Chap 4. Context Sensitive Analysis

COMP321 컴파일러

2007년 가을학기

경북대학교 전자전기컴퓨터학부

4.1 Introduction

Beyond Syntax

- example

int x, y = 20;

string z = "123";

…

x = y;

x = z;

…

- grammar

assign → *variable* = *expression* ;

- scanner, parser 모두 OK !

- 그러나…
- x = z : *integer* ← *string*
  - type은 어떻게 알 수 있나?
  - type이 다름은 어떻게 아나?
  - type이 다르면, error 인가?
  - error가 아니면, 어떻게 하나?
  - string을 integer로 어떻게?
- function 처리 시에는 더 많은 문제
  - parameter는 어떻게 구별하나?
  - type이 다르면, 어떻게 하나?
- **scanner, parser 이상의 처리가 필요!**

Beyond Syntax

Context-Sensitive Analysis의 도입

- Answers depend on values, not parts of speech
- Questions & answers involve **non-local information**
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - **Attribute Grammar**
- Use *ad-hoc* techniques
  - **symbol tables**
  - **Ad-hoc code**

4.2 An Introduction to Type Systems

Type Systems

- 현대 programming language의 선택
  - 모든 variable, 모든 expression에 type을 binding
- "**declare before use**" rule
  - CFG: *Procedure* → *Declarations Statements*
  - **Declarations** : 미리 모든 variable을 declare
  - **Statements** : 이제 variable들을 use
- how to implement ?
  - **symbol table**에 정보를 저장

Type System의 목적

- type system을 사용하여 얻게 되는 이득
- **run-time safety** : 잘못된 연산을 미리 방지
  - double + integer : OK !
  - double + complex : illegal in Fortran
- **improving expressiveness**
  - type을 구별할 수 있으면, 다른 연산이 가능
  - 예: operator overloading, function overloading
- **run-time efficiency**
  - un-typed language는 run-time에 일일이 type check
  - in Perl: i = 3; s = "123"; b = i + s; → 연산 가능 !
  - in C: type error 거나, 미리 conversion routine 삽입

Components of a Type System

- built-in types
  - char, int, float, bool, …
- compound types: 새로운 type을 만드는 방법 필요
  - array, enumeration, structure, class, …
- **type equivalence rules**: type 끼리 같은 지 판별
  - name equivalence, structural equivalence
- **type inference rules**
  - type mismatch 시, type을 변환하는 방법

Components of a Type System

- type equivalence의 예

```
struct TreeNode {
    struct TreeNode* left;
    struct TreeNode* right;
};

struct SearchTree {
    struct SearchTree* left;
    struct SearchTree* right;
};
```

- 둘은 equivalent한가, 아닌가?

- function prototype의 도입
  - 예: C++, Java

- int Add(int, int);
- float Add(float, float);
- double Add(double, double);

- x = Add(3, 4.0);
  - 어느 것을 call 해야 하나?

4.3 The Attribute Grammar Framework

Attribute Grammar

- A **context-free grammar** augmented with **a set of rules**
  - Each symbol in the derivation has a set of values, or ***attributes***
  - The rules specify how to compute a value for each attribute
- Example CFG : get a signed binary number

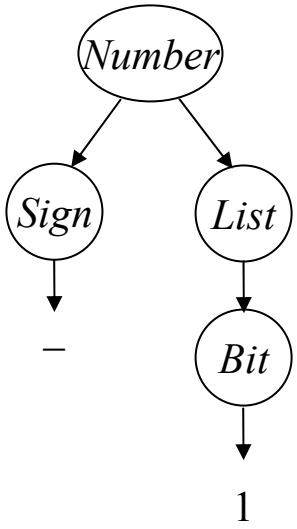
Number → *Sign List*
Sign → +
| –
List → *List Bit*
| *Bit*
Bit → 0
| 1

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Example: parse trees

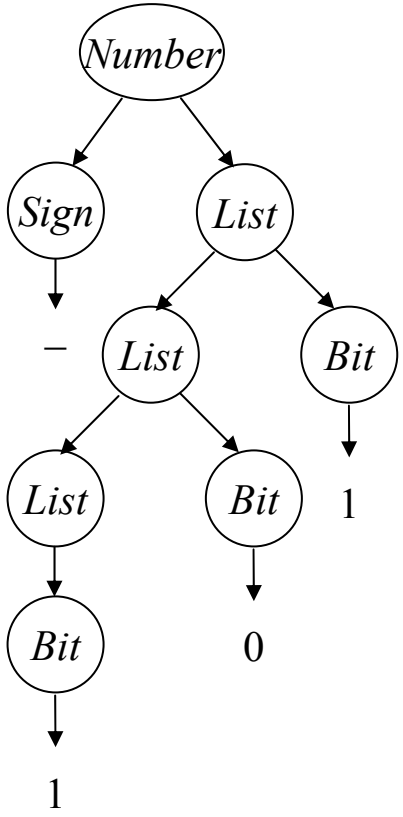
- for "–1"

