Backus–Naur Form (BNF)

Overview

BNF is one of the two main notation techniques for context-free grammars, often used to describe the syntax of languages used in computing; the other main technique for writing context-free grammars is the van Wijngaarden form.

History

John **Backus**, the principle designer of FORTRAN, and Peter **Naur**, a journalist for a computer magazine, both attend a conference on **Algol** in 1960 in Paris, France. On their return trip, they discussed the definition of Algol-60, as they had perceived it. They had both attended the same meetings and discussions and had come away from the conference with widely differing interpretations of what the language would look like and how it would work. Together they worked on refining a notation for describing the grammars of languages. This notation is called **BNF**, or Backus Naur Form. Some authors have dropped Peter Naur's name from their definitions and called it **Backus Normal Form** instead.

Note: ALGOL (ALGOrithmic Language) is one of several high level languages designed specifically for programming scientific computations.

Example

An **example** of a BNF grammar that describes the rules for the +, -, *, and / operators:

```
EXP    => EXP + TERM
EXP    => EXP – TERM
EXP    => TERM
TERM   => TERM * FACTOR
TERM   => TERM / FACTOR
TERM   => FACTOR
FACTOR => (EXP)
FACTOR => identifier
```

The names EXP, TERM, and FACTOR are just arbitrary names, which represent groups of symbols, for example, the FACTOR is defined as either an identifier (variable name) or an expression in parenthesis. Symbols that are composed of other symbols are called **Non-Terminal** symbols. Symbols that are composed of key words, operators, or names or any other symbol that may be found in a program are called **Terminal symbols**.
The **Terminal** symbols for this grammar are +, -, *, /, (,), and identifier. The **Non-Terminal** symbols are EXP, TERM, and FACTOR.

Parsing Methods Evaluation

Grammar Embedded in the Code

The earliest compilers were written with the definition of the language buried deeply within the code. With these compilers, it was very difficult to verify that the compiler accepted all of the language syntax and only the language syntax. This became especially difficult when the definition of the language was changed for later versions. All compilers before the early 1960's were of this type because there wasn't any uniform method of describing the language grammars.

Recursive Descent

With the advent of the **BNF** notation for describing the languages, compiler writers designed the structure of their subroutines and functions to correspond to the structure of the BNF definition of the language. To use our example grammar above, there would be **separate functions** to handle EXP's, TERM's, and FACTOR's. The EXP function would call itself and the TERM function, etc. This way, when it came time to update the language to meet changing standards, it would be easier to find where the changes should be made. It also made it much easier to verify that the language accepted all legal syntax and only the legal syntax.

Top Down Parsing (LL)

The top down parsing method, also called a **predictive** parse or **LL** parse, requires that we reorganize the grammar so that the first symbol of each rule defining a given **Non-Terminal** will indicate which rule to choose for the Non-Terminal. This transformation can be done to any grammar, but is sometimes **awkward**. There are also some cases that **cannot** be parsed **correctly** with Top Down Parsing methods.

Bottom UP Parsing (LR)

The bottom up parse, also called an **LR** parse is the most **powerful** parsing method. It also has the most **complicated** set of algorithms for building the parse tables. There are a set of algorithms for building LR parse tables. The same algorithm is used for all of the LR parse tables.

LR(0)

The first of the LR parse generation algorithms is called LR(0) and it generates tables that are somewhat **large**. The LR(0) parse algorithm do not parse grammars with certain types of **ambiguities**.

LR(1)

The second algorithm, which **handles all of the grammars** correctly is LR(1). The LR(1) algorithm generates correct parse tables for grammars **with all** of the ambiguities that are found in most useful languages. The biggest strike against LR(1) parse tables is the fact that the tables generated are much **larger** than the LR(0).

SLR

The third algorithm attempts to **handle some** of the ambiguities that LR(0) fails at and **keeps the size** of the parse tables the same as those generated by LR(0). It is called Simple LR.

LALR

The last algorithm, **Look Ahead LR**, generates parse tables that are the **same size** of LR(0), but **handles most** of the ambiguities that are handled by LR(1).

Look Ahead Left to Right Parser (LALR)

Overview

LALR parser or **Look-Ahead LR** parser is a **simplified** version of a **canonical** LR parser, to parse a text according to a set of **production rules** specified by a formal grammar for a computer language. ("**LR**" means **left-to-right**, rightmost derivation).

