

Syntactic Analysis

Basic definitions

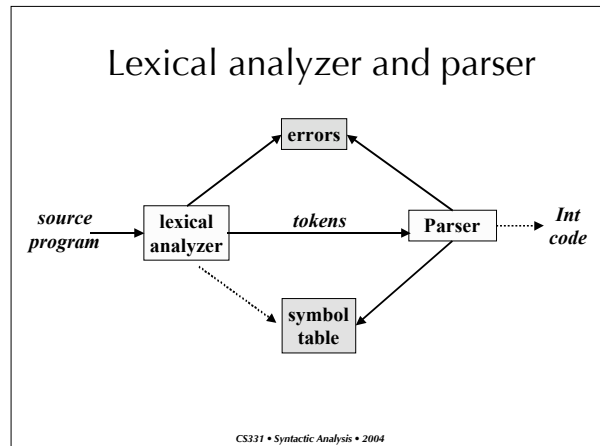
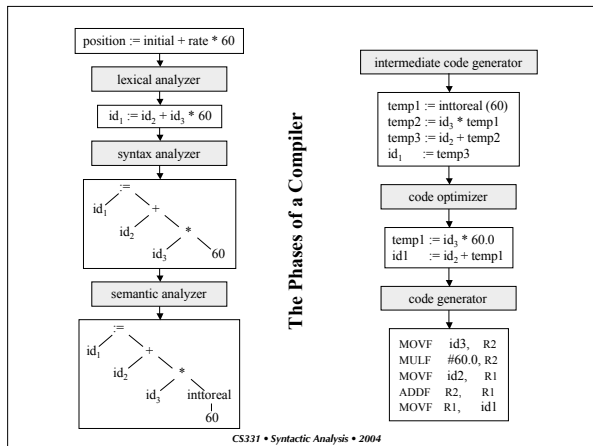
syntax: the way in which words are put together to form phrases, clauses, or sentences. The rules governing the formation of statements in a programming language.

syntax analysis: the task concerned with fitting a sequence of tokens into a specified syntax.

parsing: Breaking a sentence down into its component parts of speech with an explanation of the form, function, and syntactical relationship of each part.

parsing = lexical analysis + syntax analysis

CS331 • Syntactic Analysis • 2004



The role of the parser

- perform context-free syntax analysis
- guide the context-sensitive analysis (→ attribute grammars)
- construct an intermediate representation (→ attribute grammars)
- produce meaningful error messages
- attempt error correction

For the next several lectures, we will look at parser construction.

CS331 • Syntactic Analysis • 2004

Syntax analysis

Context-free syntax is specified with a **grammar**.

Formally, a context-free grammar G is a four-tuple (T, NT, S, P)

T is the set of terminal symbols in the grammar. For our purposes, the set of terminals is equivalent to the set of tokens returned by the lexical analyzer.

NT is a set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.

S is a distinguished nonterminal ($S \in NT$) that denotes the entire set of strings in $L(G)$. This is sometimes called a goal symbol. S cannot appear on the right hand side of some production.

P is a set of productions that specify the way that terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

CS331 • Syntactic Analysis • 2004

Syntax analysis

Grammars are often written in **BNF**, or **Backus-Naur form**.

E.g., a BNF grammar for simple expressions over numbers and identifiers:

```
1 <goal> ::= <expr>
2 <expr> ::= <expr> <op> <expr>
3           | number
4           | id
5 <op> ::= +
6       | -
7       | *
8       | /
```

In a BNF for a grammar, we represent

1. non-terminals with brackets or capital letters,
2. terminals with **typewriter font** or underline,
3. productions as in the example.

CS331 • Syntactic Analysis • 2004

Why use context-free grammars?

Many advantages:

- **precise syntactic specification** of a programming language
- **easy to understand**, avoids ad hoc definition
- **easier to maintain**, add new language features
- can **automatically construct efficient parser**
- parser construction **reveals ambiguity**, other difficulties
- **imparts structure** to language
- **supports syntax-directed translation**

CS331 • Syntactic Analysis • 2004

Grammars for regular languages

Can we place a restriction on the **form** of a grammar to ensure that it describes a regular language?

Provable fact:

For any RE r , there is a grammar g such that $L(r) = L(g)$.

