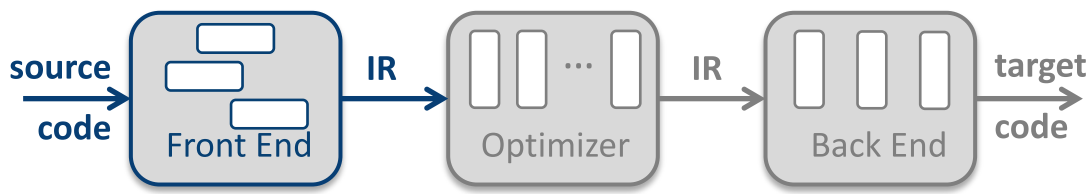
COMP 412
FALL 2018

Building a Parser

(from a Lab 1 perspective)

Comp 412



Chapter 1 & 3 in EaC2e

Parsing Command Line Arguments

The handling of command line arguments in (Linux, Mac OS X, Windows) is language specific. You need to consult the appropriate documentation.

With that caveat:

- Most of the languages you choose follow the C/Bell Labs Unix® model
- The "main" procedure receives an array of strings
- Each argument is in its own string, in order
- There may (not) be an integer that contains the number of strings

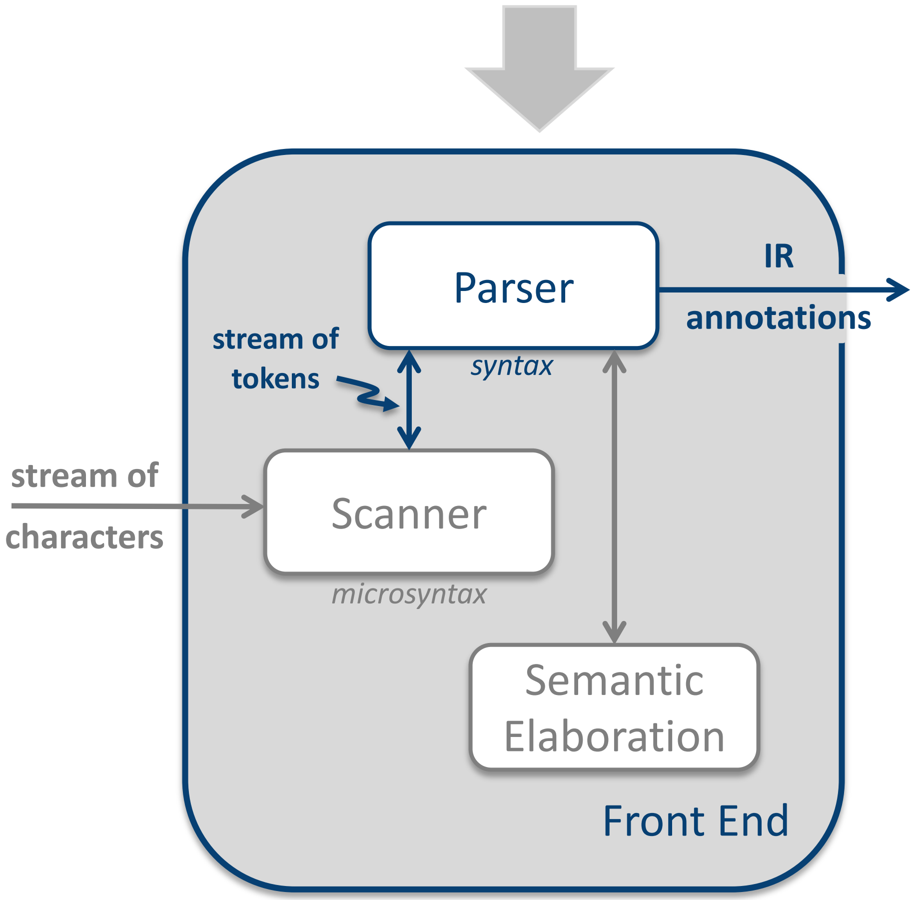
In C, that is:

```
int main(int argc, char *argv[])
```

- You write code to iterate over argv and process the strings
- argv[0] is the name of the command from the command line (e.g., "ls")

Posted some links (python, Java, C/C++) on piazza yesterday

The Front End



Scanner and **Parser** collaborate to check the syntax of the input program.

