COP 3402 Systems Software

Syntax analysis (Parser)

Outline

1. Parsing
2. Context Free Grammars
3. Ambiguous Grammars
4. Unambiguous Grammars

Parsing

In a regular language nested structures can not be expressed.

Nested structures can be expressed with the aid of recursion.

For example, A FSA cannot suffice for the recognition of sentences in the set

$\{ \mathbf{a}^n \mathbf{b}^n \mid n \text{ is in } \{0, 1, 2, 3, \ldots\}\}$

where **a** represents "(" or "{"

and **b** represents ")" or "}"

Parsing

So far we have been working with three rules to define regular sets (regular languages):

Concatenation → (s r)

Alternation (choice) → (s | r)

Kleene closure (repetition) → (s)*

Regular sets are generated by regular expressions and recognized by scanners (FSA).

Adding recursion as an additional rule we can define context free languages.

Context Free Grammars

Any string that can be defined using concatenation, alternation, Kleene closure and recursion is called a Context Free Language (CFL).

CFLs are generated by Context Free Grammars (CFG) and can recognize by Pushdown Automatas.

"**Every language displays a structure called its grammar**"

Parsing is the task of determining the structure or syntax of a program.

Context Free Grammars

Let us observe the following three rules (grammar):

1) <sentence> → <subject> <predicate>

   Where "→" means "is defined as"

2) <subject> → **John** | **Mary**

3) <predicate> → **eats** | **talks**

   where " | " means "or"

With this rules we define four possible sentences:

John eats **John talks** **Mary eats** **Mary talks**

Context Free Grammars

We will refer to the formulae or rules used in the former example as :

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

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

Using a shorthand notation we can write the following syntax rules

$S \rightarrow A\ B$

$A \rightarrow a \mid b$

$B \rightarrow c \mid d$

L = { ac, ad, bc, bd} = set of sentences

L is called the language that can be generated by the syntax rules by repeated substitution.

Context Free Grammars

- Definition : A language is a set of strings of characters from some alphabet.
- The strings of the language are called sentences or statements.
- A string over some alphabet is a finite sequence of symbols drawn from that alphabet.
- A meta-language is a language that is used to describe another language.

Context Free Grammars

A very well known meta-language is BNF (Backus Naur Form)

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

Noam Chomsky in the early 50s developed context free grammars which can be expressed using BNF.

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 can not 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 {S, A, B} by other symbols

non terminal symbols

$$S \rightarrow \langle A\rangle \langle B\rangle$$

$$\langle A\rangle \rightarrow a \mid b$$

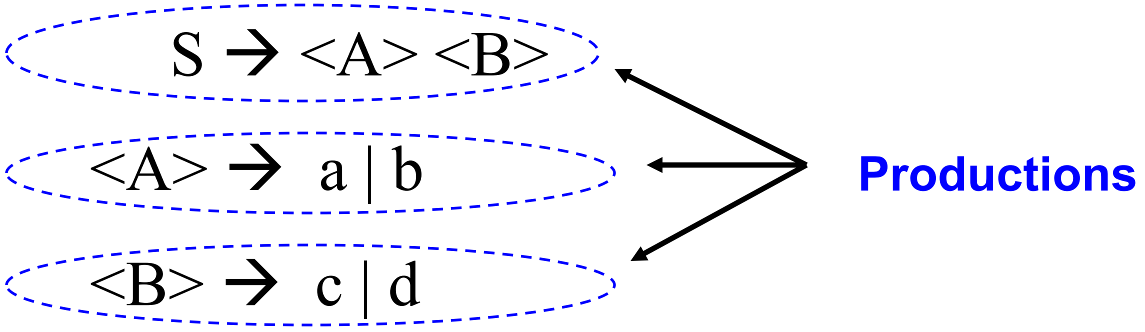
$$\langle B\rangle \rightarrow c \mid d$$

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)

$$\boxed{S} \rightarrow \langle A\rangle\ \langle B\rangle$$

$$\langle A\rangle \rightarrow a \mid b$$

$$\langle B\rangle \rightarrow c \mid d$$

Context Free Grammars

Example of a grammar for a small language:

```
<program>    → begin <stmt-list> end
<stmt-list>  → <stmt> | <stmt> ; <stmt-list>
<stmt>       → <var> = <expression>
<expression> → <var> + <var> | <var> - <var> | <var>
```

Context Free Grammars

A sentence generation is called a derivation.

