DSL course: individual work #2

DSL context-free grammar work

Tasks

- ☑ a data structure for grammar;
- ☑ remove useless non-terminal symbols(dead and unreachable);
- ☑ determination of vanishing symbols;
- ☑ removing of vanishing symbols;
- ☑ determination of left-recursion in a grammar;
- ☑ remove chain productions;
- ☑ remove left-recursion;
- ☑ factorization;
- ☑ recursive descent parsing.

Additional:

- ☑ left-recursion removing optimization(order based on distance to the initial non-terminal);
- ☑ recursive descent parsing optimization(prediction using FIRST mapping).

How to use

Clone/download the repository and execute `python app.py`.

Use `python3` (the project was tested by `python3.6`), because the whole project uses it features(type hints, print, next and etc built-in functions(!)).

It imports only standard libraries and local grammar library, no external dependencies.

Testing

The application was tested by 3 grammars and related test cases:

- simple HTML, words;
- arithmetic operations, words;
- positive integers, words.

The application requires file of grammar and its file of words.

Grammar format

The format is similar to BFN(`<>` describes non-terminal, `::=` separates non-terminal from its rules), except terminals are not written in quotes(') and there is limitation of using escaped symbols (for example, newline symbol separates production set one from another). But you can use escaped `\<`, `\>`, `\|` to present the symbols of `<`, `>` and `|` in your grammars (see HTML example).

As non-terminals you can use any strings, but for application representation they will be transformed into integers in the order they were met by grammar "parser".

Examples of grammars are listed above.

Word format

Every line of the file is word to check. Every file must begins with `[true]` or `[false]` control sequences. They tell the application, should the words below be true or false (truth expectation). Also, you can write something like that:

```
[true]
...
[false]
...
[true]
...
```

where `...` is some words.

Also, limitations that have been described above are suitable here.

Grammar definitions

Context-free grammar consist of two main things:

- initial non-terminal;
- mapping from non-terminals to derivation sets, where derivation is a tuple of terminals or non-terminals.

Terminal is any character. Non-terminal is represented as integer in the implementation, but formally it is symbol that can be replaced by its rules.

Formally grammar looks like:

```
S —> Aab|BC
A —> a|b
C —> c
```

where `S`, `A`, `B` and `C` — non-terminals, `a`, `b`, `c` — terminals. Here `B` is a dead symbol(there are no terminal derivations of any step from it) and should be deleted as well all productions where it occurs.

```
S —> Aab
A —> a|b
C —> c
```

where `C` became unreachable — there are no derivations from `S` that contain `C`.

There are left-recursion of two types: direct and indirect.

This is grammar with direct recursion.

```
S —> S|a|b
```

This is grammar with indirect recursion.

```
S —> A|a|b
A —> S
```

Grammar implementation

The whole project is properly documented, so it is not needed to be brave to dig into it. :D

Project implements grammar class, related data types, and required in algorithms data structures.

The process of word checking consists of two parts:

- preparing grammar to recursive descent parsing;
- descent parsing itself.

Because recursive descent parsing is applied there, there is problem of left recursion. That and other cases are handled in `prepare_for_checking` method. It returns parsing-ready grammar, formed from input grammar.

The method does the following steps:

- determines left-recursion in the grammar;
- if yes:
  - remove vanishing symbols;
  - remove chain productions;
  - remove useless(because they can exist in the initial grammar and removing of vanishing and chain productions can generate some of them);
  - remove left-recursion;
  - add empty word to the grammar if it was in initial.

- if no:
  - left factorization;

- remove useless.

Removing of useless

Useless non-terminal symbol is dead or unreachable one.

To determine a dead symbol, undead must be determined. If some rule of the non-terminal symbol is fully terminal it is undead. If there is some terminal derivation from non-terminal, the symbol is undead too.

To determine a unreachable symbol, enough to run one of graph bypass algorithms.

Removing of vanishing

1. determine all `A` non-terminals with the following rules `A –> ε|...`;
2. if $A \text{ –> } \alpha|\beta_1...$, and $\alpha := \gamma_1...\gamma_k$, and all `γ` are vanishing, `A` is also vanishing;
3. all productions of `A –> ε` must be deleted and all productions of form `B –> αAβ`, where `α` and `β` some sequences of terminals, non-terminals or empty word, and `A` is vanishing, must be split into `B –> αAβ | αβ`. Repeat until all possible variants will be present.

Removing of chain productions

Example of chain productions.

```
A –> B |...
B –> C |...
C –> ...
```

`A` has indirect transition to `C` because of `A -> B` and `B -> C` and this are called chain productions.

1. determine all direct chain productions;
2. build all possible combinations of present chain productions;
3. for every pair `A` and `B`, remove `A -> B`, add all α_i from $B \rightarrow \alpha_1|...|\alpha_k$ to `A`.

Removing of left recursion

1. order non-terminals from 0 to n;
2. removing of indirect: for productions from 0 to n, if rule of `A` has non-terminal `B`, which less by the order than `A`, at the first position, replace it by `B` rules;
3. removing of direct: for all non-terminals of form $A \rightarrow A\alpha_1...A\alpha_k|\beta_1...\beta_n$, remove rules starting with `A` and add rules $A \rightarrow \beta_1A'...\beta_kA'|\beta_1...\beta_n$ as well as $A' \rightarrow \alpha_1A'...\alpha_kA'|\alpha_1...\alpha_n$.

Left factorization

Factorization select common prefixes in rules and transfer different suffixes as rules of new non-terminal symbol.

To achieve it, I used prefix tree and tree restoring algorithm which combines rules with common prefixes to new non-terminals recursively.

About optimizations

To speed up left-recursion removing algorithm, I used order based on distance to initial non-terminal. Initial non-terminal has number 0(has no effect with the current grammar parser, because it writes integer representations of non-terminal symbols in order of their appearing in rules).

To speed up recursively descent method, I used FIRST dictionary: it contains pairs of non-terminal and symbol which are mapped into a set of derivations that have that symbol at theirs first position. It means, that you can predict possible substitution rules using the dictionary.