Maphoon: A C^{++} based Parser Generator

Hans de Nivelle

Nazarbayev University, Nur-Sultan City, Kazakhstan

Aspen, 05.05.2022

Pronunciation

Let's start at the very beginning:

How do we pronounce 'Maphoon'?

I will be honoured in whichever way you choose to pronounce it, but my preferred pronunciation is 'Mah-phone' with the stress on 'Mah'.

'Maphoon' does not rhyme with 'Lagoon'.

Kazakhstan and Nazarbayev University

- The present state of Kazakhstan exists since 1991, but Kazakh as national identity exists much longer, since European middle ages.
- Kazakhstan has approximately 19 million inhabitants, and a large surface area $2,724,900\,\mathrm{km}^2$ (1,052,100 square miles), 9-th largest in the world.
- Kazakhstan is rich in natural resources, oil and gas, coal and metals (chromium, lead, uranium, copper, gold and zinc.)
- Unlike some its neighbours, the Kazakh government invests a large fraction of the income from natural resources into development.

- Nazarbayev University was established in 2010. Before that, the best students were sent to universities abroad.
- NU has 4000 undergraduate students, 1500 graduate students, and 700 students in the foundational year.
- Proficiency in English is a requirement for everyone.
- Students are required to live on campus, in order to reduce inequality caused by background. All important decisions are merit only.
- School of Engineering and Digital Sciences has 2000 undergraduate students, 300 graduate students.

Motivation

I am developing a programming language for implementation of logic (verification of mathematical proofs and theorem proving). For this language, I need a parser.

I have been teaching compiler construction, using Java as implementation language.

There are very nice tools for Java, namely JFlex (`www.jflex.de/`) for automated tokenizer generation, and CUP (`www2.cs.tum.edu/projects/cup/`) for automated parser generation.

I want to use C^{++} for the implementation of my programming language, I wanted to use similar tools, but they were lacking.

In future version of the compiler construction course, I want to use C^{++}.

Motivation (2)

For big, established languages like C^{++} or C, one probably does not need automatic parser generators.

Syntax does not change often, and the resources spent to implementing the parser are small, compared to the total resources spent.

For experimental languages, like mine, they are useful. It is very easy to make changes in syntax, and use them.

I decided to write a tokenizer generator and parser generator by myself.

