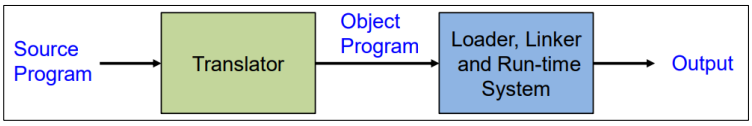
[Compiler Construction]

Why compilers?

compiler: 어떤 프로그래밍 언어로 작성된 코드를 다른 프로그래밍 언어(혹은 같은 언어의 다른 버전)로 바꾸는 tool. 컴파일러는 source program의 semantic을 다 알고, target language의 효율적인 버전을 만들어야함.

초기 컴퓨터는 machine language만 있었는데 점점 발전함에 따라 assembly, HLL이 등장. low-level language만 사용하기에는 machine structure가 complex해지고 sw 관리가 어려워짐.

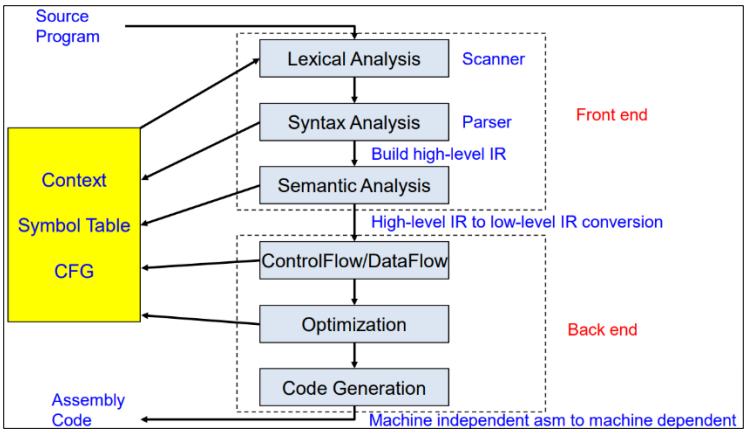
Compiler structure



source program이 translator를 거쳐 object program이 되고, object program에 library 등을 붙여 executable한 output binary로 만듦.

compile time (static time): compile을 하기 전에 행동을 취함. (ex. positioning of variables)
run time (dynamic time): 실제로 프로그램을 구동 시킬 때 작업을 수행함. (ex. SP values, heap)

General structure of a modern compiler



front end

: source program의 logic을 이해하고 자체적인 자료구조로 저장함. 예를들어 C 코드가 input으로 들어오면 C의 문법적인 부분을 없애고 코드가 궁극적으로 하려는 것이 무엇인지를 이해. 3가지 analysis를 통해 자신만의 assembly code로 분석.

back end

: 어떤 프로그램을 target hardware에 맞는 코드(target machine에 맞는, 실제 구동시킬 수 있는 코드)로 만듦. 효율적인 코드를 만들기 위해 optimization이 필수.

Lexical analysis (scanner)

source stream에서 가장 낮은 level의 lexical elements를 detect하고 추출하는 작업. text로 작성된 프로그램을 하나씩 읽어서(scan) 단어를 추출하고, 그 단어가 reserved words(특별한 행동을 하도록 지정된 특수 단어들. for, if, switch 등)인지, identifiers(변수)인지, 숫자인지 symbol인지 찾아냄. 공백이나 주석같은 non-grammatical element를 없애고 FSA를 이용해 구현됨.

Lex/Flex

각 source program에 따라 다른 lexical analysis를 수행해야함. 자바 프로그램과 c 프로그램은 문법이 서로 다르기 때문에 language마다 다른 형태의 lexical analysis가 필요한 것. 같은 언어라도 버전이나 사용 library에 따라 조금씩 다른 analysis가 필요.

regular expression의 specification을 받아서 table driven FSA를 만듦. Lex/Flex의 output은 c 프로그램인데, 그 프로그램을 컴파일하면 scanner 프로그램이 만들어짐.

Parser

input token stream이 문법적으로 올바른지 판단하고(문법적으로 허용되지 않는 단어를 제외시킴), 올바르다면 그 구조를 자료구조로 저장하는 작업을 수행함.

Static semantic analysis

compile time에 문법을 제대로 체크해야함.

ex) *i = i+1;* 은 문법적으로 맞는 것 같지만, 이전에 i가 define되지 않았다면 틀린 문장.

즉, identifier의 definition를 체크하거나 usage가 올바른지 확인해야하고, overloaded된 operator의 ambiguity를 없애야함.

semantic analysis까지 통과하면 이 프로그램은 문법적으로 틀리지 않은 프로그램. 컴파일러가 가지고 있는 assembly의 형태로 자료구조에 저장하게 됨.

