컴파일러: 7장

국민대학교 소프트웨어학부
강 승 식

제7장 컴파일러 구현 방법

- Lex를 이용한 어휘 분석
  - 정규표현식 인식 → action code 작성(C 언어 문장)
  - 스트링 패턴(정규표현식) 인식에 사용되는 범용 도구
  - 주 활용 분야: 컴파일러 개발, 에디터에서 스트링 탐색 등

- Lex를 이용한 컴파일러의 어휘 분석기 구현
  - <token number, token value>를 파서(yacc) 입력으로 전달
  - 토큰값 전달 필요가 없으면 token number만 전달

    정규식_i　{ return token_number_i; }

  - 토큰값 전달 방법: 토큰값을 yylval에 저장한 후에 token number를 전달
  - 변수명, 함수명 등 명칭(identifier)과 정수/실수/스트링 상수 등

    정규식_j　{ set_token_value_j; return token_number_j; }

Token number, token value

- 토큰값 이외의 다른 값을 파서에게 전달할 필요가 있는 경우
  - 전달하고자 하는 데이터를 저장할 변수를 전역 변수로 선언하고, 파서에서 외부 변수(external variable)로 전달
- Lex는 규칙부에 기술된 각 정규표현식들을 인식하는 함수 yylex()를 생성
  - 각 정규표현식의 액션 코드에 기술된 return값은 함수 yylex()의 return값
  - yylex()의 return값은 정수형(integer)이므로 인식된 토큰에 대한 token number는 정수형으로 정의
  - 다만, '+', '-' 등과 같이 한 문자로 구성된 연산 기호는 아스키 코드값이 정수이므로 아스키 코드를 직접 return 가능
- Lex에서 파서로 전달하는 token number(정규표현식의 return값)
  - Yacc의 정의부에서 토큰으로 정의된 것
  - 규칙부에서 직접 터미널 문자로 사용된 아스키 문자
- Yacc에서 사용되는 토큰 유형(터미널 기호)
  - %token, %left, %right 으로 정의된 토큰
  - 생성규칙에서 직접 사용된 아스키 문자

Yacc를 이용한 구문 분석

- Yacc 입력 파일
  - Lex의 입력 파일과 유사
    - 차이점: 규칙부에서 정규식 대신에 생성규칙 기술
  - 규칙부의 생성규칙이 적용되면 실행할 액션 코드를 C 언어 문장으로 기술 – Lex와 유사함

- Yacc의 용도
  - CFG 문법으로 기술되는 구문 구조(syntactic structure)를 인식하는 시스템을 개발할 때 매우 유용하게 사용
  - 계산기(calculator)와 같이 수식 연산을 하는 경우에 목적 코드를 생성하지 않고 바로 실행하는 인터프리터 형태로 구현 가능

```
exp: exp '+' exp  { $$ = $1 + $3; }
           | NUMBER  { $$ = $1; }
```

- BASIC, LISP, PROLOG 등 고급 언어 인터프리터를 구현할 때
  - 조건문과 반복문 등 구조화된 문장이 여러 개의 생성규칙으로 기술되므로 구조화된 문장을 처리하는 방법이 고려되어야 함

- 파싱 결과를 목적 코드로 생성하는 가장 단순한 방법은 각 생성규칙마다 이에 해당하는 목적 코드를 출력하는 것이다.

- 예) 덧셈 연산에 대한 스택 기계용 목적 코드를 출력

```
exp:    exp '+' exp   { printf("add₩n"); }
        | NUMBER    { printf("push %d₩n", yylval); }
```

- 입력 문장의 100 + 200 + 300 에 대한 우단 유도 과정

```
exp => exp + exp              (exp -> exp + exp)
    => exp + 300              (exp -> 300)
    => exp + exp + 300        (exp -> exp + exp)
    => exp + 200 + 300        (exp -> 200)
    => 100 + 200 + 300        (exp -> 100)
```

- Yacc는 우단 역유도 방식
  - right parse에 의해 reduce할 때 목적 코드 생성

```
push 100
push 200
add
push 300
add
```