The grammars that generate regular sets are called **regular grammars**.

Definition:

In a regular grammar, all productions have one of two forms:

1. $A \rightarrow aA$
2. $A \rightarrow a$

where A is a non-terminal and a is a terminal symbol.

These are also called type 3 grammars (Chomsky)

CS331 • Syntactic Analysis • 2004

Scanning vs. parsing

Where do we draw the line?

$term \rightarrow [a-zA-z] ([a-zA-z] | [0-9])^*$
 $\quad \quad \quad | 0 | [1-9][0-9]^*$
 $op \rightarrow + | - | * | /$
 $expr \rightarrow (term\ op)^* term$

Regular expressions are used to classify

- identifiers, numbers, keywords

Context-free grammars are used to count

- brackets (), begin - end, if -then - else
- imparting structure - expressions

CS331 • Syntactic Analysis • 2004

Review of some definitions

Parse tree (derivation tree):

"Graphical" representation for a derivation $S \xRightarrow{*} w$; filters out the choice regarding non-terminal replacement order

Leftmost (rightmost) derivation:

The leftmost (rightmost) non-terminal is replaced at each step

Sentential form:

Any α such that $S \xRightarrow{*} \alpha$

Left (Right) sentential form:

Any α such that $S \xRightarrow{*}_{lm} \alpha$ ($S \xRightarrow{*}_{rm} \alpha$)

CS331 • Syntactic Analysis • 2004

Derivations

We can view the productions of a cfg as rewriting rules.

Using our example

$\langle goal \rangle \Rightarrow \langle expr \rangle$
 $\Rightarrow \langle expr \rangle \langle op \rangle \langle expr \rangle$
 $\Rightarrow \langle expr \rangle \langle op \rangle \langle expr \rangle \langle op \rangle \langle expr \rangle$
 $\Rightarrow \langle id, x \rangle \langle op \rangle \langle expr \rangle \langle op \rangle \langle expr \rangle$
 $\Rightarrow \langle id, x \rangle + \langle expr \rangle \langle op \rangle \langle expr \rangle$
 $\Rightarrow \langle id, x \rangle + \langle num, 2 \rangle \langle op \rangle \langle expr \rangle$
 $\Rightarrow \langle id, x \rangle + \langle num, 2 \rangle * \langle expr \rangle$
 $\Rightarrow \langle id, x \rangle + \langle num, 2 \rangle * \langle id, y \rangle$

We have *derived* the sentence $x + 2 * y$.

We denote this $\langle goal \rangle \Rightarrow * id + num * id$.

Such a sequence of rewrites is a *derivation* or a *parse*.

The process of discovering a derivation is called *parsing*.

CS331 • Syntactic Analysis • 2004

Derivations

At each step, we chose a non-terminal to replace.
This choice can lead to different derivations.

Two are of particular interest

- leftmost derivation
 - the leftmost non-terminal is replaced at each step
- rightmost derivation
 - the rightmost non-terminal is replaced at each step

The example was a leftmost derivation.

CS331 • Syntactic Analysis • 2004

Rightmost Derivation

For the string $x + 2 * y$:

```

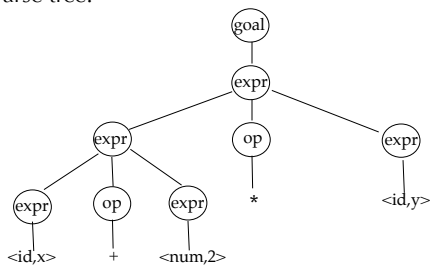
<goal>  => <expr>
        => <expr> <op> <expr>
        => <expr> <op> <id,y>
        => <expr> * <id,y>
        => <expr> <op> <expr> * <id,y>
        => <expr> <op> <num,2> * <id,y>
        => <expr> + <num,2> * <id,y>
        => <term> + <num,2> * <id,y>
        => <id,x> + <num,2> * <id,y>
    
```

Again, $\langle \text{goal} \rangle \Rightarrow * \text{id} + \text{num} * \text{id}$.