Backend

우리가 이해한 프로그램을 output executable binary로 만드는 과정. 기본적으로 machine independent assembly가 backend process의 input이 됨.

사람이 작성하는 코드는 기본적으로 비효율적이므로 코드 최적화를 시행하고, 그 코드를 실제 하드웨어와 맵핑하는 작업을 수행.

*machine independent: CPU 같은 processing unit은 computation을 할 수 있는 resource와 data를 저장할 수 있는 register가 필요함. target CPU마다 ALU의 개수나 register의 개수가 다르기 때문에 output program이 달라지게 됨. 이런 상황을 피하기 위해 resource와 register의 개수가 무한대라고 가정하는 것이 machine independent. 기본적으로 3-address code(A=B+C처럼 3개의 operand를가지고 하나의 action을 취하는 형태)

Dataflow and control flow analysis

프로그램이 어떻게 흘러가는지 알기 위해서는 각 instruction들의 dependencies와 변수들이 어디서 생기고 어디서 사용되는지에 대한 분석이 필요. 분석에 따라 코드를 변경하고, 변경된 코드가 원래 코드와 동일한지 판단해야함.

dataflow: 변수에 저장된 data들 간의 관계, 어떤 instruction들이 어떤 값을 만들어내고 사용하는지, data dependency를 체크함.

control flow: control statement (if 등으로 발생하는 jump), execution flow가 변했을 때 실행되는 instruction이 어떻게 변하는지 분석, control dependency를 체크함.

Optimization

코드를 빠르게 혹은 energy efficient하게 구동시킬 수 있는 방법을 적용.

classical optimizations: 필요없는 코드를 지우고 중복되는 코드를 수정하는 등 코드 자체를 최적화하는 기본적인 최적화 방식.

machine independent: 대부분의 아키텍처에 적용할 수 있는 최적화 기법.

machine dependent: 각각의 프로세서 아키텍처마다 적합한 최적화.

Code generation

machine independent한 코드(virtual)를 target architecture에 맞는 코드(physical)로 맵핑.

instruction selection: 컴파일러가 가지고 있는 assembly의 operation을 실제 ARM이나 intel 하드웨어에서 대응되는 instruction으로 맞춰줌

register allocation: virtual register를 N개의 physical register로 맵핑함.

scheduling: instruction들의 순서를 dependency에 맞춰서 바꿈

assembly emission

output으로 만들어진 machine assembly를 가지고 assembler와 linker가 binary로 바꿈.

Why are compilers important?

computer architecture

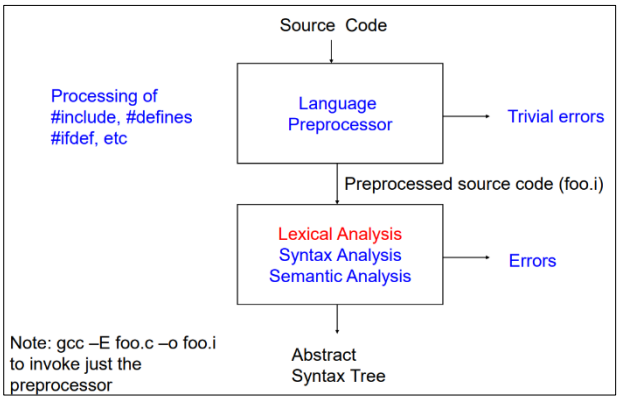
: OS와 하드웨어를 연결함. 예를 들어, intel CPU를 사용하다가 새로운 intel CPU가 나왔을 때 이전에 없던 instruction이 추가되었다면, 새로 추가된 instruction을 사용할 수 있도록 compiler가 변경되어야함. CPU의 버전이 바뀌었을 때 예전에는 잘 되던 최적화가 안될 수도 있는데, 거기에 맞춰 또 다른 최적화를 수행할 수 있도록 해야함.

software developers

: compiler가 코드를 어떻게 변경하는지 알면 프로그램을 효율적으로 짤 수 있음. 컴파일러가 어떠한 최적화를 수행할 때 최적화가 적용되기 위한 코드구조가 있을 수 있기 때문에, 정해진 코드 구조에 맞게 프로그램을 짜야하는 경우도 있음(그렇지 않으면 최적화가 적용되지 않는 문제 발생 가능).

[Lexical analysis 1]

Frontend structure



source code -> IR로 된 machine independent 한 assembly -> abstract syntax tree에 저장

analysis에 들어가기 전에 preprocessor를 거쳐서 preprocess를 먼저 처리함.

