Basic Components — Syntax

Steven Zeil

Sep. 7, 2001

Contents

3 Translation **1**
2.1 Syntax . . . 1
2.1.1 Separation of Form from Intent . . . 1
2.1.2 Abstract Syntax . . . 3
2.1.3 Concrete Syntax . . . 3
2.2 Recursive Descent Parsing . . . 9
2.3 Miscellaneous notes . . . 11

Basic Components of Programming Languages

1. History
2. Classification
3. Translation

---

3 Translation

1. phases of Translation
2. Lexical: What are the *words*?
3. Syntax: What is the *grammatical structure* combining the words into sentences?
4. Semantics: What do the sentences *mean*?

2.1 Syntax

The **syntax** of a language is the set of rules describing how tokens can be combined to form sentences.

("sentence" = "program", for our purposes)

---

1. Separation of Form from Intent
2. Abstract Syntax
3. Concrete Syntax

2.1.1 Separation of Form from Intent

The same sentence can be expressed in many different ways by altering the syntactic rules.

Idea: Add `A` to the product of `B` and `C`, then divide the result by `B`.

How to express this?: The form we choose must capture such details as "the `+` operator is applied to `A` and to the *result* of the $B * C$ calculation."

Some possibilities are

Infix form: Put each operator in between its operands.

Prefix form: Put each operator before its operands.

Postfix form: Put each operator after its operands.

Idea: Add `A` to the product of `B` and `C`, then divide the result by `B`.

Expressed as:

Infix form: $(A + B * C)/B$

Prefix form: $/ + A * B\,C\,B$

Postfix form: $A\,B\,C * +B\,/$

Infix Notation

- most familiar
- actually more complicated than prefix or postfix

For example, how do we know to write

$$(A + B * C)/B$$

instead of

$$A + B * C/B \text{ ?}$$

How do we know that we don't need to write

$$(A + (B * C))/B \text{ ?}$$

We augment infix notation using *associativity* and *precedence*.

Associativity

Associativity is the rules by which sequences of the *same* operation are evaluated.

In conventional algebra,

- $+$, $-$, $/$, and $*$ are **left associative**.
  $4 - 2 - 1$ is grouped as (4-2)-1
- $**$ (raises to the power) is **right associative**.
  $4 * *2 * *3 = 4^{2^3}$ is grouped as $4 * *(2 * *3) = 4^{(2^3)}$.

Precedence

Precedence is the rules by which sequences of *different* operations are evaluated.

In conventional algebra, we have the following precedences
higher $\Rightarrow$ lower

$**$	$*, /$	$+, -$

So $1 + 2 * 3$ groups as $1 + (2 * 3)$.

Confusion often results because

- Programmers sometimes forget the algebraic rules.

Confusion often results because

- Programmers sometimes forget the algebraic rules.
  - esp. right-assoc. of $**$

Confusion often results because

- Programmers sometimes forget the algebraic rules.
- Some HLL's have unexpected precedence and associativity rules.

Confusion often results because

- Programmers sometimes forget the algebraic rules.
- Some HLL's have unexpected precedence and associativity rules.
  - APL gave all operators the same precedence and right associativity.

Confusion often results because

- Programmers sometimes forget the algebraic rules.
- Some HLL's have unexpected precedence and associativity rules.
  - Some languages give unary $-$ the highest precedence; some give it the lowest.
    What is $-x * -y$?
    How about $-2 * -1$?

Confusion often results because

- Programmers sometimes forget the algebraic rules.
- Some HLL's have unexpected precedence and associativity rules.
- HLL's have many more operators, for which the expected precedence and associativity is unclear
  - Pascal boolean ops

    ```
    if A < B and C > D then
    ```

    groups as

    ```
    A < (B and C) > D
    ```

    causing a syntax error.

  - In C, if `p` is a pointer to a personnel record, then `(*p).salary` gets the salary. What about `*p.salary`?

2.1.2 Abstract Syntax

So far, we have been pre-occupied with

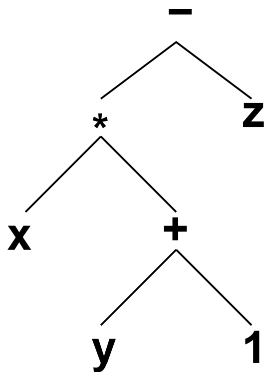
- how tokens group together
- which operators apply to which operands

We call these properties the **abstract syntax** of the language.

Abstract Syntax Trees

Grouping and application of operators are easily illustrated using trees.

