



프로그래밍언어론

2nd edition

Tucker and Noonan

2장 구문구조

컴파일러로 처리하기 간단한 언어는
프로그래머가 보기에
간단하다.

니클라우스 워드 (*Niklaus Wirth*) [1974]



소목차

- 2.1 문법
 - 2.1.1 백커스/나우어 형식(BNF)의 문법
 - 2.1.2 유도
 - 2.1.3 파스 트리
 - 2.1.4 결합성과 우선순위
 - 2.1.5 모호한 문법
- 2.2 확장 BNF
- 2.3 소규모 언어 Clite의 구문구조
 - 2.3.1 어휘 구문구조
 - 2.3.2 구체 구문구조
- 2.4 컴파일러와 실행기
- 2.5 구문구조와 의미구조의 연결
 - 2.5.1 추상 구문
 - 2.5.2 추상 구문 트리
 - 2.5.3 Clite의 추상 구문 트리

구문구조란?

프로그래밍 언어의 구문구조(*syntax*)는 문법적으로 맞는 모든 프로그램을 정확하게 기술한 것이다.

정형적으로 기술된 최초의 구문구조는 Algol 60의 정의에서 처음 등장한 이래, 대부분의 언어에서 사용

세 단계:

- 어휘 구문구조(*Lexical syntax*)
- 구체 구문구조(*Concrete syntax*)
- 추상 구문구조(*Abstract syntax*)



구문구조의 단계

어휘 구문구조 = 언어를 구성하는 기본 기호(이름, 값, 연산자 등)를 정의

구체 구문구조 = 계산식, 문장, 프로그램을 작성하는 규칙

추상 구문구조 = 구두점이나 괄호와 같은 구문 인식전용 구조를 제외한 핵심적인 구문정보만으로 구성된 구문 정의






2.1 문법

메타언어(*metalanguage*) 다른 언어를 기술하는데 사용하는 언어를 말한다.

문법(*grammar*)은 언어의 문법구조를 정의하는데 사용하는 메타언어이다.

목적: 문법으로 프로그래밍언어의 구문구조를 정의하는 것



2.1.1 백커스/나우어 형식 (BNF)

- (참스키 계층의 하나인) 문맥자유문법을 실용적으로 사용할 수 있도록 만든 것
- 백커스/나우어 형식(Backus Normal Form)이라 함
- Algol 60의 구문구조를 정의하는데 처음 사용
- 현재 대부분 언어의 구문구조를 정의하는데 사용

BNF 문법

다음의 집합으로 구성

생성 규칙(*Productions*): P

단말자 기호(*Terminal symbols*): T

비단말자 기호(*Nonterminal symbols*): N

시작 기호(*Start symbol*): $S \in N$

생성규칙은 다음과 같이 쓴다.

$$A \rightarrow \omega$$

여기서 $A \in N$ 이고, $\omega \in (N \cup T)^*$ 이다.



예: 이진숫자(Binary Digits) 만드는 규칙

다음 문법을 보자.

$binaryDigit \rightarrow 0$

$binaryDigit \rightarrow 1$

이렇게 써도 같은 표현이다.

$binaryDigit \rightarrow 0 \mid 1$

여기서 수직선(\mid)은 양자택일을 나타내는 메타기호이다.



2.1.2 유도

다음 문법을 보자.

$$Integer \rightarrow Digit \mid Integer\ Digit$$
$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

352와 같은 부호가 붙지 않은 정수를 이 문법으로 유도해낼 수 있다.



352를 *Integer*로 유도하기

이 6단계로 이루어지는 유도 단계는 다음으로 시작한다.

Integer



352의 유도 (단계 1)

문법 규칙 하나를 적용하면 다음과 같이 유도된다.

Integer \Rightarrow *Integer Digit*

352의 유도 (단계 1-2)

오른쪽에 있는 비단말자 하나에 문법 규칙 하나를 적용하면 다음과 같이 유도된다.

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \end{aligned}$$

352의 유도 (단계 1-3)

같은 방식으로 계속 유도해나가면 다음과 같다.

$$\begin{aligned} \textit{Integer} &\Rightarrow \textit{Integer Digit} \\ &\Rightarrow \textit{Integer } 2 \\ &\Rightarrow \textit{Integer Digit } 2 \end{aligned}$$

352의 유도 (단계 1-4)

Integer \Rightarrow *Integer Digit*
 \Rightarrow *Integer 2*
 \Rightarrow *Integer Digit 2*
 \Rightarrow *Integer 5 2*

352의 유도 (단계 1-5)

Integer \Rightarrow *Integer Digit*
 \Rightarrow *Integer 2*
 \Rightarrow *Integer Digit 2*
 \Rightarrow *Integer 5 2*
 \Rightarrow *Digit 5 2*

352의 유도 (단계 1-6)

단말자 기호만 남으면 유도가 종료된다.

Integer \Rightarrow *Integer Digit*

\Rightarrow *Integer* 2

\Rightarrow *Integer Digit* 2

\Rightarrow *Integer* 5 2

\Rightarrow *Digit* 5 2

\Rightarrow 3 5 2

352의 유도 (다른 방식)

Integer \Rightarrow *Integer Digit*
 \Rightarrow *Integer Digit Digit*
 \Rightarrow *Digit Digit Digit*
 \Rightarrow 3 *Digit Digit*
 \Rightarrow 3 5 *Digit*
 \Rightarrow 3 5 2

이와 같이 단계마다 가장 왼쪽 비단말자가
대치되는 유도 방식을 ‘왼쪽 우선 유도’라고 한다.
(첫 번째 유도는 ‘오른쪽 우선 유도’라고 한다.)

유도 표기법

$Integer \Rightarrow^* 352$

$Integer$ 문법규칙을 사용하여 유한 번의 단계를 거쳐 352가 유도될 수 있음을 뜻한다.

$352 \in L(G)$

352는 문법 G 로 정의된 언어에 포함됨을 뜻한다.

$L(G) = \{ \omega \in T^* \mid Integer \Rightarrow^* \omega \}$

문법 G 로 정의된 언어는 문법규칙 $Integer$ 로 유도할 수 있는 모두 기호 문자열 ω 의 집합임을 뜻한다.

