3. Parsing

Oscar Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/

Roadmap

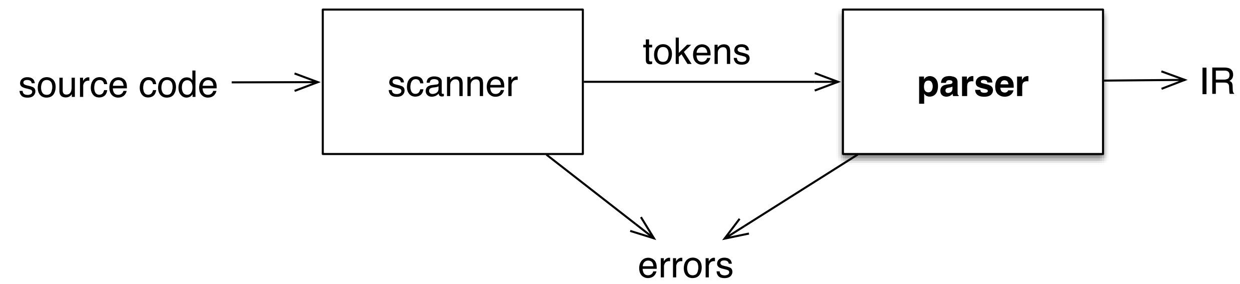
> Context-free grammars
> Derivations and precedence
> Top-down parsing
> Left-recursion
> Look-ahead
> Table-driven parsing

See, *Modern compiler implementation in Java* (Second edition), chapter 3.

Roadmap

> **Context-free grammars**
> Derivations and precedence
> Top-down parsing
> Left-recursion
> Look-ahead
> Table-driven parsing

The role of the parser



- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction

Syntax analysis

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

> Formally a CFG $G = (V_t, V_n, S, P)$, where:

- V_t is the set of *terminal* symbols in the grammar (i.e.,the set of tokens returned by the scanner)
- V_n, the *non-terminals*, are variables that denote sets of (sub)strings occurring in the language. These impose a structure on the grammar.
- *S* is the *goal symbol*, a distinguished non-terminal in V_n denoting the entire set of strings in L(G).
- P is a finite set of *productions* specifying how 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.

> The set $V = V_t \cup V_n$ is called the *vocabulary* of *G*

Notation and terminology

- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $U, V, W, \ldots \in V$
- $\alpha, \beta, \gamma, \ldots \in V^*$
- $u, v, w, \ldots \in V_t^*$

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *single-step derivation* using $A \rightarrow \gamma$
$\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps
If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G
$L(G) = \{ w \in V_t^* \mid S \Rightarrow^+ w \}$, w in L(G) is called a *sentence* of G

NB: $L(G) = \{ \beta \in V^* \mid S \Rightarrow^* \beta \} \cap V_t^*$

Syntax analysis

Grammars are often written in Backus-Naur form (BNF).

Example:

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

In a BNF for a grammar, we represent

1. non-terminals with <angle brackets> or CAPITAL LETTERS
2. terminals with `typewriter` font or <u>underline</u>
3. productions as in the example

This describes simple expressions over numbers and identifiers.

Scanning vs. parsing

Where do we draw the line?

```
term  ::=  [a-zA-Z] ( [a-zA-Z] | [0-9] )*
           | 0 | [1-9][0-9]*
op    ::=  + | – | * | /
expr  ::=  (term op)* term
```

Regular expressions:

—Normally used to classify identifiers, numbers, keywords …

—Simpler and more concise for tokens than a grammar

—More efficient scanners can be built from REs

CFGs are used to impose *structure*

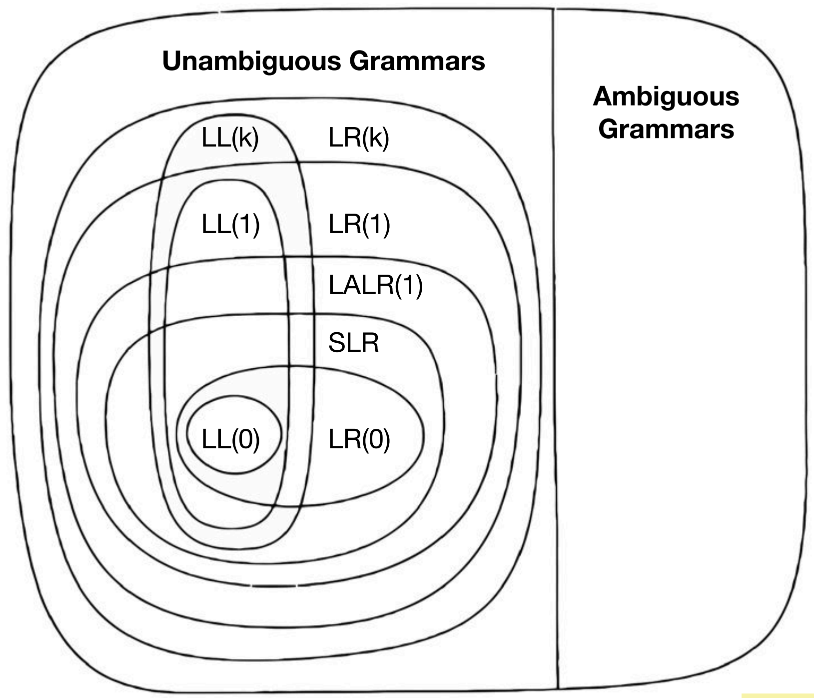
—Brackets: `()`, `begin … end`, `if … then … else`

—Expressions, declarations …

Factoring out lexical analysis simplifies the compiler

Syntactic analysis is complicated enough: grammar for C has around 200 productions.