Number → *Sign List*
→ – *List*
→ – *Bit*
→ – 1



- for "–101"

Number → *Sign List*
→ *Sign List Bit*
→ *Sign List 1*
→ *Sign List Bit 1*
→ *Sign List 1 1*
→ *Sign Bit 0 1*
→ *Sign 1 0 1*
→ – *101*



Example: Attribute Grammar

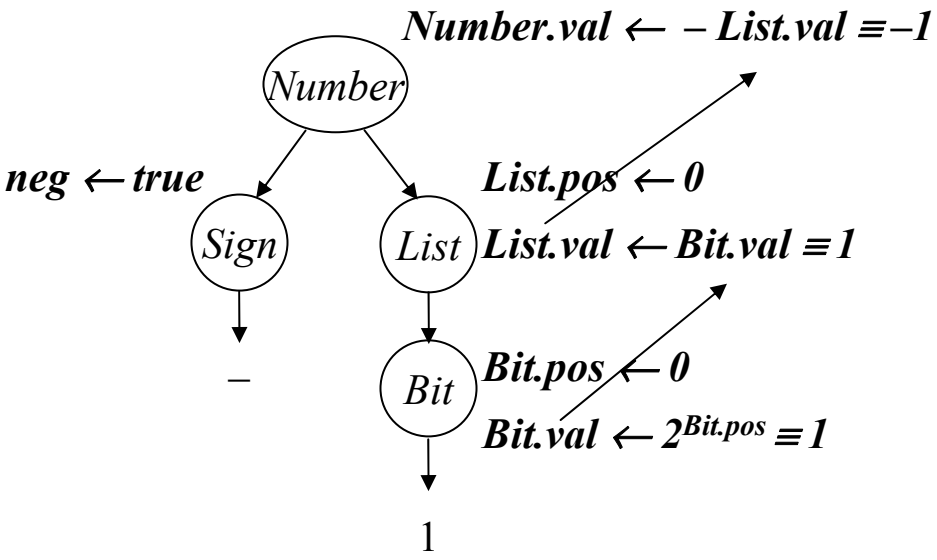
- Add rules to compute the decimal value of a signed binary number
  - AG: a signed binary number → the decimal value

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* if *Sign.neg* then *Number.val* ← – *List.val* else *Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← **false**
	|	–	*Sign.neg* ← **true**
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Example: AG applied for "–1"

- parse tree + AG rules → attribute dependence graph



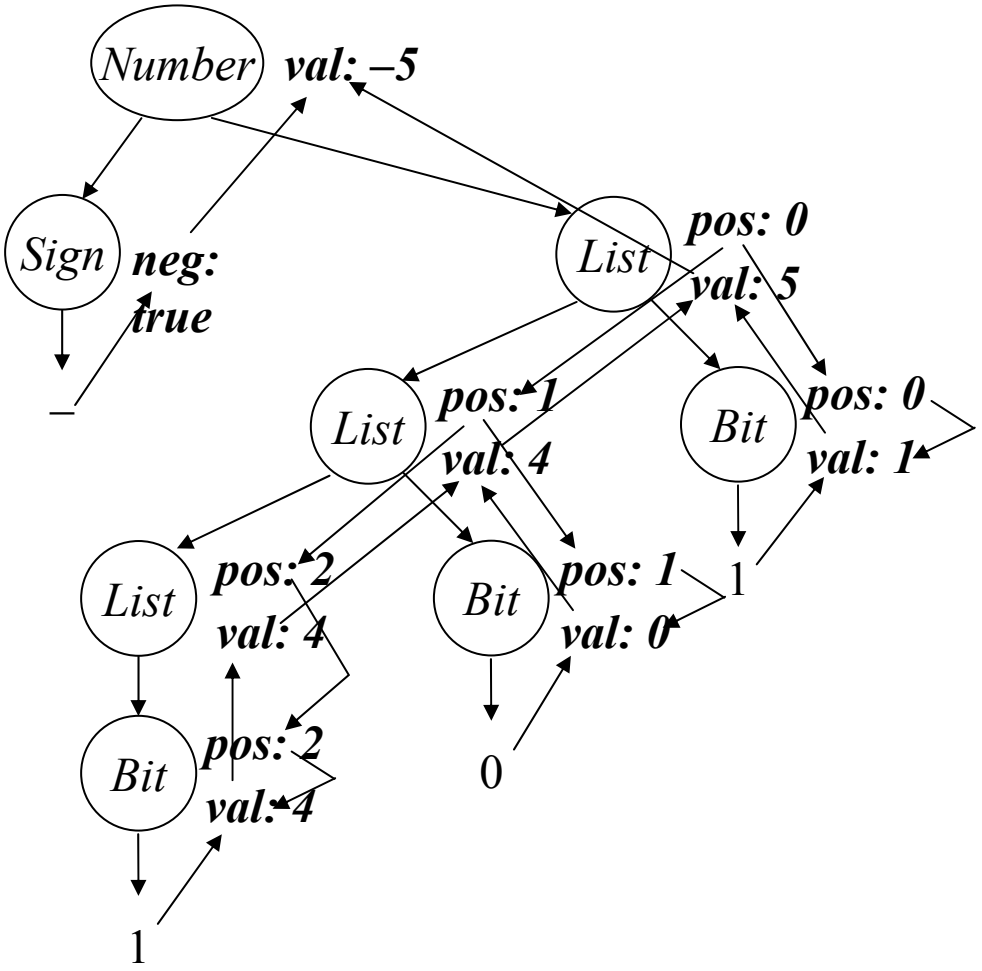
One possible evaluation order:

1 List.pos
2 Sign.neg
3 Bit.pos
4 Bit.val
5 List.val
6 Number.val

Other orders are possible

- original AG by Knuth : use a **data-flow model**
  - independent attributes first
  - others in order as input values become available

Example: AG applied for "–101"



pos : downward
val : upward

Attribute Grammar

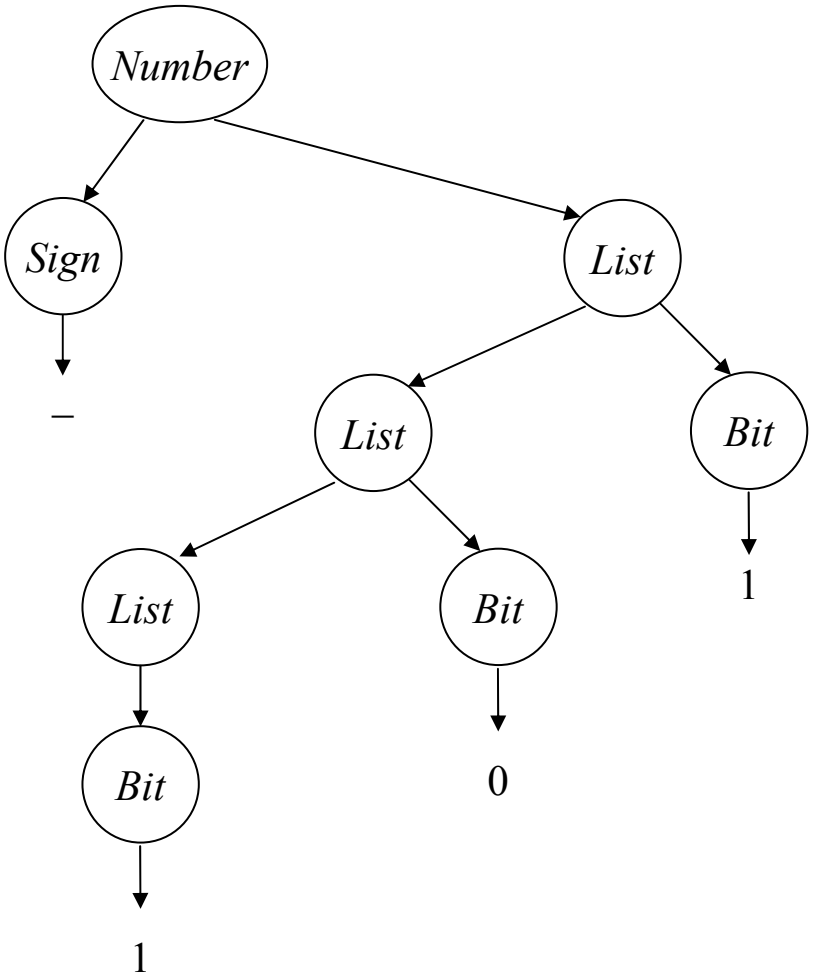
- nodes in the parse tree : 각각에 attribute 결합
- AG rules : assignment for a CFG production
- attributes : local information
- **parse tree + AG rules** → an attribute dependence graph
  - must be non-circular !