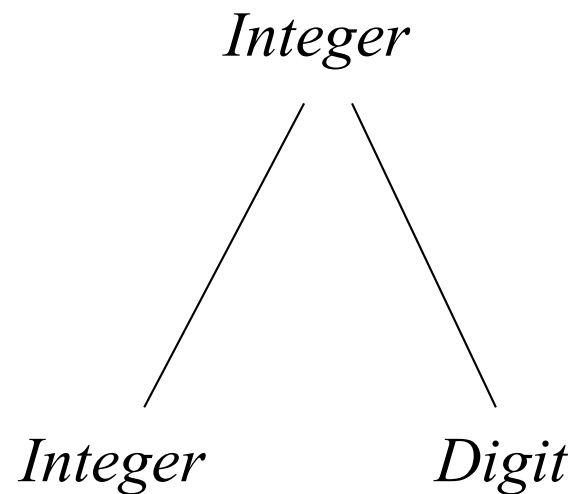
2.1.3 파스 트리

파스트리(*parse tree*)는 유도를 그림 형식으로 표현한 것이다.

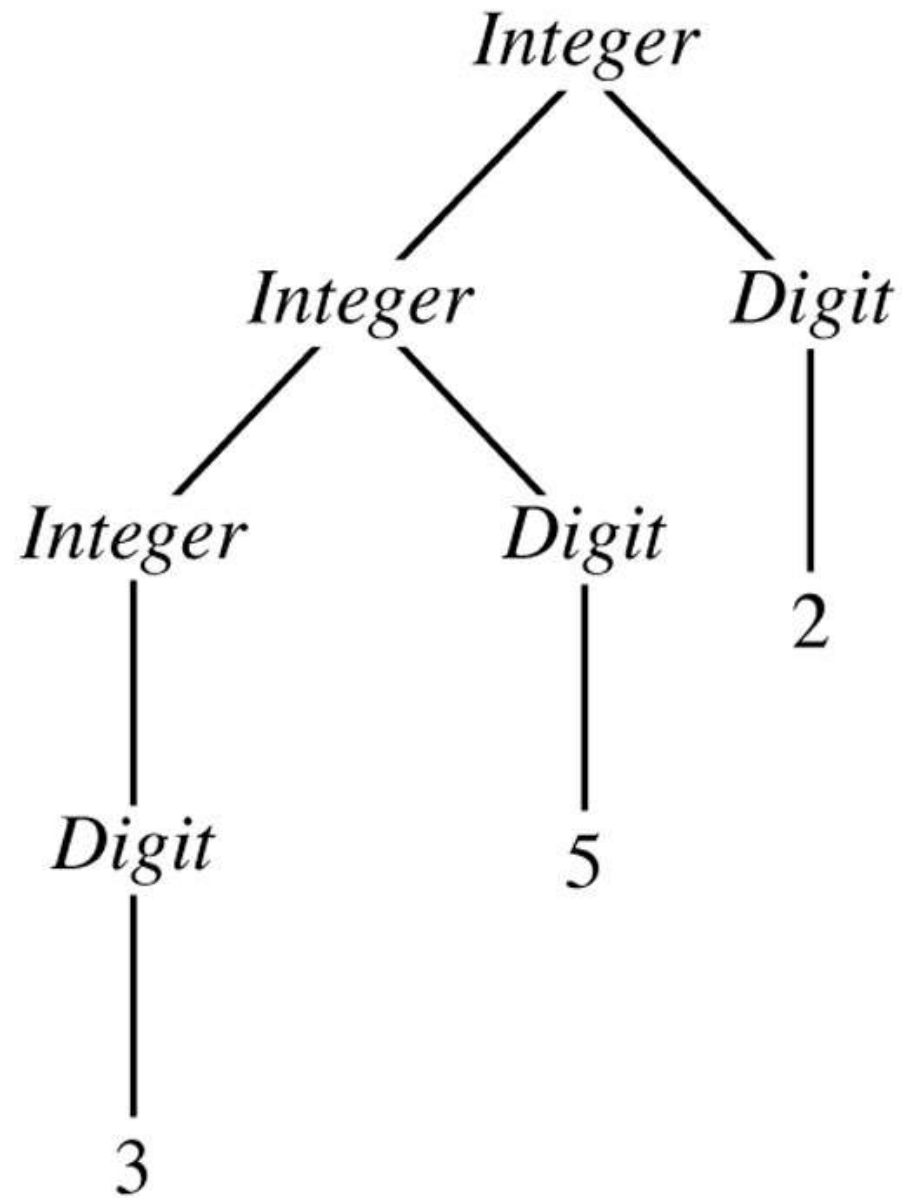
- *트리의 내부 노드는 유도의 한 단계에 해당한다.*
- *노드의 자식 노드들은 생성규칙의 우변을 나타낸다.*
- *각 잎새 노드를 왼쪽에서 오른쪽으로 읽으면 유도된 문자열 기호가 된다.*

예를 들어, 다음 단계는 아래와 같은 파스트리로 표현된다.

Integer \Rightarrow *Integer Digit*



Integer로서
352에 대한 파스 트리
그림 2.1

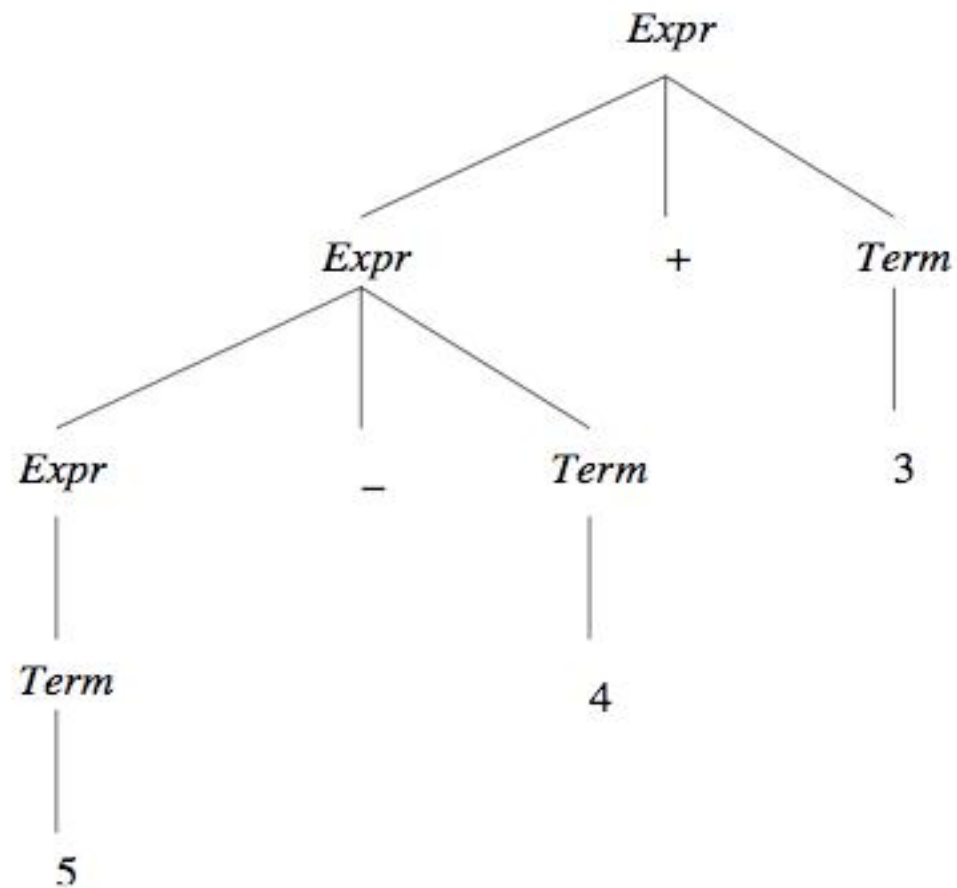


산술계산식 문법

다음 문법은 한자리 정수, 덧셈, 뺄셈을 표현할 수 있는 산술계산식 언어를 정의하고 있다.