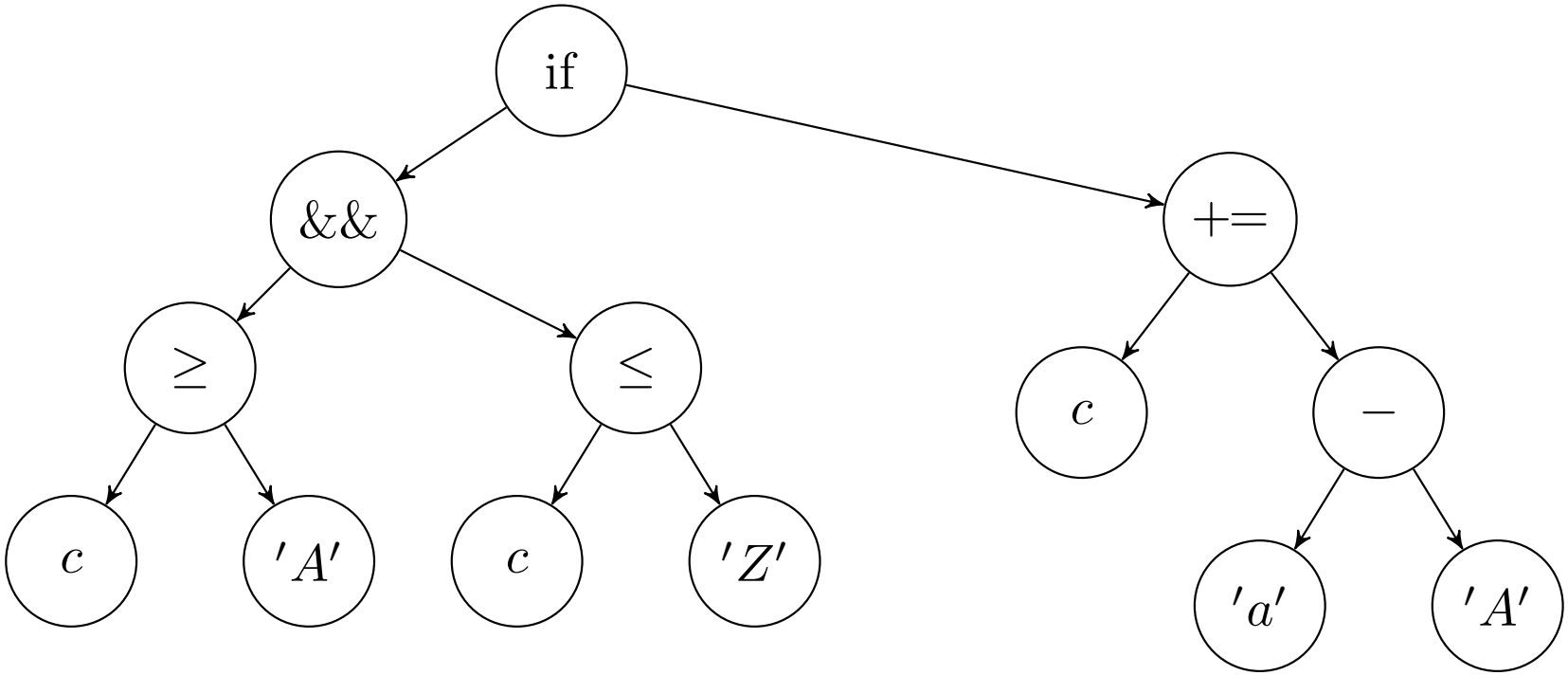
What is Parsing?

In computer science, nearly everything can be represented as a tree.

We are given an input as sequence of characters. The task of the parser is to extract the tree structure. The resulting tree is called abstract syntax tree.

Example of an AST

```
if( c >= 'A' && c <= 'Z' ) c += 'a' - 'A';
   // If c is upper case, make it lower case.
```



Tokenizing

Tokenizing is the first step.

Tokens typically have the following forms:

- Numbers (integers or floating point)
- Strings (Can be quite complicated, with escape codes)
- Reserved words, operators
- Comments
- Indentation changes (In Python, these are tokens)

Automatic Generation of Tokenizers

There exists a very nice theory for recognizing and classifying tokens:

Regular Expressions ⇒ Non-Determinstic Finite Automata ⇒ Deterministic Finite Automata ⇒ Minimal Deterministic Finite Automaton.

There exist many excellent tools that implement this theory, also for C or C^{++} :

flex: (`github.com/westes/flex`),

re2c: (`re2c.org/`)

(There are more)

Why not use an existing implementation?

The problem is lack of flexibility.

There is always something that doesn't fit in:

- My logic language uses Python-style indentation.
- It also uses comments of form `#< ... >#` , which can be nested. (They play the role of `#if 0 ... #endif`). These tokens are non-regular.
- Long comments of form `/* ..... */` should not be stored in the buffer. It is better to handle these separately.
- For some languages (I will not mention their names), the tokenizer must have access to type definitions.
- Some languages allow a certain token `>>` only in certain contexts. (Again, I am not mentioning any names.)

I hesitated very long if it is worth automating tokenizing. In the end, I did it, keeping the following goals in mind:

- Flexibility: It must be possible to handle some tokens by hand.

  The tokenizer must not dictate how source information is handled.

- Usable in education. I am a university professor and I want to show automata to students.

- Efficiency (until flexibility has to be sacrificed)

I created the building blocks for the tokenizer, but not the tokenizer itself.

On the next slide, I show a **filereader** class. It forms the buffer between the tokenizer and the input source.