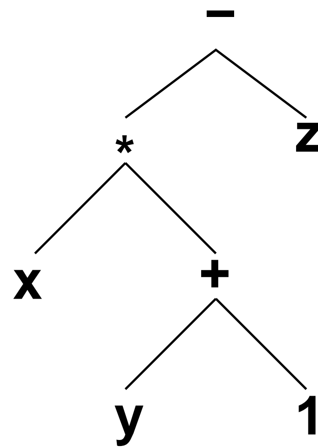
For example, our interpretation of the infix $x * (y + 1) - z$ is



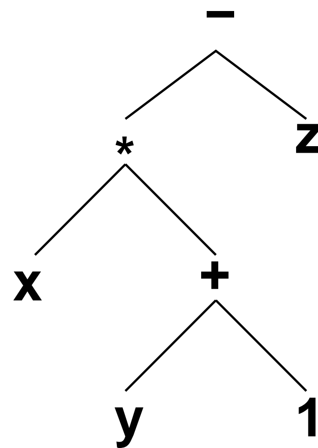
Such **abstract syntax trees** (AST's) are interpreted as

- internal nodes:
  - labelled with tokens denoting operators
  - children are the operands
- leaves contain non-operator tokens (e.g., variables, constants)

Note that the () do not appear in the AST. These are "syntactic sugar" — they don't matter once we have grasped the appropriate grouping and application structure.



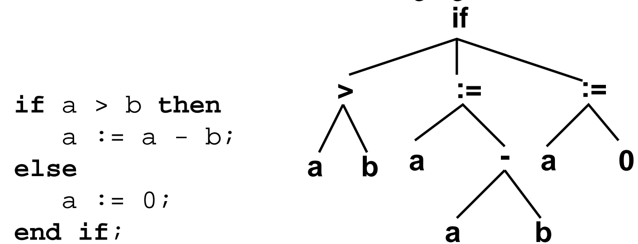
$x * (y + 1) - z$



$x * (y + 1) - z$

How would the AST change if we considered the postfix $x\, y\, 1 + * z -$?

AST's can be extended to other language structures:



2.1.3 Concrete Syntax

The **concrete syntax** of a language is the set of rules governing how it is actually written.

- generally described using *grammars*
  - for HLL's, context-free grammars

A **language** is a set of strings.
A **grammar** $G = [T, N, P, S]$ is a description of a language:

T: set of tokens or **terminals**

N: set of **nonterminals**, symbols representing "sub-languages"

P: set of **productions**, rules for producing strings

S: **start symbol**, a nonterminal that denotes the entire language ($S \in N$)

Kinds of Grammars

There are 4 major kinds of grammars, depending upon the form of the productions:

- Let α, β, γ denote strings of grammar symbols (from N and/or T)
- Let A, B denote a single nonterminal
- Let a, b denote a single terminal

if all the productions have the form	the grammar is called	type
$A \rightarrow a$ or $A \rightarrow aB$	regular	3
$A \rightarrow \alpha$	context-free	2
$\alpha A \beta \rightarrow \alpha \gamma \beta$	context-sensitive	1
(no restriction)	general	0

Each type of grammars includes all grammars of larger type #

- Regular grammars describe the same set of strings as do regular expressions.
- General parsing techniques exist for regular and context-free languages.
  - Lexemes usually described by regular expressions
  - Modern HLL's described via context-free grammars
    - FORTRAN, C are not — headache for compiler writers

Context-Free Grammars

Productions in a CFG have

- a single nonterminal on the left-hand side,
- a "produces" or "can be" symbol in the middle
- and a string of terminals and nonterminals on the right.

Backus-Naur Form

Used to describe the syntax of ALGOL 60.

- Nonterminals are enclosed in $\langle\ \rangle$
  e.g., $\langle$*expression*$\rangle$, $\langle$*statement*$\rangle$
- Terminals are written "as is" or quoted
  e.g., `+`, `-`, `'<'`
- The "can be" symbol is ::=.
- The notation

  $$\langle nonterm \rangle ::= \texttt{string}_1 | \texttt{string}_2 | \ldots$$

  is understood as shorthand for

  $$\langle nonterm \rangle ::= \texttt{string}_1$$
  $$\langle nonterm \rangle ::= \texttt{string}_2$$
  $$\langle nonterm \rangle ::= \ldots$$

Sample Productions

$$
\begin{array}{rcl}
\langle statement \rangle & ::= & \langle assignmentt \rangle \\
 & | & \langle if\text{-}stmt \rangle \\
 & | & \langle loop\text{-}stmt \rangle \\
\langle assignment \rangle & ::= & \langle expr \rangle \texttt{ := } \langle expr \rangle \texttt{ ;} \\
\langle loop\text{-}stmt \rangle & ::= & \texttt{while(} \langle expr \rangle \texttt{)} \\
 & & \langle statement \rangle
\end{array}
$$

Writing Grammars

Not as mysterious as it may seem:

- Nonterminals name "meaningful" substrings
- The structure of the grammar should intuitively reflect the structure of the sentences

---

Example: Consider part of a telephone directory:

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

What can we say about the structure of this sentence?

We are defining the language of "TelephoneDirectories":

$$\langle phoneDir \rangle \quad ::= \quad \ldots$$

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

We can recognize certain internal structures: names, addresses, phone numbers, spacers.