$$Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$$
$$Term \rightarrow 0 \mid \dots \mid 9 \mid (Expr)$$

문자열 5-4+3의 파스 트리
그림 2.2



2.1.4 결합성과 우선순위

문법은 계산식에서 연산자의 결합성과 우선순위를 정의하는데 사용된다.

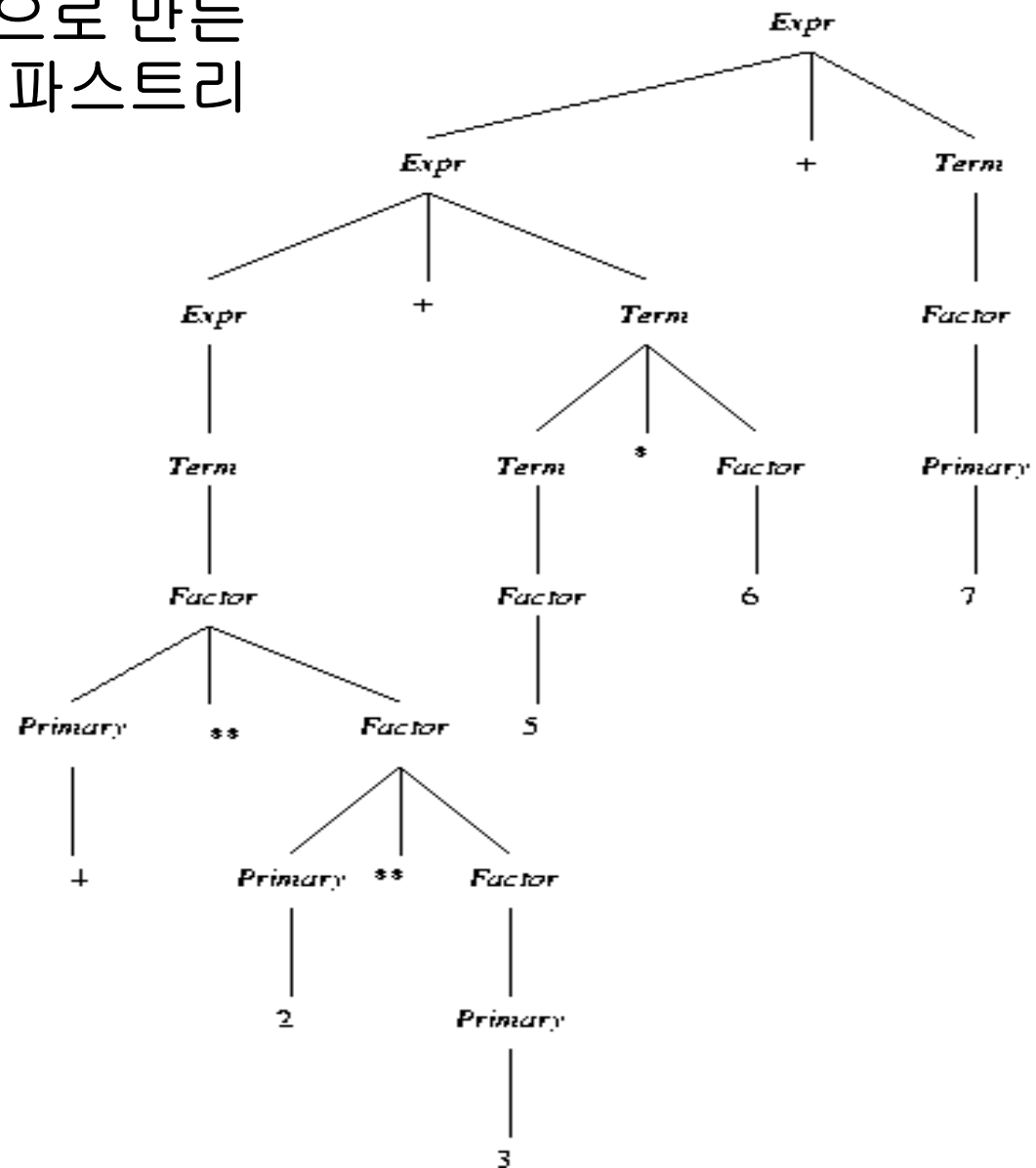
예를 들어, + 와 - 는 수학에서 좌결합 연산자이고,

* 와 / 는 +와 - 보다 우선순위가 높다.

좀 더 복잡한 문법 G_1 을 살펴보자.

$$Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$$
$$Term \rightarrow Term * Factor \mid Term / Factor \mid$$
$$Term \% Factor \mid Factor$$
$$Factor \rightarrow Primary ** Factor \mid Primary$$
$$Primary \rightarrow 0 \mid \dots \mid 9 \mid (Expr)$$

문법 Grammar G_1 으로 만든
 $4**2**3+5*6+7$ 의 파스트리
 그림 2.3



문법 G_1 의 결합성과 우선순위
표 2.1

우선순위	결합성	연산자
3	right	**
2	left	* / %
1	left	+ -

주목: 이 관계는 파스트리의 구조를 보면 다음과 같이 관찰할 수 있다. 우선순위가 높으면 아래쪽에 위치, 좌결합이면 각 수준에서 왼쪽으로 쏠림.

2.1.5 모호한 문법

문법으로 만들어내는 문자열 하나로 두 개 이상의 다른 모양의 파스트리를 만들 수 있는 경우, 그 문법은 모호하다(*ambiguous*)고 한다.

예를 들어, 위의 문법 G_1 은 모호하지 않다.

C, C++, Java 는

- 다양한 연산자와
- 다양한 우선순위 수준을 제공하고 있다.

