Maciej Bielarz
Artur Jabłoński

Styczeń 2006

Error Recovery w parserach typu bottom-up

Błędy w programie mogą występować na kilku poziomach:

- leksykalnym ("literówki" identyfikatorów, operatorów, słów kluczowych)
- syntaktycznym (np. wyrażenie arytmetyczne z niepoprawnie zamykanymi nawiasami)
- semantycznym (np. użycie operatora do nieodpowiednich operandów)
- logicznym (np. nieskończone wywołanie rekursywne)

Znaczna część procesu wykrywania błędów skoncentrowana jest wokół analizy syntaktycznej. Spowodowane jest to faktem, że większość błędów ma z natury charakter syntaktyczny lub ich wykrycie związane jest z naruszeniem reguł gramatyki danego języka, przez napływające z analizatora leksykalnego tokeny. Dokładne wykrywanie błędów natury semantycznej czy logicznej w czasie kompilacji jest o wiele bardziej skomplikowanym zadaniem i często wręcz niemożliwym.

Obsługa błędów w parserach ma następujące zadania:

- jasne i dokładne zgłaszanie obecności błędów
- po wykryciu błędu szybki powrót do dalszego parsowania pozostałej części pliku źródłowego
- stosunkowo nie duże zwolnienie parsowania poprawnych plików przez mechanizmy obsługi błędów

Spełnienie wszystkich wymienionych założeń obsługi błędów jest zadaniem nietrywialnym. Upraszcza je znacznie fakt, że statystyczna większość błędów występujących w programach jest prosta z natury i do ich wykrycia wystarczą stosunkowo proste strategie wykrywania błędów. W niektórych przypadkach jednak, błąd może wystąpić w pliku źródłowym o wiele wcześniej niż nastąpi jego wykrycie. W przypadkach takich zadanie rozpoznania natury błędu może być bardzo trudne a w skrajnych przypadkach oparte jedynie na pewnych przypuszczeniach co do zamysłu programisty tworzącego plik źródłowy.
Niektóre metody parsingu, takie jak LL czy LR, wykrywają błędy zaraz po ich wystąpieniu, dzięki mechanizmowi aktywnego przedrostka. Metody te potrafią wykryć przedrostki pojawiające się na wejściu, które nie są przedrostkami żadnego prawidłowego ciągu znaków danego języka.

Sposób zgłaszania błędu:

W najprostszym przypadku, kiedy wykryty zostanie błąd, kompilator powinien poinformować o miejscu wykrycia błędu w pliku źródłowym, ponieważ istnieje duże prawdopodobieństwo, że błąd wystąpił kilka tokenów przed miejscem wykrycia.

Powracanie do dalszego parsingu po wykryciu błędu:

Istnieje kilka strategii powrotu do parsingu, przy czym żadna z nich nie jest metodą dominującą czy mającą zastosowanie w każdym wypadu. W większości przypadków nie dopuszczalne jest, aby kompilator przerwał pracę po wykryciu pierwszego błędu, gdyż plik wejściowy może zawierać ich więcej.

Strategie powrotu do parsingu:

- powrót w trybie paniki
- poziom frazy
- produkcje uwzględniające błędy
- globalna korekta

Tryb paniki:
Jest to najprostsza strategia mająca zastosowanie w większości metod parsingu. Po wykryciu błędu parser pobiera z analizatora leksykalnego dalsze tokeny aż do pojawienia się jednego z określonych wcześniej tokenów synchronizujących. Od tego miejsca parser rozpoczyna dalszą analizę syntaktyczną. Tokenami synchronizującymi zwykle są tokeny ograniczające wyrażenia. Np. średnik czy słowo kluczowe end. Pomimo, iż powracanie do parsingu w trybie paniki związane jest z pomijaniem pewnej ilości tokenów, jego prosta konstrukcja zapobiega np. nieskończonym pętlom. Tryb ten stosowany jest szczególnie często w przypadkach, gdy wystąpienie wielu błędów w jednym wyrażeniu jest rzadkie.

Powrót na poziomie frazy:
W metodzie tej po wykryciu błędu, parser może dokonać lokalnych zmian w strumieniu wejściowym, tj. zamienić przedrostek pozostałego strumienia wejściowego na taki, który pozwala na przeprowadzenia dalszego parsingu. Typowym przykładem jest dodanie brakującego dwukropka, zamiana przecinka na dwukropek, usunięcie nadmiarowego dwukropka. Wadą metody jest trudność radzenie sobie z sytuacjami w których błąd wystąpił znacznie wcześniej niż jego wykrycie.