Since my tokenizer generator does not generate the complete tokenizer, the user has to interact with this class.

Class Filereader

A filereader contains a pointer to a file. In addition to that, it keeps track of line number and column position. It has an unbounded buffer of characters that is initially empty.

The main methods are:

- `bool has( size_t n )` : Read characters from the source until the buffer has size n and return `true` on success.
- `char peek( size_t i ) const` : Get the i-th character from the buffer.
- `string_view view( size_t i ) const` : Get the first i characters in the buffer as `string_view`.
- `commit( size_t i )` : Remove the first i characters from the buffer. The buffer must contain $n \geq i$ characters, and after committing, it contains $n - i$ characters.

Recognizing Tokens by Hand (1)

For the
symbols that you want to recognize by hand, write a function of form:
`std::pair< symboltype, size_t > try_X( filereader& inp )`.

If the attempt failed, the second field must be 0.

It is possible to recognize more than one type of symbol in a single function.

The following function may return different symbol types
dependent on the type of number:

```
std::pair< symboltype, size_t >
                    try_number( filereader& inp )
```

Recognizing Tokens by Hand (2)

This is how one interacts with `filereader`:

```
std::pair< symboltype, size_t >
tokenizer::try_identifier( filereader& inp )
{
   if( inp. has(1) && starts_ident( inp. peek(0)) )
   {
      size_t i = 1;
      while( inp. has(i+1) &&
                continues_ident( inp. peek(i) ))
         ++ i;
      return std::pair( sym_IDENT, i );
   }
   else
      return std::pair( sym_IDENT, 0 );
}
```

Automatic Recognition

For the remaming tokens, we build a finite automaton:

Building the automaton. For the time being, we recompute automaton every time the program is started:

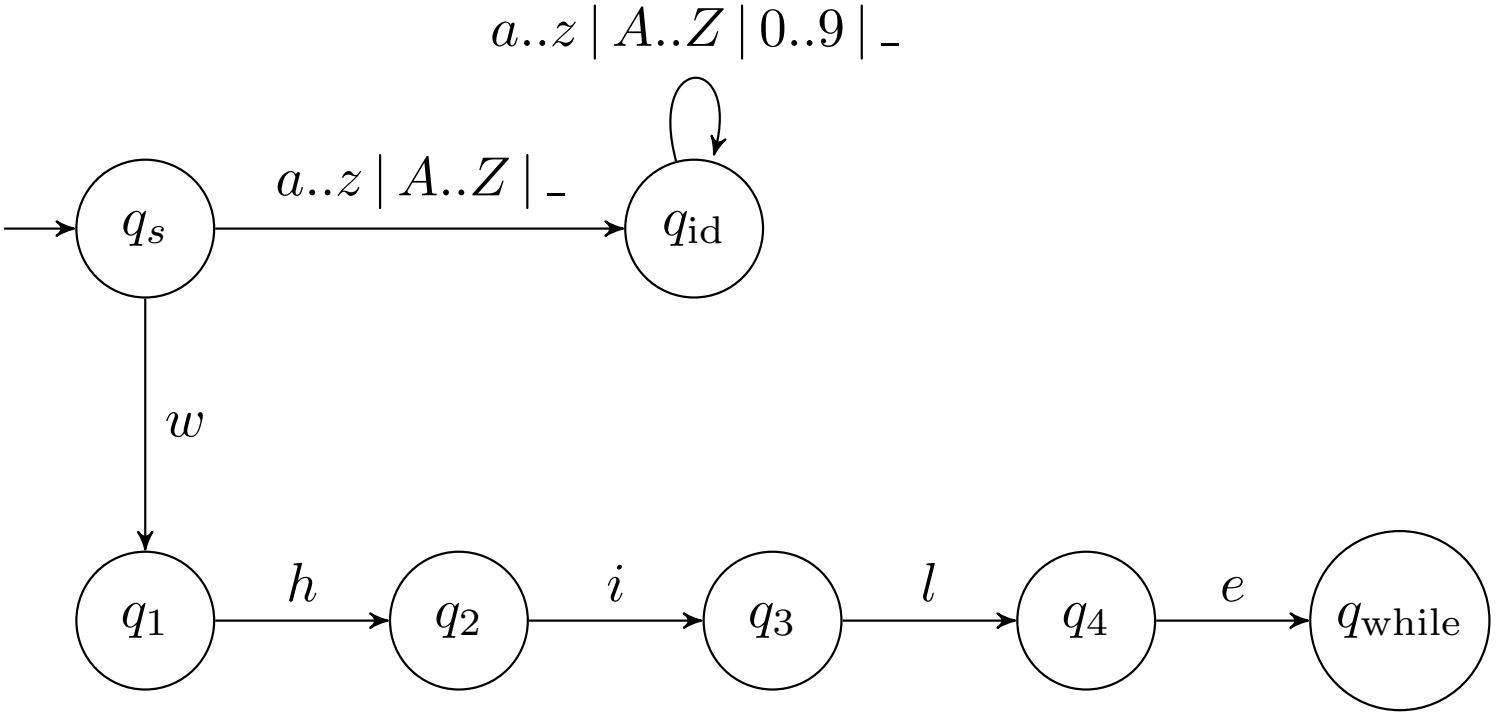
```
static lexing::classifier< char, symboltype >
   cls = buildclassifier( );
```