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>**

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

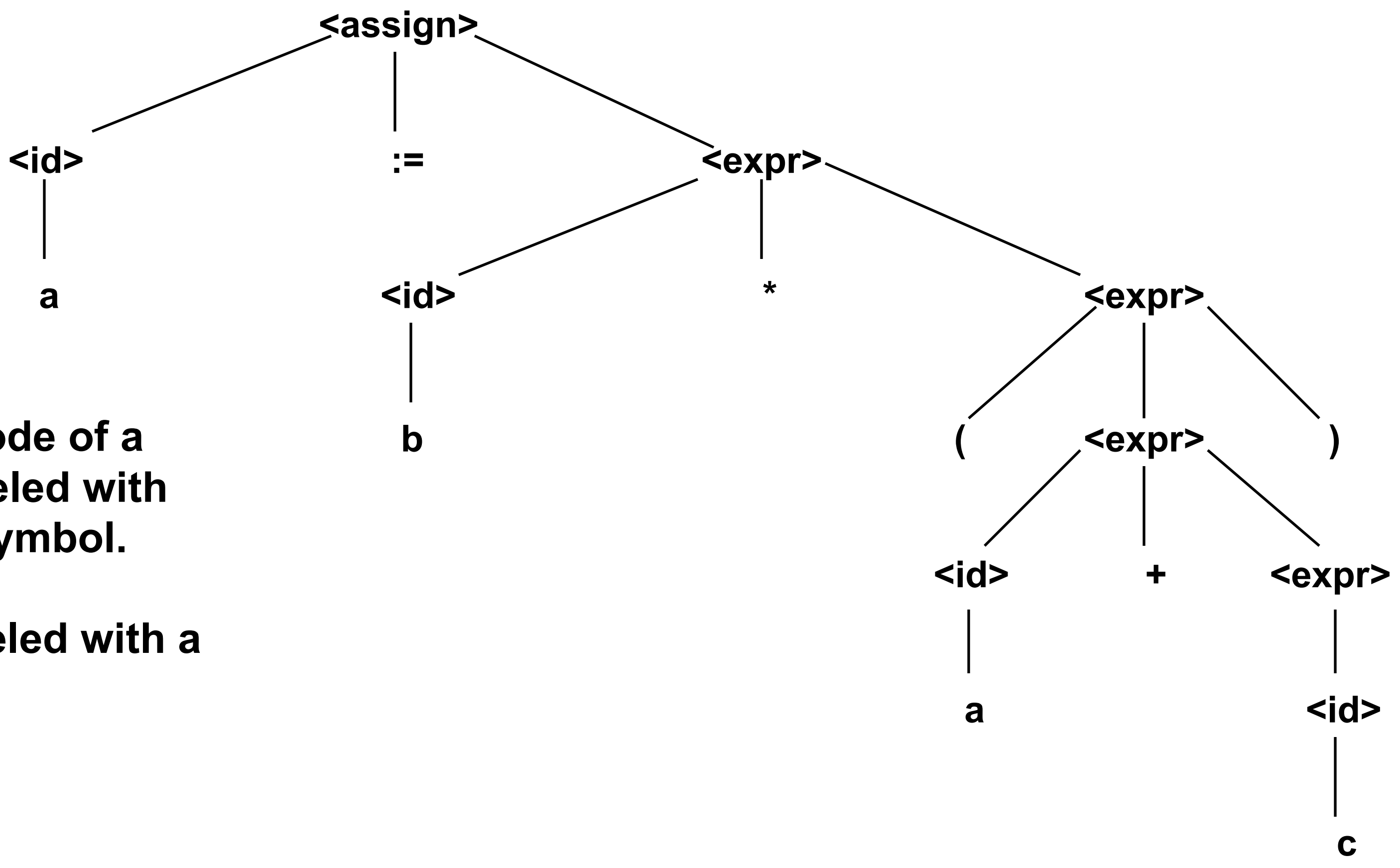
The statement a := b * (a + c)
Is generated by the left most derivation:

<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**

Parse Trees

<u>A parse tree is a graphical representation of a derivation</u>

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.