Produkcje uwzględniające błędy:
Jeśli znane są najczęstsze przypadki występowania błędów, wtedy gramatykę danego języka można rozszerzyć o takie produkcje, które produkują błędne konstrukcje i wykorzystać je przy konstrukcji parsera.

Korekta globalna:
Istnieją algorytmy, które na podstawie pewnego strumienia wejściowego x oraz gramatyki G potrafią znaleźć drzewo rozbioru syntaktycznego strumienia y takiego, że transformacja z x do y odbywa się najmniejszym kosztem w sensie ilości modyfikacji tokenów. Metody te w związku ze swoją dużą złożonością obliczeniową rozpatrywane są jedynie w kategoriach teoretycznych.

Implementacja strategii powrotu do parsingu w parserach bottom-up:

1. Parsing oparty na hierarchii operatorów.

Parsery tej klasy są prostymi parserami stosowanymi dla dość wąskiej, lecz istotnej klasy gramatyk operatorowych bez eta-produkcji.

Trzy sytuacje w czasie których występuje błąd składniowy:

1. nie istnieje relacja hierarchii między symbolem na wierzchołku stosu a następnym symbolem wejściowym - błąd pary symboli
2. relacje hierarchii wskazują, że należy dokonać redukcji, natomiast nie istnieje

nieterminal według którego redukcja mogłaby być dokonana - błąd redukcji
3. po wykonaniu redukcji, nie istnieje relacja hierarchii między aktualnie znajdującym sie na wierzchu stosu symbolem a kolejnym symbolem wejścia - błąd stackability

Obsługa błędów redukcji:

Jeśli w czasie parsingu zredukowany ma zostać ciąg symboli, który nie jest żadną z prawych produkcji gramatyki na której skonstruowany został parser, wtedy kompilator musi zdecydować do której z istniejących prawych produkcji najbardziej podobny jest ten ciąg. Dla przykładu, jeśli z analizy stosu wynika że zredukowany ma zostać ciąg abc i nie jest on redukowalny poprzez produkcje gramatyki, lecz redukowalny jest np ciąg aEcE wtedy kompilator może wypisać błąd o nieprawidłowym "b". Można także rozważać modyfikację lub dodanie terminala. Na przykład, jeśli istnieje produkcja abEdc, wtedy w powyższym przypadku wypisany może zostać komunikat o brakującym "d". Można wreszcie rozpatrywać prawe strony produkcji, które mają pożądaną, w danym kontekście redukcji, kombinację terminali lecz nieodpowiednią symboli nieterminalnych. W odniesieniu do poprzedniego przykładu jeśli ciąg abc ma być zredukowany bez żadnych symboli nieterminalnych między symbolami, a istnieje prawa strona produkcji postaci aEbc, wtedy kompilator może wypisać komunikat o brakującej kategorii syntaktycznej reprezentowanej przez terminal E.

Obsługa błędów shift/reduce

W sytuacji, gdy parser sprawdza, w oparciu o tablicę hierarchii, czy dokonać operacji shift czy reduce, i okaże się że nie ma określonej relacji hierarchii między symbolem znajdującym się na wierzchu stosu a kolejnym symbolem wejściowym, wtedy sygnalizowany jest błąd. Aby kontynuować parsing należy zmodyfikować stos, wejście lub jedno i drugie. Jeśli wprowadzamy bądź usuwamy symbol ze stosu bądź wejścia, wtedy należy uważać aby nie zapętlić parsera w nieskończoność. Jedną ze strategii zapobiegania takim sytuacjom jest zagwarantowanie, że po procedurze powracania do parsingu kolejny symbol wejścia może być przeniesiony na stos. Na przykład, jeśli na stosie znajduje się *ab* a na wejściu *cd*, oraz jeśli między *a* i *c* istnieje relacja < lub = hierarchii wtedy można usunąć symbol *b*, lub też, jeśli odpowiednie relacje istnieją między *b* i *d*, wtedy można zdecydować się na usunięcie *c*. Jeszcze jedną opcją jest znalezienie takiego symbolu *x*, który jest w relacjach *b <= x <= c*, wtedy możemy taki symbol wstawić na wejście przed *c*. Jeśli nie można znaleźć pojedynczego symbolu spełniającego podany warunek, wtedy możemy starać się znaleźć ciąg symboli *q r s .. x* takich że *b <= q <= r <= s <= ... <= x <= c*. Dokładny wybór ciągu zależy od intuicji projektanta kompilatora co do natury danego błędu. Dla każdego pustego pola w tablicy hierarchii należy określić procedurę powrotu do parsingu.