- **synthesized attributes** (downward)
  - depends on values from children & constants
- **inherited attributes** (upward)
  - depends on values from parents & siblings

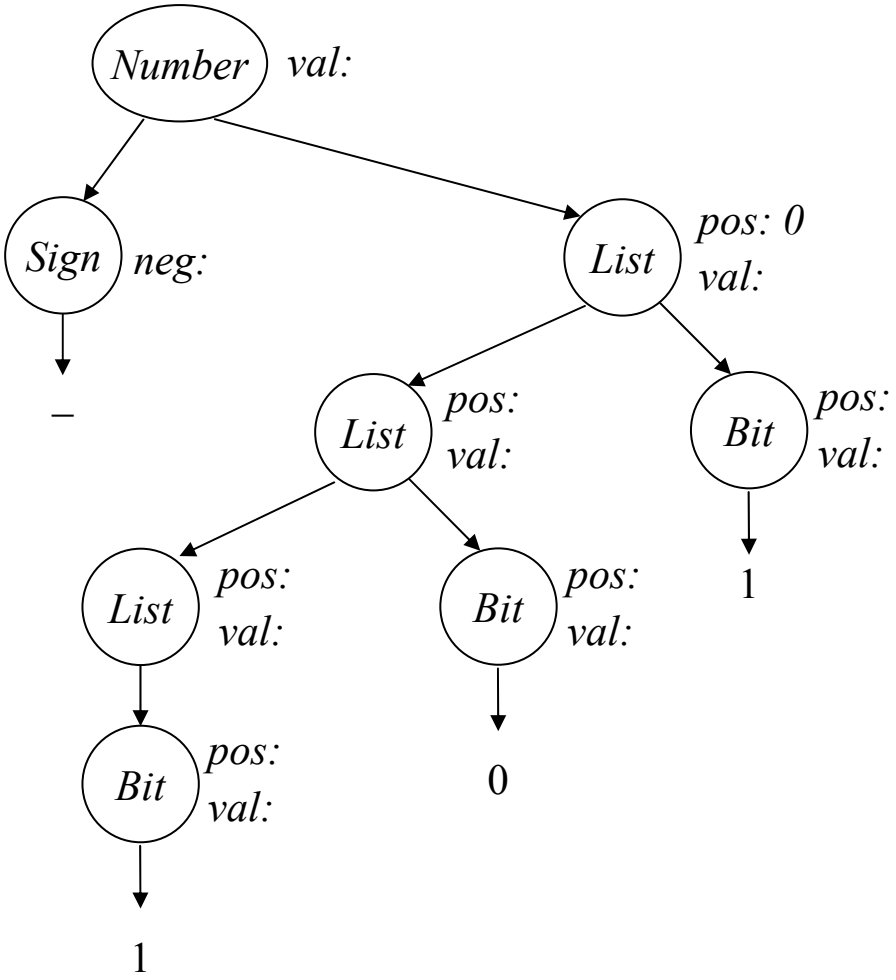
Attribute Evaluation

- 최종 목표 : 모든 **attribute value**를 계산
  - 문제점 : synthesized attribute 와 inherited attribute의 혼합 → evaluation order를 어떻게 해야 하나?
  - 보통, parse tree를 다 만든 후, 별도로 적용
- dynamic methods
  - topological sort the dependence graph
  - evaluate attributes in topological order
- oblivious methods (multi-pass method)
  - 아무 순서나 하나 정한 후, fixed-point iteration
- rule-based methods
  - compile time에 미리 evaluation 순서를 결정

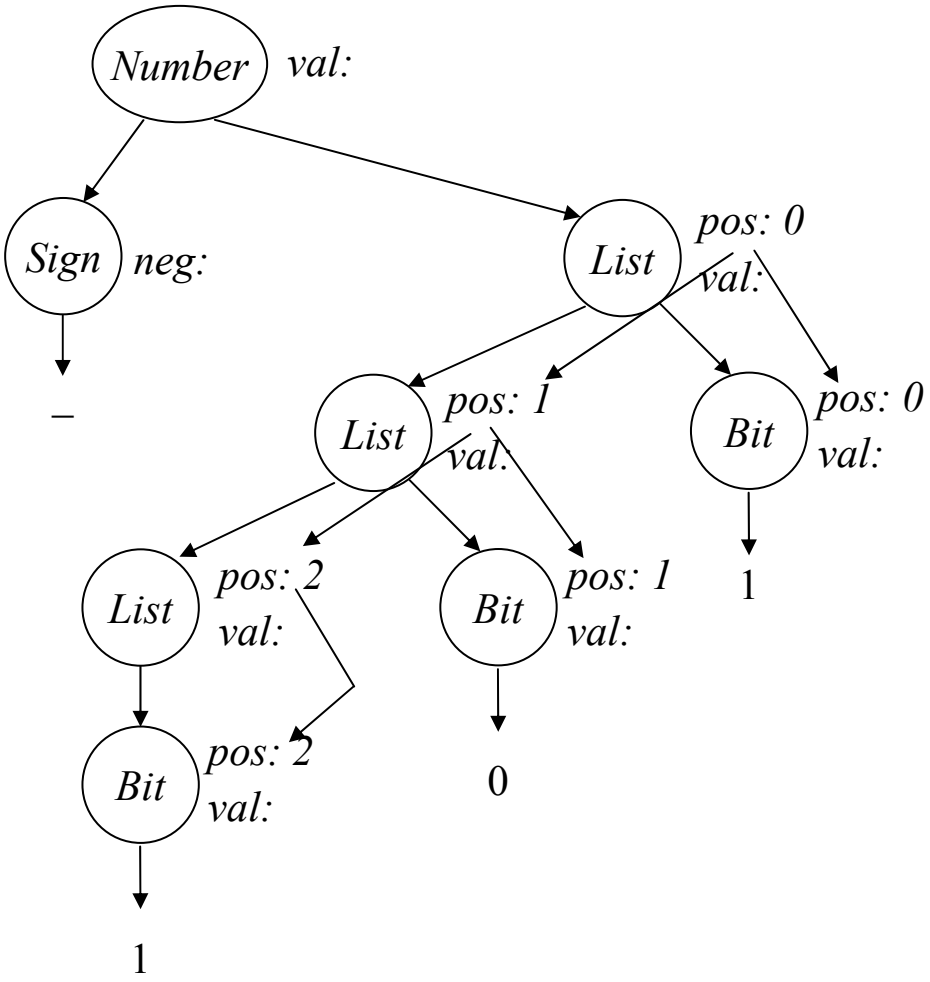
Example: for "–101"