문법 정의의 규모를 키우는 대신,

- 규모가 작은 모호한 문법을 쓰고,
- 우선순위와 결합성을 따로 정의한다 (예, 표 2.1)

모호한 문법 G_2

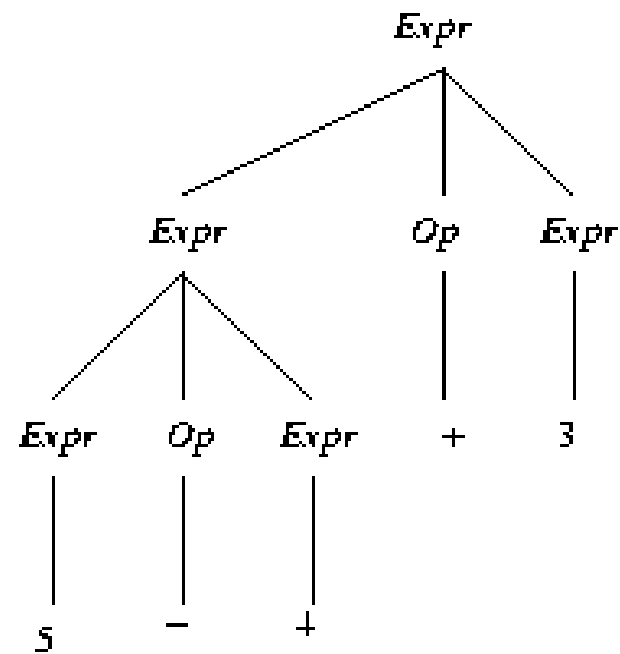
$Expr \rightarrow Expr \ Op \ Expr \mid (Expr) \mid Integer$

$Op \rightarrow + \mid - \mid * \mid / \mid \% \mid **$

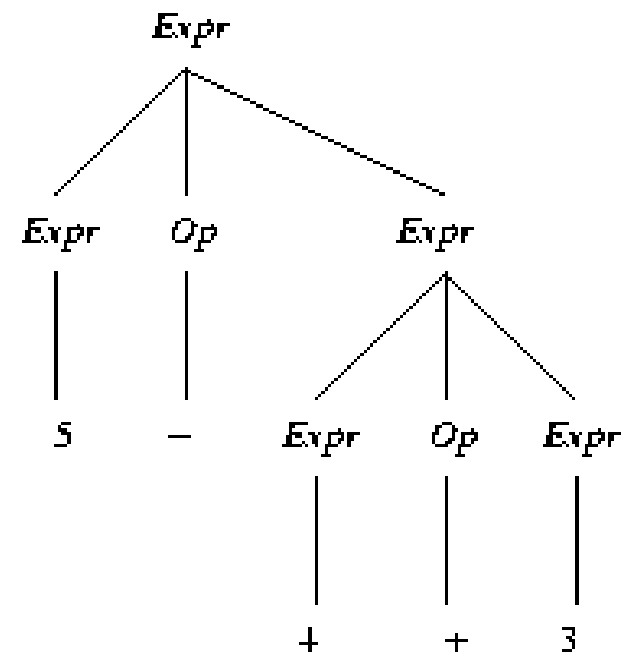
Notes:

- G_2 은 G_1 과 동일함. 즉, 같은 언어를 정의함.
- G_2 은 G_1 .보다 비단말자의 개수와 생성규칙의 개수가 작다.
- 그러나 G_2 는 모호하다.

문법 G_2 의 모호함을 보여주는 $5-4+3$ 의 두 파스 트리
그림 2.4



(a)



(b)

비대칭 선택문 문제(Dangling Else)

$\text{IfStatement} \rightarrow \text{if (Expression) Statement} \mid$
 $\text{if (Expression) Statement else Statement}$

$\text{Statement} \rightarrow \text{Assignment} \mid \text{IfStatement} \mid \text{Block}$

$\text{Block} \rightarrow \{ \text{Statements} \}$

$\text{Statements} \rightarrow \text{Statements Statement} \mid \text{Statement}$




예

어느 'if'가 'else'와 관련된 쌍인가?

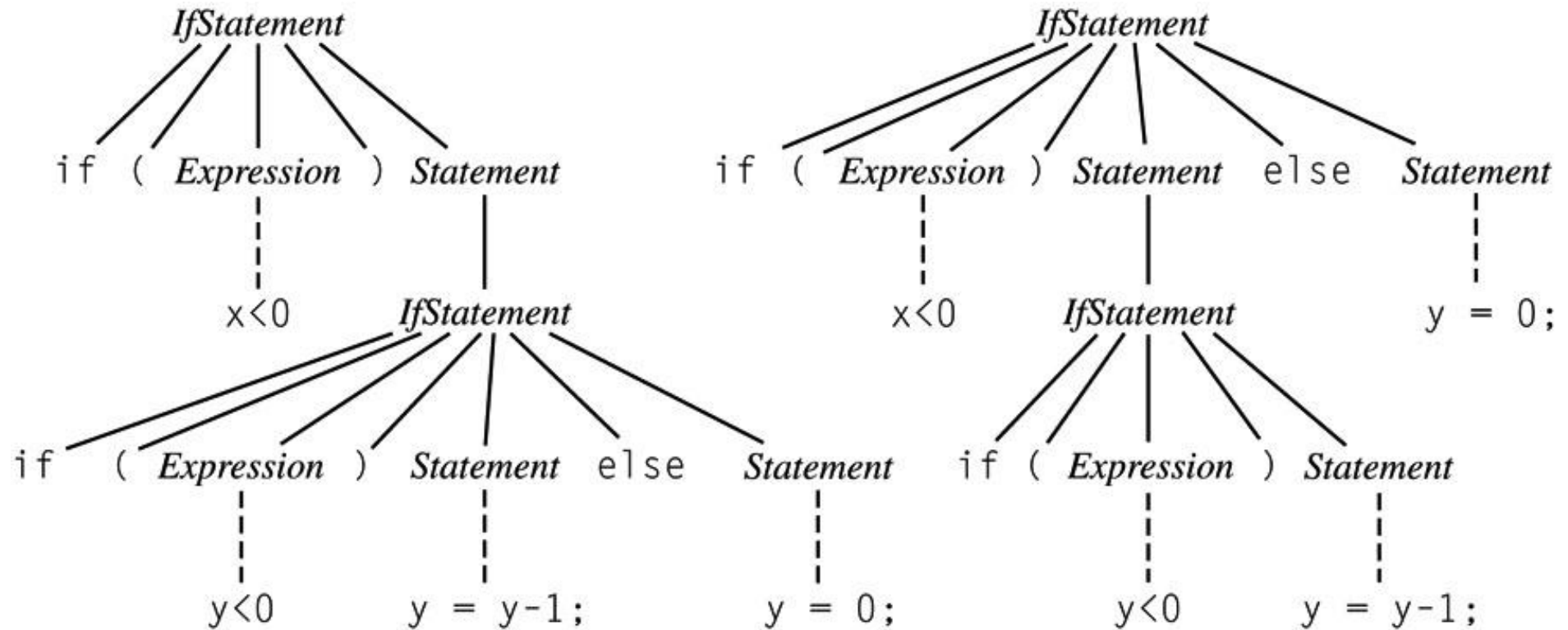
```
if (x < 0)
    if (y < 0) y = y - 1;
    else y = 0;
```

Answer: 어느 것이든 가능하다!