Przykład:

Oto fragment tablicy hierarchii dla prostej gramatyki wyrażeń arytmetycznych z uwzględnieniem procedur powrotu do parsingu po napotkanym błędzie

	Id	(	)	$
Id	C3	C3	>	>
(	<	<	=	C4
)	C3	C3	>	>
$	<	<	C2	C1

Oraz znaczenia odpowiednich procedur:

c1: brakuje całego wyrażenia
akcja: wstaw id na wejście
informacja: brakujący operand

c2: wyrażenie zaczyna się od zamykającego nawiasu
akcja: skasuj nawias z wejścia
informacja: nadmiarowy nawias zamykający

c3: po id lub) występuje id lub (
akcja: wstaw + na wejście
informacja: brakujący operatorowych

c4: wyrażenie kończy się nawiasem otwierającym
akcja: usuń (ze stosu
informacja: brak nawiasu zamykającego

Metoda error productions

Metoda *error productions* należy do grupy *ad hoc error recovery methods*. Metody tej grupy nie mogą być automatycznie wygenerowane na podstawie gramatyki. Wymagają one od twórcy gramatyki umiejętności przewidzenia ewentualnych błędów programisty.

Sama metoda *error productions* polega na tworzeniu specjalnych reguł, oraz powiązanych z nimi akcji, które informują o błędzie i podejmują odpowiednie czynności pozwalające powrócić do normalnego działanie parsera. W ten sposób błędne zdania stają się częścią generowanego przez gramatykę języka. Przykładowo w gramatyce Pascalu mamy:

```
if-statement -> IF boolean-expression THEN statement else-part
else-part -> ELSE statement | ε
```

Częstym błędem jest wstawienie średnika po wystąpieniu instrukcji ELSE (w Pascalu średnik jest separatorem a nie terminatorem i nie jest dopuszczalne przed instrukcją ELSE). Aby wykryć taką sytuację możemy zmodyfikować poprzednią gramatykę, wstawiając odpowiednią produkcję:

```
else-part -> ELSE statement | ε | ; ELSE statement
```

gdzie ostatnia produkcja z prawej jest produkcją wywołującą akcję obsługującą błąd.

Główną wadą tej metody jest fakt że tylko błędy przewidziane przez twórców gramatyki są poprawnie obsługiwane. Z drugiej strony dzięki tej metodzie można generować bardzo konkretne komunikaty dla użytkownika (precyzyjnie opisujące problem). Jest więc ona wykorzystywana jako dodatkowa, współpracująca z innymi, metoda obsługująca dość częste i oczywiste błędy.

Metoda *backward/forward move*

Jedna z metod rodziny *regional error handeling* to technika *backward/forward move*. Składa się ona z dwóch faz:

- faza pierwsza polegająca na ograniczeniu kontekstu wokół miejsca wystąpienia błędu w jak największym stopniu (*condensation phase*)
- draga faza to faza korekcji (*correction phase*). Polega ona na odpowiedniej modyfikacji stosu lub/i bufora wejściowego tak aby proces parsingu mógł być kontynuowany.

Metoda ta najczęściej stosowana jest w *precedence parsers*.

Dla przykładu posłużymy się właśnie taką gramatyką (z gramatyki zostały już usunięte konflikty stąd symbole E' i T'):

```
S -> # E′ #
E′-> E
E -> E + T′
E -> T′
T′-> T
T -> T × F
T -> F
F -> n
F -> ( E )
```

Odpowiednia tablica pierwszeństwa operatorów będzie wyglądać następująco:

	#	***E'***	***E***	***T'***	***T***	***F***	***n***	*+*	*×*	*(*	*)*
#		=·	<·	<·	<·	<·	<·			<·	
E'	=·										
E	·>							=·			=·
T'	·>							·>			·>
T	·>							·>	=·		·>
F	·>							·>	·>		·>
n	·>							·>	·>		·>
+				=·	<·	<·	<·			·>	
×						=·	<·			·>	
(			=·	<·	<·	<·	<·			·>	
)	·>							·>	·>		·>

