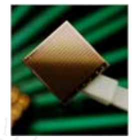
COMP 181

Lecture 5

Parsing

September 19, 2006

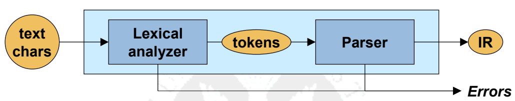
Prelude



- Microprocessor news:
  - Semiconductor with lasers
  - So what?
- Chip-to-chip communication is costly
  - **On chip:** registers, L1 and L2 cache – 1 to 8 cycles
  - ***Off chip:*** main memory – up to 150 cycles
- What's so great about optical communication?
  - Speed of light
  - No grounding, crosstalk, resistance, capacitance, etc.
  - Wave division multiplexing
    - 2.6 terabits/sec – equivalent to 30 million phone calls

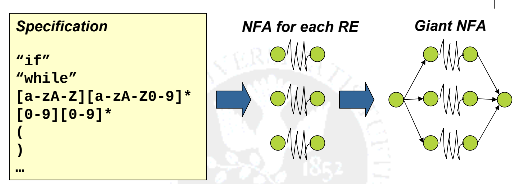
---

Where are we?



- Lexical analyzer
  - Reads characters one at a time
  - Produces a stream of tokens
    <kind, value>
- Automatically generated…

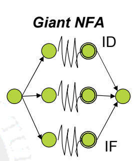
Building a lexer



- Language: `if` | `while` | `[a-zA-Z][a-zA-Z0-9]*` | `[0-9][0-9]*`…
- **Problem**:
  - Giant NFA either accepts or rejects a one token
  - We need to ***partition*** a string, and indicate the kind

---

Partitioning



- ***Input***: stream of characters

  $\mathbf{x_0, x_1, x_2, x_3, \ldots, x_n}$

  - Annotate the NFA
    - Remember the accepting state of each RE
    - Annotate with the kind of token
  - Does giant NFA accept some substring $\mathbf{x_0 \ldots x_i}$?
    - Return substring and kind of token
    - Restart the NFA at $\mathbf{x_{i+1}}$

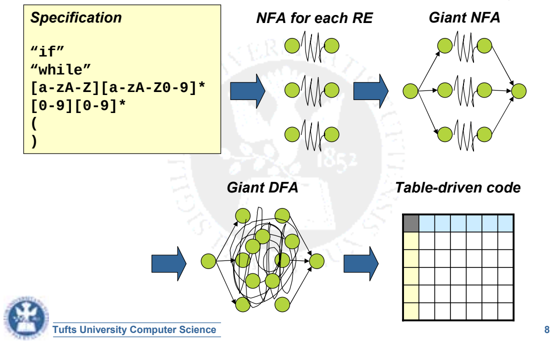
Partitioning problems

- Matching is ambiguous
  - **Example**: "**foo+3**"
  - We want <foo>,<+>,<3>
  - But: <f>,<oo>,<+>,<3> also works with our NFA
    - Can end the identifier anywhere
    - Note: "foo+" does not satisfy NFA
- Solution: ***"maximal munch"***
  - Choose the longest substring that is accepted
  - Must look at the next character to decide -- ***lookahead***
  - Keep munching until no transition on lookahead

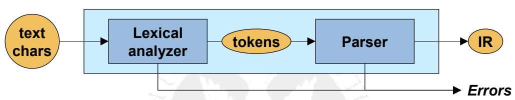
More problems

- Some strings satisfy multiple REs
  - **Example**: “`new foo`”
  - <new> could be an identifier or a keyword
- **Solution**: rank the REs
  - First, use maximal munch
  - Second, if substring satisfies two REs, choose the one with higher rank
  - Order is important in the specification
  - Put keywords first!

