Journal of Korea Multimedia Society Vol. 11, No. 5, May 2008(pp. 703-709)

리덕션 골을 이용한 LR 파서의 개선

손윤식†, 오세만††

요 약

컴파일러의 구성 방법론은 파싱 기법의 정립과 자동화 도구의 개발을 통해 많은 발전을 이루었으며, 이를 통해 다양한 컴파일러를 효과적으로 제작할 수 있는 환경이 마련되었다. 특히, 최근에는 임베디드/모바일 기기의 사용과 콘텐츠 산업이 활성화되고 있으며, 이에 따라 각 시스템과 콘텐츠에 적합한 컴파일러 개발 요구가 늘어나고 있다. 컴파일러의 모듈화와 자동적인 구성을 통해 이러한 수적인 요구는 해결되고 있지만, 개발도구로서의 컴파일러를 최적화하기 위해서는 경험적인 방법론의 사용과 이에 따른 매우 큰 비용이 필요하다. 본 논문에서는 LR 파서의 특징을 분석하여, 불필요한 reduce 행동을 경감할 수 있는 파싱 기법을 제시한다. 개선된 파싱 기법은 파싱과정에서 lookahead/상태 정보와 도달 가능한 리덕션 골의 정보를 이용하여 연속적인 reduce를 하나의 reduce로 변환하여 효율성을 높인다. 또한, 임베디드 ANSI C컴파일러의 전단부에 적용하여 실제 모바일 콘텐츠 대한 파싱 성능을 분석하였다.

Improvement of LR Parser using Reduction Goals

Yun-Sik Son†, Se-Man Oh††

ABSTRACT

The methodology of the compiler construction improved by well-defined parsing techniques and developments of automatic generation tools. Through them, a variety of compilers for the special applications can be developed effectively: particularly, the compiler for embedded/mobile devices. Also, as contents industry is proliferating recently, the necessity of developing a compiler which is suitable for contents system is highly increasing. These various demands can be resolved by modular techniques and automatic construction of compilers. But, optimization of compiler itself as development tools uses heuristic methods and it needs higher cost. In this paper, we suggest the parsing method which can decrease unnecessary reduce actions by analyzing the characteristics of LR parser. The suggested parsing technique uses lookahead/states, reachable reduction goals information in parsing process and enhances the parsing efficiency by changing continuous reduce actions to one. Actually, we applied it to the front-end of ANSI C compiler and proved the parsing performance in terms of the number of reduce actions.

Key words: LR Parsing(LR 파싱), C_0(Canonical 콜랙션), Reduction Goals(리덕션 골)

1. 서 론

최근 임베디드 시스템과 모바일 장치를 활용한 콘텐츠 산업은 그 규모가 커지고 있으며, 이에 따라 효과적인 콘텐츠 개발도구의 요구도 늘어나고 있는 추세이다. 특히 모바일 기기에서는 가상기계와 같은 기법을 이용하여 콘텐츠를 위한 플랫폼과 개발도구를 제공하는 것이 일반적이다.

※ 교신저자(Corresponding Author) : 오세만, 주소 : 서울시 중구 필동 3가 26번지(100-715), 전화 : 02)2260-3342, FAX : 02)2265-8742, E-mail : smoh@dongguk.edu
접수일 : 2008년 2월 29일, 완료일 : 2008년 3월 24일
† 준회원, 동국대학교 컴퓨터공학과 박사과정 (E-mail : sonbug@dongguk.edu)
†† 정회원, 동국대학교 컴퓨터공학과 교수
※ 이 논문은 2006년도 동국대학교 연구년 지원에 의하여 이루어졌음.

이 중 콘텐츠 개발에 있어 핵심이 되는 컴파일러는 모듈화된 구성 방법론과 자동화 도구를 기반으로 하여 특정 플랫폼에 적합하게 구성하는 것이 매우 용이해져 있으며, 다양한 컴파일러가 필요에 의해 만들어지고 있다. 반면, 컴파일러 자체에 대한 최적화는 여전히 경험적인 방법론을 사용할 수밖에 없으며, 이에 따르는 비용 또한 매우 크다. 더 나아가 특정 개발 환경을 최적화하기 위한 컴파일러는 언어의 사용에 있어서 제약을 가지는 경우가 많기 때문에 프로그래머에게 부담을 주기도 한다.