History

Frank DeRemer invented LALR parsers with his **PhD** dissertation, called "Practical LR(k) Translators", in **1969**, at MIT. This was an important breakthrough, **because** LR(k) translators, as defined by **Donald Knuth** in his **1965** paper, "On the Translation of Languages from Left to Right", were much too **large** for implementation on computer systems in the 1960s and 70's. The LALR parser became the most-powerful alternative to LR.
In **1977**, memory **optimizations** for the **LR** parser were invented **but** still the LR parser was **less** memory-**efficient** than the **LALR**.
In **1979**, **Frank DeRemer** and **Tom Pennello** announced a series of **optimizations** for the LALR parser that would further **improve** its memory **efficiency**. Their work was published in **1982**.

Advantages

In practice, **LALR** offers a good solution, because **LALR**(1) grammars are more **powerful** than **SLR**(1), and can parse **most** practical **LL**(1) grammars. **LR**(1) grammars are more **powerful** than **LALR**(1), however, **canonical** LR(1) parsers can be extremely **large** in size and are considered not practical. **Minimal** LR(1) parsers are **small** in size and **comparable** to LALR(1) parsers.

Disadvantages

Because the **LALR** parser performs a **right** derivation instead of the more intuitive left derivation, **understanding** how it works is quite **difficult**. This makes the process of finding a correct and efficient LALR **grammar** very demanding and **time-consuming**. For the same reason, **error-reporting** can be quite **hard** because LALR parser errors cannot always be interpreted into messages with high-level terms meaningful for the end user. For this reason, the **recursive descent** parser is **sometimes** preferred over the **LALR** parser. This parser requires **more hand-written** code because of its lower language-recognition power. **However**, it does not have the special difficulties of the LALR parser **because** it performs left-derivation. Notable **examples** of this phenomenon are the **C-language and C++** parsers of the **Gnu** Compiler Collection. These **started** as **LALR** parsers **but** were later **changed** to **recursive-descent** parsers.

YACC

Overview

Yacc is a computer program for the **Unix** operating system. The name is an acronym for "**Yet Another Compiler Compiler**". It is a **LALR** parser generator, generating a parser. It was originally developed in the early 1970s by "**Stephen C. Johnson**" at **AT&T** Corporation and written in the **B** programming language, **but** soon rewritten in **C**. It appeared as part of Version 3 Unix, and a full description of Yacc was published in **1975**.

Yacc produces only a **parser**; for full **syntactic** analysis this requires an external **lexical** analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper. Lexical analyzer generators, such as **Lex** or **Flex** are widely available. The **IEEE POSIX P1003.2** standard defines the functionality and requirements for both Lex and Yacc.

YACC Input

Input to YACC is divided into **three** sections. The ***definitions*** section consists of **token** declarations and **C** code **bracketed** by "**%{**" and "**%}**". The **BNF** grammar is placed in the ***rules*** section and user subroutines are added in the ***subroutines*** section.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Definitions Example

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER
```

Include standard input output library

Declaration of Lex Subroutine

Declaration of new subroutine

C code

Tokens Declaration

Rules Example

```
program:

    program expr '\n' { printf("%d\n", $2); }
    |
    ;

expr:

    INTEGER           { $$ = $1; }
    | expr '+' expr   { $$ = $1 + $3; }
    | expr '-' expr   { $$ = $1 - $3; }
    ;
```

First Non-Terminal

First Rule

Or Epsilon

Second Non-Terminal

Terminal

Or two other rules

$$ → Current Non-Terminal
$1 → Token No.1 in the rule
$2 → Token No.2 in the rule
$3 → Token No.3 in the rule

Subroutine Example

```
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();

    return 0;
}
```

Declaration for yyerror Subroutine using C-Language function

Declaration for main Subroutine

yyparse is a function to start parsing process

Subroutine is a **C-Language function** that will be called by the rules when necessary.

References

https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form
http://www2.andrews.edu/~bidwell/456/history.html
https://en.wikipedia.org/wiki/LALR_parser_generator
https://en.wikipedia.org/wiki/LALR_parser
http://epaperpress.com/lexandyacc/pry1.html