비대칭 선택문 문제로 인한 문법의 모호성

그림 2.5



비대칭 선택문의 모호성 해결하기

1. Algol 60, C, C++: `else`와 가장 가까운 `if`를 우선적으로 짝지움; `{ }` 또는 `begin...end`으로 둘러싸면 이 규칙을 무시할 수 있음.
2. Algol 68, Modula, Ada: 각 조건문마다 유일한 키워드를 사용하여 끝맺음 (예: `if...fi`)
3. Java: 문법으로 두개의 모양이 다른 조건문을 따로 정의하여 해결

IfThenStatement \rightarrow `if (Expression) Statement`

IfThenElseStatement \rightarrow `if (Expression) StatementNoShortIf
else Statement`

*StatementNoShortIf*는 *IfThenStatement*를 제외한 모든 문장을 나타낸다.

2.2 확장 BNF (EBNF)

BNF:

- 반복을 재귀적으로 표현
- 비단말자로 그룹화

EBNF: 메타문자 추가 사용

- { } 0번 이상 반복
- () 여러 개 중에서 하나는 반드시 선택
- [] 옵션, 선택하거나 버리거나 둘 중 하나

EBNF 사례

Expression 은 하나 이상의 *Terms*이 연산자 + 와 - 로 분리되어 나열된 것임

$$Expression \rightarrow Term \{ (+ | -) Term \}$$
$$IfStatement \rightarrow \text{if } (Expression) Statement \text{ [else Statement]}$$

C스타일 EBNF 에서는 아래 첨자 _{opt} 를 사용하여 옵션으로 사용하는 부분을 표시한다. 예:

IfStatement:

$$\text{if } (Expression) Statement ElsePart_{opt}$$

ElsePart:

$$\text{else Statement}$$

EBNF 를 BNF 로 변환하기

EBNF 문법은 BNF로 언제든지 변환이 가능하다.

예를 들어, 다음 EBNF문법은

$$A \rightarrow x \{ y \} z$$

다음과 같이 BNF 문법으로 변환된다.

$$A \rightarrow x A' z$$

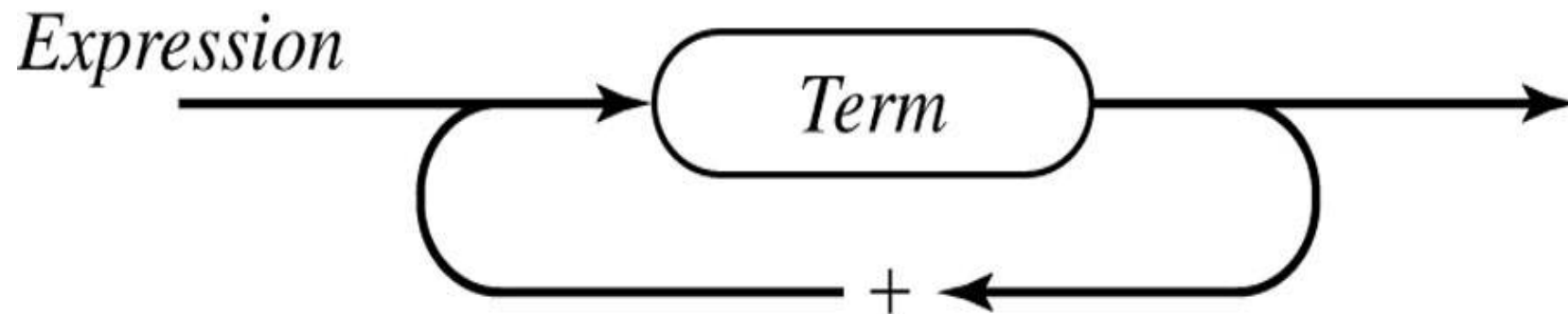
$$A' \rightarrow \mid y A'$$

()와 []로 표현된 EBNF 문법을 BNF로 변환하는 방법은 연습문제로 남겨둔다.

EBNF가 BNF보다 표현력이 더 좋은 것은 아니지만, 보기에는 일반적으로 훨씬 간단 명료하다.

덧셈 연산자가 포함된 **Expressions**에 대한
구문 다이어그램

Figure 2.6



2.3 소규모 언어 Clite의 구문구조

- C의 일부 만을 쓰는 동기:

<u>언어</u>	<u>문법 (쪽수)</u>	<u>참고문헌</u>
Pascal	5	Jensen & Wirth
C	6	Kernighan & Richie
C++	22	Stroustrup
Java	14	Gosling, et. al.

- *Clite* 문법은 1쪽에 작성 가능 (슬라이드로는 3쪽), 따라서 언어 설계를 공부하기에는
안성맞춤

그림 2.7 *Clite* 문법: Statements

$Program \rightarrow \text{int main () } \{ Declarations Statements \}$

$Declarations \rightarrow \{ Declaration \}$

$Declaration \rightarrow Type Identifier [[Integer]] \{ , Identifier [[Integer]] \}$

$Type \rightarrow \text{int} \mid \text{bool} \mid \text{float} \mid \text{char}$

$Statements \rightarrow \{ Statement \}$

$Statement \rightarrow ; \mid Block \mid Assignment \mid IfStatement \mid WhileStatement$

$Block \rightarrow \{ Statements \}$

$Assignment \rightarrow Identifier [[Expression]] = Expression ;$

$IfStatement \rightarrow \text{if (Expression) Statement} [\text{else Statement}]$

$WhileStatement \rightarrow \text{while (Expression) Statement}$

그림 2.7 *Clite* 문법: Expressions

$Expression \rightarrow Conjunction \{ \mid \mid Conjunction \}$

$Conjunction \rightarrow Equality \{ \&\& Equality \}$

$Equality \rightarrow Relation [EquOp Relation]$

$EquOp \rightarrow == \mid !=$

$Relation \rightarrow Addition [RelOp Addition]$

$RelOp \rightarrow < \mid <= \mid > \mid >=$

$Addition \rightarrow Term \{ AddOp Term \}$

$AddOp \rightarrow + \mid -$

$Term \rightarrow Factor \{ MulOp Factor \}$

$MulOp \rightarrow * \mid / \mid \%$

$Factor \rightarrow [UnaryOp] Primary$

$UnaryOp \rightarrow - \mid !$

$Primary \rightarrow Identifier [[Expression]] \mid Literal \mid (Expression) \mid$
 $Type (Expression)$