본 논문에서는 LR 파싱 과정에서 리덕션 골의 선택을 이용하여 reduce 행동에 소요되는 비용을 크게 줄일 수 있다는 점에 착안하여, 컴파일러에서 효과적으로 프로그램을 파싱할 수 있는 기법을 제안하고 임베디드 ANSI C컴파일러[1]에 적용하여 그 결과를 확인 한다.

개선된 파싱 기법은 reduce 과정의 축약을 이용하여 파서의 성능을 높였으며, 이 과정에서 중요하게 고려된 부분은 다음과 같다. 1. 프로그램 작성 시 효율적인 분석을 위한 제약이 없어야 함. 2. 기존 파싱 알고리즘의 사용이 용이 할 것. 3. 기존의 방법과 개선된 방법을 통해 파싱된 프로그램의 의미 동등. 4. 개선에 소요되는 비용 대비 큰 성능 향상 등이다.

본 논문은 다음과 같은 순서로 개선된 파싱 기법에 대해 논한다. 우선 2장에서 LR 파서에 대해 알아보며, LR 파서의 reduce 행동이 가지는 의미를 이용한 연구를 살펴본다. 이어 3장에서는 LR 파서 중 가장 많이 사용되는 LALR 파서의 특징과 연구의 기본 아이디어를 소개한다. 4장에서는 앞서의 연구를 바탕으로 개선된 파서 모델과 리덕션 골을 선택하기 위한 방법론을 제시한다. 5장에서는 실제 컴파일러에 개선된 파서를 적용하여 그 성능을 확인하며, 6장에서는 연구의 결과와 앞으로의 연구 방향에 대해 제시한다.

2. 관련연구

2.1 LR 파서

Knuth에 제시된 LR(1) 파싱 기법은 대부분의 프로그래밍 언어를 파싱할 수 있는 강력한 파싱 기법으로 특히 구문 에러를 빨리 찾아내는 장점을 가지고 있다[2,3]. 반면, 파싱 테이블의 크기가 매우 커지기 때문에 실제 컴파일러에 적용하기에 어려웠지만, 효과적인 파서 구성을 위한 다양한 기법의 제안으로 큰 발전을 이루었다. 특히, DeRemer에 의해 제안된 LR(k), SLR(k), LALR(k) 기법은 효과적으로 구문 구조를 분석함과 동시에 파싱 테이블의 크기를 줄일 수 있어, 컴파일러에 효과적으로 적용이 가능한 방법론이다[4,5].

LR 파서는 결정적인 bottom-up 방법을 특징으로 가지며, 크게 파서 구동 루틴, 파싱 테이블, 파싱 스택, 입력 버퍼의 4가지로 구성되며, 파싱 스택의 최상위 값과 현재 입력 심벌에 따라 shift, reduce, accept, error의 4가지 구문 분석 행동을 결정한다[6,7]. 그림 1은 LR 파서의 알고리즘을 간략하게 나타낸 것이다.

그림 2는 파서의 행동을 정형적으로 표기한 내용이다. 여기서 ACTION 테이블과 GOTO 테이블은 각각 파서의 4가지 행동과 reduce 행동 시 다음 상태를 지시한다.

```
Algorithm Parser :
    initialize state
    get next token
    do until accept or error
        set current state
        check action of current state
            case shift : set stack state & get next token
            case reduce :
                semantic action
                stack & symbol adjustment
                set stack state
            case accept : return result
            case error : error recovery
```

그림 1. 파싱 테이블을 사용하는 LR 파서의 기본 알고리즘

State : S_i
Grammatical symbol : X_i
Input : $a_1a_2 \cdots a_n\$$
Stack : $S_0X_1S_1 \cdots X_mS_m$
Configuration of LR parser : $(S_0X_1S_1 \cdots X_mS_m, a_ia_{i+1} \cdots a_n\$)$