Using the automaton:

```
std::pair< symboltype, size_t >
       p = readandclassify( cls, inp );
```

`inp` is a `filereader&`, and the return value is the same as for hand-written functions.

Refreshing Automata: Identifier and a reserved Word "while"



Flat Automata

We use a nice and simple way for representing finite automata.

A state is a triple (Λ, ϕ, T), in which

- Λ is a set of integers, representing the ϵ transitions.
- ϕ is an ordered map from `char` to (integer or #), representing the transitions from this state. It must have at least have a value for the minimal `char`. (I will assume it is -128.) In order to apply it on character c, find the $(c', n) \in \phi$ with maximal $c' \leq c$.
- T is the classification of the state.

A classifier is a vector of states. We use relative addressing for state references.

nr	Λ	Φ	T
0 :	$\{1, 4\}$	$\{\ (-128, \#)\ \}$	**err**
1 :	$\{\,\}$	$\{\,(-128, \#),\ (A, 1), (Z^{+1}, \#),\ (a, 1), (z^{+1}, \#)\,\}$	**err**
2 :	$\{1\}$	$\{\,(-128, \#),\ (0, 0), (9^{+1}, \#),\ (A, 0), (Z^{+1}, \#),$	
		$(\_, 0), (\_^{+1}, \#), (0, 0), (9^{+1}, \#)\,\}$	**err**
3 :	$\{\,\}$	$\{\,(-128, \#)\,\}$	**ident**
4 :	$\{\,\}$	$\{\,(-128, \#), (w, 1), (w^{+1}, \#)\,\}$	**err**
5 :	$\{\,\}$	$\{\,(-128, \#), (h, 1), (h^{+1}, \#)\,\}$	**err**
6 :	$\{\,\}$	$\{\,(-128, \#), (i, 1), (i^{+1}, \#)\,\}$	**err**
7 :	$\{\,\}$	$\{\,(-128, \#), (l, 1), (l^{+1}, \#)\,\}$	**err**
8 :	$\{\,\}$	$\{\,(-128, \#), (e, 1), (e^{+1}, \#)\,\}$	**err**
9 :	$\{\,\}$	$\{\,(-128, \#)\,\}$	**while**

Remaining Topics

I show you in code

- How the classifier is created, using code.
- The classifier.
- How to make the classifier deterministic.
- How to minimize the classifier.
- How to deal with EOF and bad files.
- How to ignore whitespace and comments.
- How to compute attributes.
- How to obtain a maximally(?) efficient tokenizer, using truly dirty trickery.