- 일반적으로 파싱 결과는 "파스 트리"로 구성

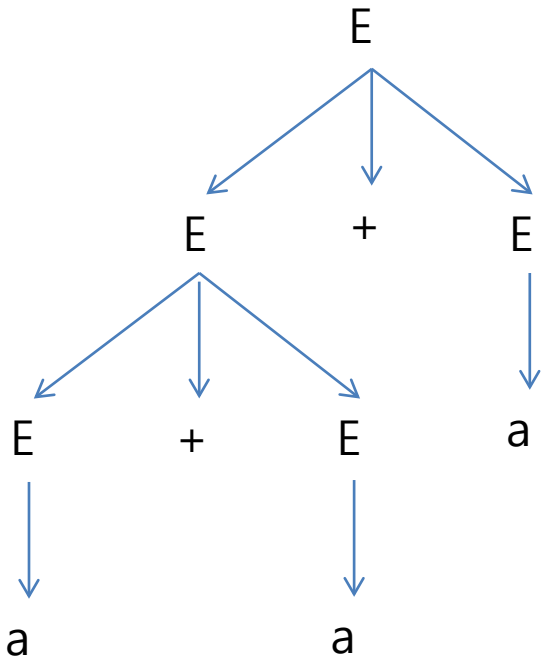
파스 트리 생성 방법

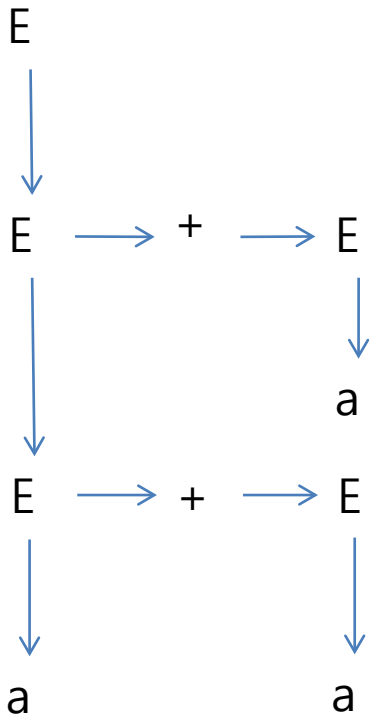
- 입력 문장에 대한 파서의 구문 분석 결과는 파스 트리 (parse tree)로 생성

- 파스 트리 생성을 위한 자료 구조
  - 트리 구조(tree structure), linked list 타입
  - 일반적인 범용 트리 구조를 구현하는 방법
    - 이진 트리 형태로 저장

```
struct NODE {
        NODE *down;          // first child에 대한 포인터
        NODE *right;         // 2nd - last child 연결
        int data;
};
```

이진 트리 구현 예: a+a+a





- 파싱 결과를 파스 트리로 구성할 때 Yacc의 규칙부에서 액션 코드 부분
  - 예) 새로운 노드 생성 및 생성규칙에 따라 sub-tree 구성

```
exp:    exp '+' exp {
              $$.node = get_newnode();
              $1.node를 $$.node의 first child로 연결
              tnode = build_node('+');
              tnode를 $$.node의 second child로 연결
              $3.node를 $$.node의 third child로 연결 }
        | NUMBER {
              tnode = build_node(NUMBER);
              $$.node = get_newnode();
              tnode를 $$.node의 first child로 연결
        }
```

심볼 테이블 구성

- 변수명, 상수명, 함수명 등 명칭들은 각 유형에 따라 속성이 다양함
  - 배열의 경우 : 차원수, 인덱스 허용 범위, 데이터 타입 등

- 비지역 변수(non-local variable)
  - 블록 구조 내에서 그 변수가 정의된 블록을 찾아가야 할 필요성에 의해 각 명칭들에 대한 정보 수집 및 관리
  - 전단부(분석 단계)에서 각 명칭들에 대한 속성 수집
  - 후단부에서 목적 코드를 생성할 때 제약 조건 검사 및 코드 생성에 활용

- 심볼 테이블 내용
  - 변수: type, 변수 유형(전역 변수, 지역 변수 등), scope 등
  - 배열: 차원수, 첨자 허용 범위, 데이터 타입 등
  - 구조체 변수: 각 field의 변수명과 타입 등
  - 함수: 인자 개수, 각 인자의 속성, return type 등

- 심볼 테이블 구성 관련 내용
  - 각 명칭들에 대한 속성 정보 수집
  - 블록 구조 언어에서 비지역 변수들의 엑세스 방법
  - 심볼 테이블 탐색 알고리즘