1. shift: $ACTION[S_m,a_i] = shift\ S$
   $(S_0X_1S_1 \cdots X_mS_m, a_ia_{i+1} \cdots a_n\$) \Rightarrow (S_0X_1S_1 \cdots X_mS_ma_iS, a_{i+1} \cdots a_n\$)$
2. reduce: $ACTION[S_m,a_i] = reduce\ A \rightarrow \alpha, |\alpha| = r$
   $(S_0X_1S_1 \cdots X_mS_m, a_ia_{i+1} \cdots a_n\$)$
   $\Rightarrow (S_0X_1S_1 \cdots X_{m-r}S_{m-r}, a_ia_{i+1} \cdots a_n\$), GOTO(S_{m-r},A) = S$
   $\Rightarrow (S_0X_1S_1 \cdots X_{m-r}S_{m-r}AS, a_ia_{i+1} \cdots a_n\$)$
3. accept: $ACTION[S_m,a_i] = accept$
4. error: $ACTION[S_m,a_i] = error$

그림 2. 파서의 4가지 행동에 대한 정의

2.2 리덕션 골을 이용한 응용 연구

LR 파서는 reduce 과정에서 사용되는 리덕션 골을 파싱 중 리덕션 시간에 결정되도록 설계되어 있다. 그러나 LR 문법의 설계 상 미리 알 수 있는 리덕션 골이 존재하며 이를 통해 파싱 과정에서 리덕션 시점 전에 해당하는 리덕션 골을 예상할 수 있다.

Hammer에 의해 연구된 이 기법[8]은 제약된 범위 내의 LR 문법을 LL 문법으로 변환하기 위한 근본 아이디어지만 LR 파서에 직접적으로 적용하지는 않았다.

최근에는 결정적인 리덕션 골의 예상 방법을 파싱에서 발생하는 에러 복구에 적용하는 연구가 진행되고 있다[9,10]. 이 방법은 리덕션 골의 결정에 대해 정형적인 방법으로 접근하고 결정론적 접근 방법 도입하여 자동적으로 파서를 생성하는 알고리즘을 제시하고 있는 반면, 실제 파서에 도입하여 에러 복구 방법론으로 사용하기에는 제약점이 많다.

3. 기존 파서 분석

3.1 LALR 파서와 문법

LALR(LookAhead LR)은 CLR과 SLR 방법에서 발전한 기법으로, lookahead 정보를 이용하기 때문에 SLR 방법보다 많은 언어를 분석할 수 있으며 CLR에서 core가 같은 아이템들을 한 데 묶음으로써 파싱테이블의 크기를 SLR과 동일하게 구성할 수 있다. 또한 모호하지 않은 context-free 문법으로 표현된 거의 모든 언어를 인식할 수 있기 때문에 최근 대부분의 파서 제작 시스템은 LALR 방법을 사용한다[7].

일반적인 LR 파서와 동일하게 LALR 파서 역시 파싱 테이블과 구동 루틴이 핵심이며, 파싱 테이블 lookahead와 상태정보를 이용하여 파서의 행동을 기술하고 있다. 이때 reduce를 위한 lookahead의 분석이 파싱 테이블 구성의 핵심이다.

LALR 파서는 lookahead 정보를 이용하여 shift와 reduce를 반복하면서 프로그램을 분석한다. 이때 발생하는 shift 행동의 수는 프로그램에 사용된 토큰 수에 의존적이며, reduce 횟수는 문법의 기술 형태에 따라 달라진다. 따라서, shift/reduce 행동이 주인 LALR 파서의 성능은 문법 기술이 얼마나 효과적인지에 따라 좌우된다. 간결하게 작성된 문법은 파서의 성능에 향상에 중요한 요소지만, 실제로 사용되는 문법은 충돌 문제 해결이나 우선순위 문제, 작성의 용이성 등에 의해 많은 요소가 추가되어 그 크기가 커지게 된다.