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

⟨*name*⟩ ::= ...
⟨*address*⟩ ::= ...
⟨*phoneNum*⟩ ::= ...
⟨*spacer*⟩ ::= ...

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

⟨*name*⟩ ::= ⟨*lastNm*⟩ `,` ⟨*1stNm*⟩ ⟨*mi*⟩
⟨*address*⟩ ::= ⟨*stNumber*⟩ ⟨*street*⟩
⟨*phoneNum*⟩ ::= ...
⟨*spacer*⟩ ::= ...

It may be tempting to expand things all the way down to the character level:

⟨*name*⟩ ::= ⟨*lastNm*⟩ `,` ⟨*1stNm*⟩ ⟨*mi*⟩
⟨*address*⟩ ::= ⟨*stNumber*⟩ ⟨*street*⟩
⟨*lastNm*⟩ ::= ⟨*noBlankString*⟩
⟨*1stNm*⟩ ::= ⟨*noBlankString*⟩
⟨*mi*⟩ ::= ⟨*char*⟩
⟨*stNumber*⟩ ::= ⟨*noBlankString*⟩
⟨*street*⟩ ::= ⟨*string*⟩

⟨*noBlankString*⟩ ::= ⟨*char*⟩
| ⟨*char*⟩ ⟨*noBlankString*⟩
⟨*string*⟩ ::= ⟨*char*⟩
| ⟨*char*⟩ ⟨*string*⟩
| ' ' ⟨*string*⟩
⟨*char*⟩ ::= $a|b|c|\ldots$

But that's neither necessary nor desirable.

- Need to ask: What are the tokens in this language?

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

⟨*name*⟩ ::= ⟨*lastNm*⟩ `,` ⟨*1stNm*⟩ ⟨*mi*⟩
⟨*address*⟩ ::= ⟨*stNumber*⟩ ⟨*street*⟩
⟨*lastNm*⟩ ::= `noBlankString`
⟨*1stNm*⟩ ::= `noBlankString`
⟨*mi*⟩ ::= `char`
⟨*stNumber*⟩ ::= `noBlankString`
⟨*street*⟩ ::= `string`

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

Next, we note that there are really 2 different kinds of lines here:

⟨*fullLine*⟩ ::= ⟨*name*⟩ ⟨*address*⟩ `spacer` `phoneNum`
⟨*partialLine*⟩ ::= ⟨*partialName*⟩ ⟨*address*⟩ `spacer` `phoneNum`
⟨*partialName*⟩ ::= ⟨*1stNm*⟩ ⟨*mi*⟩

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

And these 2 kinds of lines are arranged into a definite pattern:

⟨*nameBlock*⟩ ::= ⟨*fullLine*⟩ ⟨*partialLineList*⟩
⟨*partialLineList*⟩ ::=
| ⟨*partialLine*⟩ ⟨*partialLineList*⟩

```
Smilling, Joe T 207 Elm St.........555-1201
Smit, Robert 12 Geneva Ave.........555-2345
      Sallie 143 Whit Landing Rd...555-7834
Smith, Andrew A 427 1st St.........555-8928
       Arthur B 123 Sesame St......555-1234
       Barbara K 476 Rock Lake Dr..555-4829
```

And finally, we note that a phone listing consist of repeated such blocks:

⟨*phoneDir*⟩ ::= ⟨*blockList*⟩
⟨*blocklist*⟩ ::= ⟨*nameBlock*⟩
| ⟨*nameBlock*⟩ ⟨*blocklist*⟩

Pulling it all together:

⟨*phoneDir*⟩	::=	⟨*blockList*⟩
⟨*blocklist*⟩	::=	⟨*nameBlock*⟩
	|	⟨*nameBlock*⟩ ⟨*blocklist*⟩
⟨*nameBlock*⟩	::=	⟨*fullLine*⟩ ⟨*partialLineList*⟩
⟨*partialLineList*⟩	::=	
	|	⟨*partialLine*⟩ ⟨*partialLineList*⟩
⟨*fullLine*⟩	::=	⟨*name*⟩ ⟨*address*⟩
		`spacer` `phoneNum`

⟨*partialLine*⟩	::=	⟨*partialName*⟩ ⟨*address*⟩
		`spacer` `phoneNum`
⟨*partialName*⟩	::=	⟨*1stNm*⟩ ⟨*mi*⟩
⟨*name*⟩	::=	⟨*lastNm*⟩ `,` ⟨*1stNm*⟩ ⟨*mi*⟩
⟨*address*⟩	::=	⟨*stNumber*⟩ ⟨*street*⟩
⟨*lastNm*⟩	::=	`noBlankString`
⟨*1stNm*⟩	::=	`noBlankString`
⟨*mi*⟩	::=	`char`
⟨*stNumber*⟩	::=	`noBlankString`
⟨*street*⟩	::=	`string`

How does this compare to our earlier definition of a grammar = (N, T, S, P)?