Lexical analysis: Specification, Recognition, and Automation

specification: 어떤 단어들을 추출할 것인지 견적을 냄. -> regular expression 이용

recognition: lexical pattern을 인식하는 방법. -> DFA 이용

ex. match0을 변수로, 512를 숫자로 인식할 수 있어야함.

automation: specification으로부터 string recognizer를 자동으로 generate

-> Thompson's construction, subset construction 이용

Lexical analysis process

if (b == 0) a = b; 코드가 있을 때 lexical analysis를 통과해 우리가 원하는 string을 뽑아냄. multi-character로 구성된 input stream을 token stream으로 바꾸는 것. 이렇게 하면 공백이나 주석이 사라지고 필요한 단어들을 token으로 인식해서 저장해 프로그램의 길이를 줄일 수 있음

syntax analysis: token stream을 이용해서 문법적으로 틀렸는지 맞았는지 확인하고 token들의 관계를 저장하는 것

Tokens

HLL에 공통적인 부분도 있고, 아닌 부분도 있음. token들은 정규표현식으로 HLL마다 다른 specification이 정해짐.

How to describe tokens

token은 정규표현식을 이용해서 표현함.

- a: ordinary character stands for itself
- ε: empty string
- $R|S$: either R or S, where R,S = RE
- RS: R followed by S (concatenation)
- R^*: concatenation of R (0 or more times)
- R^+: $R^* - \varepsilon$

Language

L(R): regular expression R로 만들어진 모든 string의 set

인식하려는 모든 token들을 지정할 수 있는 RE의 set을 만들고 language로 묶으면 우리가 원하는 lexer의 specification이 될 수 있음

RE notational shorthand

- R?: optional R: (R|ε)
- [abcd]: one of listed characters: (a|b|c|d)
- [a-z]: one character from range: (a|b|c|...|z)
- [^ab]: anything but one of the listed characters
- [^a-z]: one character not from this range

Example regular expression

How to break up text

text에서 string을 찾아낼 때 string들이 multiple match가 되는 경우가 있기 때문에 RE만으로는 충분하다고 볼 수 없음.

elsex = 0;이 있을 때

1) else x = 0; (else: keyword)
2) elsex = 0; (elsex: identifier)

2가지로 인식될 수 있기 때문에, 둘 중 무엇을 선택해야할지에 대한 rule이 필요함.

-> longest matching token wins: 둘 중 더 긴 token을 선택. 동일한 길이라면 priority를 고려함.

Automatic generation of lexers

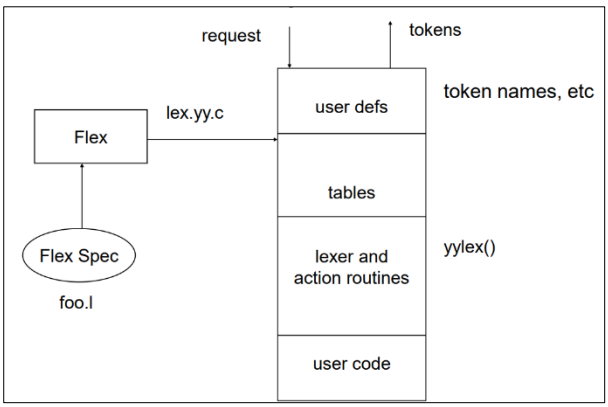
Lex/Flex: lexer를 자동으로 만들어주는 프로그램

Yacc/bison: syntax analysis를 할 때 필요

input -> longest token first matching을 기본 rule로해서 각 RE마다 관련된 associated action을 정해줌.

output -> input stream을 받아 RE에 의해 token으로 break up하는 프로그램

Lex/Flex



Flex 실행

-> lex.yy.c 생성

-(컴파일)-> lexer

* lexer: request(input stream)가 들어오면 token을 뽑아내는 프로그램

Lex specification

specification - priority가 지정된 RE의 모임. lex file은 총 3개의 section으로 나눠짐.

- definition section: 우리가 원하는 pattern과 state를 define함
- rules section: 우리가 원하는 lexical pattern과 semantic actions(의미적으로 필요한 action)에 대한 정보를 만들어줌
- user function section: 필요한 function을 만듦

Partial flex program

```
D               [0-9]
%%
if              printf ("IF statement\n");
[a-z]+          printf ("tag, value %s\n", yytext);
{D}+            printf ("decimal number %s\n", yytext);
"++"            printf ("unary op\n");
"+"             printf ("binary op\n");
```

pattern 지정 / pattern에 따라 필요한 action 지정

Flex program

flex count.l (flex에 count.l를 넣어서 실행)

-(generate)-> lex.yy.c

-(compile)-> lexical analyzer

Another flex program

RE를 통해 원하는 action을 지정해주면 이 파일을 input으로 넣어 원하는 lexical analyzer를 만들 수 있음

Lex regular expressions meta chars

- **.** match any single char (except ₩n ?)
- ***** Kleene closure (0 or more)
- **[]** Match any character within brackets
  – in first position matches, **^** in first position inverts set
- **^** matches beginning of line
- **$** matches end of line
- **{a, b}** match count of preceding pattern from a to b times, b optional
- **₩** escape for metacharacters
- **+** positive closure (1 or more)
- **?** matches 0 or 1 REs
- **|** alteration
- **/** provides lookahead
- **()** grouping of RE
- **<>** restricts pattern to matching only in that state

How does lex work

기본적으로 FSA를 이용해서 시행함. RE를 이용해 regular set을 찾고, FSA를 통해 regular sets를 인식함.