3.2 reduce 정보 분석

PGS를 통해 생성되는 파서의 성능을 개선하기 위해서는 입력 문법을 수정하는 것이 핵심이지만, 앞서 언급한 바와 같이 의도적으로 추가된 문법 요소를 사용하는 경우에 파서는 불필요한 작업을 수행한다. 일반적으로 연산자에 대한 우선순위는 문법의 추가로 해결하는 방식을 취하는데, 임베디드 시스템용 ANSI C 컴파일러 역시 이와 같은 문법을 사용하여 얻어진 파싱 테이블을 사용하고 있다[11,12]. 다음 표 1은 완전수를 계산하는 알고리즘을 구현하여 ANSI C 컴파일러를 이용하여 파싱한 후, 그 정보를 분석한 표이다.

reduce 횟수 정보는 shift 행동이 없는 reduce 행동에 대해 (reduce 수, AST 생성 수)를 묶어서 표현한 것이며, 통계 부분에서는 전체 프로그램에서 발생한 shift 횟수 / reduce 횟수 / 만들어진 AST 노드의 수, 연속으로 발생한 reduce의 평균 횟수, reduce 당 발생한 AST 생성 횟수, shift와 AST의 생성 없이 reduce만 연속으로 발생한 경우와 그 평균 길이를 나타내고 있다.

실제 컴파일러에서, 파싱 이후의 작업에 사용되는 의미를 가지는 자료구조는 AST이며, 표 1에서 본 것과 같이 AST의 생성 수에 비해 reduce 행동의 수

표 1. 완전수 알고리즘의 파싱 정보

REDUCE COUNT INFORMATION
(4,2) (1,1) (2,1) (4,2) (4,2) (4,2) (5,2) (3,2) (3,0) (21,2) (8,0) (20,2) (2,0) (18,2) (1,1) (3,0) (20,1) (3,1) (3,0) (21,2) (8,0) (5,0) (20,3) (2,0) (18,2) (1,1) (5,0) (9,1) (18,1) (1,1) (3,0) (19,1) (4,2) (6,3) (5,2) (9,0) (17,1) (2,1) (6,3) (6,3) (5,3) (0,0)

STATISTICS
total shift count : 72 total reduce count : 319 total make Tree count : 53 average of continuous reduce count : 7.780488 make Tree count per reduce : 0.166144 only reduce count & average length : (12, 4.250000)

표 2. 다양한 알고리즘에 대한 파싱 정보

count \ program	Perfect number	Josephus problem	Spanning tree	hash	matrix	DES
total shift	1179	2352	3131	2803	3043	4884
total reduce	3798	7690	11859	10659	11602	30058
total make Tree	1563	3097	3643	3352	3610	2646
avg. continuous Reduce	4.01	4.07	4.93	4.9	4.99	9.87
make tree per reduce	0.41	0.4	0.3	0.31	0.31	0.08
only reduce & avg. length	(17, 4.35)	(60, 2.93)	(223, 6.47)	(142, 6.01)	(186, 6.97)	(1312, 15.45)

가 크며, AST의 생성 없이 reduce만을 반복한 경우도 상당수가 있다. 이러한 불필요한 reduce의 수를 줄이게 되면 파싱과정에서 reduce에 필요한 여러 작업을 줄일 수 있기 때문에 보다 효율적인 분석과정을 거칠 수 있다.

표 2는 다양한 알고리즘을 구현하고 알고리즘에서 사용된 여러 라이브러리를 위한 헤더파일을 포함 결과이다. 여러 알고리즘에서 연속적인 reduce가 발생하는 것을 확인할 수 있으며, 특히 다수의 배열을 이용한 작업이 필요한 DES 알고리즘의 경우 연속적인 reduce의 수뿐만 아니라 AST 생성 없이 reduce만 발생한 경우도 크게 증가한 것을 확인할 수 있다. 이는 배열 및 상수 처리를 위한 문법의 기술에 여러 가지 비효율적인 부분이 있다는 것을 나타내는 결과이다.

반면, 이러한 비효율을 해결하기 위해 문법을 변경하게 되면 프로그램 구현에 많은 제약이 발생한다. 따라서, 파싱 과정에서 의미 분석에 영향을 주지 않고, reduce 행동의 수를 경감시키는 것은 파서의 성능향상에 도움이 된다.