Hierarchy of grammar classes



LL(*k*):

— **L**eft-to-right, **L**eftmost derivation, *k* tokens lookahead, *top-down*

LR(*k*):

— **L**eft-to-right, **R**ightmost derivation, *k* tokens lookahead, *bottom-up*

SLR:

— **S**imple **LR** (uses "follow sets")

LALR:

— **L**ook**A**head **LR** (uses "lookahead sets")

http://en.wikipedia.org/wiki/LL_parser …

LL(1) and LR(1) are "sweet spots"

Roadmap

- Context-free grammars
- **Derivations and precedence**
- Top-down parsing
- Left-recursion
- Look-ahead
- Table-driven parsing

Derivations

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

<goal>	$\Rightarrow$	**<expr>**
	$\Rightarrow$	**<expr>** <op> <expr>
	$\Rightarrow$	**<expr>** <op> <expr> <op> <expr>
	$\Rightarrow$	<id,x> **<op>** <expr> <op> <expr>
	$\Rightarrow$	<id,x> + **<expr>** <op> <expr>
	$\Rightarrow$	<id,x> + <num,2> **<op>** <expr>
	$\Rightarrow$	<id,x> + <num,2> * **<expr>**
	$\Rightarrow$	<id,x> + <num,2> * <id,y>

We have derived the sentence: `x + 2 * y`

We denote this *derivation* (or *parse*) as: <goal> $\Rightarrow^*$ `id + num * id`

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

Derivation

> At each step, we choose a non-terminal to replace.

—*This choice can lead to different derivations.*

> Two strategies are especially interesting:

1. *Leftmost derivation*: replace the leftmost non-terminal at each step
2. *Rightmost derivation*: replace the rightmost non-terminal at each step

The previous example was a leftmost derivation.

Rightmost derivation

For the string: `x + 2 * y`

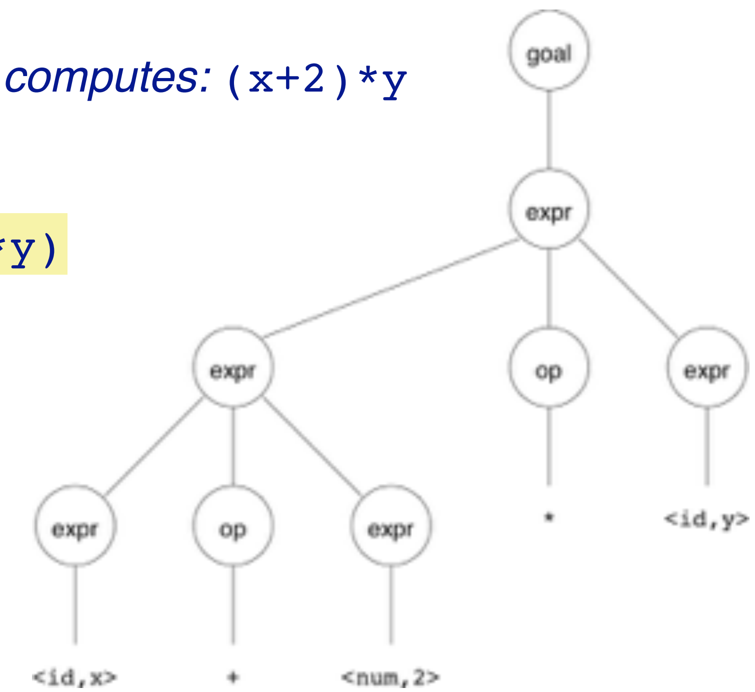
<goal>	⇒	**<expr>**
	⇒	<expr> <op> **<expr>**
	⇒	<expr> **<op>** <id,y>
	⇒	**<expr>** * <id,y>
	⇒	<expr> <op> **<expr>** * <id,y>
	⇒	<expr> **<op>** <num,2> * <id,y>
	⇒	**<expr>** + <num,2> * <id,y>
	⇒	<id,x> + <num,2> * <id,y>

Again we have: <goal> $\Rightarrow^*$ `id + num * id`

Precedence

Treewalk evaluation computes: `(x+2)*y`

Should be: `x+(2*y)`



Precedence

> **Our grammar has a problem:** 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>	::=	`num`
9.		|	`id`

> This grammar enforces a precedence on the derivation:

—terms *must* be derived from expressions

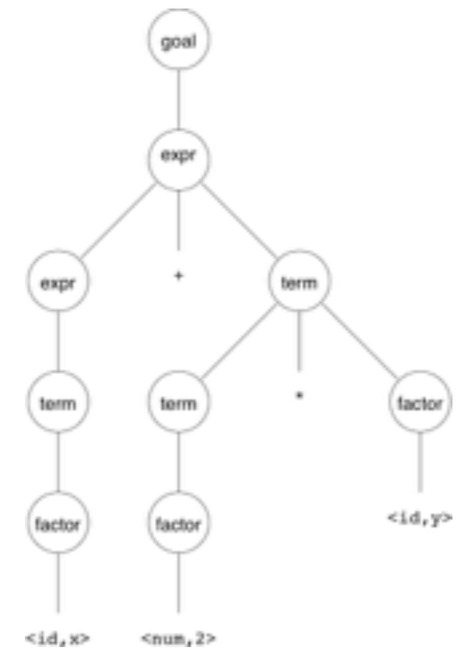
—forces the “correct” tree

Forcing the desired precedence

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

<goal>	$\Rightarrow$	**<expr>**
	$\Rightarrow$	<expr> + **<term>**
	$\Rightarrow$	<expr> + <term> * **<factor>**
	$\Rightarrow$	<expr> + **<term>** * <id,y>
	$\Rightarrow$	<expr> + **<factor>** * <id,y>
	$\Rightarrow$	**<expr>** + <num,2> * <id,y>
	$\Rightarrow$	**<term>** + <num,2> * <id,y>
	$\Rightarrow$	**<factor>** + <num,2> * <id,y>
	$\Rightarrow$	<id,x> + <num,2> * <id,y>

Again we have: <goal> $\Rightarrow^*$ `id + num * id`, but this time with the desired tree.



Ambiguity

If a grammar has more than one derivation for a single sentential form, then it is *ambiguous*

```
<stmt>    ::= if <expr> then <stmt>
            | if <expr> then <stmt> else <stmt>
            | …
```

> Consider: `if` E_1 `then` `if` E_2 `then` S_1 `else` S_2

— This has two derivations
— The ambiguity is purely grammatical
— It is called a *context-free ambiguity*

Resolving ambiguity

Ambiguity may be eliminated by rearranging the grammar:

<stmt>	::=	<matched>
	|	<unmatched>
<matched>	::=	`if` <expr> `then` <matched> `else` <matched>
	|	…
<unmatched>	::=	`if` <expr> `then` <stmt>
	|	`if` <expr> `then` <matched> `else` <unmatched>

This generates the same language as the ambiguous grammar, but applies the common sense rule:

—*match each* `else` *with the closest unmatched* `then`

Ambiguity

> Ambiguity is often due to confusion in the context-free specification. Confusion can arise from *overloading*, e.g.:

```
a = f(17)
```

> In many Algol-like languages, `f` could be a function or a subscripted variable.

> Disambiguating this statement *requires context:*

—need *values* of declarations

—not *context-free*

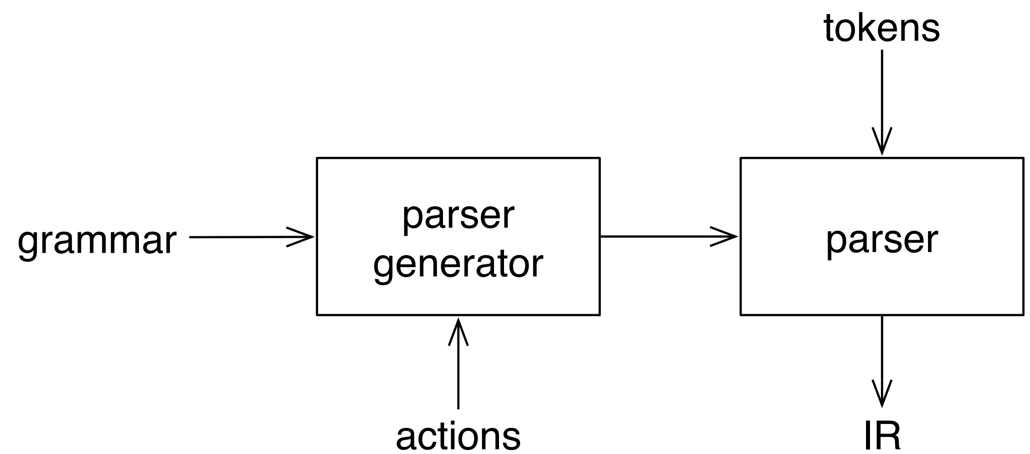
—really an issue of *type*

Rather than complicate parsing, we will handle this separately.