그림 2.7 *Clite* 문법: 어휘 정의

Identifier \rightarrow *Letter* { *Letter* | *Digit* }

Letter \rightarrow a | b | ... | z | A | B | ... | Z

Digit \rightarrow 0 | 1 | ... | 9

Literal \rightarrow *Integer* | *Boolean* | *Float* | *Char*

Integer \rightarrow *Digit* { *Digit* }

Boolean \rightarrow true | False

Float \rightarrow *Integer* . *Integer*

Char \rightarrow ` ASCII Char `

이 문법에서 빠진 구문 관련 이슈

- 주석
- 공백문자의 역할
- `<=`를 하나의 기호로 취급하는지, 아니면 `<`와 `=`를 두 개의 기호로 취급하는지의 구별
- `if`와 같은 키워드와 식별자의 구별

이 이슈는 다음의 두 단계로 구분하여 설명한다.

- 어휘 단계
- 구문 단계

2.3.1 어휘 구문구조

- 입력: 프로그래머가 입력한 ASCII 문자 스트림
- 출력: 다음과 같이 분류된 토큰 또는 기본 기호의 스트림
 - 식별자 *Identifiers* e.g., Stack, x, i, push
 - 리터럴 *Literals* e.g., 123, 'x', 3.25, true
 - 키워드 *Keywords* bool char else false float if int
main true while
 - 연산자 *Operators* = || && == != < <= > >= +
- * / !
 - 구두점 *Punctuation* ; , { } ()

공백문자(Whitespace)

- 공백문자는 빈칸(space), 탭(tab)문자, 줄바꿈(end-of-line) 문자, 주석 내부의 문자들을 말한다.
- 토큰 내부에는 공백문자가 포함될 수 없다.
(문자 또는 문자열 제외)

- 예:

`>=` 토큰 한 개

`> =` 토큰 두 개

Pascal의 공백 문자 예

while a < b do

인정

while a<b do

<의 양쪽에 공백문자 불필요

whilea<bdo

불인정

whilea < bdo

토큰 사이의 경계 구별 불가능



주석(Comments)

- 문법에 정의되어 있지 않음
- *Clite* 는 C++ 형식을 따라 // 를 사용



식별자(Identifier)

- 문자로 시작하여 문자 또는 숫자의 나열
 - Clite 문법에 따르면 if 는 식별자도 되고 키워드도 된다
 - 대부분의 언어는 식별자는 키워드와 구별하여 다르게 취급한다
- 일부 언어에서 식별자를 미리 정해주기도 한다.
필요에 따라 프로그래머가 재정의할 수도 있다.

식별자의 재정의는 위험

```
program confusing;  
const true = false;  
begin  
    if (a < b) = true then  
f(a)  
    else ...
```


식별자의 대소문자는 구별해야 하나?


구식 언어: 구별하지 않는다. 왜?

- *Pascal*: 구별하지 않음
- *Modula*: 구별
- *C, C++*: 구별
- *Java*: 구별
- *PHP*: 일부는 구별하고, 일부는 구별하지 않음.
이렇게 하면, 직교성에는 어떤 영향이?


2.3.2 구체 구문구조

- Based on a parse of its *Tokens*
 - C는 ; 를 문장 종결자로 사용
 - Algol-60, Pascal 은 ; 를 문장 분리자로 사용
- *IfStatement* 의 문법 규칙은 모호함
 - “비대칭 선택문 모호성은 else 없는 가장 가까운 if와 해당 else를 연결하여 해결한다.”

[Stroustrup, 1991]



Clite의 계산식(*Expressions*)

- 13의 문법 규칙
 - 메타 중괄호의 사용 – 연산자는 좌결합
 - C++ 는 계산식을 정의하는 문법 규칙이 4쪽에 달함 [Stroustrup]
 - C 는 우선순위와 결합성을 문법으로 정의하지 않고, 모호한 계산식 문법을 그대로 사용 [Kernighan and Ritchie]
- 

결합성과 우선순위

<u>Clite 연산자</u>	<u>결합성</u>
단항 - !	없음
* /	좌결합
+ -	좌결합
< <= > >=	없음
== !=	없음
&&	좌결합
	좌결합

Clite 동등/관계 연산자

- ... 결합성은 없음.
(Ada에서 따온 아이디어)

- C++에서는 좌결합임

C++에서 다음 관계식은

if (a < x < b)

다음 관계식과 의미가 다르다.

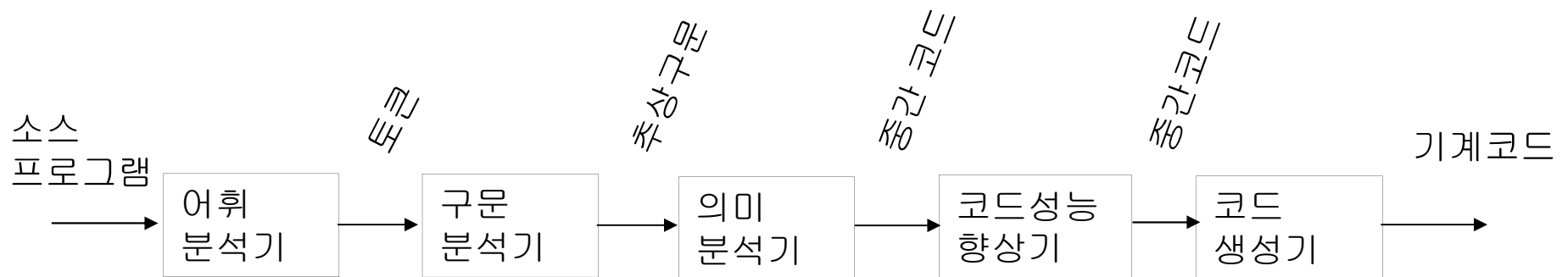
if (a < x && x < b)

전자는 항상 true가 된다!

왜?

연산자에 결합성을 부여한 설계가 바람직하였나?