4. 개선된 파서

reduce 정보를 이용한 파싱 액션의 경감을 위해서는 리덕션 골의 정보를 미리 알아야한다. 이러한 정보는 문법과 이를 이용한 C_0 정보의 분석을 통해 이루어지는데, 실제 프로그래밍 언어를 위한 문법을 대상으로 분석하기에는 많은 제약이 따른다. 본 논문에서는 배열과 상수 처리를 위한 부분에 한정하여 정보를 분석하고 리덕션 골을 선택한다.

4.1 개선된 파서 알고리즘

개선된 파서 알고리즘은 기존의 컴파일러에 쉽게 적용하는 것을 목표로 하고 있기 때문에 파서의 구동 루틴 변경을 최소화하고 shift/accept/error 행동에 영향을 주지 않게 설계하였다.

그림 3에 언급된 개선된 파서 알고리즘은 reduce 행동에 한하여 필요한 상태 정보를 확인하는 부분과 여러 리덕션 골의 후보에서 lookahead 정보를 이용하여 올바른 리덕션 골을 선택하는 것을 보이고 있다. 이러한 리덕션 골 적용을 위해 사용되는 정보는 파싱 시간 이전에 모두 미리 계산되고, reduce 시에만 사용되기 때문에 추가적인 분석 비용이 실행시간에 미치는 영향은 매우 적다. 그리고, reduce 과정에서 적용되는 리덕션 골로 인한 프로그램의 의미 변질을 막기 위해 리덕션 골은 AST를 만들지 않고 연속된 reduce만 행하는 경우에만 적용한다. 이러한 경우는 문법의 우선순위와 기술 편의성 때문에 생기는 단일 생성 규칙에서 쉽게 찾아볼 수 있다.

4.2 리덕션 골 후보 및 선택

리덕션 골의 선택은 C_0에서 reduce 아이템만을 대상으로 한다. reduce 아이템은 이미 lookahead 정보 통해 그 상태가 결정되었기 때문에 현재 파싱 상태의

```
Algorithm Enhanced Parser :
    initialize state
    get next token
    do until accept or error
        set current state
        check action of current state
            case shift : set stack state & get next token
            case reduce :
                check current state for array
                    find candidate reduction goal
                    check lookahead
                    select reduction goal
                semantic action
                stack & symbol adjustment
                set stack state
            case accept : return result
            case error : error recovery
```

그림 3. 개선된 파서 알고리즘

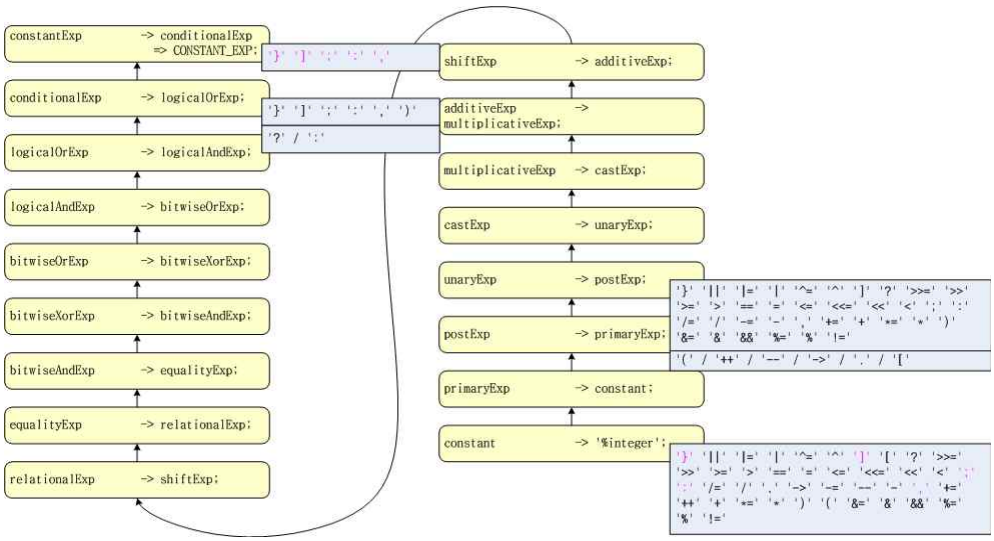

그림 4. 상수의 reduce 과정

파악과 파싱 과정에서 리덕션 골의 적용이 쉽게 이루어진다.