- N is the set of nonterminals: ⟨*phoneDir*⟩, ⟨*blocklist*⟩, . . . , ⟨*street*⟩
- T is the set of terminals (tokens): `spacer`, `phoneNum`, `string`, . . .
- S is the starting nonterminal: ⟨*phoneDir*⟩
- P is the set of productions we have just written

Using Grammars

How do we know when a grammar describes what we want?

- Need to show that the grammar **generates** the strings in our language.
  - begin with start symbol
  - expand nonterminals using production rules
  - continue until all nonterminals have been removed

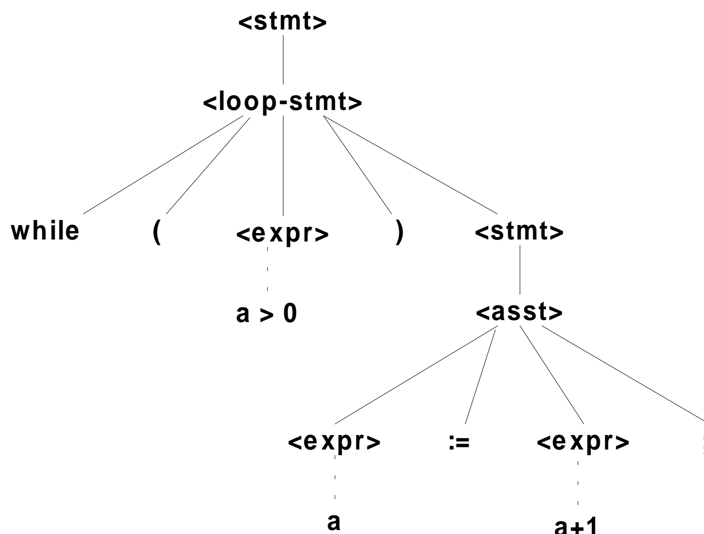
⟨*stmt*⟩	::=	⟨*assignment*⟩
	|	⟨*if-stmt*⟩
	|	⟨*loop-stmt*⟩
⟨*assignment*⟩	::=	⟨*expr*⟩ `:=` ⟨*expr*⟩ `;`
⟨*loop-stmt*⟩	::=	`while(` ⟨*expr*⟩ `)`
		⟨*stmt*⟩

⟨*stmt*⟩ → ⟨*loop-stmt*⟩ → `while(` ⟨*expr*⟩ `)` ⟨*stmt*⟩
. . .
→ `while(a>0)` ⟨*stmt*⟩
→ `while(a>0)` ⟨*assignment*⟩
→ `while(a>0)` ⟨*expr*⟩ `:=` ⟨*expr*⟩ `;`
. . . → `while(a>0)a:=a+1;`

Parse Trees

A more picturesque way to demonstrate the production of strings from a grammar is to give a **parse tree**.

- Start symbol is at root of tree
- Leaves of tree are terminals
- Each non-leaf node is a nonterminal, and its children are the RHS of a production for that nontermimal.



Recursion in CFG's

Note that CFG's are often recursive:

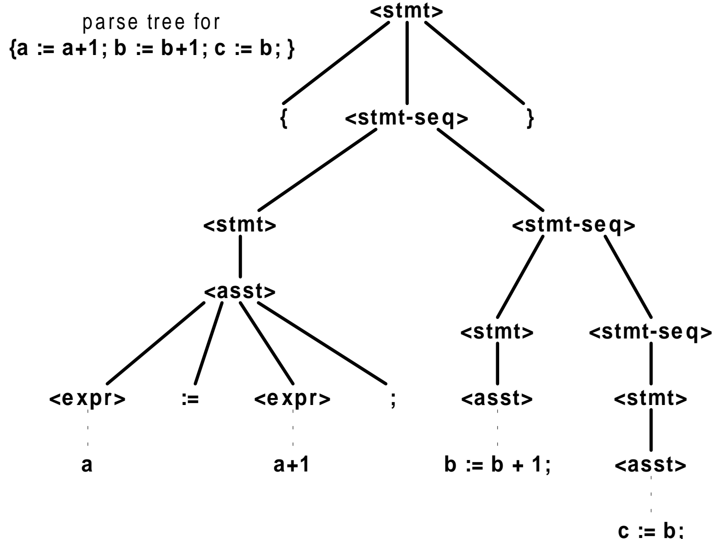
⟨*stmt*⟩	::=	⟨*loop-stmt*⟩
⟨*loop-stmt*⟩	::=	`while(` ⟨*expr*⟩ `)`
		⟨*stmt*⟩

Recursion in a grammar may be

- essential, because self-inclusion is part of the abstract syntax
- incidental, because recursion is used to capture repetition