- Scanner maps stream of characters into words (words are the fundamental unit of syntax)
- Scanner produces a stream of **tokens** for the parser
  - Token is <part of speech, word>
  - "Part of speech" is a unit in the grammar
- Parser maps stream of **tokens** into a sentence in a grammatical model of the input language

The Study of Parsing

Parsing is the process of discovering a *derivation* for some sentence

- Given a stream of parts of speech, is it a valid sentence?

We need

- A mathematical model of syntax — a context-free grammar *G*
- An algorithm to test for membership in *G*
  - Given a sentence, is it a member of *L(G)* — the language defined by *G* ?
- To remember that our goal is to build parsers, not to study the fascinating, if arcane, mathematics of arbitrary languages

For lab 1, the language and its grammar are quite simple

- **ILOC** looks like the assembly language for a simplified **RISC** processor
- Grammar is easily handled by a small, hand-coded parser

Specifying Syntax: Context-Free Grammars

Context-free syntax is specified with a context-free grammar (CFG)

$$
\begin{aligned}
\textit{SheepNoise} &\rightarrow \underline{\text{baa}}\ \textit{SheepNoise} \\
&\quad |\ \ \underline{\text{baa}}
\end{aligned}
$$

This grammar defines the set of noises that a sheep makes under normal circumstances

This grammar is written in a variant of Backus–Naur Form (BNF)

Formally, a grammar $G = (S,N,T,P)$

- S is the *start symbol* (*SheepNoise*)
- N is a set of *non-terminal symbols* (*SheepNoise*)
- T is a set of *terminal symbols* or *words* (baa)
- P is a set of *productions* or *rewrite rules* (shown above)
  $(P : N \rightarrow N \cup T)$

(Example due to Dr. Scott K. Warren)

Specifying Syntax: Context-Free Grammars

We can use a grammar, like *SheepNoise*, to generate sentences

SheepNoise → <u>baa</u> *SheepNoise*
  | <u>baa</u>

Rule	***Sentential Form***
—	*SheepNoise*
1	<u>baa</u>

Rule	***Sentential Form***
—	*SheepNoise*
0	<u>baa</u> *SheepNoise*
1	<u>baa</u> <u>baa</u>

Rule	***Sentential Form***
—	*SheepNoise*
0	<u>baa</u> *SheepNoise*
0	<u>baa</u> <u>baa</u> *SheepNoise*
1	<u>baa</u> <u>baa</u> <u>baa</u>

And, so on …

While this example is cute, it becomes trite pretty quickly …

Specifying Syntax: Context-Free Grammars

We can put context-free grammars to better uses

- This simple grammar generates the set of expressions over identifiers and the operators +, -, ×, and ÷

1	*Expr*	→	**id** *Op Expr*
2		|	**id**
3	*Op*	→	**+**
4		|	**–**
5		|	**×**
6		|	**÷**

L(G) includes sentences such as
a × b, c + d, and e

A **CFG** is a four tuple, $G = (S, N, T, P)$

- S is the *start symbol* of the grammar
  L(G) is the set of sentences that can be derived from S
- N is a set of *nonterminal symbols* or syntactic variables { *Expr, Op* }
- T is the set of *terminal symbols* or words {+, -, ×, ÷, **id**}
- P is a set of *productions* or *rewrite rules*, shown in the table to the left {**1, 2, 3, 4, 5, 6**}

This grammar is a simple but bad example of an expression grammar. See later lectures or Chapter 3 in EaC2e.

Parsing

The point of parsing is to discover a grammatical derivation for a sentence

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

- S is the start symbol of the grammar
- Each γ_i is a sentential form
  - If γ contains only terminal symbols, γ is a **sentence** in *L(G)*
  - If γ contains 1 or more non-terminals, γ is a **sentential form**
- To get γ_i from γ_{i-1}, expand some **NT** $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
  - Replace the occurrence of $A \in \gamma_{i-1}$ with β to get γ_i
  - Replacing the leftmost **NT** at each step, creates a **leftmost** derivation
  - Replacing the rightmost **NT** at each step, creates a **rightmost** derivation

NT means <u>n</u>on<u>t</u>erminal symbol

Parsing

The point of parsing is to discover a grammatical derivation for a sentence

Derivation ↓   Parse ↑

Rule	Sentential Form
—	*Expr*
1	<u>**id**</u> *Op Expr*
3	<u>**id**</u> + *Expr*
5	<u>**id**</u> + <u>**id**</u>

Derivation of "a + b"

1	*Expr*	→	<u>**id**</u> *Op Expr*
2		|	<u>**id**</u>
3	*Op*	→	**+**
4		|	**−**
5		|	**×**
6		|	**÷**

(Bad) Expression grammar

In the general case, discovering a derivation looks expensive

- Many alternatives and combinations, possible backtracking
- Derivation must be guided by the actual words in the input stream
- Fortunately, most programming languages have simple syntax that can be parsed efficiently
  - Studying parsing will help you understand why **PL**s look as they do!