This completes the topic of tokenizing.

Parsing

We are now at the second stage. We have cut the input in bite-sized pieces, and we need to build a tree from them.

We, as professors, torment our students with the following definition:

A context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$ is a quadruple consisting of:

- A finite set of non-terminal symbols V.
- A finite set of terminal symbols Σ,
- A set of rules of form $\alpha \to w$, with $\alpha \in V$ and w a finite word over $V \cup \Sigma$,
- A start symbol S.

Unfortunately, this definition is not usable in practice.

Attribute Grammars

Definition: An attribute grammar has form $\mathcal{G} = (\Sigma, A, R, S, T)$, in which

- Σ is the set of symbols. We don't distinguish anymore between terminal symbols and variable symbols.
- A is a function that attaches to each $\sigma \in \Sigma$ a non-empty attribute set $A(\sigma)$.
- R is a set of rewrite rules.
- $S \in \Sigma$ is the start symbol, $T \subseteq \Sigma$ is the set of terminator symbols. These are symbols that follow after a correct input (e.g. EOF or ;).

We define $\Sigma \otimes A = \{\ (\sigma, a)\ \mid\ \sigma \in \Sigma \text{ and } a \in A(\sigma)\ \}$. These are the valid symbols.

The formal notation may look a bit frightening, but it is totally natural.

For C language, one could have (extremely simplified):

$$
\begin{aligned}
\text{Stat} &\rightarrow \textbf{while} \text{ (} E \text{) Stat} \\
&\rightarrow \textbf{do} \text{ Stat } \textbf{while} \text{ (} E \text{) ;} \\
&\rightarrow \textbf{if} \text{ (} E \text{) Stat} \\
&\rightarrow \textbf{if} \text{ (} E \text{) Stat } \textbf{else} \text{ Stat} \\
&\rightarrow \textbf{for} \text{ (Stat Stat) Stat} \\
&\rightarrow \textbf{for} \text{ (Stat Stat Stat) Stat) Stat}
\end{aligned}
$$

$\sigma(\text{Stat})$ is the set of all possible ASTs that represent a statement. Similarly, $\sigma(E)$ is the set of all possible ASTs that represent an expression.

Standard Example: Calculator

The calculator can evaluate simple expressions of form:

```
1 + 2 * 3;
   --> 7
a := 1 + 1;
   --> assigning a := 2
a - a * 4;
   --> -6
```

We use the standard operators: `+`, `-`, `*`, `/`, where `*` and `/` take priority over `+` and `-`.

It is possible to assign to variables using `:=`

The rewrite rules are (first attempt):

$$S \rightarrow \epsilon \mid S\ C$$

$$C \rightarrow E\,; \mid \text{ident} := E\,;$$

$$E \rightarrow E + E \mid E - E \mid E * E \mid E\,/\,E \mid -E \mid (\,E\,)$$
$$E \rightarrow \text{double} \mid \text{ident} \mid (\,E\,) \mid \text{ident}(\,A\,)$$

$$A \rightarrow E \mid A,\ E$$

S is the start symbol. It represents a complete session. C is a single command.

Ambiguity

Problem: No control over evaluation order.

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow \cdots \Rightarrow \text{double} + \text{double} * \text{double}$$

($*$ evaluated first.)

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow \cdots \Rightarrow \text{double} + \text{double} * \text{double}.$$

($+$ evaluated first.)

When a same input word can be obtained in different ways by applying the grammar, this is called ambiguity. Ambiguity is bad, because different derivations will result in different meanings.