Note how recursion is used here, where our intuitive idea is "repetition":

$$\begin{array}{rcl} \langle \textit{stmt} \rangle & ::= & \{ \langle \textit{stmt-seq} \rangle \} \\ \langle \textit{stmt-seq} \rangle & ::= & \langle \textit{stmt} \rangle \\ & | & \langle \textit{stmt} \rangle \langle \textit{stmt-seq} \rangle \end{array}$$



Compare this

$$\begin{array}{rcl} \langle \textit{stmt-seq} \rangle & ::= & \langle \textit{stmt} \rangle \\ & | & \langle \textit{stmt} \rangle \langle \textit{stmt-seq} \rangle \end{array}$$

to

$$\begin{array}{rcl} \langle \textit{stmt-seq} \rangle & ::= & \\ & | & \langle \textit{stmt} \rangle \langle \textit{stmt-seq} \rangle \end{array}$$

The right hand side of a production can be empty!

Compare this

$$\begin{array}{rcl} \langle \textit{stmt-seq} \rangle & ::= & \langle \textit{stmt} \rangle \\ & | & \langle \textit{stmt} \rangle \langle \textit{stmt-seq} \rangle \end{array}$$

to

$$\begin{array}{rcl} \langle \textit{stmt-seq} \rangle & ::= & \langle \textit{stmt} \rangle \\ & | & \langle \textit{stmt} \rangle \texttt{;} \langle \textit{stmt-seq} \rangle \end{array}$$

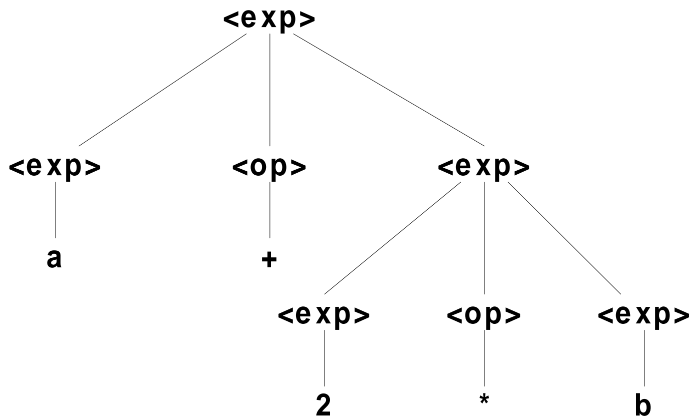
Grammars for Expressions

The simplest approach to representing expressions would be

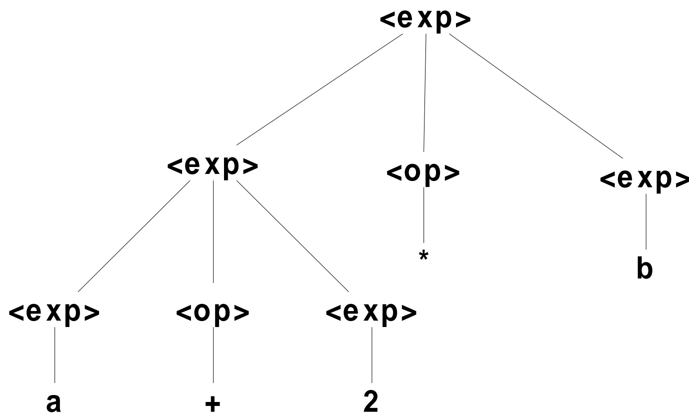
$$\begin{array}{rcl} \langle \textit{exp} \rangle & ::= & \texttt{id} | \texttt{number} \\ & | & \langle \textit{exp} \rangle \langle \textit{op} \rangle \langle \textit{exp} \rangle \\ & | & \texttt{(} \langle \textit{exp} \rangle \texttt{)} \\ \langle \textit{op} \rangle & ::= & \texttt{+} | \texttt{-} | * | \texttt{/} \end{array}$$

This does indeed generate the strings we want, but it fails to reflect the *structure* we want.

This would be a parse tree for $a + 2 * b$:



But this is also a parse tree for $a + 2 * b$:



Ambiguity

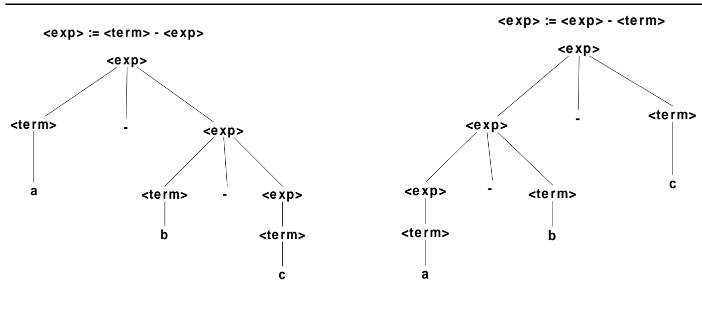
A grammar is **ambiguous** if it allows more than one parse tree for some string in its language.

The simple expression grammar is ambiguous because it fails to reflect the rules of *associativity* and *precedence* that we use to interpret infix expressions.