CS331 • Syntactic Analysis • 2004

Precedence

Parse tree:



Treewalk evaluation would give the "wrong" answer:
 $(x + 2) * y$ instead of $x + (2 * y)$

CS331 • Syntactic Analysis • 2004

Precedence

- These two derivations point out a problem with the grammar
 - It has no notion of *precedence*, or implied order of evaluation.
- To add precedence takes additional machinery

```

1 <goal> ::= <expr>
2 <expr> ::= <expr> + <term>
3           | <expr> - <term>
4           | <term>
5 <term> ::= <term> * <factor>
6           | <term> / <factor>
7           | <factor>
8 <factor> ::= number
9           | id
    
```

- This grammar enforces a precedence on the derivation
 - terms must be derived from expressions
 - forces the "correct" tree

CS331 • Syntactic Analysis • 2004

Precedence

Now, for the string $x + 2 * y$:

```

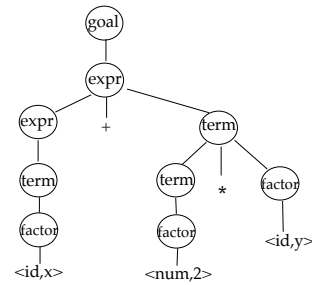
<goal>      => <expr>
            => <expr> + <term>
            => <expr> + <term> * <factor>
            => <expr> + <term> * <id,y>
            => <expr> + <factor> * <id,y>
            => <expr> + <num,2> * <id,y>
            => <term> + <num,2> * <id,y>
            => <factor> + <num,2> * <id,y>
            => <id,x> + <num,2> * <id,y>
    
```

Again, $\langle \text{goal} \rangle \Rightarrow \text{id} + \text{num} * \text{id}$, but this time, we build the desired tree.

CS331 • Syntactic Analysis • 2004

Precedence

This time we get the desired tree:



Treewalk evaluation computes $x + (2 * y)$.

CS331 • Syntactic Analysis • 2004

Ambiguity

A grammar is *ambiguous*

iff

There is a string that has multiple parse trees

iff

There is a string that has multiple leftmost derivations

iff

There is a string that has multiple rightmost derivations

Example

```

<stmt> ::= if <expr> then <stmt>
        | if <expr> then <stmt> else <stmt>
        | other stmts
    
```

Consider deriving the sentential form:

if E_1 then if E_2 then S_1 else S_2

It has two derivations.

- This ambiguity is purely grammatical.

CS331 • Syntactic Analysis • 2004

Ambiguity

- We may be able to eliminate ambiguities by rearranging the grammar.

```

<stmt> ::= <ms>
        | <us>
<ms> ::= if <expr> then <ms> else <ms>
        | other stmts
<us> ::= if <expr> then <stmt>
        | if <expr> then <ms> else <us>
    
```

- This grammar generates the same language as the ambiguous grammar, but applies the common sense rule: "match each else with the closest unmatched then"

This is pretty clearly the language designer's intent.

CS331 • Syntactic Analysis • 2004

Ambiguity

- Ambiguity generally refers to a confusion in the *context-free* specification
- *Context-sensitive* confusions can arise from overloading. E.g.,
 $a = f(17)$
 - In many Algol-like languages, f could be either a function or a subscripted variable
- Disambiguating this statement requires context
 - need values of declarations
 - not context free
 - really an issue of type

Rather than complicate parsing, we handle this separately

CS331 • Syntactic Analysis • 2004

Top-down versus bottom-up parsing

Top-down parsers

- start at the root of derivation tree and fill in
- pick a production and try to match the input
- may require backtracking
- some grammars are backtrack-free (predictive)

Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to record possibilities (recognize valid prefixes)
- use a stack to store both state and sentential forms

CS331 • Syntactic Analysis • 2004

Top-down parsing

A top-down parser starts with the root of the parse tree. The root is labeled with the start symbol or goal symbol of the grammar.