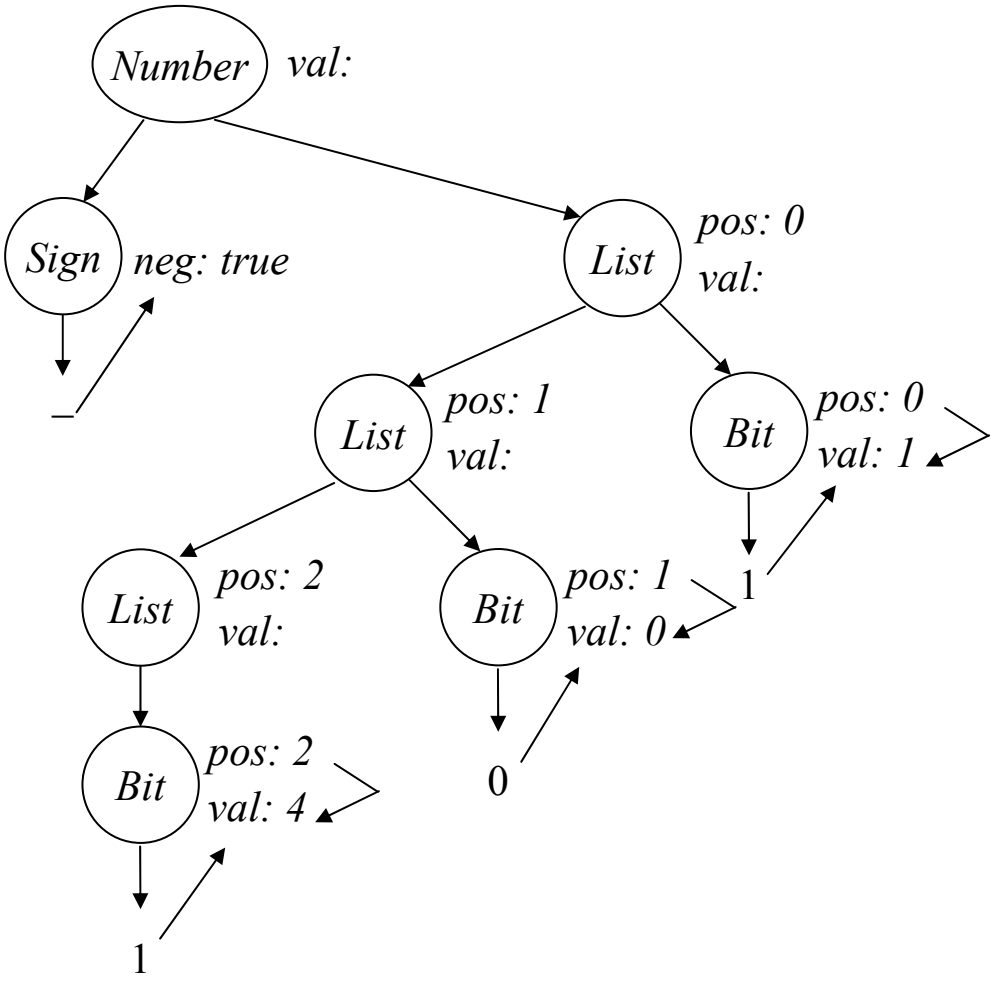
Example: constants



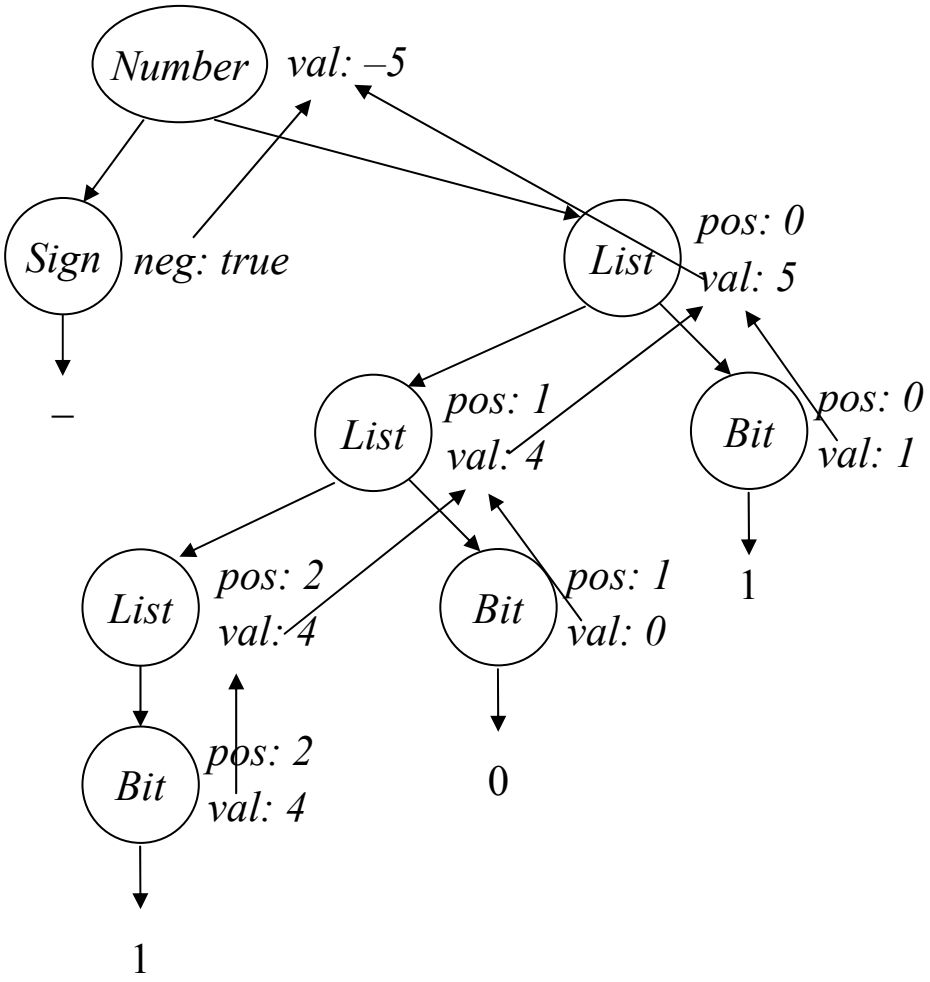
Example: inherited attributes