Roadmap

- Context-free grammars
- Derivations and precedence
- **Top-down parsing**
- Left-recursion
- Look-ahead
- Table-driven parsing

Parsing: the big picture



Our goal is a flexible parser generator system

Top-down versus bottom-up

> *Top-down parser (LL):*
—starts at the root of derivation tree and fills in
—picks a production and tries to match the input
—may require backtracking
—some grammars are backtrack-free (*predictive*)

> *Bottom-up parser (LR):*
—starts at the leaves and fills in
—starts in a state valid for legal first tokens
—as input is consumed, changes state to encode possibilities (*recognize valid prefixes*)
—uses a *stack* to store both state and sentential forms

Hand-written parsers are normally top-down.

Bottom-up parsers are normally built by parser generators.

Parser generators can be used to build either top-down or bottom-up parsers.

LL parsers are top-down. LR parsers are bottom-up.

Top-down parsing

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

To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string

1. At a node labeled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
2. When a terminal is added to the fringe that doesn't match the input string, *backtrack*
3. Find the next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1

$\Rightarrow$ should be guided by input string

Simple expression grammar

Recall our 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>	::=	`num`
9.		|	`id`

Consider the input string `x — 2 * y`

Top-down derivation

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 + ⟨term⟩	x ↑– 2 * y
–	⟨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 – ⟨term⟩	x ↑– 2 * y
–	id – ⟨term⟩	x – ↑2 * y
7	id – ⟨factor⟩	x – ↑2 * y
8	id – num	x – ↑2 * y
–	id – num	x – 2 ↑* y
–	id – ⟨term⟩	x – ↑2 * y
5	id – ⟨term⟩ * ⟨factor⟩	x – ↑2 * y
7	id – ⟨factor⟩ * ⟨factor⟩	x – ↑2 * y
8	id – num * ⟨factor⟩	x – ↑2 * y
–	id – num * ⟨factor⟩	x – 2 ↑* y
–	id – num * ⟨factor⟩	x – 2 * ↑y
9	id – num * id	x – 2 * ↑y
–	id – num * id	x – 2 * y ↑

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

The horizontal lines denote the backtracking points.

Whenever a token cannot be read, or input is left, then we must backtrack to an alternative rule.

NB: This example does not show how we pick which rule to expand! (be patient)

Roadmap

- > Context-free grammars
- > Derivations and precedence
- > Top-down parsing
- > **Left-recursion**
- > Look-ahead
- > Table-driven parsing

Non-termination

Another possible parse for `x – 2 * y`

Prod'n	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow$x – 2 * y
1	$\langle\text{expr}\rangle$	$\uparrow$x – 2 * y
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow$x – 2 * y
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow$x – 2 * y
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow$x – 2 * y
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow$x – 2 * y
2	$\cdots$	$\uparrow$x – 2 * y

If the parser makes the wrong choices, expansion doesn't terminate!

Left-recursion

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

Formally, a grammar is ***left-recursive*** if

$$\exists A \in V_n \text{ such that } A \Rightarrow^{+} A\alpha \text{ for some string } \alpha$$

Our simple expression grammar is left-recursive!

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

Eliminating left-recursion

To remove left-recursion, we can transform the grammar

```
<foo>  ::=  <foo> α
         |  β
```

```
<foo>  ::=  β <bar>
<bar>  ::=  α <bar>
         |  ε
```

NB: α and β do not start with <foo>!

Example

```
<foo>  ::=  <foo> α
        |   β
```

```
<foo>  ::=  β <bar>
<bar>  ::=  α <bar>
        |   ε
```

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

```
<expr>   ::=  <term> <expr´>
<expr´>  ::=  + <term> <expr´>
          |   – <term> <expr´>
          |   ε
<term>   ::=  <factor> <term´>
<term´>  ::=  * <factor> <term´>
          |   / <factor> <term´>
          |   ε
```

Example

Our long-suffering expression grammar :

1.	<goal>	::=	<expr>
2.	<expr>	::=	<term> <expr´>
3.	<expr´>	::=	+ <term> <expr´>
4.		|	– <term> <expr´>
5.		|	ε
6.	<term>	::=	<factor> <term´>
7.	<term´>	::=	* <factor> <term´>
8.		|	/ <factor> <term´>
9.		|	ε
10.	<factor>	::=	`num`
11.		|	`id`

With this grammar, a top-down parser will

- *terminate*
- *backtrack on some inputs*

Example

This cleaner grammar defines the same language:

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

It is:

- *right-recursive*
- *free of ε productions*

Unfortunately, it generates different associativity.
Same syntax, different meaning!

Roadmap

> Context-free grammars
> Derivations and precedence
> Top-down parsing
> Left-recursion
> **Look-ahead**
> Table-driven parsing

How much look-ahead is needed?

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

Do we need arbitrary look-ahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms
  - *Aho, Hopcroft, and Ullman, Problem 2.34 Parsing, Translation and Compiling, Chapter 4*

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): L**eft to right scan, **L**eft-most derivation, **1**-token look-ahead; and
- **LR(1): L**eft to right scan, **R**ight-most derivation, **1**-token look-ahead

Predictive parsing

Basic idea:

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

For some RHS $\alpha \in G$, define FIRST(α) as the set of tokens that appear first in some string derived from α

I.e., for some $w \in V_t^*$, $w \in$ FIRST(α) iff $\alpha \Rightarrow^* w\gamma$

Key property:

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

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

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

The example grammar has this property!