2.4 컴파일러(Compilers)와 실행기 (Interpreters)





어휘 분석기

- 입력: 문자 스트림
- 출력: 토큰 스트림
- 어휘분석을 따로 하는 이유
 - 속도: 성능향상 부분을 제외하고, 컴파일 총 시간의 75%가 어휘 분석
 - 설계가 간단
 - 기종마다 다른 문자열 사용 (이식성 향상)
 - 줄바꿈 문자의 관례도 기종마다 다름 (이식성 향상)



파서

- BNF/EBNF 문법 기반
 - 입력: 토큰 스트림
 - 출력: 추상 구문 트리 (파스 트리)
 - 추상 구문: 파스트리에서 더 이상 필요없는
구두점 및 비단말자를 제거한 트리
- 





의미 분석

- 식별자의 선언 여부 검사
- 타입 검사
- 타입변환 연산자 삽입
 - 묵시적 타입 변환 → 명시적 타입 변환




코드 성능 향상

- 컴파일하면서 상수 계산식 실행
 - 캐시 수행 성능 향상을 위하여 코드 재배치
 - 공통 부분계산식 삭제
 - 불필요한 코드 삭제
- 





코드 생성

- 출력: 기계 코드
 - 명령 선택
 - 레지스터 관리
 - 피플 성능향상
- 



실행기

- 컴파일러 단계 중 마지막 두 단계를 대체
 - 입력:
 - 혼합형: 중간 코드
 - 순수형: *ASCII 문자 스트림*
 - 혼합형 실행기
 - *Java, Perl, Python, Haskell, Scheme*
 - 순수형 실행기:
 - 대부분 *Basic 언어, shell 명령*
- 