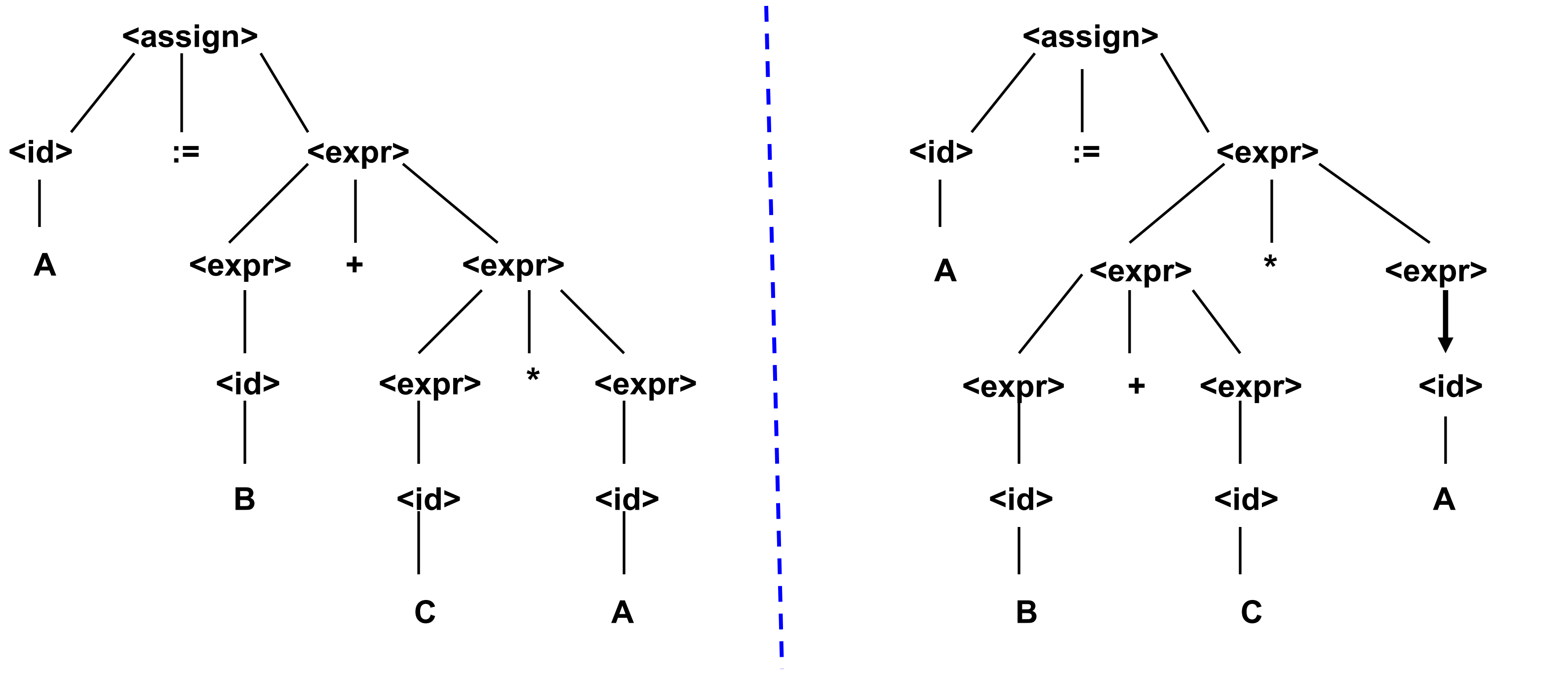
Ambiguity

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

For instance, the following grammar is ambiguous because it generates distinct parse trees for the expression a := b + c * a

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

Ambiguity



This grammar generates two parse trees for the same expression.

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

Ambiguity

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 unambiguos 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

Ambiguity

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
```

Ambiguity

Dealing with ambiguity:

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

Example:

a + c * 3 → a + (c * 3)

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

Example:

a + c + 3 → (a + c) + 3

Ambiguity

Dealing with ambiguity:

Rewrite the grammar to avoid ambiguity.

The grammar:

<expr> → <expr> <op> <expr> | id | int | (<expr>)
<op> → + | - | * | /

Can be rewritten it as:

<expr> → <term> | <expr> + <term> | <expr> - <term>
<term> → <factor> | <term> * <factor> | <term> / <factor>.
<factor> → id | int | (<expr>)

Ambiguity

Is this grammar ambiguous?

$E \rightarrow T \mid E + T \mid E - T$
$T \rightarrow F \mid T * F \mid T / F$
$F \rightarrow id \mid num \mid (E)$

Parse id + id – id

Is this grammar ambiguous?

$E \rightarrow T \mid E + T$
$T \rightarrow F \mid T * F$
$F \rightarrow id \mid (E)$

Parse id + id + id

Ambiguity

Is this grammar ambiguous?

$E \rightarrow E + E \mid id$

Parse id + id + id

Is this grammar ambiguous?

$E \rightarrow E + id \mid id$

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)

Find the parse three for:

Id + id * id

Recursive descent parsing

Ambiguity if not the only problem associated with recursive descent parsing. Other problems to be aware of are left recursion and left factoring:

Left recursion: A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \rightarrow A\,\alpha$ for some string α. Top-down parsing methods can not handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

For example, the pair of productions: $A \rightarrow A\,\alpha \mid \beta$

could be replaced by the non-left-recursive productions:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Recursive descent parsing

Left factoring: Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive (top-down) parsing. When the choice between two alternative A-production is not clear, we may be able to rewrite the production to defer the decision until enough of the input has been seen thus we can make the right choice.

For example, the pair of productions: $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$

could be left-factored to the following productions: $A \rightarrow \alpha\ A'$

$A' \rightarrow \beta_1 \mid \beta_2$

Recursive descent parsing

Given the following grammar:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow id \mid (E)$

To avoid left recursion we can rewrite it as:

$E \rightarrow T E'$
$E' \rightarrow + T E' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

COP 3402 Systems Software

Euripides Montagne
University of Central Florida