먼저 C_0 분석을 통해 연속적인 reduce의 발생 가능 여부를 파악하고 필요한 lookahead 정보를 수집한다. 이때 해당 문법 심벌의 교차 참조를 확인한다. 교차 참조가 있을 경우 연속적인 reduce시 리덕션 골이 다수 발생하기 때문에 추가 정보가 필요하다. 반면, 교차 참조가 존재하지 않는 경우 하나의 리덕션 골만이 있기 때문에 추가 정보가 필요하지 않다.

교차 참조가 존재할 경우, 리덕션 골의 선택은 lookahead를 통해 이루어진다. lookahead에 의해 리덕션 골이 선택 가능할 경우, 해당 lookahead에 다시 연속적인 reduce가 가능한지 분석하고, 가장 긴 reduce 축약을 행할 수 있는 리덕션 골을 선택한다. lookahead에 의한 구분이 불가능할 경우 리덕션 골은 선택되지 않으며, 이 경우 일반적인 reduce 행동을 취한다.

그림 4는 ANSI C 컴파일러에서 사용된 문법 중 상수가 어떤 식으로 reduce되는지 보여주고 있다. 간단한 상수를 분석하기 위해 총 17번의 reduce가 발생하는데, lookahead에 따라 reduce되는 문법 심벌이 다르다. 특히 상수는 연속적인 reduce 과정에서 lookahead 정보가 중복되는 경우가 없기 때문에 해당 lookahead 정보에 근거해 리덕션 골을 선택할 수 있다. 그림 5는 이와 같은 리덕션 골의 선택을 통해 배열 선언 시 reduce 과정이 축약되는 것을 보여주고 있다. 이 경우 커널 97에서 커널 79까지 축약되며, 커널 72는 reduce 아이템이 아니므로 더 이상 축약되지 않는다.

교차 참조가 발생하는 경우 리덕션 골에 의해 축약되는 부분은 제한되기도 하는데, 이때 이전 상태 정보를 이용하면 효과적으로 축약이 가능하다. 그림 5에서 보인 상수의 constantExp로 리듀스되는 과정은 커널 79에만 국한되는 것이 아니기 때문에, 이전 상태 정보까지 고려하면 보다 다양한 경우에 리덕션

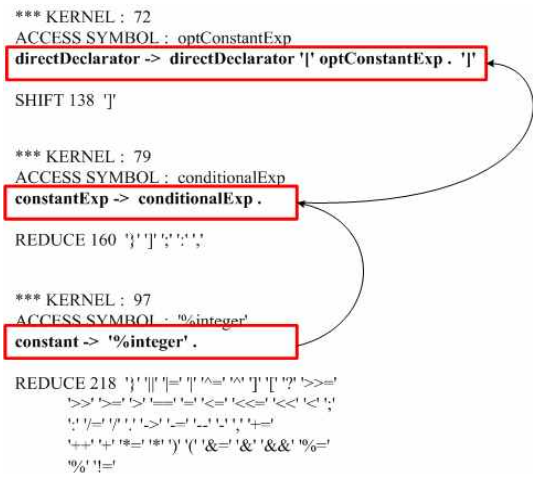

그림 5. 배열 선언에서 상수의 사용

표 3. 실험 대상 프로그램의 토큰 수와 파일 크기

	parsing table (c)	parsing table (c++)	DES	image data	mobile game (gaza)	mobile game (aiolos)
Tokens	131,033	699,154	3,785	226,195	3,029,647	466,320
File size(KB)	396	2,089	13.6	651	7,844	1,605

골을 적용할 수 있다.

5. 구현 및 성능평가

개선된 파서의 구현은 임베디드 시스템을 위한 ANSI C 컴파일러의 파서 모듈을 이용하였다. ANSI C 컴파일러의 파서는 LALR 파서이며, 문법의 생성규칙 수는 224, 평균 생성규칙 길이 3이며, PGS를 통해 생성된 파싱테이블 정보에서 커널의 수는 371이며, 터미널과 논터미널의 수는 각각 87개와 81개이다.