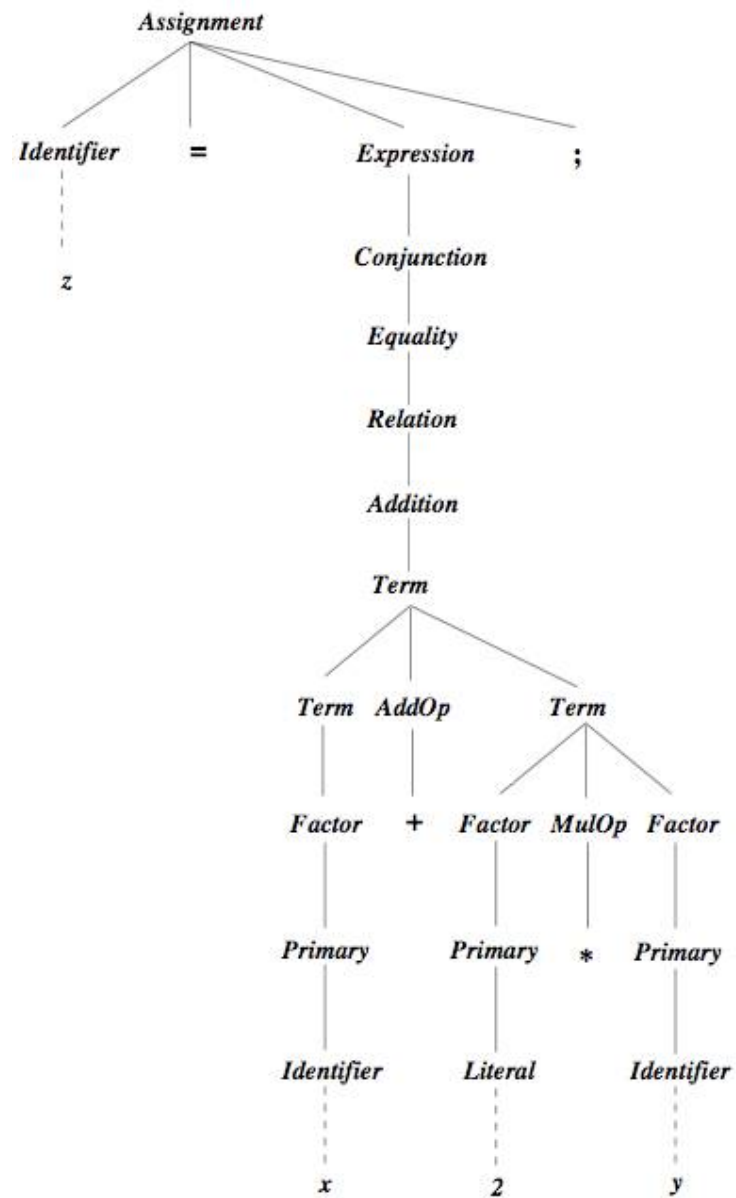


2.5 구문구조와 의미구조의 연결

출력: 파스트리 는 효율적이지 않음

예: 그림 2.9

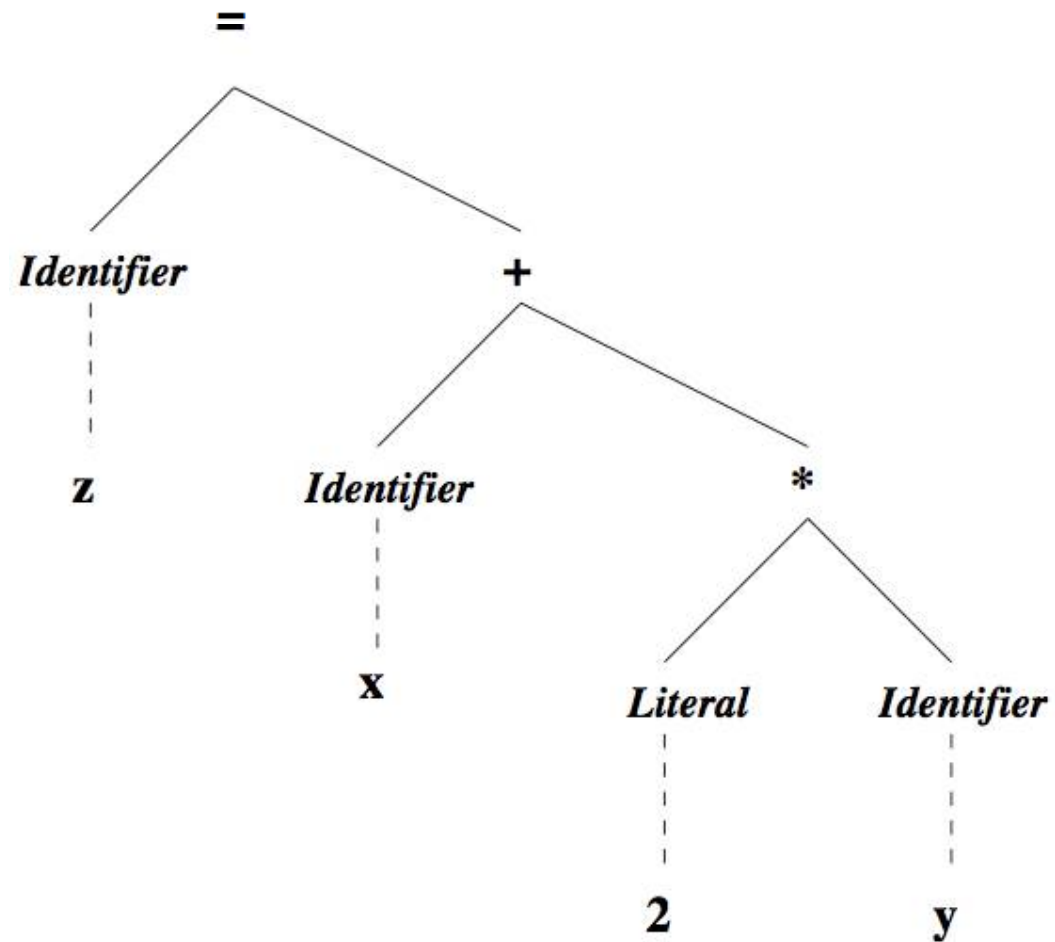
$z = x + 2 * y;$
의 파스 트리
그림 2.9



더 효율적인 트리 찾기

- 파스트리의 모양은 프로그램의 의미를 드러낸다.
- 따라서 모양은 그대로 유지한 채, 비효율적인 부분이 제거된 트리로 만들면 좋겠다.
 - 분리자/구둣점과 같은 단말자 기호 제거
 - 사소하게 루트가 되는 비단말자 모두 제거
 - 나머지 비단말자는 모두 해당 입새 단말자로 대체
- 예: 그림 2.10

$z = x + 2 * y;$
에 대한 추상 구문 트리
그림 2.10



추상 구문구조(Abstract Syntax)

“구문적 설탕(syntactic sugar)”을 제거하고, 꼭 필요한 부분만 가지고 있다.

예: 동일한 의미를 가진 다음 두 루프 구조를 보자.

Pascal

```
while i < n do begin
```

```
    i := i + 1;
```

```
end;
```

C/C++

```
while (i < n) {
```

```
    i = i + 1;
```

```
}
```

이 두 루프에서 꼭 필요한 정보는

- 1) 루프라는 사실과
- 2) 종료 조건은 $i < n$ 이라는 사실과
- 3) 몸통에서 i 의 현재 값을 1씩 증가한다는 사실이다.

Clite 저장문의 추상 구문구조

Assignment = *Variable* target; *Expression* source

Expression = *VariableRef* | *Value* | *Binary* | *Unary*

VariableRef = *Variable* | *ArrayRef*

Variable = *String* id

ArrayRef = *String* id; *Expression* index

Value = *IntValue* | *BoolValue* | *FloatValue* | *CharValue*

Binary = *Operator* op; *Expression* term1, term2

Unary = *UnaryOp* op; *Expression* term

Operator = *ArithmeticOp* | *RelationalOp* | *BooleanOp*

IntValue = *Integer* intValue

...

추상 구문구조를 Java Class로 표현

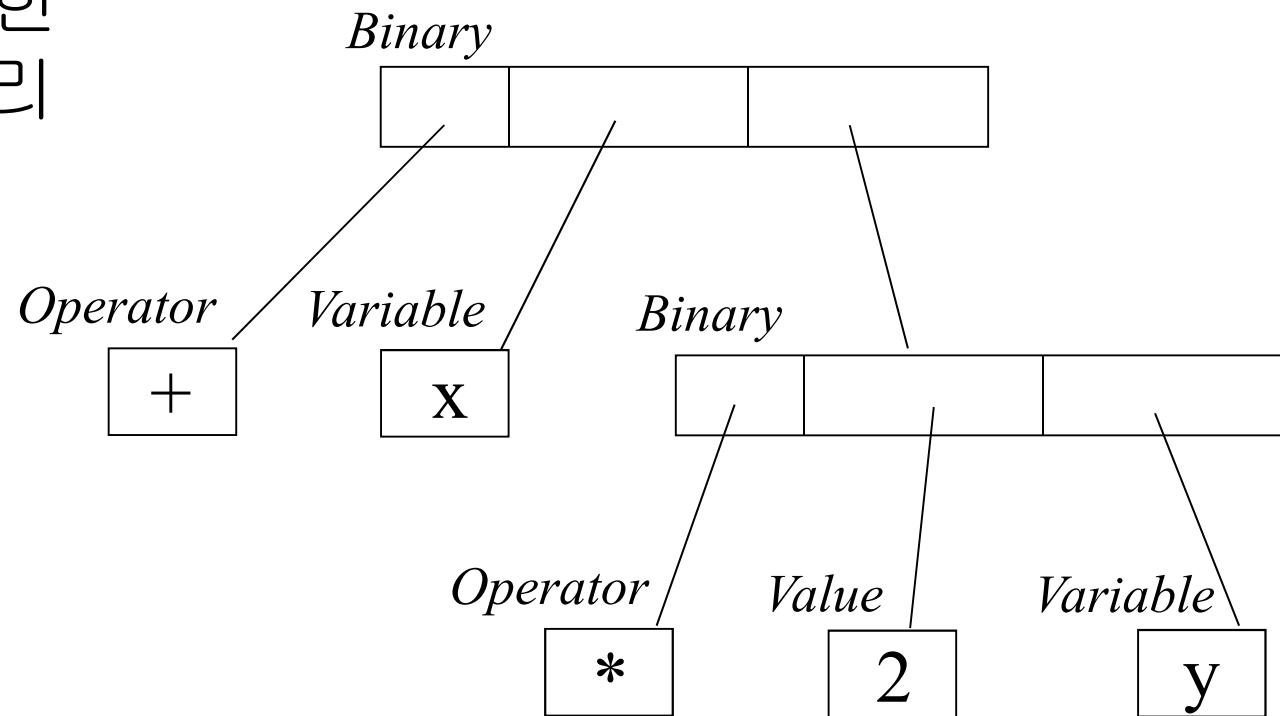
```
abstract class Expression { }
abstract class VariableRef extends Expression { }
class Variable extends VariableRef { String id; }
class Value extends Expression { ... }
class Binary extends Expression {
  Operator op;
  Expression term1, term2;
}
class Unary extends Expression {
  UnaryOp op;
  Expression term;
}
```

추상 구문 트리의 예

이진 노드의 형식

op	term1	term2

$x+2*y$ 에 대한
추상구문트리
(그림 2.13)



Clite 다른 부분의 추상구문구조 (선언 및 문장)

그림 2.14

```
Program = Declarations decpart; Statements body;  
Declarations = Declaration*  
Declaration = VariableDecl | ArrayDecl  
VariableDecl = Variable v; Type t  
ArrayDecl = Variable v; Type t; Integer size  
Type = int | bool | float | char  
Statements = Statement*  
Statement = Skip | Block | Assignment | Conditional | Loop  
Skip =  
Block = Statements  
Conditional = Expression test; Statement thenbranch, elsebranch  
Loop = Expression test; Statement body
```