Two kinds of FSA

NFA: 같은 상황이라도 다양한 경우의 수를 가지고 있음. multiple transition 또는 empty transition 가능.

DFA: 같은 상황에서는 항상 같은 결과가 나오도록 정해져있고, empty transition은 불가능함.

NFA example

transition을 하는 중에 Error로 가는 상황을 lexical analysis 단계에서 찾음.

4번이나 5번 state에서 transition이 끝났다면 우리가 원하는 string을 찾은 것.

DFA example

DFA의 조건

- state transition에서 하나의 input string에 대한 transition option은 하나만 존재함
- upsilon transition(empty string)은 허용되지 않음

NFA vs DFA

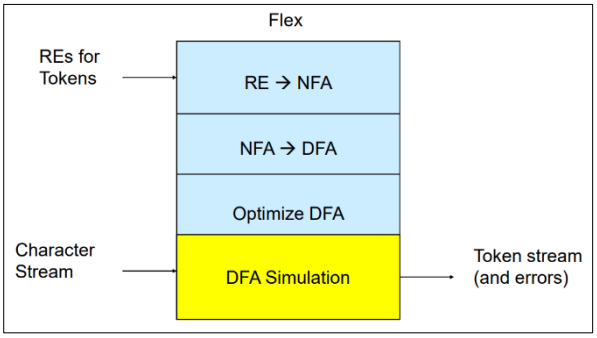
NFA	DFA
- 각 state에서 하나의 input에 대해 여러 개의 transition choice가 존재할 수 있음 - accepting state로 가는 path가 하나라도 있으면 그 string은 accept됨 - table-driven approach가 불가능해 프로그램을 작성하기에 not obvious	- 각 input에 따라 해야 할 action(transition)이 하나만 있음 - table-driven approach가 가능함 - RE를 finite automata로 만들 때 더 많은 state가 필요

NFA는 그때그때 어떠한 state로 가야 제대로 동작할 것인지를 알기 어려움.
token을 검출할 때는 state transition table을 만들어 table-driven approach로 simulate하기 위해 DFA가 적합함

[Lexical analysis 2]

How does lex work

각 language에 맞는 spec(RE)이 input으로 들어오면 FLEX를 구동시켜 DFA simulation을 할 수 있는 C code를 output으로 만듦



NFA는 예측할 수 없는 경우가 많아 실제 프로그램으로 만들이 어렵기 때문에 DFA로 변경.