Building a lexer



Next step



- ***Parsing:*** Organize tokens into “sentences”
  - Do tokens conform to language ***syntax*** ?
  - ***Good news:*** token types are just numbers
  - ***Bad news:*** language syntax is fundamentally more complex than lexical specification
  - ***Good news:*** we can still do it in linear time in most cases

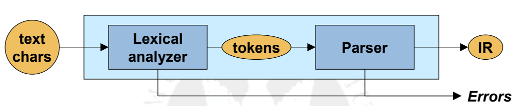
Parsing introduction

- Is the following sentence grammatically correct:

  The horse ran past the barn fell
- Why?
  - We can use run as a transitive verb
    - I ran the horse past the barn
    - The horse that was ran past the barn
  - Structure
    - Subject: the horse ran past the barn
    - Verb: fell

Hopefully, parsing programming languages won’t be this hard!

Parsing



- Parser
  - Reads tokens from the scanner
  - Checks organization of tokens against a ***grammar***
  - Constructs a ***derivation***
  - Derivation drives construction of IR

Study of parsing

- Discovering the derivation of a sentence
  - “Diagramming a sentence” in grade school
  - Formalization:
    - Mathematical model of syntax – a grammar G
    - Algorithm for testing membership in L(G)
- Roadmap:
  - Context-free grammars
  - Top-down parsers

    Ad hoc, often hand-coded, recursive decent parsers
  - Bottom-up parsers

    Automatically generated LR parsers

Specifying syntax with a grammar

- Limitations of regular expressions
  - Later in lecture, and in homework *(yes, homework)*
  - Need something more powerful
  - Still want formal specification *(for automation)*
- Context-free grammar
  - Set of rules for generating sentences
  - Expressed in Backus-Naur form (BNF)

Context-free grammar

"produces" or "generates"

- Example:

#	Production rule
1	*sheepnoise* → *sheepnoise* baa
2	| baa

Alternative (shorthand)

- Formally: ***context-free grammar*** is
  - **G** = (s, N, T, P)
  - ***T*** : set of terminals *(provided by scanner)*
  - ***N*** : set of non-terminals *(represent structure)*
  - $s \in N$: start or goal symbol
  - $P : N \rightarrow (N \cup T)^*$: set of production rules

Language L(G)

- Language L(G)

  L(G) is all sentences generated from start symbol
- Generating sentences
  - Use productions as ***rewrite rules***
  - Start with goal (or start) symbol – a non-terminal
  - Choose a non-terminal and "expand" it to the right-hand side of one of its productions
  - Only terminal symbols left → sentence in L(G)
  - Intermediate results known as ***sentential forms***

Examples

- Grammar:

#	Production rule
1	*sheepnoise* → *sheepnoise* baa
2	| baa

Rule	Sentential form
-	*sheepnoise*
2	baa

Rule	Sentential form
-	*sheepnoise*
1	*sheepnoise* baa
2	baa baa

Rule	Sentential form
-	*sheepnoise*
1	*sheepnoise* baa
1	*sheepnoise* baa baa
2	baa baa baa

Sheep noises aren't that interesting for compilers…

Better example

- Language of expressions
  - Numbers and identifiers
  - Allow different binary operators
  - Arbitrary nesting of expressions

#	Production rule
1	*expr* → *expr* *op* *expr*
2	| `number`
3	| `identifier`
4	*op* → +
5	| -
6	| *
7	| /

Expressions can consist of other expressions connected by operators

Numbers and identifiers can fill in any of the positions in the overall expression

Language of expressions

- What's in this language?

#	Production rule
1	*expr* → *expr* *op* *expr*
2	| `number`
3	| `identifier`
4	*op* → +
5	| -
6	| *
7	| /

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	<id,x> *op expr*
5	<id,x> - *expr*
1	<id,x> - *expr op expr*
2	<id,x> - <num,2> *op expr*
6	<id,x> - <num,2> * *expr*
3	<id,x> - <num,2> * <id,y>

⇨ We can build the string "**x - 2 * y**"

This string is in the language

Derivations

- Using grammars
  - A sequence of rewrites is called a ***derivation***
  - Discovering a derivation for a string is ***parsing***
- Different derivations are possible
  - At each step we can choose any non-terminal
  - ***Rightmost derivation***: always choose right NT
  - ***Leftmost derivation***: always choose left NT

