Parser 1

Ronghui Gu
Spring 2019
Columbia University

* Course website: https://www.cs.columbia.edu/~rgu/courses/4115/spring2019
** These slides are borrowed from Prof. Edwards.
The Big Picture
How do we describe/construct a program?
Solution: Use a Discrete Combinatorial System

Use combinations of a small number of things to represent (exponentially) many different things.
How do we combine characters into words?
int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

front-end

middle-end

back-end

Scanner
How do we combine words into sentences?
int avg (int a, int b) ...

Lexical Analysis

Syntax Analysis

Semantic Analysis

Intermediate Code Generation

Optimization

Code Generation

010110101...
### Choices: CS Research Jargon Generator

Pick one from each column

<table>
<thead>
<tr>
<th>an integrated</th>
<th>mobile</th>
<th>network</th>
</tr>
</thead>
<tbody>
<tr>
<td>a parallel</td>
<td>functional</td>
<td>preprocessor</td>
</tr>
<tr>
<td>a virtual</td>
<td>programmable</td>
<td>compiler</td>
</tr>
<tr>
<td>an interactive</td>
<td>distributed</td>
<td>system</td>
</tr>
<tr>
<td>a responsive</td>
<td>logical</td>
<td>interface</td>
</tr>
<tr>
<td>a synchronized</td>
<td>digital</td>
<td>protocol</td>
</tr>
<tr>
<td>a balanced</td>
<td>concurrent</td>
<td>architecture</td>
</tr>
<tr>
<td>a virtual</td>
<td>knowledge-based</td>
<td>database</td>
</tr>
<tr>
<td>a meta-level</td>
<td>multimedia</td>
<td>algorithm</td>
</tr>
</tbody>
</table>

E.g., “a responsive knowledge-based preprocessor.”

http://www.cs.purdue.edu/homes/dec/essay.topic.generator.html
hey Jude

make it bad → take a sad song and make it better

don't
be afraid → you were made to go out and get her
let me down → you have found her, now go and get her

remember to
let her into your heart
let her under your skin

then you
con start
begin → to make it better

better better better better better waaaaa

na

http://loveallthis.tumblr.com/post/506873221
How about more structured collections of things?

The boy eats hot dogs.
The dog eats ice cream.
Every happy girl eats candy.
A dog eats candy.
The happy happy dog eats hot dogs.

Pinker, *The Language Instinct*
If the boy eats hot dogs, then the girl eats ice cream.

Either the boy eats candy, or every dog eats candy.

Does this work?
Want to “remember” whether it is an “either-or” or “if-then” sentence. Only solution: duplicate states.
Automata in the form of Production Rules

Problem: automata do not remember where they’ve been

\[
\begin{align*}
S & \rightarrow \text{Either } A \\
S & \rightarrow \text{If } A \\
A & \rightarrow \text{the } B \\
A & \rightarrow \text{the } C \\
A & \rightarrow \text{a } B \\
A & \rightarrow \text{a } C \\
A & \rightarrow \text{every } B \\
A & \rightarrow \text{every } C \\
B & \rightarrow \text{happy } B \\
B & \rightarrow \text{happy } C \\
C & \rightarrow \text{boy } D \\
C & \rightarrow \text{girl } D \\
C & \rightarrow \text{dog } D \\
D & \rightarrow \text{eats } E \\
E & \rightarrow \text{hot dogs } F \\
E & \rightarrow \text{ice cream } F \\
E & \rightarrow \text{candy } F
\end{align*}
\]
Context-Free Grammars have the ability to “call subroutines:”

- \[ S \rightarrow \text{Either } P, \text{ or } P. \]  
- \[ S \rightarrow \text{If } P, \text{ then } P. \]  
- \[ P \rightarrow A \ H \ N \text{ eats } O \]  
- \[ A \rightarrow \text{the} \]  
- \[ A \rightarrow \text{a} \]  
- \[ A \rightarrow \text{every} \]  
- \[ H \rightarrow \text{happy } H \]  
- \[ H \rightarrow \varepsilon \]  
- \[ N \rightarrow \text{boy} \]  
- \[ N \rightarrow \text{girl} \]  
- \[ N \rightarrow \text{dog} \]  
- \[ O \rightarrow \text{hot dogs} \]  
- \[ O \rightarrow \text{ice cream} \]  
- \[ O \rightarrow \text{candy} \]  

Exactly two \( P \)s  
One each of \( A, H, N, \) and \( O \)  
\( H \) is “happy” zero or more times
n 0’s followed by n 1’s, e.g., 000111, 01

\[ S \rightarrow 0 \ S \ 1. \]
\[ S \rightarrow \epsilon. \]
Constructing Grammars and Ocamlyacc
Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

\[ 2 \times 3 + 4 \Rightarrow \]

Goal: verify the syntax of the program, discard irrelevant information, and “understand” the structure of the program. Parentheses and most other forms of punctuation removed.
I shot an elephant in my pajamas
The Dangling Else Problem

Who owns the *else*?

```c
if (a) if (b) c(); else d();
```

```
stmt : IF expr THEN stmt
    | IF expr THEN stmt ELSE stmt
```