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 α common to two or more of its alternatives.
- if $\alpha \neq \varepsilon$ then replace all of the A productions

  $$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n$$

  with

  $$A \rightarrow \alpha A'$$

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

  where A' is fresh
- Repeat until no two alternatives for a single non-terminal have a common prefix.

Example

Consider our *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>	::=	`num`
9.		|	`id`

To choose between productions 2, 3, & 4, the parser must see past the `num` or `id` and look at the `+`, –, `*` or `/`.

$$\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \emptyset$$

This grammar *fails* the test.

I.e., they all have the same FIRST set, namely { num, id }
NB: *This grammar is right-associative.*

Example

Two non-terminals must be left-factored:

<expr>	::=	<term> + <expr>
	|	<term> – <expr>
	|	<term>
<term>	::=	<factor> * <term>
	|	<factor> / <term>
	|	<factor>

<expr>	::=	<term> <expr´>
<expr´>	::=	+ <expr>
	|	– <expr>
	|	ε
<term>	::=	<factor> <term´>
<term´>	::=	* <term>
	|	/ <term>
	|	ε

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>	::=	`num`
11.		|	`id`

Now, selection requires only a single token look-ahead.

NB: *This grammar is still right-associative.*

NB: This is a different grammar than the one we obtained by factoring out left recursion in the previous chapter.

Example derivation

	Sentential form	Input
–	⟨goal⟩	↑x – 2 * y
1	⟨expr⟩	↑x – 2 * y
2	⟨term⟩⟨expr′⟩	↑x – 2 * y
6	⟨factor⟩⟨term′⟩⟨expr′⟩	↑x – 2 * y
11	id⟨term′⟩⟨expr′⟩	↑x – 2 * y
–	id⟨term′⟩⟨expr′⟩	x ↑– 2 * y
9	idε ⟨expr′⟩	x ↑– 2
4	id– ⟨expr⟩	x ↑– 2 * y
–	id– ⟨expr⟩	x – ↑2 * y
2	id– ⟨term⟩⟨expr′⟩	x – ↑2 * y
6	id– ⟨factor⟩⟨term′⟩⟨expr′⟩	x – ↑2 * y
10	id– num⟨term′⟩⟨expr′⟩	x – ↑2 * y
–	id– num⟨term′⟩⟨expr′⟩	x – 2 ↑* y
7	id– num* ⟨term⟩⟨expr′⟩	x – 2 ↑* y
–	id– num* ⟨term⟩⟨expr′⟩	x – 2 * ↑y
6	id– num* ⟨factor⟩⟨term′⟩⟨expr′⟩	x – 2 * ↑y
11	id– num* id⟨term′⟩⟨expr′⟩	x – 2 * ↑y
–	id– num* id⟨term′⟩⟨expr′⟩	x – 2 * y↑
9	id– num* id⟨expr′⟩	x – 2 * y↑
5	id– num* id	x – 2 * y↑

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>	::=	`num`
11.		|	`id`

The next symbol determines each choice correctly.

Back to left-recursion elimination

- Given a left-factored CFG, to eliminate left-recursion:
  - if $\exists\ A \rightarrow A\alpha$ then replace all of the A productions

    $$A \rightarrow A\alpha \mid \beta \mid \ldots \mid \gamma$$

    with

    $$A \rightarrow NA'$$
    $$N \rightarrow \beta \mid \ldots \mid \gamma$$
    $$A' \rightarrow \alpha A' \mid \varepsilon$$

    where N and A' are fresh
  - Repeat until there are no left-recursive productions.

Generality

> ***Question:***
>
> — By *left factoring* and *eliminating left-recursion*, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token look-ahead?

> ***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:

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

S	:=	R0 | R1
R0	:=	a R0 b | 0
R1	:=	a R1 bb | 1

> Must look past an arbitrary number of *a*'s to discover the 0 or the 1 and so determine the derivation.

Recursive descent parsing

Now, we can produce a simple recursive descent parser from the (right- associative) grammar.

```
goal:
   token ← next_token();
   if (expr() = ERROR | token ≠ EOF) then
      return ERROR;
expr:
   if (term() = ERROR) then
      return ERROR;
   else return expr_prime();
expr_prime:
   if (token = PLUS) then
      token ← next_token();
      return expr();
   else if (token = MINUS) then
      token ← next_token();
      return expr();
   else return OK;
term:
   if (factor() = ERROR) then
      return ERROR;
   else return term_prime();
term_prime:
   if (token = MULT) then
      token ← next_token();
      return term();
   else if (token = DIV) then
      token ← next_token();
      return term();
   else return OK;
factor:
   if (token = NUM) then
      token ← next_token();
      return OK;
   else if (token = ID) then
      token ← next_token();
      return OK;
   else return ERROR;
```