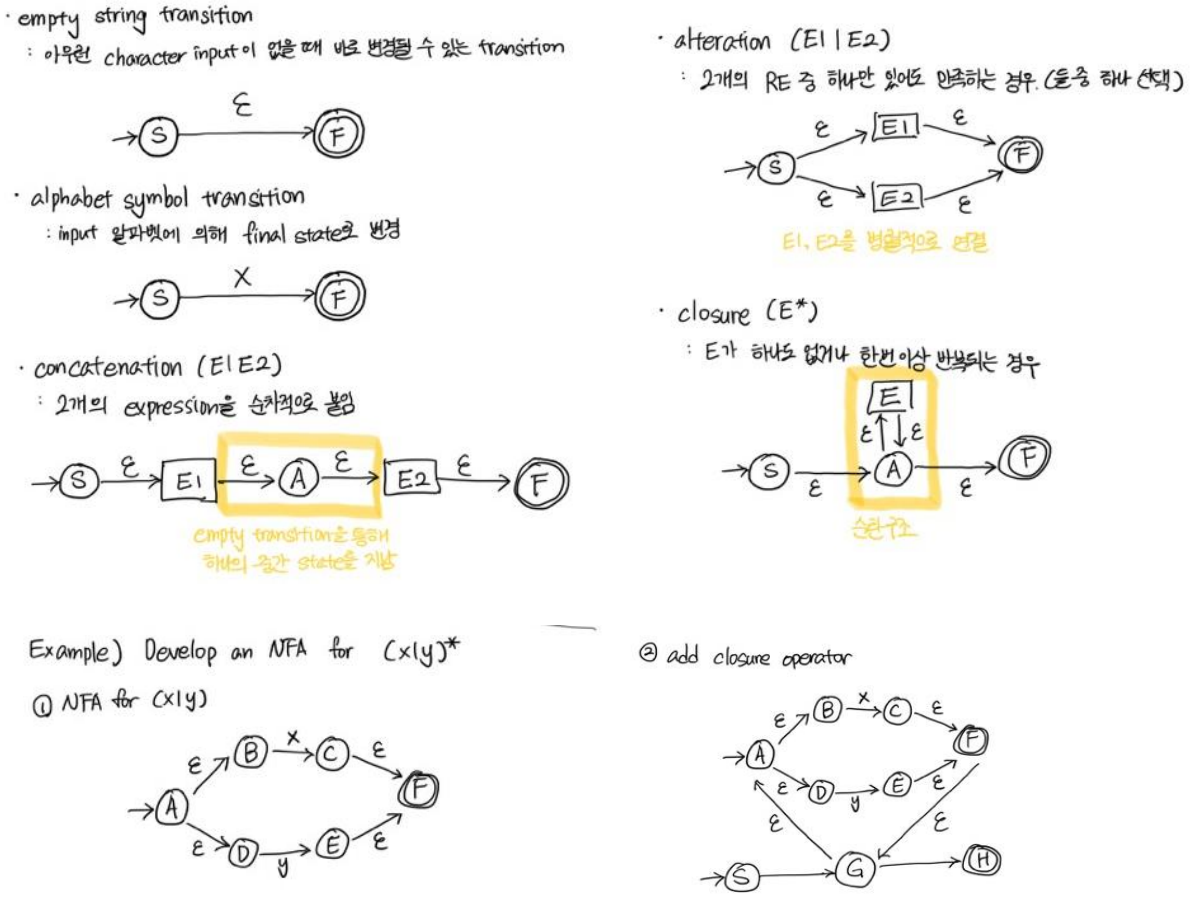
DFA로 바꾸면 state수가 많아지고 복잡해지므로 최적화.

최종적으로 만들어진 프로그램: DFA를 이용해서 lookup table을 구축하고, DFA를 simulation할 수 있도록 프로그램을 만들어줌. character stream에서 문자를 하나씩 받으며 DFA simulation을 통해 token stream이나 lexical error를 output으로 내보냄.

Regular expression to NFA

Thompson's construction algorithm: NFA를 귀납적으로 만드는 알고리즘. 조그만 NFA를 만들어서 연결하고 키워서 최종 NFA를 만듦. 각각의 기본적인 RE마다 NFA를 만드는 rule을 만들고 기본 expression으로 만들어진 NFA들을 이미 정해진 rule에 따라 합쳐서 최종 NFA를 만드는 것.

Thompson construction



NFA to DFA

NFA에만 있고, DFA에는 없는 특징을 지워야함

1. multiple outgoing edges to same input (multiple transition)

   solve by subset construction

   - 하나의 input에서 갈 수 있는 state가 여러 개일 때, 그 state들을 모두 포함하는 하나의 큰 state(subset state)를 만들어 새로운 state transition graph를 build하고, 원래의 final state가 포함된 state들을 final state로 지정.
   - 위의 방법으로 바꾸고 나면 하나의 input에 대해 중복된 transition이 사라짐.
   - state의 수가 늘어날 수 있음. 그렇게 되면 state transition table에서 row 또는 column이 늘어나서 table의 크기가 커지기 때문에 나중에 state의 개수를 줄이는 최적화가 중요.

2. empty transitions

   ε closure: ε transition으로 도달할 수 있는 모든 state.

   ε closure를 하나의 큰 state로 만들어 새로운 DFA를 생성.

State minimization

NFA를 DFA로 바꿨을 때, state의 수가 늘어날 수 있음.

redundant하거나 equivalent한 state들이 많은데, 이 의미는 메모리 접근이 자주 발생하고 사용하는 메모리가 많다는 것. state minimization을 통해서 크기를 줄여야함.

equivalent states의 그룹을 찾고 merge해서 minimize함

Simulating the DFA

각 state에서 input에 따라 transition이 발생하는 것을 table로 만들어주고, table lookup을 이용해 실제 프로그램을 수행함.

```
trans_table[NSTATES][NINPUTS];
accept_states[NSTATES];
state = INITIAL;

while (state != ERROR) {
  c = input.read();
  if (c == EOF) break;
  state = trans_table[state][c];
}
return accept_states[state];
```

trans_table: 현재 state와 input을 넣어 다음 state를 찾을 수 있음

accept_state: final state가 될 수 있는 state

1) INITIAL state에서 시작 -> while문 안의 코드 동작으로 구동
2) character stream(input)으로부터 character를 하나 읽음
3) trans_table에서 [state][c]를 이용해 next state를 찾음
4) character가 더 이상 없을 때까지 반복
5) ERROR없이 accept state에 도달하면 제대로 된 token

Handling multiple REs

RE 여러 개를 각각의 NFA로 만들고 combine해서 single NFA로 만듦.