Parsing

The point of parsing is to discover a grammatical derivation for a sentence

Derivation ↓ Parse ↑

Rule	Sentential Form
—	*Expr*
1	**id** *Op Expr*
4	**id** – *Expr*
1	**id** – **id** *Op Expr*
5	**id** – **id** × *Expr*
2	**id** – **id** × **id**

Derivation of a – b × c

1	*Expr*	→	**id** *Op Expr*
2		|	**id**
3	*Op*	→	+
4		|	–
5		|	×
6		|	÷

(Bad) Expression grammar

We denote this particular derivation: $Expr \Rightarrow^* \underline{id} - \underline{id} \times \underline{id}$

Parsing

For a sufficiently-simple, well-behaved grammar, we can construct a *top-down*, *recursive-descent* parser

Sufficiently-simple, well-behaved grammar?

- For non-terminal A, if $A \rightarrow B \mid C \mid D$, the parser must be able to choose between B, C, & D based on the first symbol on the right hand side.
- The bad expression grammar does not quite have this property

The **ILOC** grammar does.

Top-down, recursive-descent parser

- For each non-terminal in the grammar, construct a routine to parse it
- To parse non-terminal on a production's right-hand side, call the appropriate routine for that non-terminal
  – In practice, these parsers quickly become recursive …

Sufficiently-simple, well-behaved grammars have a name: ***LL*(1)** grammars

Parsing

The Bad Expression Grammar has two non-terminals

- The productions for Op have the desired property
  - The first symbol in each right-hand side is a unique terminal symbol
- The productions for Expr do not have the desired property
  - They both begin with **id**
- Fortunately, we can fix this problem easily, in this case [1]

1	*Expr*	→	**id** *Op* *Expr*
2		|	**id**
3	*Op*	→	**+**
4		|	**−**
5		|	**×**
6		|	**÷**

(Bad) Expression Grammar

1	*Expr*	→	**id** *Tail*
2	*Tail*	→	*Op Expr*
3		|	ε
4	*Op*	→	**+**
5		|	**−**
6		|	**×**
7		|	**÷**

Repaired (Bad) Expression Grammar

[1] While this fix, *left-factoring* the production, works in this case, it does not cure all problems with all grammars.

Parsing

A simple, recursive-descent style parser for the repaired (bad) expression grammar

1	*Expr*	→	**id** *Tail*
2	*Tail*	→	*Op Expr*
3		|	ε
4	*Op*	→	**+**
5		|	**–**
6		|	**×**
7		|	**÷**

```
Expr( ) {
   result = false;
   word ← Next Token()
   if  (word = id ) {
       result = Tail()
   else
       throw an error;
   return result;
}
```

```
Tail( ) {
   result = false;
   if (Op( ) = true) {
      if ( Expr( ) )
          result = true;
   }
   else if (word = EOF)
      result = true;
   else
       throw an error;
   return result;
}
```

```
Op( ) {
   result = false;
   word ← Next Token()
   if  (word = + )
       result = true
   else if (word = – )
       result = true
   else if (word = × )
       result = true
   else if (word = ÷ )
       result = true
   return result;
}
```

The Input to Lab 1

ILOC was discussed in Chapter 1 of EaC2e, as well in Appendix A. You should have read Chapter 1 already.

In Lab 1, the input is written in a subset of ILOC

Syntax			Meaning	Latency
load	r_1	=> r_2	$r_2 \leftarrow$ MEM(r_1)	3
store	r_1	=> r_2	MEM(r_2) $\leftarrow$ r1	3
loadI	x	=> r_2	$r_2 \leftarrow$ x	1
add	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 + r_2$	1
sub	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 - r_2$	1
mult	r_1, r_2	=> r_3	$r_3 \leftarrow$ r1 * r_2	1
lshift	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 << r_2$	1
rshift	r_1, r_2	=> r_3	$r_3 \leftarrow r_1 >> r_2$	1
output	x		prints MEM(x) to stdout	1
nop			idles for one cycle	1

ILOC is an abstract assembly language for a simple RISC processor. We will use it at times as an **IR** and at other times as a target language.