Problem comes after matching the first statement. Question is whether an “else” should be part of the current statement or a surrounding one since the second line tells us “stmt ELSE” is possible.
The Dangling Else Problem

Should this be

\[
\begin{array}{c}
\text{if} \\
\text{a} \\
\text{b} \\
\text{c()} \\
\text{d()}
\end{array}
\]

or

\[
\begin{array}{c}
\text{if} \\
\text{a} \\
\text{b} \\
\text{c()}
\end{array}
\]

Grammars are usually ambiguous; manuals give disambiguating rules such as C’s:

As usual the “else” is resolved by connecting an else with the last encountered elseless if.
The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling “then” (“dstmt”) and those that do not (“cstmt”). A statement may be either, but the statement just before an “else” must not have a dangling clause because if it did, the “else” would belong to it.

```
stmt : dstmt
    | cstmt

dstmt : IF expr THEN stmt
    | IF expr THEN cstmt ELSE dstmt

cstmt : IF expr THEN cstmt ELSE cstmt
    | other statements
```

```
if (a) if (b) c(); else d();
```
Another Solution to the Dangling Else Problem

We are effectively carrying an extra bit of information during parsing: whether there is an open “then” clause. Unfortunately, duplicating rules is the only way to do this in a context-free grammar.
Another Solution to the Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```plaintext
if a < b then a else b fi ;
```

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.
Ambiguity can be a problem in expressions. Consider parsing

$$3 - 4 \times 2 + 5$$

with the grammar

$$e \rightarrow e + e \mid e - e \mid e \times e \mid e / e \mid N$$
Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions
Like you were taught in elementary school:
“My Dear Aunt Sally”
Mnemonic for multiplication and division before addition and subtraction.
Operator Precedence

Defines how “sticky” an operator is.

\[ 1 \times 2 + 3 \times 4 \]

* at higher precedence than +:

\[(1 \times 2) + (3 \times 4)\]

+ at higher precedence than *:

\[1 \times (2 + 3) \times 4\]
Associativity

Whether to evaluate left-to-right or right-to-left

Most operators are left-associative

\[
1 - 2 - 3 - 4
\]

left associative

\[
((1 - 2) - 3) - 4
\]

right associative

\[
1 - (2 - (3 - 4))
\]
Fixing Ambiguous Grammars

A grammar specification:

```
expr :
    expr PLUS expr
| expr MINUS expr
| expr TIMES expr
| expr DIVIDE expr
| NUMBER
```

Ambiguous: no precedence or associativity.

Ocamlyacc’s complaint: “16 shift/reduce conflicts.”

```
1 * 2 + 3?
```
Assigning Precedence Levels

Split into multiple rules, one per level

expr : expr PLUS expr
| expr MINUS expr
| term

term : term TIMES term
| term DIVIDE term
| atom

atom : NUMBER

Still ambiguous: associativity not defined

Ocamlyacc’s complaint: “8 shift/reduce conflicts.”

1 * 2 + 3
1 * 2 * 3?
Assigning Associativity

Make one side the next level of precedence

```plaintext
expr : expr PLUS term
   | expr MINUS term
   | term

term : term TIMES atom
     | term DIVIDE atom
     | atom

atom : NUMBER
```

This is left-associative.

No shift/reduce conflicts.

```
1 * 2 * 3
```
Ocamlyacc Specifications

{%
(* Header: verbatim OCaml; optional *)
%
/* Declarations: tokens, precedence, etc. */
%
/* Rules: context-free rules */
%
(* Trailer: verbatim OCaml; optional *)
%}
Declarations

- `%token symbol …
  Define symbol names (exported to .mli file)
- `%token < type > symbol …
  Define symbols with attached attribute (also exported)
- `%start symbol …
  Define start symbols (entry points)
- `%type < type > symbol …
  Define the type for a symbol (mandatory for start)
- `%left symbol …
- `%right symbol …
- `%nonassoc symbol …
  Define precedence and associativity for the given symbols, listed in order from lowest to highest precedence
nonterminal :  
  symbol ... symbol { semantic-action }  
  | ...  
  | symbol ... symbol { semantic-action }

- *nonterminal* is the name of a rule, e.g., “program,” “expr”
- *symbol* is either a terminal (token) or another rule
- *semantic-action* is OCaml code evaluated when the rule is matched
- In a *semantic-action*, $1, 2, ...$ returns the value of the first, second, ... symbol matched
- A rule may include “%prec symbol” to override its default precedence
An Example .mly File

```ml
%token <int> INT
%token PLUS MINUS TIMES DIV LPAREN RPAREN EOL

%left PLUS MINUS /* lowest precedence */
%left TIMES DIV
%nonassoc UMINUS /* highest precedence */

%start main /* the entry point */
%type <int> main

main:
   expr EOL { $1 }

expr:
   INT { $1 }
   | LPAREN expr RPAREN { $2 }
   | expr PLUS expr { $1 + $3 }
   | expr MINUS expr { $1 - $3 }
   | expr TIMES expr { $1 * $3 }
   | expr DIV expr { $1 / $3 }
   | MINUS expr %prec UMINUS { - $2 }
```