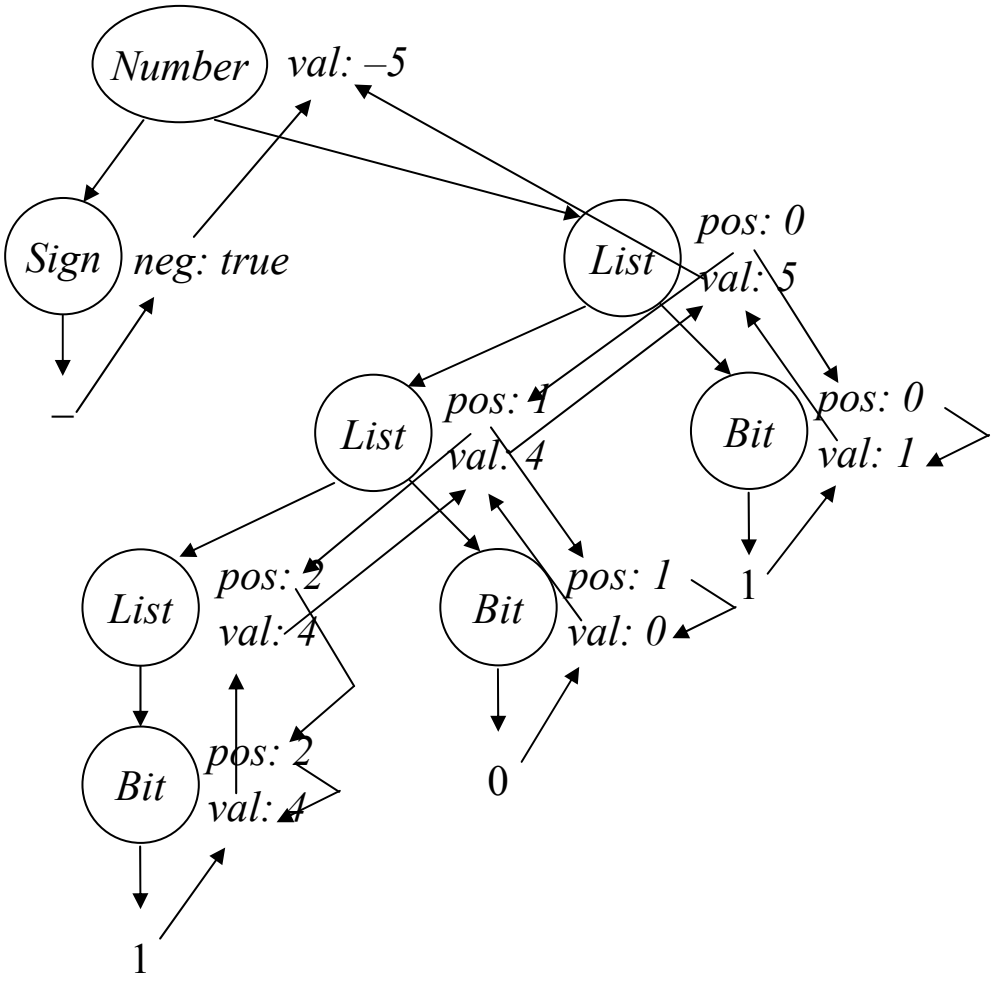
Example: synthesized attributes



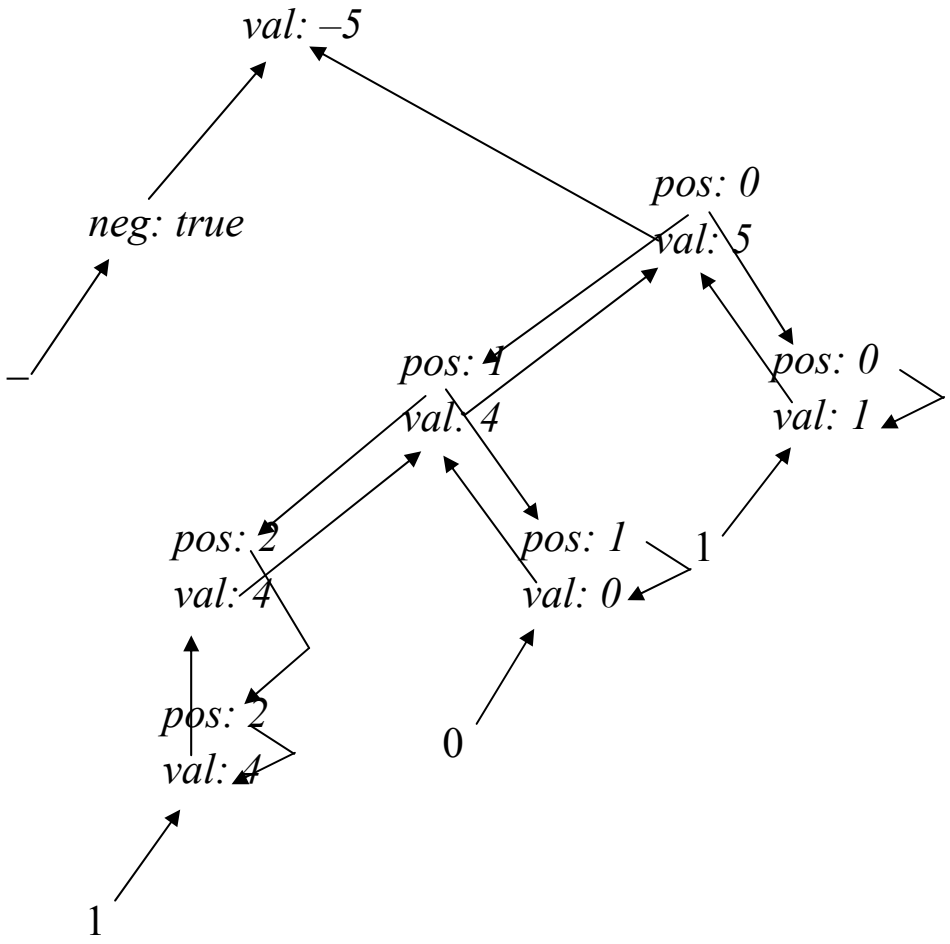
Example: more synthesized attributes



Example: overall sequence



Example: dependence graph



must be acyclic !