- To build a parse tree, it repeats the following steps until the fringe of the parse tree matches the input string.
 - At a node labeled A , select a production with A on its lhs, and for each symbol on its rhs, construct the appropriate child.
 - When a terminal is added to the fringe that doesn't match the input string, backtrack.
 - Find the next node to be expanded. (Must have a label in NT)

The key is selecting the right production in step 1
 \Rightarrow should be guided by input string

CS331 • Syntactic Analysis • 2004

Simple expression grammar

Recall grammar for simple expressions:

```
1 <goal> ::= <expr>
2 <expr> ::= <expr> + <term>
3           | <expr> - <term>
4           | <term>
5 <term> ::= <term> * <factor>
6           | <term> = <factor>
7           | <factor>
8 <factor> ::= number
9           | id
```

Consider the input string $x - 2 * y$

CS331 • Syntactic Analysis • 2004

Example

Prod'n	Sentential form	Input
-	@<goal>	@x - 2 * y
1	@<expr>	@x - 2 * y
2	@<expr> + <term>	@x - 2 * y
4	@<term> + <term>	@x - 2 * y
7	@<factor> + <term>	@x - 2 * y
9	@id * <term>	@x - 2 * y
id	id @ + <term>	x @ - 2 * y
Backtrack →	@<expr>	@x - 2 * y
3	@<expr> - <term>	@x - 2 * y
4	@<term> - <term>	@x - 2 * y
7	@<factor> - <term>	@x - 2 * y
9	@id - <term>	@x - 2 * y
id	id @ - <term>	x @ - 2 * y
Backtrack →	id - @<term>	x - @2 * y
7	id - @<factor>	x - @2 * y
9	id - @num	x - @2 * y
id	id - num @	x - 2 @ * y
5	id - @<term> * <factor>	x - @2 * y
7	id - @<factor> * <factor>	x - @2 * y
9	id - @num * <factor>	x - @2 * y
num	id - num @ * <factor>	x - 2 @ * y
*	id - num * @<factor>	x - 2 * @y
9	id - num * id	x - 2 * y@
id	id - num * id@	x - 2 * y@

CS331 • Syntactic Analysis • 2004

Example

Another possible parse for x - 2 * y:

Prod'n	Sentential form	Input
-	<goal>	@x - 2 * y
1	<expr>	@x - 2 * y
2	<expr> + <term>	@x - 2 * y
2	<expr> + <term> + <term>	@x - 2 * y
2	<expr> + <term> + ...	@x - 2 * y
2	<expr> + <term> + ...	@x - 2 * y
2	...	@x - 2 * y

- If the parser makes the wrong choices, the expansion doesn't terminate.
- This isn't a good property for a parser to have. (Parsers should terminate!)

CS331 • Syntactic Analysis • 2004

Left Recursion

Top-down parsers cannot handle left-recursion in a grammar.

Formally,

- a grammar is left recursive if for $A \in NT$ and there is a derivation $A \xRightarrow{*} A\alpha$ for some string α .
- The simple expression grammar is left recursive
- But left recursion can be removed without changing the language generated
 - Did this in language theory...

CS331 • Syntactic Analysis • 2004

Example

- The expression grammar contains two cases of left recursion

```

<expr> ::= <expr> + <term>
        | <expr> - <term>
        | <term>
<term> ::= <term> * <factor>
        | <term> / <factor>
        | <factor>
    
```

- Applying the transformation to eliminate left recursion gives

```

<expr> ::= <term> <expr'>
<expr'> ::= + <term> <expr'>
        | ε
        | - <term> <expr'>
<term> ::= <factor> <term'>
<term'> ::= * <factor> <term'>
        | ε
        | / <factor> <term'>
    
```

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

CS331 • Syntactic Analysis • 2004

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production

- Do we need arbitrary lookahead to parse CFGs?
 - in general, yes
 - use the Earley or Cocke-Younger, Kasami algorithms
Aho, Hopcroft, and Ullman, Problem 2.34
- Fortunately
 - large subclasses of CFGs can be parsed with limited lookahead
 - most programming language constructs can be expressed in a grammar that falls in these subclasses
 - Among the interesting subclasses are LL(1) and LR(1).

CS331 • Syntactic Analysis • 2004

Predictive Parsing

Basic idea:

For any two productions $A ::= \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.

For some $rhs \alpha \in C$, define $FIRST(\alpha)$ as the set of tokens that appear as the first symbol in some string derived from α .

That is, $x \in FIRST(\alpha)$ iff $\alpha \Rightarrow^* x\gamma$ for some γ

Key Property:

Whenever two productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, we would like

$$FIRST(\alpha) \cap FIRST(\beta) = \Phi$$

This would allow the parser to make a correct choice with a lookahead of only one symbol!

