Compilation
TP 2: Parsing

C. Alias & S. Filip[*]

credits: G. Iooss

Part I – Top-Down Analysis

Exercise 0. *Warming up*

Compute `NULLABLE`, `FIRST` and `FOLLOW` for each token of the following grammar:

$$
\begin{aligned}
S &\to uBDz \\
B &\to Bv|w \\
D &\to EF \\
E &\to y|\epsilon \\
F &\to x|\epsilon
\end{aligned}
$$

Exercise 1. *Implementing LL(k)*

Download and unzip `LL.tar.gz`. This package contains (besides `main.cpp` and `Makefile`):

- `Token.h` defines the `Token` class (terminal and non-terminal symbols) building blocks of a formal grammar. This class provides 2 constructors: `Token()` to build ϵ and `Token(string nLex, bool isTerm)` to build a token (`isTerm` indicate a terminal token).
- `Grammar.h` defines the `Grammar` class which represents a context-free grammar. This class provides a constructor: `Grammar(multimap<Token*, vector<Token*>> rules, Token* base)` where `rules` is the set of rules and `base` is the start symbol. The `lexGram` attribute is the set of all the tokens appearing somewhere in the grammar. The `nullable`, `first` and `follow` have to be computed by calling `fst_follow()`.

  By the way, a `multimap` has the same behavior than a `map` except that it allows multiple value for a given key and returns the last one.

To do:

- Build the grammar from Exercise 0.
- Compute NULLABLE, FIRST and FOLLOW sets, then display them to check your answer to Exercise 0.[1]
- Compute and display the directors of grammar rules. Show clearly if yes or no this grammar is LL(1).
- Modify (slightly) the grammar of Exercise 0 such that the same language is accepted but the grammar is LL(1).
- Compute and display the transition table for this new grammar.

[*] `http://perso.ens-lyon.fr/silviuioan.filip/compilation`

[1] Tool tip: `operator<<` has been overloaded for your convenience

Part II – Bottom-Up Analysis

Exercise 2. *Warming up*

Lets take into consideration the following grammar over $\Sigma = \{\text{if}, \text{then}, \text{else}, \text{inst}\}$:

$$\begin{aligned} Z &\to S\$ \\ S &\to \text{if } E \text{ then } S \text{ else } S \\ S &\to \text{if } E \text{ then } S \\ S &\to \text{inst} \end{aligned}$$

Questions.

- Draw the automaton LR(0)
- Is the grammar LR(0)? SLR(1)? LR(1)?

Exercise 3. *Bison*

As we see, it would be tedious to build the automaton for a big actual grammar... A lot of tools exists that automatically generates C code for a fully functional syntactic analyzer directly from a grammar. *Bison* is one of them (it is based on *Yacc*, thus the pun).

A) Integrating flex and bison

Download and unzip `src0_if.tar.gz`. The file `parser.ypp` contains the specification of the grammar. As with `flex`, this file contains three parts (separated by $$):

- **First part** contains declarations:

  %{ ... %} (lines 5 – 23) contains raw C code that will be copy/pasted at the beginning of the generated analyzer. Typically, the is where we add `#include` of files that declares objects of attributes we want to generate or this is where we declare variables used in the semantic actions (see below). Bison is setup to build C++ code simply by giving a `.ypp` extension instead of the `.y` one. Thus allowing to put C++ code here.

  %union { ... } (lines 30 – 33) enumerate the types that can be used for attributes (see below). Here we only allow `char*` attributes that will be referenced as "`string`".

  %token (lines 40 – 41) enumerate terminal symbols of the grammar. Generally, they are the tokens generated by a lexical analyzer, like *flex*.

  %type (line 43) associate an attribute type to a symbol (terminal or not). For terminal symbols, the attribute is produced by the lexical analyzer. Therefore the syntactic analyzer and the lexical analyzer have to use the same data structures.

  %start (line 46) define the start symbol of the grammar.

- **Second part (lines 49 – end)** declare the rules of the grammar. Productions emanating from the same non-terminal are regrouped in one rule ended by "`;`" where each part is separated by "`|`".
- **Third part (absent)** begin by %% and can contain raw C code that is appended at the end of the syntactic analyzer. Typically a `main` function for grammar driven compilers.