그 후에 NFA를 DFA로 변환, DFA minimization 수행.

[Syntax analysis 1_1]

Where is syntax analysis performed?

lexical analysis의 output으로 나온 token stream을 input으로 받아 syntax analysis 수행.

token들이 문법에 맞게 stream이 이루어져있는지 검사하고, abstract syntax tree를 만들어줌

Parsing analogy

parsing == 언어를 분석함

- 문장이 문법적으로 맞는지 분석
- 각 word(token)의 역할을 지정
- token들의 관계를 지정

Syntax analysis overview

목표: input token stream이 프로그램의 syntax를 만족하는지 판단

analysis을 하기 위해 필요한 것

- syntax를 표현하는 방법 (spec)
- input token stream이 spec을 만족하는지 판단하는 메커니즘

Just use regular expressions?

RE만으로는 recursive한 부분을 해결할 수 없음

ex) 왼쪽 괄호와 오른쪽 괄호의 개수가 맞는지를 detect

Context-free grammars

4 components

- terminal symbols (token or ε)
- non-terminal symbols (syntactic variables)
- start symbol S (special non-terminal)
- productions of the form LHS->RHS
  * LHS (left hand side): single non-terminal / RHS (right hand side): string of terminals and non-terminals

L(G) = language generated by grammar G (set of strings of terminals)

CFG example

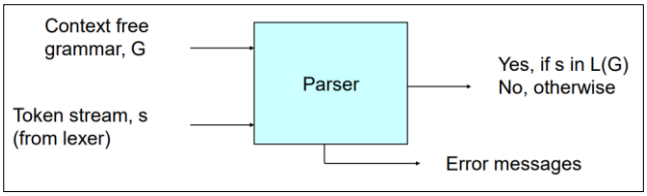
grammar for balanced-parentheses language: S -> (S)S | ε

"()(())"

➔ S => (S)S => ()S => ()(S)S => ()((S)S)S => ()(()S)S => ()(())S => ()(())

start symbol로부터 production rule을 반복적으로 수행해서 우리가 원하는 string이 도출된다면 (derivation) 그 string은 문법적으로 문제가 없는 것

Parser



input: CFG로 지정된 spec + token stream from lexer

output: S로부터 derive되면 Yes, 아니면 No | Error

RE is a subset of CFG

- ε S -> ε
- a S -> a
- R1R2 S -> S1S2
- R1 | R2 S -> S1 | S2
- R1* S -> S1S1 | ε

G1 = grammar for R1 with start symbol S1, G2 = grammar for R2 with start symbol S2

Grammar for sum expression

S -> E+S | E

E -> number | (S)

: 4 productions, 2 non-terminals (S,E), 4 terminals ("(", ")", "+", number), start symbol S

Constructing a derivation

S에서 시작해서 production rule을 반복적으로 적용해 token을 derive

arbitrary strings α, β, γ, production A -> β

single step of derivation: $\alpha A \gamma$ -> $\alpha\beta\gamma$ (substitute β for A)

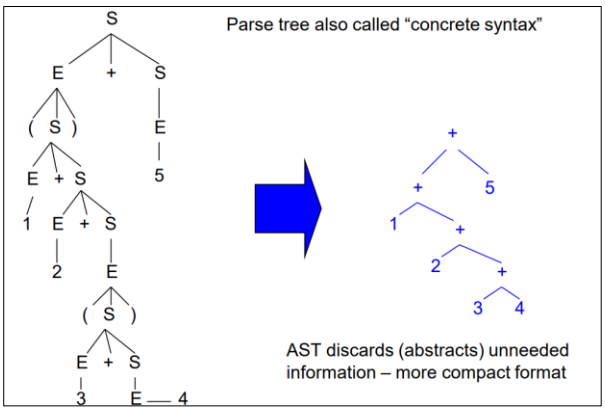
Parse tree

derivation 과정을 tree로 나타낸 것.