Circularity

- dependence graph에 cycle이 생길 수 있다
  - 해결책 ?
- avoidance
  - AG rule들을 수정해서, cycle을 제거
  - 예: synthesized attribute만 사용하면, acyclic 보장
- evaluation
  - fixed point iteration again…
  - 값이 변하지 않을 때까지 계속 evaluation 반복

4.4 Ad Hoc Syntax-Directed Translation

Realist's Alternative

- Attribute Grammar
  - 이론적으로는 완벽
  - 그런데, 정말 이렇게 할 필요가 있나?
- ad hoc approach
  - 기본 아이디어: CFG에서 parsing 되면, 바로 값 계산
  - **simplification on attributes**
    - single value for each CFG symbol
      - AG 에서는 여러 개도 가능
    - one-directional value flow
      - synthesized attribute만 사용

Yacc Approach

- LR(1) stack의 변형: 각 symbol의 현재 값도 stack에 저장
  - <*symbol*, *state*> 저장 → **<*value*, *symbol*, *state*>** 저장
    - value가 복잡하면, pointer만 저장
- simple naming convention
  - symbol 별로 value 이름을 쓰기 힘들다.
  - for a CFG production: $A \rightarrow \beta_1 \beta_2 \beta_3 \ldots \beta_n$
  - $$: LHS value
  - $1, $2, $3, …, $*n* : RHS values
- LR(1) parsing 중에, reduce action 마다
  - synthesized attribute 계산을 추가할 수 있음

Example: signed binary number again

- Yacc's approach

	production	syntax-directed actions
1	*Number* → *Sign List*	$$ ← $1 * $2
2	*Sign* → +	$$ ← 1
3	*Sign* → –	$$ ← –1
4	*List* → *Bit*	$$ ← $1
5	$List_0 \rightarrow List_1\ Bit$	$$ ← 2 * $1 + $2
6	*Bit* → 0	$$ ← 0
7	*Bit* → 1	$$ ← 1

Example: expression evaluation

- more practical example

Goal	→	*Expr*	$$ = $1;
Expr	→	*Expr* + *Term*	$$ = Add($1, $3);
	|	*Expr* – *Term*	$$ = Subtract($1, $3);
	|	*Term*	$$ = $1;
Term	→	*Term* * *Factor*	$$ = Multiply($1, $3);
	|	*Term* / *Factor*	$$ = Divide($1, $3);
	|	*Factor*	$$ = $1;
Factor	→	(*Expr*)	$$ = $1;
	|	number	$$ = GetValue(token);
	|	id	$$ = GetValue(token);

Actions at Other Points

- Ad-hoc approach의 한계
  - CFG의 reduce action에서만 code 실행 가능
- shift action 에서 code를 실행 하려면?
  - *Factor* → (*Expr*)
    - *Factor* → (*Expr*) • normal case
    - *Factor* → (• *Expr*) 어떻게 할 것인가?
- 해결책: CFG의 개선 (새로운 NT의 도입)
  - *Factor* → *Prefix* • *Expr*)
  - *Prefix* → (• 여기서 처리 !

Reality

- 현재 상황: 대부분의 parser가 ad-hoc style을 사용
- 장점: efficient, flexible
  - attribute grammar의 단점을 극복
- 단점: must hand-written
  - automatic이 아니므로, program으로 만들기 힘듦
  - 결국, programmer가 직접 detail까지 작성해야
- Most parser generators support a yacc-like notation

Typical Scenario

- Building a symbol table
  - declaration syntax 완료 → symbol table에 정보 입력
  - symbol table 정보는 이후 semantics check에 사용
- Error checking/type checking
  - define before use rule
    → variable 사용 시, symbol table 참조
  - variable dimension, type :
    symbol table에서 바로 check
  - expression의 type check : bottom-up 방식으로 수행
- Procedure interfaces are harder
  - parameter list, type : 미리 symbol table에 저장
  - call 할 때 마다 check 해야

Comparison

- Attribute Grammars
  - Pros: formal, powerful, can deal with propagation strategies
  - Cons: works on parse tree
- Ad-hoc methods
  - Pros: simple and functional,
    can be specified in grammar (Yacc),
    but does not require parse tree
  - Cons: rigid evaluation order, no context inheritance

4.5 Advanced Topics

Declaration-free Languages

- 일부 programming language는 type을 선언하지 않음
  - 왜? 편리할 수 있음
  - 예: BASIC, Perl, ML, …
- 실제로는 각 variable의 type이 계속 변화
  - x = y * 3.14159; y를 floating number로 해석
  - z = a[y]; y를 integer로 해석
  - printf(y); y를 string으로 해석
- 극단적인 경우
  - 수식을 계산할 때, type을 맞추기 위해, fixed-point iteration 필요 !

Last Comment

- Why context-sensitive analysis ?
  - type checking
  - parse tree generation
    - reduce action 마다 tree 생성 / 수정 가능
  - interpreter language 구현
    - reduce action 마다 실제 값을 계산
    - 계산기 프로그램: interpreter로 구현 가능