For details on Lab 1 **ILOC**, see the lab handout, the **ILOC** simulator document and Appendix A in EaC2e. (**ILOC** appears throughout the book.)

Note that **ILOC** is case sensitive. The 'I' in "loadI" must be an uppercase letter and the others must be lowercase letters.

'x' represents a constant.

The same **ILOC** subset (syntax & meaning) will be used in Lab 3, with different per-operation latencies. You should plan to reuse your **ILOC** front end in Lab 3.

The Syntax of Lab 1 **ILOC**

Rewriting the rules in a more formal notation

operation	→	**MEMOP REG INTO REG**
	|	**LOADI CONSTANT INTO REG**
	|	**ARITHOP REG COMMA REG INTO REG**
	|	**OUTPUT CONSTANT**
	|	**NOP**
block	→	*operation block*
	|	ε

Where

- *Italics* indicates a syntactic variable
- '→' means "derives"
- '|' means "also derives"
- 'ε' is the empty string

- **All CAPITALS** indicate a category of word (*e.g.*, **INTO** contains one word, "=>")
- Categories may contain > 1 word

These rules form a ***grammar***. See the digression on page 87 in EaC2e titled "Backus-Naur Form".

The Syntax of Lab 1 **ILOC**

ILOC has a “sufficiently-simple, well-behaved” grammar

operation	$\rightarrow$	**MEMOP REG INTO REG**
	|	**LOADI CONSTANT INTO REG**
	|	**ARITHOP REG COMMA REG INTO REG**
	|	**OUTPUT CONSTANT**
	|	**NOP**
block	$\rightarrow$	*operation block*
	|	ε

Each rule in the grammar (*production*) starts with a unique token type

- Can use tokentype to determine syntax of rest of the operation
- Suggests a simple decision procedure: switch on the first token

Parsing Lab 1 **ILOC**

```
Word = NextToken();
While (Word ≠ ENDFILE) {
   Switch (Word.TokenType) into {
       case MEMOP:        finish_memop();
                          break;
       case LOADI:        finish_loadI();
                          break;
       case ARITHOP:      finish_arithop();
                          break;
       case OUTPUT:       finish_output();
                          break;
       case NOP:          finish_nop();
                          break;
       default:           throw an error;
                          break;
       }
    Word = NextToken();
}
```

Structure of a simple ILOC parser

- One procedure to select the rule
  - Sketched at left
- One procedure per rule
  - Checks the rest of the tokens
  - If needed (scanner design) skips whitespace and end-of-line characters
  - Keep to one procedure per rule

If you are writing in Python, use a nest of *if-then-else* constructs

This parser is a simplified variant of a recursive-descent parser, a form of top-down parser that we will study in more depth.

See § 3.3.2 in EaC2e

Building a Representation for ILOC

Your front end must also build an IR for the code it parses

- **IR** must be useful in two labs that you have not seen
- Simplest **IR** for **ILOC** might be a $k \times 4$ array

Opcode	Op 1	Op 2	Op 3
0	2	—	0
0	5	—	1
4	0	1	2
3	2	—	0
10	2		

An ILOC Program as an Array

Opcode numbers are indexes into an array of strings that contain the mnemonic names of the opcodes.

Other numbers are register numbers or values of non-negative constants

This would work for Lab 1

- Several problems for future labs
- Array expansion is slow
  - Initializing size in python arrays
  - **numpy** arrays are not so good
  - ArrayList is **O**(n^2) on insertions
  - See T128k.i
- Need many more fields for lab 2 and lab 3
- Need ability to traverse both top to bottom and bottom to top

Building a Representation for ILOC

Your front end must also build an IR for the code it parses

- The reference implementation uses an **IR** that looks like this one

Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
loadI (0)	12	—	—	—	—	—	—	—	13	—	—	—

Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
load (1)	3	—	—	—	—	—	—	—	4	—	—	—

Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
mult (4)	3	—	—	—	4	—	—	—	4	—	—	—

SR: *source reg.*
VR: *virtual reg.*
PR: *physical reg.*
NU: next use

Doubly linked list might be built inside another other structure, like blocks of records.

lab1_ref allocates large blocks of records and has a cheap way to return a new record (cheap unless it needs to get a new block).

Lab 1 does not need all these fields. Labs 2 and 3 will.
You might also add a source-line number for debugging and error messages.

Building a Representation for ILOC

You must make some design choices on this representation

- Pointers? Records? Arrays?
- Built-in data structures or classes?
  - Lists versus ArrayLists