Building the tree

> *One of the key jobs of the parser is to build an intermediate representation of the source code.*

> To build an abstract syntax tree, we can simply insert code at the appropriate points:
—factor() can stack nodes `id`, `num`
—term_prime() can stack nodes `*`, `/`
—term() can pop 3, build and push subtree
—expr_prime() can stack nodes `+`, `-`
—expr() can pop 3, build and push subtree
—goal() can pop and return tree

Roadmap

> Context-free grammars
> Derivations and precedence
> Top-down parsing
> Left-recursion
> Look-ahead
> **Table-driven parsing**

Non-recursive predictive parsing

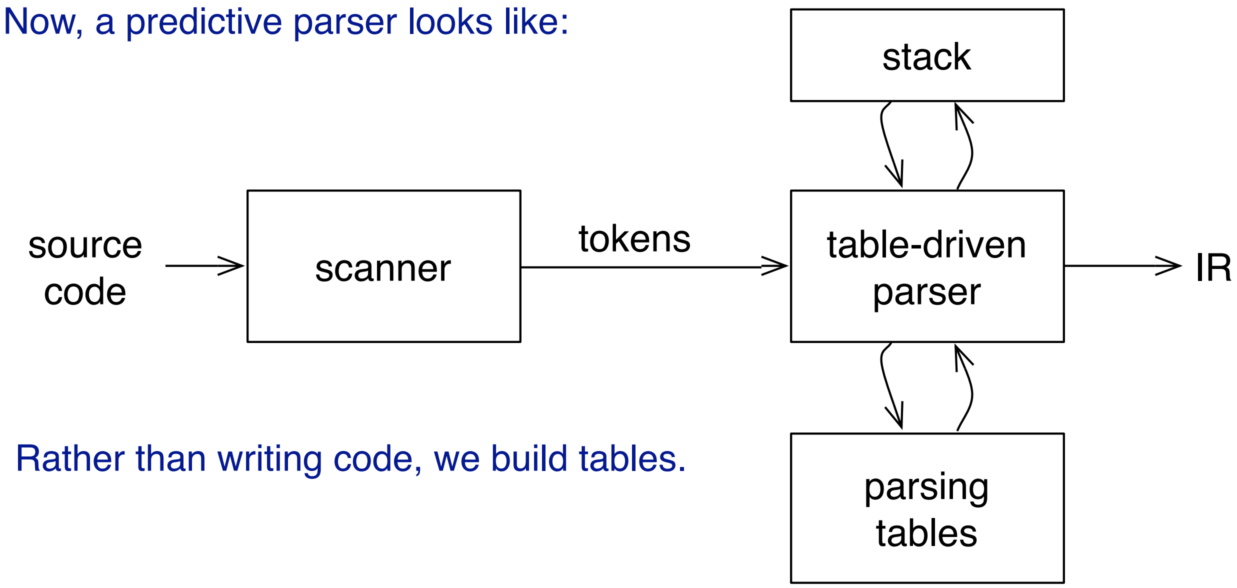
> Observation:
 —*Our recursive descent parser encodes state information in its run-time stack, or call stack.*

> Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

> This suggests other implementation methods:
 —explicit stack, hand-coded parser
 —stack-based, table-driven parser

Non-recursive predictive parsing

Now, a predictive parser looks like:

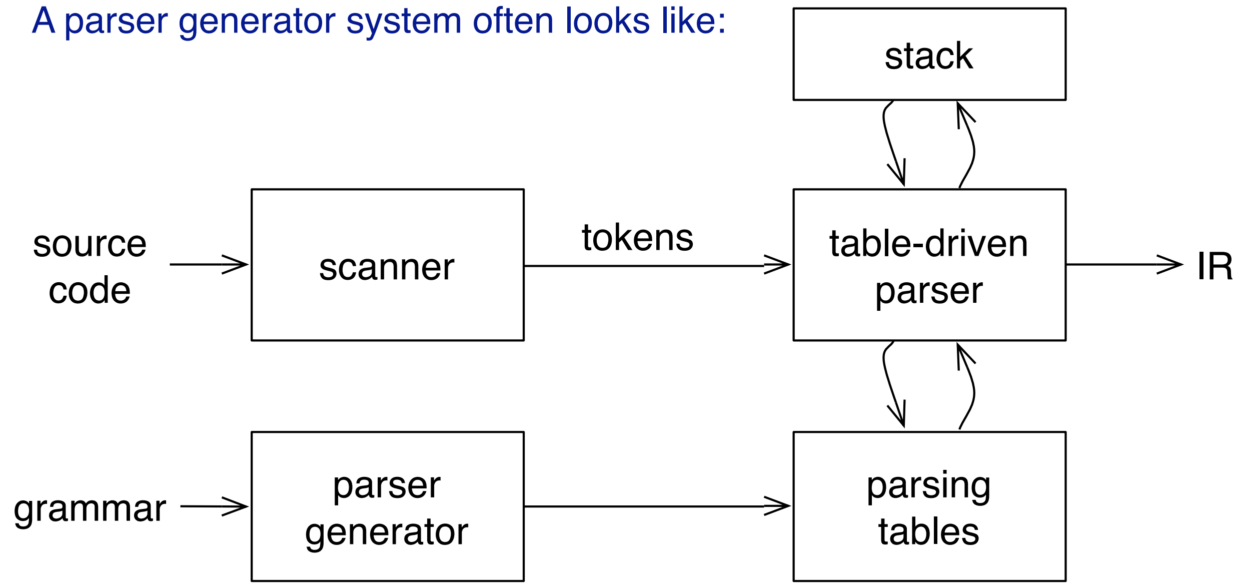


Rather than writing code, we build tables.