- 심볼 테이블 관리
  - 블록 구조 언어에서는 중첩된 블록에서 각각 변수 선언이 가능하므로 동일한 명칭의 변수들이 각 블록에 선언될 수 있음
  - 특정 블록에서 가시적인 변수(visible variable)와 비가시적인 변수(invisible variable)를 구분하여 엑세스할 수 있어야 함

- 특정 블록에서 사용된 변수 참조 방법
  - 해당 블록에서 선언된 변수를 우선적으로 참조
  - 그 블록에서 선언되지 않은 변수는 비지역 변수로 간주
    - 바깥 블록 링크(static link)를 추적하여 그 변수의 속성을 찾아감

- 심볼 테이블 탐색 방법
  - 선형 탐색(linear search), 이진 탐색(binary search), 해시(hash) 테이블 탐색 등

오류 처리 기능

- 컴파일러의 각 단계에서 오류 발생
  - 오류가 발생한 정확한 위치 및 오류 유형에 대하여 오류 수정에 도움이 되는 구체적인 메시지를 출력
  - 예) 변수를 선언하지 않고 사용했을 때
    - 이 변수가 사용된 모든 문장들을 오류로 판단?
    - 컴파일러 각 단계에서 오류 수정에 도움이 되는 오류 메시지 생성

- 어휘 분석 단계에서 발생하는 오류
  - 삽입 오류(insertion error)
  - 삭제 오류(deletion error)
  - 교체 오류(substitution error)
    - int를 imt로 입력하는 경우
  - 자리바꿈 오류(transposition error)
    - swtich

- 구문 분석(파싱) 단계에서 발생하는 오류
  - 대부분의 오류는 구문 분석 단계에서 발생하는 구문 오류(syntax error)
  - 구문 오류란?
    - 생성규칙 적용 실패
  - 첫 번째 구문 오류를 발견했을 때 생성규칙을 적용할 수 없으므로 더 이상 파싱을 진행하지 못함
    - "syntax error" 메시지를 출력하고 파싱 중단?
  - 컴파일할 때마다 한번에 오류를 1개씩 발견하고 컴파일을 중단한다면
    - 오류를 1개 수정한 후에 다시 컴파일?

- 컴파일러의 오류 처리 루틴
  - 오류 수정에 도움이 되는 구체적인 오류 메시지 작성 방법
  - 가급적 한 번의 컴파일로 모든 오류들을 발견
  - 어떤 오류가 2차 오류를 발생하지 않도록 함 – error triggering 방지

- 컴파일러가 한 번의 컴파일에 의해 모든 오류를 발견해 주기를 원함
  - 다수의 오류가 포함되어 있을 때 가급적 한꺼번에 모든 오류를 발견
  - 오류 수정의 편의성: 오류 수정에 불필요한 시간을 낭비하지 않도록 함

- 파싱 과정에서 구문 오류가 발생했다는 것의 의미
  - 현재 상태에서 입력 심벌에 대해 파싱표 탐색 오류
    - shift 혹은 reduce 연산이 지정되지 않은 파싱표의 빈 곳
  - 파싱 과정에서 파싱표의 빈곳을 만나게 되면 파서는 더 이상 파싱을 진행할 수가 없음
    - 이 상태에서는 파서가 두 번째, 세 번째 오류들을 한꺼번에 발견할 수가 없다.

- 오류 메시지를 구체적으로 자세하게 생성하려면
  - 파싱표 빈곳에 오류 유형을 기술하고,
  - 각 오류 유형에 적합한 오류 메시지 생성

- 자동 오류 복구 루틴(automatic error recovery routine)
  - 파싱 과정에서 오류가 발생했다고 하더라도 계속해서 파싱을 진행하여 그 다음 오류를 발견
    - 오류 메시지 출력 후에 첫 번째 오류를 자동으로 복구하여 정상적으로 파싱을 계속 진행
    - 컴파일러가 오류를 자동으로 복구할 수 있어야 함

- 자동 오류 복구: panic mode
  - 스택의 내용을 적당히 변경시키거나 오류 발생 부분 제거
  - 입력 버퍼에서 오류 발생 부분을 변경하거나 제거