Leave는 terminal, internal node는 non-terminal

derivation step의 순서에 대한 정보는 tree에 저장되지 않음

Parse tree vs Abstract syntax tree



AST: parse tree에서 중요하지 않은 것들을 지우고 만들어진 tree

Derivation order

순서를 정해서 production rule을 적용하면 조금 더 체계적으로 derivation 할 수 있음

- Leftmost derivation: leftmost non-terminal에 production 적용
  ex) *E* + S -> *1* + S
- Rightmost derivation: rightmost non-terminal에 적용
  ex) E + *S* -> E + *E + S*

derivation order에 관계없이 결과는 같아야함

Ambiguous grammar

leftmost derivation과 rightmost derivation의 결과가 다른 grammar

+ operator (S -> E + S)는 right-recursive production이기 때문에 derivation order에 관계없이 identical parse tree가 만들어짐

S -> S+S | S*S | number

위 grammar는 expand되는 순서에 따라 derivation 결과가 달라져 different parse tree가 되는 ambiguous grammar임. -> 수정 필요

ex) 1 + 2 * 3

1) S -> S+S -> 1+S -> 1+S*S -> 1+2*S -> 1+2*3 (7)
2) S -> S*S -> S+S*S -> 1+S*S -> 1+2*S -> 1+2*3 (9)

parse tree에 따라 evaluation이 달라지기 때문에 잘못된 grammar임

ambiguity는 language가 아닌 grammar의 문제이므로 CFG를 잘 지정하는 것이 중요.

Eliminating ambiguity

nonterminal을 추가하고, recursion이 왼쪽 혹은 오른쪽 한 쪽으로만 일어날 수 있도록 수정함.

- S -> S+T | T //왼쪽으로만 expand되는 left associative grammar
- T -> T*num | num //T를 통해 곱셈에 우선순위를 부여함

Closer look at eliminating ambiguity

- 연산자들의 우선순위 level에 따라 각각에 맞는 non-terminal 생성
- level에 따른 production rule 생성
- 높은 우선순위 operator가 그 다음 순위 non-terminal을 참조하도록함

Associativity

operator는 left, right or non-associative할 수 있음

left: a + b + c = (a + b) + c

right: a ^ b ^ c = a ^ (b ^ c)

non: a < b < c is illegal

production rule 생성시 left (right) recursion하게 만들면 left (right) associativity가 부여됨

non-associativity의 경우 recursive하지 않고 both sides of operator에서 다음 우선순위의 non-terminal을 참조하도록 만들면 됨

[Syntax analysis 1_2]

Parsing top-down

input token stream을 하나씩 읽어가며 주어진 CFG를 이용해 leftmost derivation을 수행함. stream을 끝까지 무리없이 읽고 derivation을 수행할 수 있다면 해당 stream은 문법적으로 오류가 없다고 판단.

top-down parser는 다른 parser에 비해 coverage가 작음

S -> E+S | E

E -> num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→E + S	(	(1+2+(3+4))+5
→(S) + S	1	(1+2+(3+4))+5
→(E+S)+S	1	(1+2+(3+4))+5
→(1+S)+S	2	(1+2+(3+4))+5
→(1+E+S)+S	2	(1+2+(3+4))+5
→(1+2+S)+S	2	(1+2+(3+4))+5
→(1+2+E)+S	(	(1+2+(3+4))+5
→(1+2+(S))+S	3	(1+2+(3+4))+5
→(1+2+(E+S))+S	3	(1+2+(3+4))+5
→ **…**		

Problem with top-down parsing

다음 symbol에 대해 어떤 production을 수행해야하는지 결정하는 문제

Ex1: "(1)"	S → E → (S) → (E) → (1)
Ex2: "(1)+2"	S → E+S → (S)+S → (E)+S → (1)+E → (1)+2

lookahead는 Ex1, 2 모두 "("로 동일하지만, 1은 E로, 2는 E+S로 apply해야함

Grammar is problem

grammar의 spec이 잘못된 것이 아님. spec은 잘 맞추고 있지만, top-down parser에서 제대로 동작할 수 없는 형태인 것.

top-down parsing이 올바르게 수행되려면 LL(1) grammar여야하므로 주어진 grammar를 LL(1)으로 변경해야함.

Making a grammar LL(1)

S -> E+S | E

E -> num | (S)

Problem

: S에서 E+S로 갈 수도 있고, E로 갈 수 있음. E 다음이 +S인지 아닌지는 판단할 수 없음

Sol1) Left-factoring: E+S | E 에서 겹치는 부분인 E를 묶고 나머지는 새로운 nonterminal S′로 만듦.

S′ -> ε | + S rule을 추가

Sol2) left recursion을 right recursion으로 변경

S -> E+S | E

E -> num | (S)

=>

S -> ES′

S′ -> ε | + S

E -> num | (S)

Parsing with new grammar

S -> ES′ S′ -> ε | + S E -> num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→ES'	(	(1+2+(3+4))+5
→(S)S'	1	(1+2+(3+4))+5
→(ES')S'	1	(1+2+(3+4))+5
→(1S')S'	+	(1+2+(3+4))+5
→(1+ES')S'	2	(1+2+(3+4))+5
→(1+2S')S'	+	(1+2+(3+4))+5
→(1+2+S)S'	(	(1+2+(3+4))+5
→(1+2+ES')S'	(	(1+2+(3+4))+5
→(1+2+(S)S')S'	3	(1+2+(3+4))+5
→(1+2+(ES')S')S'	3	(1+2+(3+4))+5
→(1+2+(3S')S')S'	+	(1+2+(3+4))+5
→(1+2+(3+E)S')S'	4	(1+2+(3+4))+5
→ ...		