Solution: Split E into different symbols, representing the different priority levels:

$$
\begin{array}{lcl}
S & \rightarrow & C \mid S\ C \\
\\
C & \rightarrow & E\,; \mid \text{ident} := E\,; \\
\\
E & \rightarrow & E + F \mid E - F \mid F \\
F & \rightarrow & F * G \mid F \,/\, G \mid G \\
G & \rightarrow & -G \mid H \\
H & \rightarrow & \text{double} \mid \text{ident} \mid (\,E\,) \mid \text{ident}(\,A\,) \\
\\
A & \rightarrow & E \mid A,\, E
\end{array}
$$

Rest I show in Maphoon syntax:

```
%startsymbol Session EOF
   // startsymbol with terminator EOF.

%symbol                                EOF BAD
%symbol{ std::string }                 SCANERROR IDENT
%symbol                                SEMICOLON ASSIGN COMMA
%symbol{ double }                      DOUBLE
%symbol                                PLUS TIMES MINUS DIVIDES
%symbol                                LPAR RPAR
%symbol{ double }                      E F G H
%symbol{ std::vector<double> }         Arguments
%symbol                                Session Command

%symbol                                COMMENT WHITESPACE EMPTY
```

Going back to the definition on slide 22, we now defined $\Sigma, A, S,$ and T.

Directions of Parsing

Given a sequence of tokens (with attributes), the parser constructs a derivation of this sequence of tokens, and use this derivation to attach a meaning to the input.

Parsing can be either top-down or bottom-up.

Maphoon constructs a bottom-up parser, but I will shortly discuss top-down parsing.

Top-Down Parsing

With top-down parsing, the parser starts with the start symbol S, and rewrites towards the given symbol sequence.

At each point during parsing, the parser knows which symbol it needs to obtain, looks at the next symbol in the input, and decides which rule must be applied.

For example, if one needs to obtain a Stat, and the next symbol is **while**, the parser knows that rule Stat $\rightarrow$ **while** (E) Stat must be applied.

Major disadvantage of top-down parsing is that decisions must be made at the beginning of rules. Rules that have common beginnings are a probem.

For example, if one needs Stat, and the next symbol is **if**, the parser cannot select the rule.

Bottom-Up Parsing

The parser starts with the input word, and applies the rewrite rules from right-to-left.

As far as I know, bottom-up parsing is always shift-reduce parsing.

The parser uses two variables: The parsestack (stack of symbols with attributes), and a lookahead (optional symbol).

shift (Lookahead must be defined): Push the current lookahead to the stack. Make the lookahead undefined.

reduce (The top of the stack must contain the right hand side of a rule): Remove the right hand side from the stack, and replace it by the left hand side of the rule. Compute the attribute of the new symbol.

read (Lookahead must be undefined): Read a symbol from the input source and put it in lookahead.

Bottom-Up Parsing: Computing the Attributes

When a rule is applied from right to left, one must compute the attribute of the left hand side.

In order to do this, we attach code fragments to rules:

```
E  => E:e PLUS F:f    { return e + f; }
    | E:e MINUS F:f   { return e - f; }
    | F:f             { return f; }
    ;
```

The code fragments are usually called action code. It must return the attribute of the left hand side (if it is not `void`).

Examples of Action Code

```
E => IDENT: id
     {
        if( memory. contains(id))
           return *memory. lookup(id);
        else
        {
           errorlog. push_back(
                std::string( "variable " ) +
                id + " is undefined " );
           return 0.0;  // An arbitrary choice.
        }
     }
     | DOUBLE : d   { return d; }
     ;
```

Examples of Action Code (2)

```
E => IDENT:id LPAR Arguments:args RPAR
     {
        if( id == "sin" && args. size( ) == 1 )
           return sin( args[0] );

        if( id == "pow" && args. size( ) == 2 )
           return pow( args[0], args[1] );

        errorlog. push_back(
           std::string( "unrecognized function " ) + id );

        return 0.0;  // An arbttrary choice.
     }
     ;
```

Examples of Action Code (3)

```
Arguments => E:e
{
   return { e };
}
          | Arguments:a COMMA E:e
{
   a. push_back(e);
   return a;
}
          ;
```