Associativity in Grammars

Start with a simpler case: expressions involving $-$ only.
Compare parse trees for `a - b - c` using the 2 grammars:

$$\begin{array}{rcl} \langle \textit{exp} \rangle & ::= & \langle \textit{term} \rangle \texttt{-} \langle \textit{exp} \rangle \\ & | & \langle \textit{term} \rangle \\ \langle \textit{term} \rangle & ::= & \texttt{id} \end{array} \qquad \begin{array}{rcl} \langle \textit{exp} \rangle & ::= & \langle \textit{exp} \rangle \texttt{-} \langle \textit{term} \rangle \\ & | & \langle \textit{term} \rangle \\ \langle \textit{term} \rangle & ::= & \texttt{id} \end{array}$$



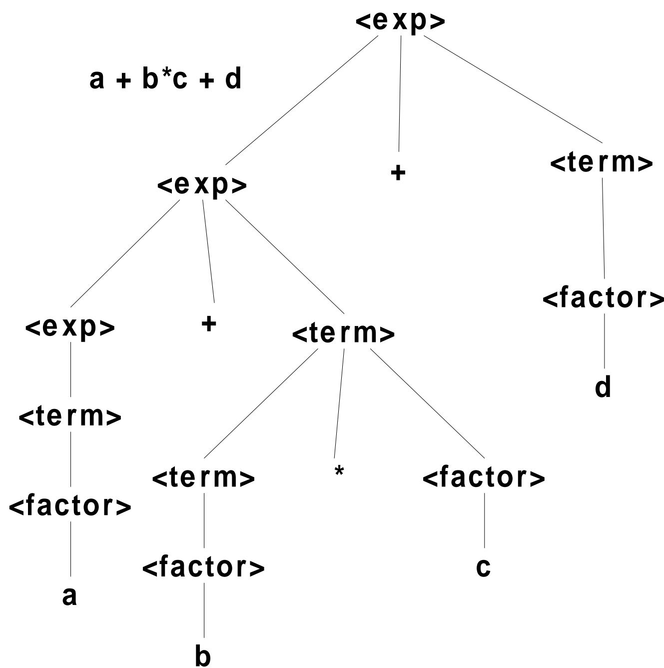
Precedence in Grammars

Consider expressions involving $+$ and $*$ only.

The grammar

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle exp \rangle + \langle term \rangle \\
 & | & \langle term \rangle \\
\langle term \rangle & ::= & \langle term \rangle * \langle factor \rangle \\
 & | & \langle factor \rangle \\
\langle factor \rangle & ::= & \texttt{id}
\end{array}
$$

uses different "levels" of recursion to group $*$ more tightly than $+$.



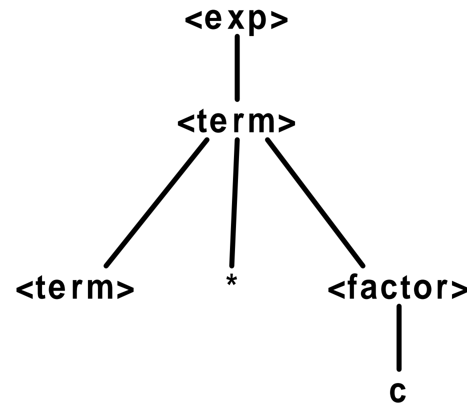
$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle exp \rangle + \langle term \rangle \\
 & | & \langle term \rangle \\
\langle term \rangle & ::= & \langle term \rangle * \langle factor \rangle \\
 & | & \langle factor \rangle \\
\langle factor \rangle & ::= & \texttt{id}
\end{array}
$$

Note that, with this grammar, we cannot get a wrong interpretation of $a + b * c$, such as $(a + b) * c$:

$a + b * c$

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle exp \rangle + \langle term \rangle \\
 & | & \langle term \rangle \\
\langle term \rangle & ::= & \langle term \rangle * \langle factor \rangle \\
 & | & \langle factor \rangle \\
\langle factor \rangle & ::= & \texttt{id}
\end{array}
$$

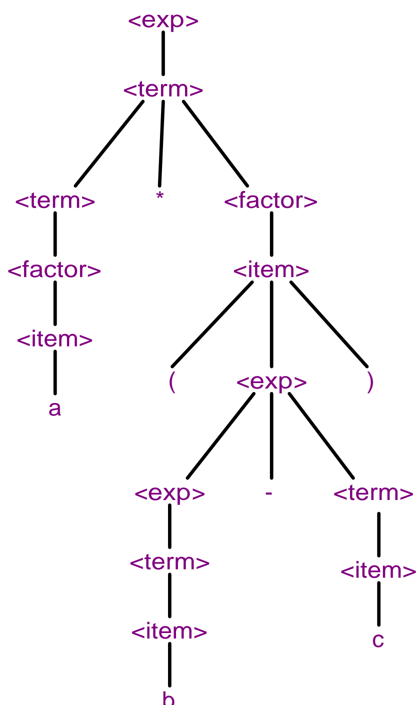
$$
\begin{array}{rcl}
\langle exp \rangle & \rightarrow & \langle term \rangle \\
 & \rightarrow & \langle term \rangle * \langle factor \rangle
\end{array}
$$



We're already stuck, because there's no way that either $\langle term \rangle$ or $\langle factor \rangle$ will ever expand to a string containing a "+".