Załóżmy że na wejściu mamy ciąg #n×+n#. Poniżej przedstawiony jest proces parsingu aż do miejsca wystąpienia błędu:

```
#<·n·>          n×+n#      shift n
#<·F·>          ×+n#       reduce n
```

```
#<·T=·×          +n#        reduce f, shift ×
```

Ponieważ w komórce zawierającej relacje pomiędzy ´ a + nie ma żadnego wpisu więc zostanie wygenerowany komunikat informujący, że + nie jest oczekiwany. Uruchomiony zostaje mechanizm error recovery:

1. *condensation phase*

   Do badania kontekstu wykonywane są dwa typy ruchów:

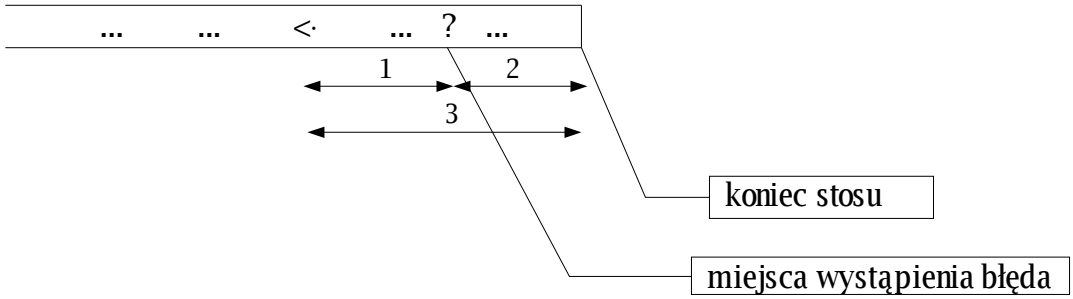
   - ruch wstecz (*backward move*) – zakładamy że pomiędzy plusem z gwiazdką występuje ·>. Wykonać należało by w takim razie wszystkie możliwe redukcje. W naszym przypadku żadna redukcja nie jest możliwa.

   - ruch do przodu (*forward move*) – zakładamy że symbolem pomiędzy plusem a gwiazdką jest <· lub =· . Przy takim założeniu możliwe jest kontynuowanie parsingu. Otrzymamy kolejno:

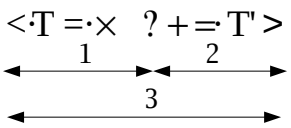
```
#<· T=· × =·/<· + <· n ·>#      shift +, shift n
#<· T=· × =·/<· + <· F ·>#      reduce n
#<· T=· × =·/<· + <· T ·>#      reduce F
#<· T=· × =·/<· + =· T' ·>#     reduce T
```

2. *correction phase*

   Otrzymany z poprzedniej fazy wynik jest tutaj następująco interpretowany:



   Dopuszczane są trzy obszary wymagające korekty odpowiednio ponumerowane cyframi 1, 2 i 3. Obszar 1 jest dopuszczalny ponieważ brakującym symbolem mógł być ·>. Obszar 2 odpowiada sytuacji gdy symbolem tym był <· a dla obszaru 3 zakładamy że brakujący symbol to =·.
   Mamy więc:



   Obszar numer jeden możemy zastąpić poprzez E, T', T i F (ponieważ pomiędzy tymi nieterminalami a plusem występuje ·>). Analogicznie dla obszaru 2 mamy F a dla obszaru 3 są to E, T', T i F. Powstało dość dużo możliwości. Pareser wybiera tylko jedną z nich na podstawie obliczonych kryteriów. Mianowicie, z każdym możliwym symbolem na stosie powiązany jest koszt wstawienie (I) oraz koszt usuniecie

danego symbolu (D). Koszty te są ustalane przez autora gramatyki. Przykładowo kosz zamiany T+ na F to D(T) + D(+) + I(F). Wybierany jest najtańszy wariant. Po podliczeniu kosztów dla naszego przykładu otrzymamy - jako najlepsze rozwiązanie - operację usuniecie symbolu´ z pierwszej części. Parser wykona kolejno:

```
#<· T =· × ? + =· T' ·>#    error situation
#<· T ·> + =· T' ·>#        error corrected
#<· T ·> + =· T' ·>#        reduce T
#<· E =· + =· T' ·>#        reduce T'
#<· E  ·>#                  reduce E+T'
#<· E' ·>#                  reduce E
#<· S  ·>#                  reduce E'
```