The example grammar has this property

CS331 • Syntactic Analysis • 2004

What about ϵ ?

What happens if $\alpha \Rightarrow^* \epsilon$?

- Can't rely just on $FIRST(\alpha)$, also have to consider $FOLLOW(A)$
- If α goes to ϵ , the next token might be from $FOLLOW(A)$

$FOLLOW(A)$ is the set of tokens that can appear immediately after an A in some sentential form

If $S \Rightarrow xAy$, $FOLLOW(A)$ includes $FIRST(y)$

CS331 • Syntactic Analysis • 2004

Left Factoring

What if a grammar does not have this property?

Sometimes, we can transform a grammar to have this property.

- For each non-terminal A find the longest prefix common to two or more of its alternatives.
- if $\alpha \neq \epsilon$, then replace all of the A productions $A ::= \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$ with
$$A ::= \alpha L \mid \gamma$$
$$L ::= \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$
where L is a new non-terminal.
- Repeat until no two alternatives for a single non-terminal have a common prefix.

Aho, Sethi, and Ullman, Algorithm 4.2

CS331 • Syntactic Analysis • 2004

Example

- Consider a right-recursive version of the expression grammar:

```

1 <goal> ::= <expr>
2 <expr> ::= <term> + <expr>
3         | <term> - <expr>
4         | <term>
5 <term> ::= <factor> * <term>
6         | <factor> / <term>
7         | <factor>
8 <factor> ::= number
9         | id
    
```

- To choose between productions 2, 3, & 4, the parser must see past the **number** or **id** and look at the +, -, *, or /.
- $FIRST(2) \cap FIRST(3) \cap FIRST(4) \neq \emptyset$
- This grammar fails the test.

CS331 • Syntactic Analysis • 2004

Example

There are two nonterminals that must be left factored:

```

<expr> ::= <term> + <expr>
        | <term> - <expr>
        | <term>
<term> ::= <factor> * <term>
        | <factor> / <term>
        | <factor>
    
```

Applying the transformation gives us:

```

<expr> ::= <term> <expr'>
<expr'> ::= + <expr>
        | - <expr>
        | ε
<term> ::= <factor> <term'>
<term'> ::= * <term>
        | / <term>
        | ε
    
```

CS331 • Syntactic Analysis • 2004

Example

Substituting back into the grammar yields

```

1 <goal> ::= <expr>
2 <expr> ::= <term> <expr'>
3 <expr'> ::= + <expr>
4         | - <expr>
5         | ε
6 <term> ::= <factor> <term'>
7 <term'> ::= * <term>
8         | / <term>
9         | ε
10 <factor> ::= number
11         | id
    
```

Now, selection requires only a single token lookahead.

CS331 • Syntactic Analysis • 2004

Example

Sentential form	Input
@<goal>	@x - 2 * y
@<expr>	@x - 2 * y
@<term> <expr'>	@x - 2 * y
@<factor> <term'> <expr'>	@x - 2 * y
@ id <term'> <expr'>	@x - 2 * y
id @<term'> <expr'>	x @ - 2 * y
id @ <expr'>	x @ - 2 * y
id @ - <expr>	x @ - 2 * y
id - @<expr>	x - @2 * y
id - @<term> <expr'>	x - @2 * y
id - @<factor> <term'> <expr'>	x - @2 * y
id - @ num <term'> <expr'>	x - @2 * y
id - num @<term'> <expr'>	x - 2 @ * y
id - num @ * <term> <expr'>	x - 2 @ * y
id - num * @<term> <expr'>	x - 2 * @y
id - num * @<factor> <term'> <expr'>	x - 2 * @y
id - num * @ id <expr'>	x - 2 * @y
id - num * id @<term'> <expr'>	x - 2 * y@
id - num * id @<expr'>	x - 2 * y@
id - num * id @	x - 2 * y@

The next symbol determines each choice correctly

CS331 • Syntactic Analysis • 2004

Generality

Question:

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary context free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer:

Given a context free grammar that doesn't meet our conditions, it is **undecidable** whether an equivalent grammar exists that does meet our conditions.

Many context free languages do not have such a grammar.

$$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$$