Building tables can be automated!

Table-driven parsers

A parser generator system often looks like:



This is true for both top-down (LL) and bottom-up (LR) parsers

Non-recursive predictive parsing

Input: a string *w* and a parsing table *M* for *G*

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
   X ← Stack[tos]
   if X is a terminal or EOF then
      if X = token then
         pop X
         token ← next_token()
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         push Yk,Yk-1,···,Y1
      else error()
until X = EOF
```


tos = “top of stack”
The top of the stack holds the current symbol (terminal or non-terminal) you are trying match.
The bottom of the stack, then, is the target. The lookahead tells you which rule to use to expand a NT, and then the top of stack is replaced by pushing all the symbols of the RHS of the rule.

Non-recursive predictive parsing

What we need now is a parsing table M.

Our expression grammar :

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>	::=	`num`
11.		|	`id`

Its parse table:

	`id`	`num`	+	−	*	/	$[†]
⟨goal⟩	1	1	–	–	–	–	–
⟨expr⟩	2	2	–	–	–	–	–
⟨expr′⟩	–	–	3	4	–	–	5
⟨term⟩	6	6	–	–	–	–	–
⟨term′⟩	–	–	9	9	7	8	9
⟨factor⟩	11	10	–	–	–	–	–

† we use $ to represent `EOF`

LL(1) grammars

Previous definition:

—A grammar G is LL(1) iff for all non-terminals A, each distinct pair of productions $A \to \beta$ and $A \to \gamma$ satisfy the condition $FIRST(\beta) \cap FIRST(\gamma) = \varnothing$

> But what if $A \Rightarrow^* \varepsilon$?

Revised definition:

—A grammar G is LL(1) iff for each set of productions
$A \to \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

1. $FIRST(\alpha_1)$, $FIRST(\alpha_2)$, …, $FIRST(\alpha_n)$ are pairwise disjoint
2. If $\alpha_i \Rightarrow^* \varepsilon$ then $FIRST(\alpha_j) \cap FOLLOW(A) = \varnothing$, $\forall\ 1 \leq j \leq n, i \neq j$

NB: If G is ε-free, condition 1 is sufficient

FOLLOW(A) must be disjoint from FIRST(a_j), else we do not know whether to go to a_j or to take a_i and skip to what follows.

FIRST

For a string of grammar symbols α, define FIRST(α) as:

- the set of terminal symbols that begin strings derived from α: $\{ a \in V_t \mid \alpha \Rightarrow^* a\beta \}$
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in$ FIRST(α)

FIRST(α) contains the set of tokens valid in the initial position in α.

To build FIRST(X):

1. If $X \in V_t$, then FIRST(X) is $\{ X \}$
2. If $X \rightarrow \varepsilon$ then add ε to FIRST(X)
3. If $X \rightarrow Y_1 Y_2 \ldots Y_k$
   a) Put FIRST(Y_1) — $\{\varepsilon\}$ in FIRST(X)
   b) $\forall i$: $1 < i \leq k$, if $\varepsilon \in$ FIRST(Y_1) $\cap \ldots \cap$ FIRST(Y_{i-1})
      (i.e., $Y_1 Y_2 \ldots Y_{i-1} \Rightarrow^* \varepsilon$)
      then put FIRST(Y_i) — $\{\varepsilon\}$ in FIRST(X)
   c) If $\varepsilon \in$ FIRST(Y_1) $\cap \ldots \cap$ FIRST(Y_k)
      then put ε in FIRST(X)

Repeat until no more additions can be made.

Straightforward recursive algorithm to build the FIRST set of a NT.

FOLLOW

- For a non-terminal A, define FOLLOW(A) as:
  - the set of terminals that can appear immediately to the right of A in some sentential form
  - I.e., a non-terminal's FOLLOW set specifies the tokens that can legally appear after it.
  - A terminal symbol has no FOLLOW set.
- To build FOLLOW(A):

1. Put $ in FOLLOW(<goal>)
2. If $A \rightarrow \alpha B \beta$:
   a) Put FIRST(β) – {ε} in FOLLOW(B)
   b) If $\beta = \varepsilon$ (i.e., $A \rightarrow \alpha B$) or $\varepsilon \in$ FIRST(β) (i.e., $\beta \Rightarrow^* \varepsilon$) then put FOLLOW(A) in FOLLOW(B)

Repeat until no more additions can be made

Nothing tricky here.

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. $\forall$ production $A \rightarrow \alpha$:
   a) $\forall a \in \mathrm{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A,a]$
   b) If $\varepsilon \in \mathrm{FIRST}(\alpha)$:
      I. *$\forall b \in \mathrm{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,b]$*
      II. *If $\$ \in \mathrm{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,\$]$*
2. Set each undefined entry of M to `error`

If $\exists M[A,a]$ with multiple entries then G is not LL(1).

NB: recall that $a, b \in V_t$, so $a, b \neq \varepsilon$

Example

Our long-suffering expression grammar:

$S \to E$
$E \to TE'$
$E' \to +E \mid -E \mid \varepsilon$
$T \to FT'$
$T' \to * T \mid / T \mid \varepsilon$
$F \to \texttt{num} \mid \texttt{id}$

	FIRST	FOLLOW
S	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E	$\{\texttt{num}, \texttt{id}\}$	$\{\$\}$
E'	$\{\varepsilon, +, -\}$	$\{\$\}$
T	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, \$\}$
T'	$\{\varepsilon, *, /\}$	$\{+, -, \$\}$
F	$\{\texttt{num}, \texttt{id}\}$	$\{+, -, *, /, \$\}$
id	$\{\texttt{id}\}$	—
num	$\{\texttt{num}\}$	—
$*$	$\{*\}$	—
$/$	$\{/\}$	—
$+$	$\{+\}$	—
$-$	$\{-\}$	—

	id	num	$+$	$-$	$*$	$/$	$\$$
S	$S \to E$	$S \to E$	—	—	—	—	—
E	$E \to TE'$	$E \to TE'$	—	—	—	—	—
E'	—	—	$E' \to +E$	$E' \to -E$	—	—	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	—	—	—	—	—
T'	—	—	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *T$	$T' \to /T$	$T' \to \varepsilon$
F	$F \to \texttt{id}$	$F \to \texttt{num}$	—	—	—	—	—

Properties of LL(1) grammars

1. No left-recursive grammar is LL(1)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. An ε–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example:

$$S \rightarrow aS \mid a$$

is not LL(1) because FIRST(aS) = FIRST(a) = { a }

$$S \rightarrow aS'$$

$$S' \rightarrow aS \mid \varepsilon$$

accepts the same language and is LL(1)

A grammar that is not LL(1)

<stmt>	::=	`if` <expr> `then` <stmt>
	|	`if` <expr> `then` <stmt> `else` <stmt>
	|	…

Left-factored:

<stmt>	::=	`if` <expr> `then` <stmt> <stmt´> | …
<stmt´>	::=	`else` <stmt> | ε

Now, FIRST(<stmt´>) = { ε, `else` }
Also, FOLLOW(<stmt´>) = { `else`, $}
But, FIRST(<stmt´>) ∩ FOLLOW(<stmt´>) = { `else` } ≠ ∅
On seeing `else`, conflict between choosing
<stmt´> ::= `else` <stmt> and <stmt´> ::= ε
⇒ grammar is not LL(1)!

Note that since <stmt> precedes <stmt’>, by recursion <stmt’> precedes <stmt’>, so FIRST(<stmt’>) is in FOLLOWS(<stmt’>).
NB: The fix, as before, is to put priority on <stmt´> ::= else <stmt> to associate else with closest previous then.

Error recovery

Key notion:

> For each non-terminal, construct a set of terminals on which the parser can synchronize

> When an error occurs looking for A, scan until an element of SYNC(A) is found

Building SYNC(A):

1. $a \in \text{FOLLOW}(A) \Rightarrow a \in \text{SYNC}(A)$
2. place keywords that start statements in SYNC(A)
3. add symbols in FIRST(A) to SYNC(A)

If we can't match a terminal on top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse

I.e., $\text{SYNC}(a) = V_t - \{a\}$

NB: popping the terminal means we matched it – since it wasn't really there, in effect we have inserted it

What you should know!

- ✎ *What are the key responsibilities of a parser?*
- ✎ *How are context-free grammars specified?*
- ✎ *What are leftmost and rightmost derivations?*
- ✎ *When is a grammar ambiguous? How do you remove ambiguity?*
- ✎ *How do top-down and bottom-up parsing differ?*
- ✎ *Why are left-recursive grammar rules problematic?*
- ✎ *How do you left-factor a grammar?*
- ✎ *How can you ensure that your grammar only requires a look-ahead of 1 symbol?*

Can you answer these questions?

- ✎ *Why is it important for programming languages to have a context-free syntax?*
- ✎ *Which is better, leftmost or rightmost derivations?*
- ✎ *Which is better, top-down or bottom-up parsing?*
- ✎ *Why is look-ahead of just 1 symbol desirable?*
- ✎ *Which is better, recursive descent or table-driven top-down parsing?*
- ✎ *Why is LL parsing top-down, but LR parsing is bottom up?*