yatta (Yet Another Tool for Text Analysis) parser generator
by Maciej Piróg

PODRĘCZNIK UŻYTKOWNIKA

WSTĘP

Yatta Parser Generator to prosty generator rekurencyjnych, zstępujących parserów LL z nawracaniem oraz lekserów. Jako parametr pobiera plik z gramatyką, w odpowiedzi tworzy pliki zawierające klasy służące do analizy leksykalnej i składniowej języków opisanych daną gramatyką. Kod generowany jest w języku C++.

WEJŚCIE

Yatta rozpoznaje własny format gramtyk (pliki .yg – yatta gramar). Plik yg składa się z dwóch części – definicji leksera i definicji parsera.

DEFINICJA LEKSERA

Definicja leksera rozpoczyna się od słowa kluczowego `LEXER:`, po którym następuje ciąg definicji symboli leksykalnych. Każda definicja rozpoczyna się od słowa kluczowego `token`, `skip` lub `priv`, po którym następuje nazwa symbolu.

Symbole oznaczone jako skip są rozpoznawane podczas analizy leksykalnej, lecz są ignorowane w dalszych fazach. Np. komentarz w języku C może być oznaczony jako `skip`, wówczas zostanie rozpoznany przez lekser, lecz nie będzie widoczny dla parsera.

Tokeny `priv` nie są rozpoznawane jako jednostki leksykalne, lecz służą do budowy innych tokenów, np. `cyfra`, choć może nie stanowić osobnego symbolu, może być przydatna przy budowaniu tokenu `liczba_rzeczywista`.

Każda defnicja tokenu jest konkatenacją grup, które zapisywane są w nawiasach `(` i `)`. Każda grupa stanowi alternatywę symboli podstawowych: liter, napisów lub innych tokenów. Kolejne składniki alternatywy oddzielamy symbolem `|`. Dodatkowo każda grupa G może być oznaczona jako `(G)*` (domknięcie; wystąpienie 0 lub więcej razy) lub jako `(G)?` (wystąpienie grupy jest opcjonalne).

Symbolem podstawowym może być pojedynczy znak (zapisywany jako `'a'`), przedział znaków

(zapisywany jako `'a'..'z'`, czyli alternatywa wszystkich znaków od a do z), napis (zapisywany jako `"napis"`) lub identyfikator inego tokenu. Dodatkowo przed symbolem podstawowym może znajdować się negacja (zapisywana jako `~`), która oznacza “dowolny znak, za wyjątkiem...”.

Defnicja symbolu leksykalnego kończy się symbolem `;` (średnikiem).

Przykładem definicji symboli leksykalnych mogą być:

```
priv DIGIT ('0'..'9'); − dowolna cyfra
token NUMBER (DIGIT) (DIGIT)*; − liczba całkowita
skip WHITESPACE ('\n' | '\t' | ' ' | '\r'); − biały znak
skip COMMENT ("/*") (~"*/")* ("*/"); − komentarz w języku C
priv LETTER ('a'..'z' | 'A'...'Z' | '_');
token ID (LETTER) (LETTER | DIGIT)*; − identyfikator
```

Wygenerowany lekser jest zawsze lekserem zachłannym, tj. jako token rozpoznawany jest najdłuższy ciąg symboli spełniających daną regułę. Jeśli najdłuższy ciąg spełnia więcej niż jedną definicję, jest rozpoznawany jako token, którego definicja zadeklarowana jest później, np. jeśli lekser zadeklarowany jest następująco:

```
LEXER:
token STRING ('a'..'z' | 'A'...'Z' | '0'..'9')*;
token ID ('a'..'z' | 'A'...'Z') ('a'..'z' | 'A'...'Z' | '0'..'9')*;
```

ciąg “`napis`“ zostanie zinterpretowany jako `ID`. Definicja leksera powinna więc przechodzić od definicji symboli najogólniejszych, do szczegółowych. Definicje słów kluczowych powinny znajdować się pod definicjami identyfikatorów.

DEFINICJA PARSERA

Definicja parsera rozpoczyna się słowem kluczowym `PARSER:`, po którym następuje ciąg definicji produkcji. Każda produkcja rozpoczyna się słowem kluczowym rule, po którym następuje jej nazwa.

Każda produkcja jest alternatywą (zapisywaną przy pomocy symbolu `|`) konkatenacji grup (zapisywanych w nawiasach `(` i `)`). Każda grupa jest konkatenacją tokenów lub identyfikatorów innych produkcji. Obowiązują modyfikatory `*` i `?` z takim samym znaczeniem jak w przypadku definicji leksera.

Defnicja produkcji kończy się symbolem `;` (średnikiem).