  (Other "random" derivations – not of interest)

Left vs right derivations

- Two derivations of "**x – 2 * y**"

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Left-most derivation

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Right-most derivation

Derivations and parse trees

- Two different derivations
  - Both are correct
  - Do we care which one we use?
- Represent derivation as a ***parse tree***
  - Leaves are terminal symbols
  - Inner nodes are non-terminals
  - To depict production $\alpha \rightarrow \beta\ \gamma\ \delta$ show nodes β,γ,δ as children of α

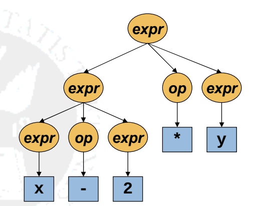
➡ Tree is used to build internal representation

Example (I)

Right-most derivation

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Parse tree



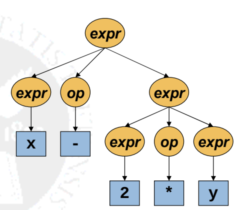
- Problem: evaluates as **(x – 2) * y**

Example (I)

Left-most derivation

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

Parse tree



- Solution: evaluates as **x – (2 * y)**

Derivations and precedence

- **Problem**:
  - Two different valid derivations
  - Shape of tree implies its meaning
  - One captures semantics we want – ***precedence***
- Can we express precedence in grammar?
  - Notice: operations deeper in tree evaluated first
  - Idea: add an intermediate production
    - New production isolates different levels of precedence
    - Force higher precedence "deeper" in the grammar

Adding precedence

- Two levels:

Level 1: lower precedence – higher in the tree

Level 2: higher precedence – deeper in the tree

#	Production rule			
1	*expr*	→	*expr*	+ *term*
2		|	*expr*	- *term*
3		|	*term*	
4	*term*	→	*term*	* *factor*
5		|	*term*	/ *factor*
6		|	*factor*	
7	*factor*	→	<u>number</u>	
8		|	<u>identifier</u>	

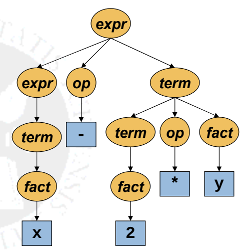
- Observations:
  - Larger: requires more rewriting to reach terminals
  - Produces same parse tree under both left and right derivations

Expression example

Right-most derivation

Rule	Sentential form
-	*expr*
2	*expr* - *term*
4	*expr* - *term* * *factor*
8	*expr* - *term* * <id,y>
6	*expr* - *factor* * <id,y>
7	*expr* - <num,2> * <id,y>
3	*term* - <num,2> * <id,y>
6	*factor* - <num,2> * <id,y>
8	<id,x> - <num,2> * <id,y>

Parse tree



Now right derivation yields `x – (2 * y)`

Typical patterns

- One or more:

#	Production rule
1	*list* → *element list*
2	| *element*

- Zero or more:

#	Production rule
1	*list* → *element list*
2	|

- Comma separated list:

#	Production rule
1	*list* → *element* , *list*
2	| *element*

C expressions

```
primary.expression:            /* P */ /* 6.3.1 EXTENDED */
        | constant
        | string.literal.list
        | '(' expression ')'
        ;

postfix.expression:            /* P */ /* 6.3.2 CLARIFICATION */
          primary.expression
        | postfix.expression '[' expression ']'
        | postfix.expression '(' ')'
        | postfix.expression '(' argument.expression.list ')'
        | postfix.expression '.' identifier
        | postfix.expression ctokARROW identifier
        | postfix.expression ctokICR
        | postfix.expression ctokDECR
        ;
```


