Semantic Analysis

Ettore Speziale

Politecnico di Milano

Contents

Contents

Syntax

"The study of the rules whereby words or other elements of sentence structure are combined to form grammatical sentences."

The American Heritage Dictionary

Syntactic Analysis I

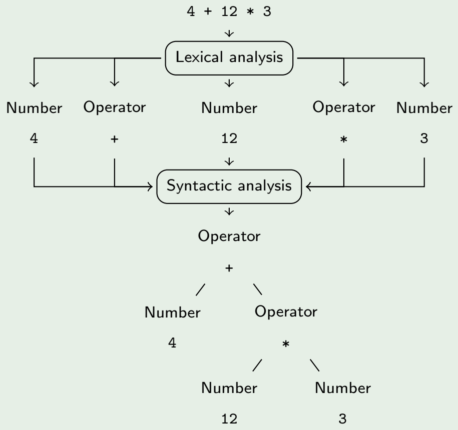
Given an input text we need to determine its *structure*:

- how statements are linked together
- operator precedence rules
- ...

The structure is defined by mean of a *grammar*.
Syntactic analysis is performed over *words*:

- the input is a tokenized stream
- usually a lexical analyzer prepares input for the semantic analysis

Structure of an algebraic expression



Semantic Analysis

It is the evaluation of the meaning of each (terminal and non-terminal) symbol, achieved by *decorating the Abstract Syntax Tree*:

Syntactic analysis

```
           Operator
              +
            /   \
      Number     Operator
        4           *
                  /   \
            Number     Number
              12         3
```

Semantic analysis

```
             Value 40
             Operator
                +
              /   \
      Value 4       Value 36
      Operator      Operator
         4             *
                     /   \
             Value 12     Value 3
              Number      Number
                12          3
```

Contents

Parsing

A *parser* is a program that performs syntactic analysis.

Typically:

- **LL** descending parsing, can be constructed by hand (`c-parser.c` in `GCC` sources) or automatically (`ANTLR` Java parsers generator)
- **LR** ascending parsing, usually too complex to be constructed manually

Common duty: building the Abstract Syntax Tree.

bison

The standard tool to generate LR parsers is `bison`:

- free implementation of `yacc`
- strongly coupled with flex
- actually a LALR(1) parser generator

Getting `bison`

Available in your distribution repositories:

Debian `aptitude install bison`

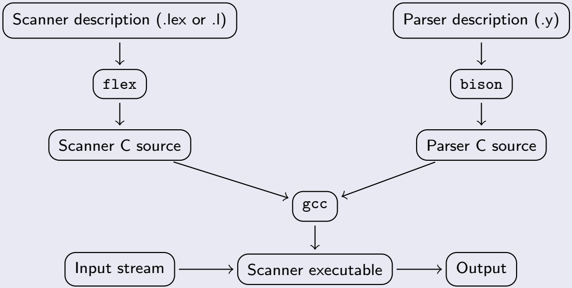
Fedora `yum install bison`

Parser Building

A parser consume tokens:

- a scanner must produces tokens
- natural choice is `flex`

Using `bison` and `flex` together



A Simple Example

Let's try to build a reverse polish notation calculator.

Grammar

$$
\begin{aligned}
S &\rightarrow E|\epsilon \\
E &\rightarrow NUMBER \\
E &\rightarrow EE+|EE*
\end{aligned}
$$

Don't worry about terminals:

- it is a scanner duty

The bison Input File

The `bison` input file resemble the one of `flex`:

C definitions header inclusions, var declarations, ...

definitions tokens, precedences, ...

grammar rules rules and semantic actions

user code main and service functions

bison input file [a]

```
%{
/* C definitions */
%}
/* Definitions */
%%
/* Grammar rules */
%%
/* User code */
```

[a] C89-style comments can appear in any of the sections.

Do You Remember flex? I

We must provide a scanner to bison:

- just implement the yylex function
- maybe better to exploit flex

scanner.l global section

```
%option noyywrap
%{
#include "rpn.tab.h"
#define UNKNOWN -1
%}
DIGIT [0-9]
BLANK [ \n\r\t]
%%
```

Do You Remember flex? II

scanner.l rules section

```
{BLANK}
{DIGIT}+ { return NUMBER; }
"+" { return OP_PLUS; }
"*" { return OP_MUL; }
. {
    yyerror("Unknown␣char");
    return UNKNOWN;
  }
```

There is no need to add extra C code:

- flex is only used to tokenize the input

Parser Definition I

Let's start with a parser that *recognize* reverse polish notation expressions:

rpn.y definitions section

```
%{
#include <stdio.h>
%}
%token NUMBER
%token OP_PLUS
%token OP_MUL
%%
```

The `%token` directive allows to define words read by the parser.