본 논문에서는 이러한 상태 정보 중 배열과 상수처리를 위한 정보만을 대상으로 리덕션 골과 look-ahead 정보를 구성하였다.

표 3은 실험 대상으로 다수의 배열이 존재하는 코드와 실제 모바일 게임 콘텐츠 상의 토큰 수와 파일 크기를 나타낸 것이다. 토큰은 프로그램에 대한 어휘분석의 결과로 토큰 수는 파서가 처리해야할 단위수와 동일하다.

그림 6은 구현된 파서에서 사용한 상태와 look-ahead 정보의 일부이다. 상태정보는 식과 배열 초기화에 사용되는 상수의 효과적인 처리를 위한 분석결과이다.

그림 6에서 사용되는 문법 규칙은 교차 참조가 존재하기 때문에 lookahead와 이전 상태 정보에 따라

```
Reduce rule : 217, 218, 219, 220, 221, 222, 223
  Previous State : Kernel 207, Kernel 336
    Lookahead : ','
    Reduction goal : 156
  Previous State : all kernel
    Lookahead : '}' '||' '|=' '|' '^=' '^' ']' '[' '?' '>>=' '>>'
                '>=' '>' '==' '=' '<=' '<<=' '<<' '<' ';' ':'
                '/=' '/' '-=' '-' ',' '+=' '++' '+' '*=' '*' ')'
                '&=' '&' '&&' '%=' '%' '!=' '*/'
    Reduction goal : 191
  Previous State : all kernel
    Lookahead : '.' '->' '--' '++' '(' '{'
    Reduction goal : 202
```

그림 6. 배열 원소의 상태 정보에 따른 lookahead와 리덕션 골

선택해야하는 리덕션 골이 다른 것을 확인 할 수 있다. 특히 이전 상태 정보를 고려하지 않는 경우, lookahead 정보로 구분되지 않으므로 리덕션 골 156은 리덕션 골 191에 포함되게 된다. 이전 상태 정보를 고려할 경우, 파서를 구성할 때 리덕션 골 선택에 따르는 정보 분석이 보다 복잡해지는 단점이 있지만, 성능을 보다 높일 수 있다.

실제 실험 결과 그림 7과 같이 이전 상태를 고려하지 않은 경우, 기존 파서의 성능에 비해 84%정도의 파싱 비용이 발생하였으며, 이전 상태 정보를 고려한 경우 다수의 배열 정보가 있는 프로그램의 경우 27%로 파싱 비용이 감소하였다. 또한 DES 알고리즘과 같이 다양한 수식 및 배열식이 혼합된 프로그램에서는 48%의 비용만 발생하였으며, 멀티미디어 콘텐츠용 이미지 자료와 게임 콘텐츠는 21~26%의 비용이 발생한

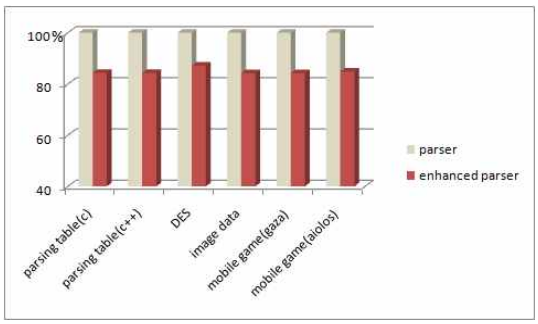

그림 7. 기존 파서와의 reduce 횟수 비교

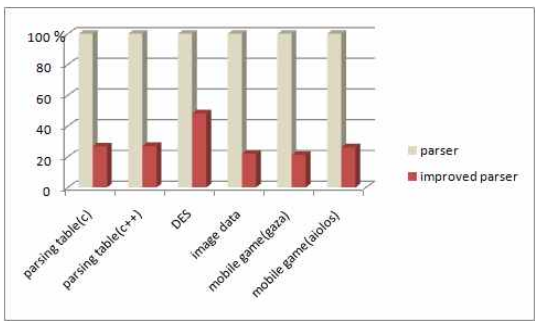

그림 8. 이전 상태정보가 고려된 reduce 횟수 비교

것을 확인하였다. 이러한 결과는 제시된 방법론과 구현된 개선 파서의 타당성을 보여준다고 판단된다.