Examples of Action Code (4)

```
Command => E:e SEMICOLON
{
   if( errorlog. size( ))
   {
      printerrors( errorlog, std::cout );
      errorlog. clear( );
   }
   else
   {
      std::cout << "---> " << e << "\n";
   }
}
```

Examples of Action Code (5)

```
Command => IDENT:id BECOMES E:e SEMICOLON
{
   if( errorlog. empty( ))
   {
      std::cout << " assigning: ";
      std::cout << id << " := " << e << "\n";
      memory. assign( id, e );
   }
   else
   {
      printerrors( errorlog, std::cout );
      errorlog. clear( );
   }
}
```

Making the Decisions

The hard part of this process is deciding between shift and reduce, in case where the stack contains the complete right side of a grammar rule.

Compared to top-down parsing, bottom-up parsing has one big advantage:

A decision whether to reduce has to be made only when the complete right hand side has been read.

At the moment, the decision needs to be made, we have more information.

Bottom-Up Parsing (Decision Making) (2)

Decisions can be made as follows (see e.g. the Dragon Book):

The state of the parser is called viable, if it is possible to continue into a succesful parse.

A word is viable if there exists a viable state of the parser, in which the parse stack contains this word.

Theorem: The set of viable words is regular. Hence it can be recognized with a deterministic finite automaton (the prefix automaton).

So how do we make the decisions? $\Rightarrow$ Compute the prefix automaton in advance, and never do anything that makes the parse stack non-viable.

Maphoon

Maphoon reads the grammar and the action code.

It creates two files **symbol.h** and **symbol.cpp** containing the symbol definition.

It also creates two files **parser.h** and **parser.cpp** containing a runnable parser that correctly applies the action code when a rule is reduced.

Every class that has correct life cycle operations (constructor, assignment, destructor) can be used as attribute.

It is even better when attributes are movable.

In my view, bottom-up parsing is easier than top-down parsing if one has the proper tools.

Beyond the Dragon Book: Adding Preconditions

Some languages (e.g. Prolog) allow to define operators at run time.

That means that we cannot specify the grammar in advance. The approach on slide 22 will not work.

One could try to recompute the prefix automaton at runtime, but that will not be easy, and it will be computationally expensive.

Instead, we use a simple, ambiguous grammar, and attach runtime preconditions to rules.

A rule can only be reduced if its precondition evaluates to true.

Preconditions

Preconditions are similar to action code. It can **const**-ly see the attributes of the right hand side of the rule, the lookahead, and additional fields of the parser. It must return return `bool`.

The precondition decides if the reduction can happen.

Example from Prolog

```
Prefix => IDENTIFIER : id
%requires
   { return synt. hasprefixdef( id ) &&
            canstartterm( lookahead. value( )); }
%reduces
   { return synt. prefixdef(id); }
;

Infix => IDENTIFIER : id
%requires
   { return synt. hasinfixdef(id) &&
            canstartterm( lookahead. value( )); }
%reduces
   { return synt. infixdef(id); }
;
```

Example from Prolog (2)

```
Term => Prefix:op Term:t
%requires
   { return canreduce( synt, op, lookahead. value( )); }
%reduces
   { return
     new functional( function( op. str, 1 ), { t } ); }

|  Term:t1 Infix:op Term:t2
%requires
   { return canreduce( synt, op, lookahead. value( )); }
%reduces
   { return
     new functional( function( op. str, 2 ), { t1, t2 } );
   }
```

Another Example: Context Sensitive Keywords

```
LeftRightStat => CHAR : c
%requires
   { char c1 = toupper(c);
     return c1 == 'L' || c1 == 'S' || c1 == 'R'; }
%reduces
{
   char c1 = toupper(c);
   if( c1 == 'L' ) return -1;  // left
   if( c1 == 'R' ) return 1;   // right
   return 0;   // stationary.
}
;
```

(This comes from a Turing-Machine simulator)

Error Handling

Shift-reduce parsing detects a syntax error at the earliest possible point.

It is possible to accurately report the position of the error.