Parser Definition II

Syntax for grammar definition is straightforward:

rpn.y grammar section

```
calculus: /* Empty */
        | expression
        ;
expression: NUMBER
          | expression expression OP_PLUS
          | expression expression OP_MUL
          ;
%%
```

Parser Definition III

The last section contains:

- the error handling function `yyerror`
- the program entry point `main`

`rpn.y` C code

```
int yyerror(char* msg) {
  printf("%s\n", msg);
  return 0;
}

int main(int argc, char* argv[]) {
  return yyparse();
}
```

Compiling sources I

From the parser (`rpn.y` file) we build:

- the parser itself (`rpn.tab.c`)
- a description of tokens (`rpn.tab.h`)

Parser and scanner generation

```
$ bison -d rpn.y
$ flex scanner.l
```

Compiling sources II

To get the final executable compile and link:

Get your own polish parser

```
$ gcc rpn.tab.c lex.yy.c
```

I am lazy:

Using make [1]

```
YFLAGS=-d
rpn: rpn.o scanner.o
clean:
	rm -f rpn y.tab.h *.o
```

[1] Filenames are slightly different.

Adding semantic I

Beside each rule it is possible to add a code-block performing a semantic action:

- the semantic action is executed in the context of the associated rule

Rules full syntax

```
lhs: rhs_1 { ... }
   | rhs_2 { ... } rhs_3 { ... }
```

The `lhs` rule is an alternative:

- each alternative is independent from the other
- the first contains a semantic action
- the second contains two semantic actions

Adding semantic II

Semantic actions are executed just after the preceding rule. Given:

```
lhs: rhs { ... }
```

The parser:

1. recognizes `rhs`
2. executes the semantic action
3. recognizes `lhs`

The action is placed at rule tail:

- it is executed *every time* `lhs` is recognized

Adding semantic III

Given:

```
lhs: rhs1 { ... } rhs2 { ... }
```

The parser:

1. recognizes `rhs1`
2. executes the first semantic action
3. recognizes `rhs2`
4. executes the second semantic action
5. recognizes `lhs`

Semantic actions not at the tail of a rule are called *actions in the middle*.

Adding semantic IV

This is a logical view of semantic action execution:

- the execution of semantic actions can be postponed due to ambiguity

Semantic Values I

A variable is associated to every symbol:

- an `int` by default
- no distinction between terminal and non-terminal
- type customizable via `%union` directive [2]

Inside actions is possible to use these vars:

- accessed throuh `$n` notation
- index are 1-based
- the left-hand side semantic variable is `$$`
- counting includes semantic actions

Semantic Values II

Variables enumeration

Given:

```
lhs: rhs1 { ... } rhs2 { ... }
```

We have:

Component	Variable
`lhs`	`$$`
`rhs1`	`$1`
`{ ... }`	`$2`
`rhs2`	`$3`
`{ ... }`	`$4`

Semantic Values III

Obviously inside a semantic action we can access only variables associated to preceeding rules:

- rhs-vars mostly accessed in read-mode [3]

With an exception: the $$ variable:

- it is a *synthesized* attribute
- always written
- available only in the semantic action [4]

Default semantic action:

- `{ $$ = $1; }`

---

[2] More on this on next lesson.

[3] LALR parsing is bottom-up.

[4] The code block at rule tail.

Add and Multiply I

We must assign a semantic value to terminals:

scanner.l scanning naturals

```
{DIGIT}+ {
              yylval = atoi(yytext);
              return NUMBER;
         }
```

The `yylval` variable is declared by `bison`:

- must be filled with the semantic value of the terminal

Add and Multiply II

Sums and products must be performed by the parser:

rpn.y computing actions

```
expression:
  NUMBER { $$ = $1; }
  | expression expression OP_PLUS {
      $$ = $1 + $2;
    }
  | expression expression OP_MUL {
      $$ = $1 * $2;
    }
  ;
%%
```

Add and Multiply III

At last print the expression evaluation:

rpn.y reporting action

```
calculus:
  /* Empty */
  | expression {
      printf("Result:␣%d\n", $1);
    }
  ;
```

Ambiguity I

Consider the grammar of infix expressions:

Grammar

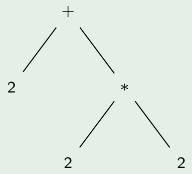
$$
\begin{aligned}
S &\rightarrow E | \epsilon \\
E &\rightarrow NUMBER \\
E &\rightarrow E + E | E * E
\end{aligned}
$$

It has a big problem: it is ambiguous!