A Full Expression Grammar

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle exp \rangle + \langle term \rangle \\
 & | & \langle exp \rangle - \langle term \rangle \\
 & | & \langle term \rangle \\
\langle term \rangle & ::= & \langle term \rangle * \langle factor \rangle \\
 & | & \langle term \rangle \ / \ \langle factor \rangle \\
 & | & \langle factor \rangle \\
\langle factor \rangle & ::= & \langle item \rangle ** \langle factor \rangle \\
 & | & \langle item \rangle \\
\langle item \rangle & ::= & \texttt{id}|\texttt{number} \\
 & | & (\ \langle exp \rangle \ )
\end{array}
$$



Extended BNF (EBNF)

Some convenient extensions to BNF form:

- $\{\ldots\}$ represents 0 or more repetitions
- $[\ldots]$ represents an optional part (0 or 1 occurrence)
- $(\ldots)$ used for grouping

Using EBNF, we can rewrite the grammar for statement sequences:

$$\begin{array}{rcl} \langle\mathit{stmt}\rangle & ::= & \{\ \langle\mathit{stmt\text{-}seq}\rangle\ \} \\ \langle\mathit{stmt\text{-}seq}\rangle & ::= & \langle\mathit{stmt}\rangle \\ & | & \langle\mathit{stmt}\rangle\ \langle\mathit{stmt\text{-}seq}\rangle \end{array}$$

as

$$\langle\mathit{stmt}\rangle \quad ::= \quad \text{'\{'}\ \{\langle\mathit{stmt\text{-}seq}\rangle\}\ \text{'\}'}$$

Repetition with separators is only slightly more complicated in EBNF:

Paramter list for a function call:

$$\begin{array}{rcl} \langle\mathit{params}\rangle & ::= & (\ \langle\mathit{paramlist1}\rangle\ ) \\ \langle\mathit{paramlist1}\rangle & ::= & \langle\mathit{exp}\rangle \\ & | & \langle\mathit{exp}\rangle\ ,\ \langle\mathit{paramlist1}\rangle \end{array}$$

in EBNF becomes

$$\langle\mathit{params}\rangle \quad ::= \quad (\ \langle\mathit{exp}\rangle\ \{\ ,\ \langle\mathit{exp}\rangle\}\ )$$

Of course, in most languages, we can supply an empty list `()` of function parameters:

$$\begin{array}{rcl} \langle\mathit{params}\rangle & ::= & (\ \langle\mathit{paramlist}\rangle\ ) \\ \langle\mathit{paramlist}\rangle & ::= & |\ \langle\mathit{paramlist1}\rangle \\ \langle\mathit{paramlist1}\rangle & ::= & \langle\mathit{exp}\rangle \\ & | & \langle\mathit{exp}\rangle\ ,\ \langle\mathit{paramlist1}\rangle \end{array}$$

in EBNF becomes

$$\langle\mathit{params}\rangle \quad ::= \quad (\ [\langle\mathit{exp}\rangle\ \{\ ,\ \langle\mathit{exp}\rangle\}]\ )$$

2.2 Recursive Descent Parsing

A simple parsing technique for (some) context-free grammars:

- For each nonterminal $\langle N\rangle$, we write a function

  ```
  bool N(Scanner& scanner);
  ```

  to recognize if the next sequence of tokens belongs to that non-terminal set.

- The body of `N(...)` is derived from the productions with $\langle N\rangle$ on the left.

Assume we have an appropriate token class...

```
struct Token {
  enum Kinds {integer, string, plus, ...};
    // depends on the language

  Kinds kind;
  string lexeme;
};
```

...and a scanner:

```
class Scanner {
public:
    Scanner(istream&);
    Token peek (int numTokens) const;
    bool match (Token::Kinds kind);
private:
  ⋮
};
```

- `peek(k)` shows us the kth next token.

  - Often, scanners can only "look ahead" one token

- `match(t)` checks to see if `peek(1).kind == t`.
  - If so, the next token is discarded and `match` returns true.
  - If not, `match` returns false and the scanner state is unchanged.

In recursive descent parsing, we write recognition functions to match each nonterminal against strings of tokens, and use `Scanner::match` to recognize a terminal.

A production like

$$\langle N\rangle ::= \langle S\rangle\ \mathtt{T}\ \langle U\rangle$$

is handled like this:

```
bool N(Scanner& scanner)
{
 ⋮
 return S(scanner)
        && scanner.match(Token::T)
        && U(scanner);
 ⋮
}
```

The tricky part is figuring out *which* production to use if there's more than 1 for $\langle N\rangle$.