To do:

- **Compile your** `.ypp` to analyze the grammar of the previous exercise.
- **Compile using** `make`. `make` produce the C code of the analyzer using the following command: `bison --defines=parser.h -o parser.cc parser.ypp`. *Bison* also generate `parser.h` which contains the declarations. **Open the file** `parser.h`. This file is used by the lexical analyzer (flex) to produce the tokens of the non-terminals and their attributes. This way, Flex use the same structures than Bison.
- **Open the file** `lexer.l` to see that we `#include "parser.h"` (ligne 7) and that we use TK_IF, TK_THEN, etc. declared in the file `.ypp`. How is the identifier string forwarded?

B) Resolving shift/reduce conflicts

As we saw it, this grammar produce conflicts that are not resolved by LALR(1). This is why Bison display a warning: `parser.ypp: conflicts: 1 shift/reduce`. Obviously, it would be useful if we had informations as to where it comes from, in order to correct the grammar.

To do:

- **Open** `Makefile` and add the options `--report=lookahead --report-file=bison_report` in the bison command line.
- **Recompile and open the file** `bison_report`. It contains, among others, a text representation of the grammar LALR(1) automaton with indications on conflict resolutions. **Go to state 8**. What is the default choice of the analyzer? In general, shift/reduce conflicts are unavoidable on big grammars. We should verify that bison makes a good choice every time.

C) Resolving reduce/reduce conflicts

Those are real bugs in the grammar that need correction. The idea is that a reduce/reduce conflict mean that a word have two different syntactic interpretation. For example: the string `int*x` could be a declaration of a variable `x` of type `int*`, or the computation of the product between the variable `int` and `x`.

To do:

- Download and unzip **src1_stmt.tar.gz**. Look at the grammar. Where does the reduce/reduce conflict comes from? How does bison solve it?

Exercise 4. *The C grammar*

Or at least a significant subset. Download and unzip **src2_parser.tar.gz**.

To do:

- **Inspect the grammar** with the help of `tests/tree.c`. Identify the syntactic categories.
- **Compile**. How many shift/reduce conflicts? **Open the file** `bison_report` and thoroughly analyse each conflict and the solution proposed by bison.

Exercise 5. *Action!*

Until now, our analyzers are passive oracles. We would like to execute code during the analysis and produce the intermediate representation. This is what allows *attribute grammar*. We associate at each production a piece of code that will be executed each time the production will be reduced. This piece of code is called *semantic action* and compute the attributes of non-terminals. Lets consider the following grammar:

$$
\begin{aligned}
&Z \to E\$ \\
&E \to E + T \\
&E \to T \\
&T \to T * F \\
&T \to F \\
&F \to id \\
&F \to (E)
\end{aligned}
$$

Questions.

- Attribute the grammar to evaluate arithmetic expressions.
- Execute your grammar against 1+(2*3). We will apply a dynamic evolution (we evaluate things at each non-terminal instead of at the end).
- If we use a LR analyzer, in which order will the actions be executed? At which tree traversal strategy does this corresponds?

To do:

- **Download and unzip src3_ETF.tar.gz**. **Open parser.ypp**. Bison only allows one attribute per symbol. We declare its type using %type (line 55). In a rule, the attribute of the `i`th symbol is collected in `$i` (line 68). The attribute of the current derived non-terminal is collected in `$$`. Every attribute is synthesized. Thus an action consists in computing `$$` from every `$i`. Which will be forwarded in its turn into a `$i` in an other rule, etc.
- **Complete the grammar** to evaluate the expression. **Display the number of each reduced rule**.
- We would like to build the abstract syntax tree (AST) from the expression. Use Ast.h/.cc and **modify your grammar adequately**.

Exercise 6. *Bonus: Return of LL*

Questions.

- Compute $First_2$, $Follow_2$ and $Directeur_2$ from the grammar of exercise 0, knowing that:
  - $First_k$ is the set of k letters words that can be the beginning of a derivation.
  - $Follow_k$ is the set of k letters words that can follow a symbol.
  - $Director_k$ is the set of k letters words that can follow the application of a rule.
- Find an algorithm to compute those sets (you can use $First$ and $Follow$ in the computation).
- Generalize to $First_k$, $Follow_k$ et $Director_k$.
- And if you are motivated, implement all that!