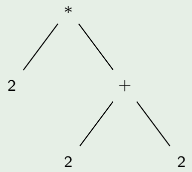
Ambiguity II

Let's try to generate $2 * 2 + 2$:

Produce $+$ first



Produce $*$ first



The grammar ambiguity between $+$ and $*$ rules generates a semantic ambiguity:

- what are the $+$ and $*$ precedences?

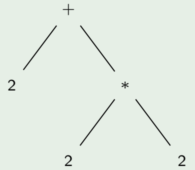
How To Resolve Ambiguity I

From theory, we can rewrite the grammar in a non ambiguous form:

Unambiguous grammar

$$
\begin{aligned}
S &\rightarrow E \mid \epsilon \\
E &\rightarrow E + T \mid T \\
T &\rightarrow NUMBER \\
T &\rightarrow T * NUMBER
\end{aligned}
$$

Unique tree



How To Resolve Ambiguity II

Since token are the same, we build only the parser:

`infix.y` rules [5]

```
expression:
  term { $$ = $1; }
  | expression OP_PLUS term {
      $$ = $1 + $3;
    }
term:
  NUMBER { $$ = $1; }
  | term OP_MUL NUMBER {
      $$ = $1 * $3;
    }
```

[5] Scaffolding is unchanged.

Precedence

Another way to handle operator precedence is to tell bison the precedence relation:

Precedence with bison [a]

```
%left TOKEN_1 TOKEN_2
%left TOKEN_3
```

[a] Precedences declared inside definitions section.

- TOKEN_1 and TOKEN_2 have the same precedence
- both have lower precedence than TOKEN_3

Associativity I

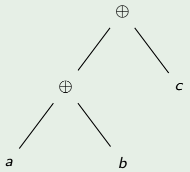
An operator $\oplus$ can be:

left-associative $a \oplus b \oplus c = (a \oplus b) \oplus c$

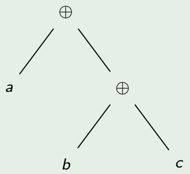
right-associative $a \oplus b \oplus c = a \oplus (b \oplus c)$

Associativity reflects on parsing:

AST of left-associative $\oplus$



AST of right-associative $\oplus$



Associativity II

Inside a `bison` file it is possible to declare the associativity of operators:

- operators are *tokens*

`bison` directives for operators associativity

Syntax	Meaning
`%left TOKEN`	`TOKEN` is left-associative
`%right TOKEN`	`TOKEN` is right-associative

Ambiguous Infix Calculator I

Declaring operator precedences allows to write ambiguous rules:

infix-ambiguous.y rules

```
expression:
  NUMBER { $$ = $1; }
  | expression OP_PLUS expression {
      $$ = $1 + $3;
  }
  | expression OP_MUL expression {
      $$ = $1 * $3;
  }
```

Ambiguous Infix Calculator II

Disambiguation is performed by `bison` consulting operator precedences:

Unambiguous tokens

```
%token NUMBER
%token OP_PLUS
%token OP_MUL
```

Ambiguous tokens

```
%token NUMBER
%left OP_PLUS
%left OP_MUL
```

Context-dependent Precedence I

Sometimes a character has a dual meaning:

- the - identifies both subtraction and unary minus

First of all, let's modify the infix scanner to recognize -:

infix-scanner.l minus token

```
"-" { return OP_MINUS; }
```

Context-dependent Precedence II

In the parser we introduce:

- the subtraction token `OP_MINUS`
- the unary minus `OP_UNARY_MINUS`

The latter is a *fake* token used to declare a precedence.

`infix-minus.y` minus token

```
%token NUMBER
%left OP_PLUS OP_MINUS
%left OP_MUL
%left OP_UNARY_MINUS
```

Context-dependent Precedence III

In the rules section we can force the right precedence:

infix-minus.y minus rules

```
expression:
  ...
  | expression OP_MINUS expression {
      $$ = $1 - $3;
    }
  ...
  | OP_MINUS expression
    %prec OP_UNARY_MINUS {
      $$ = -$2;
  }
```

Contents

Parse-First

Using `bison` requires both:

- writing the grammar
- adding semantic actions

Write the grammar first!

- try some examples
- if they are recognized, add semantic actions

Simple Grammars

As in coding, follow some conventions while writing grammars:

- terminals (tokens) are uppercase
- not-terminals are lowercase
- . . .

This keeps the grammar readable!

Contents

Bibliography

GNU.
GNU `bison` Info Pages.
info bison, 2006.

Formal Languages and Compilers Group.
Software Compilers.
http://compilergroup.elet.polimi.it, 2010.