Przykładem gramatyki może być[1]:

```
LEXER:
priv DIGIT ('0'..'9');
```

1 Mimo, iż w przykładzie nazwy tokenów pisane są wielką literą, nie jest to wymóg. Zarówno nazwy tokenów i produkcji mogą być dowolnymi identyfikatorami niezaczynającymi się od cyfry.

```
token NUMBER (DIGIT) (DIGIT)*;
token ADD ('+' | '-');
token MULT ('*' | '/');
token LP ('(');
token RP (')');

PARSER:
rule main (add_expr);
rule add_expr (mult_expr) (ADD mult_expr)*;
rule mult_expr (base_expr) (MULT base_expr)*;
rule base_expr (LP add_expr RP) | (NUMBER);
```

Należy pamiętać, że generowny parser jest parserem LL i zapętla się przy lewostronnej rekursji, np.
`rule add_expr (add_expr ADD mult_expr) | mult_expr;`
jest poprawną definicją produkcji, lecz wygenerowany parser zapętli się przy próbie analizy produkcji `add_expr`.[2]

DRZEWA ROZBIORU

Wynikiem działania wygenerowanego parsera jest drzewo wyprowadzenia ciągu danego na wejściu. Drzewo to reprezetowane jest jako obiekt klasy `node`, która generowana jest razem z parserem:

```
struct node {
    node_type type;
    token t;
    rule_types rule;
    vector<node> children;
}
```

Pole `type` przechowuje typ danego węzła drzewa. Może to być `n_token`, jeśli w węźle znajduje się symbol terminalny lub `n_rule`, jeśli jest to produkcja. Pole `rule` przechowuje nazwę zastosowanej w tym miejscu przez parser produkcji gramatyki (wszystkie nazwy produkcji zapisane są z prefiksem `r_`, by zapobiec konfliktom nazw). Pole `t` przechowuje informacje o symbolu terminalnym, zawarte w obiekcie klasy token:

```
struct token {
    token_types type;
    string type_name;
    string text;
    int line;
}
```

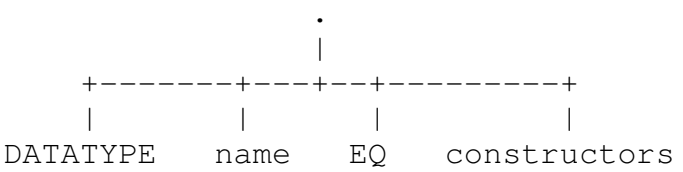
Pole `type` przechowuje nazwę typu tokenu, `type_name` – jego nazwę w reprezentacji napisowej. Pole `text` przechowuje treść tokenu (np. zawartość napisu czy identyfikator). W polu `line` zapisany jest numer linii, w której znajduje się dany symbol, jest to przydatne np. w raportowaniu o błędach.

2 W obecnej wersji Yatta nie wykrywa lewostronnej rekursji w trakcie generowania parsera, więc obowiązek zapewnienia poprawnej gramatyki spoczywa w całości na użytkowniku

W celu ułatwienia późniejszej analizy drzewa można modyfikować jego węzły już w czasie fazy analizy składniowej. Do tego celu służą symbole `!` i `^` zapisywane przed nazwami tokenów w definicji parsera. Token oznaczony jako `!` nie znajdą się w drzewie wyprowadzenia, a token oznaczony `^` zostanie zapisany w węźle o jeden poziom wyższym. Np. możliwa produkcja opisująca deklarację typu w języku SML mogłaby wyglądać następująco:

```
LEXER:
token DATATYPE ("datatype");
token EQ ('=');
// ...
PARSER:
rule dtype = (DATATYPE name EQ constructors);
```

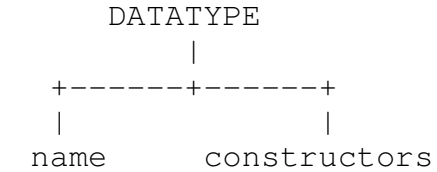
Wówczas drzewo wyprowadzenia miałoby kształt:



Zaś definicja :

```
rule dtype = (^DATATYPE typename !EQ constructors);
```

spowoduje usunięcie z drzewa całkowicie zbędnego znaku = i przesunięcie tokenu `DATATYPE` w górę:



UŻYCIE

Yatta generuje pięć plików:

nazwa pliku	***zawartość***
lexer.h	deklaracja klasy klasy `token`, `lexer`, klas pomocniczych leksera i wylicznia nazw tokenów
lexer.cpp	implementacja klasy `lexer`
parser.h	deklaracja klasy `parser`
parser.cpp	implementacja klasy `parser`
ast.h	klasa `node` i wylicznie typów produkcji

By przeprowadzić analizę leksykalną i składniową, należy wywołać metodę run klasy parser. Jako parametr pobiera ona napis wejściowy, wartością zwróconą jest obiekt klasy node, np.:

```
node n = p.run(in);
```