Shift-reduce parsing is quite good at error recovery. I copy the approach from Yacc, and it works well.

But shift-reduce parsing is not good at creating meaningful error messages. This is a traditional weakness of bottom-up parsing.

Maybe I solved this problem.

Recovery

Recovery is done by throwing away symbols, until a synchronization point is reached.

Synchronization points are defined by rules of form

```
Command =>  _recover_ SEMICOLON
{
   if( debug )
      std::cout << "recovered from syntax error\n\n";
} ;
```

After a syntax error, the parser throws away symbols until it encounters a (;). After that, it reduces the rule, and starts a trial period.

If a new error occurs during this time, the parser will treat it like a failed recovery, instead of a new error.

Error Reporting

What should one say to the user?

```
( 1 ) )
```

```
1 2
```

```
f( ,
```

```
f b
```

```
1 + *
```

```
)
```

Error Reporting (2)

In order to obtain an error message, we try to find out what is expected, and we consider the current lookahead:

expectation	lookahead	message
unknown	unknown	'syntax error'
unknown	L	'unexpected L'
X	unknown	'expected X'
X	Y	'expected X instead of Y'

Expections are obtained by matching a restricted form of regular expressions into the parse stack.

I show examples in code, because it is an empirical process.

Summary, Conclusions

I created tools for generating tokenizers and parsers. The parser generator is similar to Bison/Yacc/CUP, but supports C^{++}.

The tools fulfill my own needs. I hope they will fulfill the needs of others too.

Theory is great, it is nice to implement, it can solve your problems, but you have to be flexible.

Target group:

Tokenizer toolbox $\Rightarrow$ There is no excuse for anyone.

Parser generator $\Rightarrow$ Experimental languages, teaching.

Systems can be downloaded from `www.compiler-tools.eu/`

Thanks!

Thanks to Danel Batyrbek, Aleksandra Kireeva, Tatyana Korotkova, Dina Muktubayeva, Cláudia Nalon, and Olzhas Zhangeldinov.

I also thank Nazarbayev University for supporting this project through the Faculty Development Competitive Research Grant Program (FDCRGP), grant number 021220FD1651.

Skipped During Presentation: Top-Down Parsing

Below follows a short discussion of top-down parsing that I skipped during the presentation.

Top-Down Parsing (1)

With top-down parsing, the parser starts with the start symbol S, and rewrites towards the given symbol sequence.

At each point during parsing, the parser knows which symbol it needs to obtain, looks at the next symbol in the input, and decides which rule must be applied.

In order to do this, one needs a stack of expected symbols. The first expected symbol is on the top of the stack.

Top-down parsing can either be implementd by tables, or implemented by hand. This is called recursive descent. In case of recursive descent, no explicit stack is needed, because the call stack can be used.

Top-Down Parsing (2)

For each symbol E that occurs as left hand side of some grammar rules, write a function **parseE** that recognizes the words that can be obtained from E. It must return the attribute of the found E.

Function **parseE** looks at the next symbol in the input, decides which grammar rule for E applies, and processes the input. If necessary, it calls **parseV** for another symbol V.

Top-Down Parsing (3)

It is usually necessary to change the grammar. For example, with rules $E \to E + F \mid E - F \mid F$, the function **parseE** has to start by recursively calling **parseE** or **parseF**.

The decision can be made only when $+$ or $-$ is encountered.

In order to solve this problem, the rules have to be merged into a single rule with a regular expression to the right:
$E \to F\ (\ +F \mid -F\ )^*$.

Top-Down Parsing (4)

Now one can write (simplified):

```
double parseE( ) {
   double d = parseF( );
   while( lookahead == '+' || lookahead == '-' ) {
      op = lookahead;
      lookahead = tokenizer.read( ); // get next lookahead
      double d2 = parseF( );
      d = d op d2;
   }
   return d;
}
```

Table-driven top-down parsing has the same problem, namely that the grammar has to be changed to make it work.