C expressions

```
argument.expression.list:       /* P */ /* 6.3.2 */
          assignment.expression
        | argument.expression.list ',' assignment.expression
        ;

unary.expression:               /* P */ /* 6.3.3 */
          postfix.expression
        | ctokICR unary.expression
        | ctokDECR unary.expression
        | unary.operator cast.expression
        | ctokSIZEOF unary.expression
        | ctokSIZEOF '(' type.name ')'
        ;

unary.operator:                 /* P */ /* 6.3.3 */
          '&'  | '*'  | '+'  | '-'  | '~'  | '!'
        ;
```


C expressions

```
cast.expression:               /* P */ /* 6.3.4 */
          unary.expression
        | '(' type.name ')' cast.expression
        ;

multiplicative.expression:     /* P */ /* 6.3.5 */
          cast.expression
        | multiplicative.expression '*' cast.expression
        | multiplicative.expression '/' cast.expression
        | multiplicative.expression '%' cast.expression
        ;

additive.expression:           /* P */ /* 6.3.6 */
          multiplicative.expression
        | additive.expression '+' multiplicative.expression
        | additive.expression '-' multiplicative.expression
        ;
```

C expressions

```
shift.expression:              /* P */ /* 6.3.7 */
          additive.expression
        | shift.expression ctokLS additive.expression
        | shift.expression ctokRS additive.expression
        ;

relational.expression:         /* P */ /* 6.3.8 */
          shift.expression
        | relational.expression '<' shift.expression
        | relational.expression '>' shift.expression
        | relational.expression ctokLE shift.expression
        | relational.expression ctokGE shift.expression
        ;

equality.expression:           /* P */ /* 6.3.9 */
          relational.expression
        | equality.expression ctokEQ relational.expression
        | equality.expression ctokNE relational.expression
        ;
```


C expressions

```
AND.expression:                /* P */ /* 6.3.10 */
          equality.expression
        | AND.expression '&' equality.expression
        ;

inclusive.OR.expression:       /* P */ /* 6.3.12 */
          exclusive.OR.expression
        | inclusive.OR.expression '|' exclusive.OR.expression
        ;

logical.AND.expression:        /* P */ /* 6.3.13 */
          inclusive.OR.expression
        | logical.AND.expression ctokANDAND inclusive.OR.expression
        ;

logical.OR.expression:         /* P */ /* 6.3.14 */
          logical.AND.expression
        | logical.OR.expression ctokOROR logical.AND.expression
        ;
```


C expressions

```
conditional.expression:        /* P */ /* 6.3.15 */
      logical.OR.expression
    | logical.OR.expression '?' expression ':' conditional.expression
    ;

assignment.expression:         /* P */ /* 6.3.16 */
      conditional.expression
    | unary.expression assignment.operator assignment.expression
    ;

expression:                    /* P */ /* 6.3.17 */
      assignment.expression
    | expression ',' assignment.expression
    ;
```


Error productions

- How to provide useful error information?
- **Idea**: add productions for common errors

#	*Production rule*
1	*expr* → *expr op expr*
2	| `number`
3	| `identifier`
4	| *expr op error*

- Special "error" token – used in yacc/bison
- Emit message:

  "binary operation missing operand"

Ambiguity

- Original example has another problem:

#	*Production rule*
1	*expr* → *expr op expr*
2	| `number`
3	| `identifier`
4	*op* → +
5	| -
6	| *
7	| /

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
1	*expr op expr op expr*
3	*<id, x> op expr op expr*
5	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

- Multiple leftmost derivations – hard to automate
- Such a grammar is called ***ambiguous***

Ambiguous grammars

- A grammar is ambiguous *iff*:
  - There are multiple leftmost or multiple rightmost derivations for a single sentential form
  - ***Note***: leftmost and rightmost derivations may differ, even in an unambiguous grammar
  - Intuitively:
    - We can choose different non-terminals to expand
    - But each non-terminal should lead to a unique set of terminal symbols
- Classic example: if-then-else ambiguity

If-then-else

- Grammar:

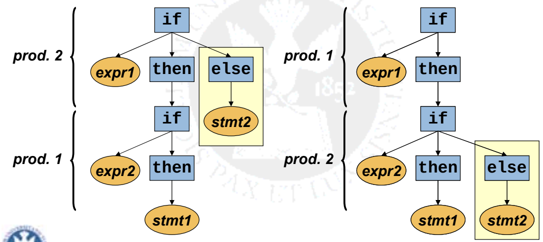
#	*Production rule*
1	*stmt* → `if` *expr* `then` *stmt*
2	| `if` *expr* `then` *stmt* `else` *stmt*
3	| *…other statements…*

- Problem: nested if-then-else statements
  - Each one may or may not have **`else`**
  - How to match each **`else`** with **`if`**

If-then-else ambiguity

- Sentential form with two derivations:

  `if` ***expr1*** **`then if`** ***expr2*** **`then`** ***stmt1*** **`else`** ***stmt2***



Removing ambiguity

- Restrict the grammar
  - Choose a rule: “else” matches innermost “if”
  - Codify with new productions

#	*Production rule*
1	*stmt* → `if` *expr* `then` *stmt*
2	| `if` *expr* `then` *withelse* `else` *stmt*
3	| *…other statements…*
4	*withelse* → `if` *expr* `then` *withelse* `else` *withelse*
5	| *…other statements…*

  - Intuition: when we have an “else”, all preceding nested conditions must have an “else”

Ambiguity

- Ambiguity can take different forms
  - Grammatical ambiguity *(if-then-else problem)*
  - Contextual ambiguity
    - In C: **`x * y;`** could follow **`typedef int x;`**
    - In Fortran: **`x = f(y);`** f could be function or array
- *Cannot be solved directly in grammar*
  - Issues of ***type*** (later in course)
- Deeper question:

  How much can the parser do?

Scanning vs parsing

- Consider the language of just matching parentheses:

`()`	legal
`(())`	legal
`(()`	*illegal*
`()(())(()())`	legal

- Can this language be expressed as an regular expression?
  - Answer: no
  - Intuitively: regular languages only expand “at the end”
  - Formally: use ***pumping lemma***

Regular vs context-free

- Using a context-free grammar

#	*Production rule*
1	*parens* → *parens* **(** *parens* **)**
2	| ε

- Difference in power:
  - Context-free grammars solve problem

    Intuitively: allow expansion “in the middle”
  - Regular grammars: less powerful

    Only allow productions of the form $\alpha \rightarrow \underline{\mathbf{x}}\ \beta$

Regular languages

- Still good for scanning
  - Efficiency (overhead, not asymptotic)
  - Comments, white space

#	Production rule
1	*expr* → *comm expr op comm expr*
2	| `number` *comm*
3	| `identifier` *comm*
…	….
8	*comm* → `/*` *text* `*/`
9	|

  - Yuck! And hard to get right

Beyond context-free?

- Is there a context-free grammar for:

  $a^n b^n c^n$
  - Must have same number of a's, b's, and c's
- No

  Intuitively: need to expand in two places
- Slightly surprising:

  $a^n b^m c^{m+n}$

  is a context-free language

#	Production rule
1	*S* → <u>a</u>*S*<u>c</u>
2	| *B*
3	*B* → <u>b</u>*B*<u>c</u>
4	| ε

Big picture

- Scanners
  - Based on regular expressions
  - Efficient for recognizing token types
  - Remove comments, white space
  - Cannot handle complex structure
- Parsers
  - Based on context-free grammars
  - More powerful than REs, but still have limitations
  - Less efficient
- Type and semantic analysis
  - Based on attribute grammars and type systems
  - Handles "context-sensitive" constructs

Roadmap

- So far…
  - Context-free grammars, precedence, ambiguity
  - Derivation of strings
- Parsing:
  - Start with string, discover the derivation
  - Two major approaches
    - Top-down – start at the top, work towards terminals
    - Bottom-up – start at terminals, assemble into tree

Next time…

- Today: homework on lexers
  - Posted on the web-page
  - Due September 26 (one week)
- Top-down parsers