6. 결론 및 향후 연구

본 논문에서는 reduce 행동 분석을 통해 파서를 개선할 수 있는 방법과 조건을 제시하였다. 또한 개선된 파서를 ANSI C 컴파일러에 구현하여 대상 프로그램을 효과적으로 분석할 수 있는 것을 보였다. 제시된 방법론은 기존의 파서 알고리즘을 기반으로 하고, 문법과 파싱테이블의 수정이 필요 없기 때문에 쉽게 기존의 컴파일러에 적용이 가능하며, 프로그램의 의미 분석에도 영향을 주지 않는다. 특히 기존의 프로그래밍 방법에 제약 없이 배열과 같은 자료처리에 효과적인 분석이 가능한 특징을 가진다.

향후 과제로는 배열과 같은 특수한 경우에서 나아가 전체 프로그램에 대해 적용하기위한 연구가 요구된다. 또한 리덕션 골의 선택을 위한 정보 분석은 전체 C_0 상태를 분석해야하기 때문에 복잡한 계산을 요구한다. 따라서 문법이 커질수록 그에 따른 복잡도가 증가하기 때문에 이를 쉽게 사용하기 위해 리덕션 골 선택 정보의 자동적인 생성에 대한 연구가 필요하다.

참 고 문 헌

[1] 이양선, 오세만, 김영근, 권혁주, 손윤식, 박성환, 임베디드 시스템을 위한 ANSI C 컴파일러 개발, 서경대학교 산학협력단, 2004.

[2] Aho, A.V. and Johnson, S.C., "LR Parsing," *ACM Computing Surveys*, Vol.6, No.2, pp. 99-124, 1974.

[3] Knuth D.E., "On the Translation of Language from Left to Right," *Information and Control*, Vol.8, No.6, pp. 607-639, 1965.

[4] DeRemer F.L., *Practical Translator for LR(k) Languages*, ph. D. Thesis MIT, 1969.

[5] DeRemer F.L., "Simple LR(k) Grammars," *Communications of ACM*, Vol.14, pp. 453-460, 1971.

[6] Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools, 2nd edition*, Addison Wesley, 2006.

[7] 오세만, 컴파일러 입문, 개정판, 정익사, 2004.

[8] Hammer, M., *A New Grammatical Transformation into deterministic top down form*, Project MAC Technical Report TR-119, MIT, 1973.

[9] 이경옥, 최광무, "미리 결정된 리덕션 골들을 가진 LR 파서," 정보과학회논문지, 제26권 제7호, pp. 931-937, 1999.

[10] 이경옥, "리덕션 골의 예상 : 결정적인 접근 방법," 정보과학회논문지, 제30권, 제5·6호, pp. 461-465, 2003.

[11] 김영근, 권혁주, 이양선, "ANSI C 컴파일러에서 중간코드의 검증과 분석을 위한 역컴파일러의 개발," 한국멀티미디어학회논문지, 제10권, 제3호, pp. 411-419, 2007.

[12] 손윤식, 시멘틱 트리를 이용한 2단계 코드 생성, 동국대학교 석사학위 논문, 2006.

손 윤 식

2004년 동국대학교 컴퓨터공학과 공학사
2006년 동국대학교 컴퓨터공학과 공학석사
2006년～현재 동국대학교 컴퓨터공학과 박사과정.

관심분야 : 프로그래밍 언어, 컴파일러, 모바일/임베디드 컴퓨팅

오 세 만

1977년 서울대 수학교육학과 졸업
1979년 한국과학기술원 전산학과 석사 졸업.
1985년 한국과학기술원 전산학과 박사 졸업.
1985년～현재 동국대학교 컴퓨터공학과 교수.
1993년 3월～1999년 2월 동국대학교 컴퓨터공학과 대학원 학과장.
2001년 11월～2003년 11월 한국정보과학회 프로그래밍언어연구회 위원장.
2004년 6월～2005년 11월 한국정보처리학회 게임연구회 위원장.

관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