  - panic mode: 오류 복구 과정은 정상적으로 파싱을 진행하는 parsing mode와는 별개의 처리 과정

코드 최적화

- 코드 최적화(code optimization)
  - 컴파일러가 생성한 목적 코드의 실행 효율을 최대화
  - 기계적인 방법으로 고급 언어 문장을 컴파일을 하게 되면 불필요한 명령어가 추가되는 경우가 많음

- 예1) X=A+B를 어셈블리어로 생성한 예

```
load r1, A
load r2, B
add r1, r2
store r1, X
```

- 예2) X=A+B+C
  - A+B의 덧셈 결과를 임시 변수에 저장
  - 이 임시 변수와 C를 더해 주는 방식으로 목적 코드 생성

```
load r1, A
load r2, B
add r1, r2
store r1, TEMP
load r1, TEMP
load r2, C
add r1, r2
store r1, X
```

- 밑줄친 두 개의 명령어 store와 load는 불필요한 명령어이므로 제거

효율적인 목적 코드 생성 기법

- Strength reduction(연산 강도 저하)
  - 단위 연산 시간이 긴 명령어를 연산 시간이 짧은 명령으로 대치
  - 예1) 2*3 → 2+2+2
  - 예2) 2*3 → 3*2 → 3+3 에 대한 명령어 생성
  - 예3) $a*2^n$ (2의 배수 곱셈, 나눗셈) → shift 연산으로 대치

- Loop 최적화
  - n번 반복하는 반복문 내에서 1개의 불필요한 명령을 제거 → 불필요한 명령 n개 제거와 동일한 효과
  - Loop 불변(loop invariant) 코드
    - 반복문 내에서 동일한 결과로 계산되는 문장
    - 반복문 밖에서 계산함 → n번의 계산을 1번 계산으로 최적화
    - 예)

```
for (i=0; i < n; i++)
        a[i] = b + i + 1;
```

    - 위 for문을 아래와 같이 변형: 덧셈 연산을 n번 줄이는 효과

```
temp = b + 1;
for (i=0; i < n; i++)
        a[i] = temp + i;
```

- 배열 a[i] 위치에 저장된 데이터를 읽어올 때
  - "배열 a의 시작주소" + i * 4
    - 곱셈 연산이 필요함
  - 배열의 모든 원소들을 순서대로 읽어올 때 a[i]의 주소 계산 방법
    - "a[i]의 주소" → "a[i-1]의 주소" + 4
    - 곱셈 연산을 덧셈 연산으로 바꿔줌
- 예)

```
sum = 0;
for (i=0; i < n; i++)
        sum = sum + a[i];
```

중간 언어의 표현 방식

- 각 기계어 유형들의 중간적인 형태
  - 가상 기계(abstract machine) 코드
  - 트리 구조(tree-structured) 코드
  - n-tuple 방식

- 가상 기계 코드
  - 가상의 기계에서 실행되는 코드: 일반적으로 스택 기계(stack machine)를 가상의 기계로 사용
  - 가상 기계 코드를 사용한 예
    - Pascal 컴파일러 제작할 때 사용한 P-code, U-code, EM-code 등
    - 자바 가상 기계(JVM: Java Virtual Machine) : 스택 기반 가상 기계의 대표적인 사례
    - 자바의 바이트 코드는 기계 독립적인 환경에서 JVM이 설치된 모든 플랫폼에서 실행 가능

- 트리 구조 코드
  - 파스 트리에서 목적 코드를 생성하는데 불필요한 논터미널 제거
  - 연산자에 해당하는 터미널을 부모 노드, 피연산자에 해당하는 터미널들을 자식 노드로 구성
  - 파싱 결과를 생성할 때 추상 구문 트리(AST: Abstract Syntax Tree)로 생성
    - 트리 구조 코드는 CMU의 컴파일러-컴파일러 프로젝트에서 사용한 TCOL(Tree structured COmmon Language)과 Ada 컴파일러를 구현할 때 사용한 Diana(Descriptive Intermediate Attribute Notation for Ada)가 대표적

- n-tuple 방식
  - 피연산자 개수에 따라 triple 코드와 quadruple 코드로 구현
  - Triple 코드는 (op, operand-1, operand-2) 형태로 구성
  - Quadruple 코드
    - (op, operand-1, operand-2, result) 형태로 구성
    - GNU에서 제작한 C/C++ 컴파일러는 RTL(Register Transfer Language)라는 중간 언어를 사용