

Generating Parsers in C^{++} with Maphoon (2)

Hans de Nivelle

Nazarbayev University, Astana, Kazakhstan

Online, 15.07.2022

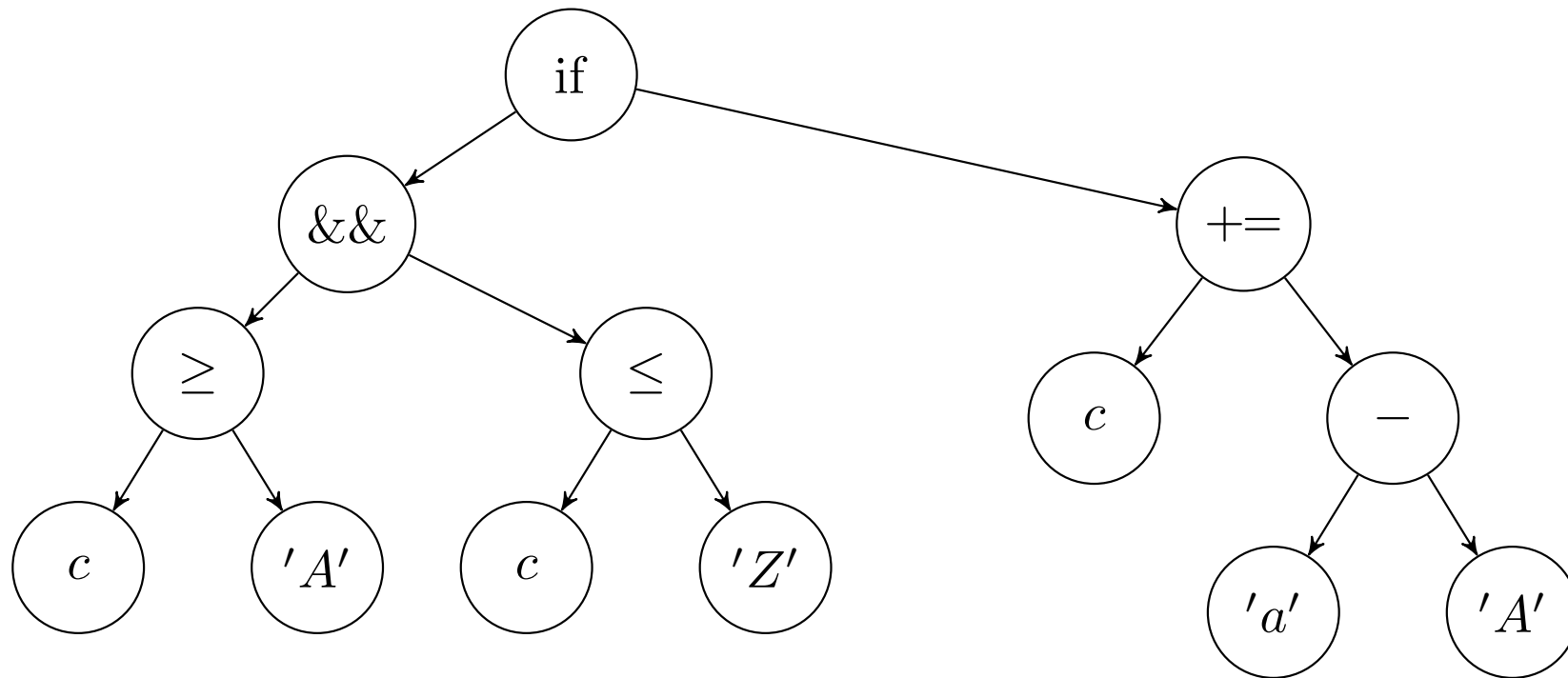
What is Parsing?

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.
```



Parsing

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 $\alpha \rightarrow w$, 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.

Here are a few examples of rewrite rules, for a Pascal-like language:

Stat \rightarrow **while** Expr **do** Stat
 \rightarrow **do** Stat **while** Expr
 \rightarrow **if** Expr **then** Stat
 \rightarrow **if** Expr **then** Stat **else** Stat
 \rightarrow **for** id **:=** Expr **to** Expr **do** Stat

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

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

(The $*$ evaluated first.)

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

(The $+$ evaluated first.)

When a same input word can be obtained in different ways by applying rules of 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:

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

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

$$E \rightarrow E + F \mid E - F \mid F$$

$$F \rightarrow F * G \mid F / G \mid G$$

$$G \rightarrow -G \mid H$$

$$H \rightarrow \text{double} \mid \text{ident} \mid (E) \mid \text{ident}(A)$$

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

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 5, we now defined Σ , 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 $\text{Stat} \rightarrow \mathbf{while} \text{ Expr } \mathbf{do} \text{ Stat}$ 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 problem.

Top-Down Parsing (2)

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

The same problem occurs with the rules $E \rightarrow E + F \mid E - F \mid F$ in the calculator.

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:

$$E \rightarrow F (+F \mid - F)^*.$$

Top-down parsing can also be implemented by hand, this is called **recursive descent parsing**.

The problems are the same.

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( "calling unknown function " ) + id );

    return 0.0; // An arbitrary 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 bottom up parsing 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 this moment, 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 5 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 `bool`.

The precondition decides if the reduction can happen.

Example from Prolog

Prefix => IDENTIFIER : id

%requires

```
{ return syntdefs. hasprefixdef( id ) &&  
    canstartterm( lookahead. value( )); }
```

%reduces

```
{ return syntdefs. prefixdef( id ); }
```

;

Infix => IDENTIFIER : id

%requires

```
{ return syntdefs. hasinfixdef( id ) &&  
    canstartterm( lookahead. value( )); }
```

%reduces

```
{ return syntdefs. infixdef( id ); }
```

;

Example from Prolog (2)

Term => Prefix:op Term:t

%requires

{ return canreduce(syntdefs, op, lookahead. value()); }

%reduces

{ return
 new functional(function(op. str, 1), { t }); }

| Term:t1 Infix:op Term:t2

%requires

{ return canreduce(syntdefs, 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 pretty good at error recovery. I copied 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.

I probably 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 we 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 '

Expectations 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 and code shown in this presentation can be downloaded from www.compiler-tools.eu/

Comparison to that Other Language

Java has no `variant`, and no `union`. The only mechanism for combining different types is inheriting from `Object`.

Exceptions in Java are annoying: You have to declare them, but if you declare them and don't use them, the compiler complains. This is annoying during development.

There is no easy way for turning code on and off
(`#if 0 ... #endif`)

Java has a portable intermediate representation. Because of that, there is no need to generate sources.

Java classes are very well documented on `docs.oracle.com`. (but `C++` has `cppreference.com`)

Thanks!

Thanks to Danel Batyrbek, Aleksandra Kireeva, Tatyana Korotkova, Akhmetzhan Kussainov, 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.