terminal is replaced

Parse Trees

A parse tree is a graphical representation of a derivation.
For instance, the parse tree for the statement `a := b * ( a + c )` is:



Every internal node of a parse tree is labeled with a non-terminal symbol.

Every leaf is labeled with a terminal symbol.

Ambiguity

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be ambiguous.

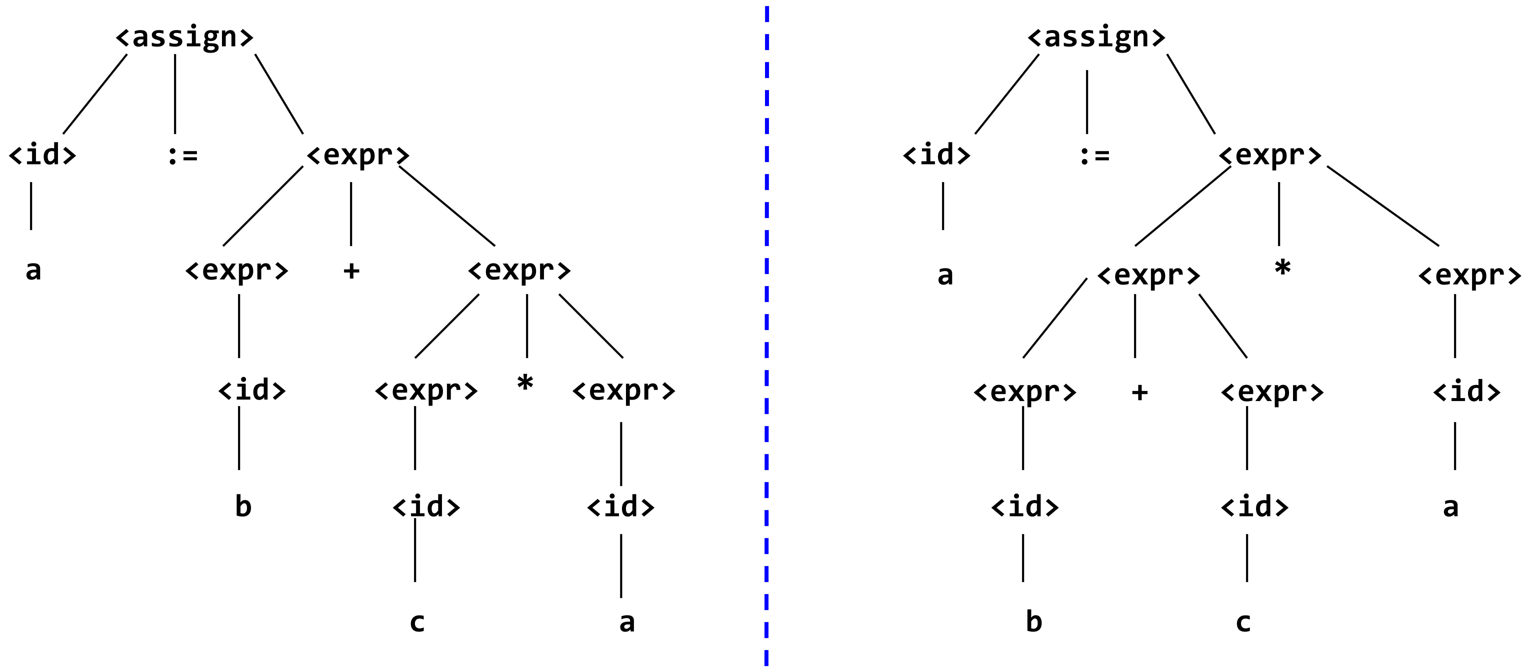
For instance, our grammar for a simple assignment statement

```
<assgn> ::= <id> := <expr>
<id>    ::= a | b | c
<expr>  ::=   <expr> + <expr>
            | <expr> *  <expr>
            | ( <expr> )
            | <id>
```

is ambiguous because there are distinct parse trees for the expression

```
a := b + c * a
```

Distinct Parse Trees



If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.

Operator Precedence

If an operator is generated lower in the parse tree, it indicates that the operator has precedence over the operator generated higher up in the tree.

An unambiguous grammar for expressions:

```
<assign> ::= <id> := <expr>

<id>     ::= a | b | c

<expr>   ::=   <expr> + <term>
            | <term>

<term>   ::=   <term> * <factor>
            | <factor>

<factor> ::=  ( <expr> )
            | <id>
```

This grammar indicates the usual precedence order of multiplication and addition operators.

This grammar generates unique parse trees independently of doing a rightmost or leftmost derivation

Example of Derivations

Leftmost derivation:

```
   <assgn>
-> <id> := <expr>
-> a := <expr>
-> a := <expr> + <term>
-> a := <term> + <term>
-> a := <factor> + <term>
-> a := <id> + <term>
-> a := b + <term>
-> a := b + <term> *<factor>
-> a := b + <factor> * <factor>
-> a := b + <id> * <factor>
-> a := b + c * <factor>
-> a := b + c * <id>
-> a := b + c * a
```

Rightmost derivation:

```
   <assgn>
-> <id> := <expr>
-> <id> := <expr> + <term>
-> <id> := <expr> + <term> * <factor>
-> <id> := <expr> + <term> * <id>
-> <id> := <expr> + <term> * a
-> <id> := <expr> + <factor> * a
-> <id> := <expr> + <id> * a
-> <id> := <expr> + c * a
-> <id> := <term> + c * a
-> <id> := <factor> + c * a
-> <id> := <id> + c * a
-> <id> :=  b + c * a
-> a := b + c * a
```

Precedence & Associativity

Dealing with ambiguity:

Rule 1: * (times) and / (divide) have higher precedence than + (plus) and – (minus).

Example:

```
a + c * 3 means a + (c * 3)
```

Rule 2: Operators of equal precedence associate to the left.

Example:

```
a + c + 3 means (a + c) + 3
```

Avoiding Ambiguity

As seen previously, the following grammar for arithmetic expression is ambiguous.

```
<expr> ::= <expr> <op> <expr> | id | int | ( <expr> )

<op>   ::= + | - | * | /
```

It is not to difficult to modify the above grammar and obtain an unambiguous grammar that generates the same sentences:

```
<expr>   ::=   <term>
             | <expr> + <term> | <expr> - <term>

<term>   ::=   <factor>
             | <term> * <factor> | <term> / <factor>

<factor> ::= id | int | (<expr>)
```

Ambiguity

Is this grammar ambiguous?

```
E -> T | E + T | E - T
T -> F | T * F | T / F
F -> id | num | ( E )
```

Parse `id + id – id`

Is this grammar ambiguous?

```
E -> T | E + T
T -> F | T * F
F -> id | ( E )
```

Parse `id + id + id`

Ambiguity

Is this grammar ambiguous?

```
E -> E + E | id
```

Parse `id + id + id`

Is this grammar ambiguous?

```
E -> E + id |  id
```

Parse `id + id + id`

Ambiguity

Example:

Given the ambiguous grammar

```
E -> E + E | E * E | ( E ) | id
```

We could rewrite it as:

```
E -> E + T | T
T -> T * F | F
F -> id | ( E )
```

Determine the parse tree for:

```
id + id * id
```