For the expression grammar:

$$\begin{array}{rcl}
\langle exp\rangle & ::= & \langle exp\rangle + \langle term\rangle \\
 & | & \langle exp\rangle - \langle term\rangle \\
 & | & \langle term\rangle \\
\langle term\rangle & ::= & \langle term\rangle * \langle factor\rangle \\
 & | & \langle term\rangle\ /\ \langle factor\rangle \\
 & | & \langle factor\rangle \\
\langle factor\rangle & ::= & \langle item\rangle ** \langle factor\rangle \\
 & | & \langle item\rangle \\
\langle item\rangle & ::= & \mathtt{id}|\mathtt{number} \\
 & | & (\ \langle exp\rangle\ )
\end{array}$$

. . . we would have functions:

```
bool exp(Scanner& scanner);
bool term(Scanner& scanner);
bool factor(Scanner& scanner);
bool item(Scanner& scanner);
```

Let's look at $\langle item\rangle$:

$$\begin{array}{rcl}
\langle item\rangle & ::= & \mathtt{id}|\mathtt{number} \\
 & | & (\ \langle exp\rangle\ )
\end{array}$$

```
bool item(Scanner& scanner)
{
  Token t = scanner.peek(1);
  if (t.kind == Token::id)
    return scanner.match(Token::id);
  else if (t.kind == Token::number)
    return scanner.match(Token::number);
  else
    return scanner.match(Token::Lparen)
        && exp(scanner)
        && scanner.match(Token::Rparen);
}
```

Unfortunately, the rest aren't so easy:

$$\begin{array}{rcl}
\langle exp\rangle & ::= & \langle exp\rangle + \langle term\rangle \\
 & | & \langle exp\rangle - \langle term\rangle \\
 & | & \langle term\rangle
\end{array}$$

```
bool exp(Scanner& scanner)
{
  if (???)
    return exp(scanner)
           && scanner.match(Token::plus)
           && term(scanner);
  else if (???)
    return exp(scanner)
           && scanner.match(Token::minus)
           && term(scanner);
  else
    return term(scanner);
}
```

We don't know how many tokens ahead to peek for a "+" or "-"!

In general, recursive descent does not work with "left-recursive" grammars:

$$\langle N\rangle \quad ::= \quad \langle N\rangle\,\alpha\,\beta\ldots$$

But it still works well for processing many structured input sets.

$$\begin{array}{rcl}
\langle phoneDir\rangle & ::= & \langle blockList\rangle \\
\langle blocklist\rangle & ::= & \langle nameBlock\rangle \\
 & | & \langle nameBlock\rangle\ \langle blocklist\rangle \\
\langle nameBlock\rangle & ::= & \langle fullLine\rangle\ \langle partialLineList\rangle \\
\langle partialLineList\rangle & ::= & \\
 & | & \langle partialLine\rangle\ \langle partialLineList\rangle \\
\langle fullLine\rangle & ::= & \langle name\rangle\ \langle address\rangle \\
 & & \mathtt{spacer}\ \mathtt{phoneNum}
\end{array}$$

⟨*phoneDir*⟩ ::= ⟨*blockList*⟩

```
bool phoneDir (Scanner& scanner)
{
  return blocklist(scanner);
}
```

⟨*blocklist*⟩ ::= ⟨*nameBlock*⟩
| ⟨*nameBlock*⟩ ⟨*blocklist*⟩

```
bool blocklist (Scanner& scanner)
{
  if (nameBlock(scanner))
   {
    if (scanner.peek(1))
      return blocklist(scanner);
    else
      return true;
   }
  else
    return false;
}
```

⟨*nameBlock*⟩ ::= ⟨*fullLine*⟩ ⟨*partialLineList*⟩

```
bool nameBlock (Scanner& scanner)
{
  return fullLine(scanner)
    && partialLineList(scanner);
}
```

⟨*partialLineList*⟩ ::=
| ⟨*partialLine*⟩ ⟨*partialLineList*⟩

```
bool partialLineList (Scanner& scanner)
{
  if (scanner.peek(2) == Token::comma)
    return true;
  else
    return partialLine(scanner)
       && partialLineList(scanner);
}
```

⟨*fullLine*⟩ ::= ⟨*name*⟩ ⟨*address*⟩
`spacer` `phoneNum`

```
bool fullLine (Scanner& scanner)
{
  return name(scanner) &&
    address(scanner) &&
    scanner.match(Token::spacer) &&
    scanner.match(Token::phoneNum);
}
```

2.3 Miscellaneous notes

- Text uses `if-then-else` to explore ambiguity
- Syntax charts
- Many editors and Unix commands use RE's for text manipulation.
- Programs exist to translate RE's and CFG's into
  - compiler code
  - test data