Predictive parsing

LL(1) grammar

- 어떤 non-terminal이 있을 때, 하나의 lookahead symbol이 apply할 수 있는 production은 unique함
- top-down parsing은 predictive parsing
- predictive parsing table을 이용해서 수행될 수 있음 (nonterminal x terminal -> production)

Parsing with table

S -> ES′ S′ -> ε | + S E -> num | (S)

non-terminal: S, S′, E

terminal: num, +, (,), $

* $는 end of file을 나타내는 special "endmarker"

	num	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-descent parser

table은 recursive descent parser로 쉽게 convert되고, 그걸 이용해서 parsing을 수행하면 됨

3 procedures: parse_S(), parse_S′(), and parse_E()

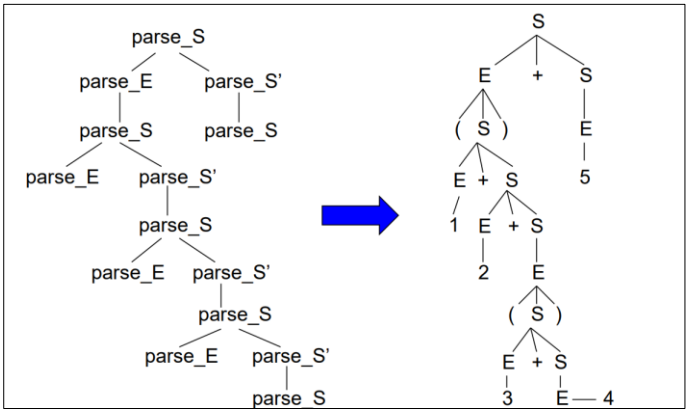
```
void parse_S() {
   switch (token) {
      case num:
            parse_E();
            parse_S′();
            return;
      case '(':
            parse_E();
            parse_S′();
            return;
      default:
            ParseError();
   }
}
```

```
void parse_S′() {
   switch (token) {
      case '+':
            token = input.read();
            parse_S();
            return;
      case ')':
            return;
      case EOF:
            return;
      default:
            ParseError();
   }
}
```

```
void parse_E() {
   switch(token) {
      case num:
            token = input.read();
            return;
      case '(':
            token = input.read();
            parse_S();
            token = input.read();
            if (token != ')')
               ParseError();
            return;
      default:
            ParseError();
   }
}
```

각 non-terminal마다 parsing table에서 production이 존재하는 terminal과 matching

Call tree (== Parse tree)



parsing function을 recursive하게 부르는 call tree가 결국 parse tree와 동일함.
실제 code를 generation할 때 parse tree에 맞게 코드를 만들면 됨.

How to construct parsing tables

grammar로부터 predictive parsing table을 자동으로 만들 수 있는 알고리즘이 필요함.
각 nonterminal에 맞는 production rule을 수행할 가능성이 있는 lookahead symbol을 찾는 것이 중요

Constructing parse tables

constructing predictive parser의 조건
: 모든 non-terminal에 대해, 모든 lookahead symbol은 최대 1개의 production으로 handle할 수 있어야함. (0개는 error를 의미, 2개는 존재할 수 없음)

FIRST(β): β의 fully expanded version에서 맨 앞에 나올 수 있는 symbols의 set
FOLLOW(X): input stream에서 X의 derivation 직후에 나올 수 있는 symbols의 set
* β: arbitrary string of terminals and non-terminals,
X: non-terminal

Parse table entries

production X -> β가 있을 때,
row X 중 FIRST(β)에 속한 symbol에 -> β를 추가
만약, β가 nullable이라면 (ε를 derive 할 수 있다면) FOLLOW(X)에 속한 symbol의 row에 ->β추가
=> conflicting entries가 없다면 주어진 grammar는 LL(1) grammar임

example)

S -> ES'	FIRST(S)는 S가 fully expand 됐을 때 맨 앞에 올 수 있는 symbols
S' -> ε | +S	-> FIRST(S) = {num, '('}
E -> num | (S)	=> row S 중 num과 '('에 ES'를 추가

Computing nullable

어떤 non-terminal X에 대해 X가 empty string을 derive할 수 있으면 nullable

- derives ε directly (X -> ε)
- has a production X -> YZ... where all RHS symbols are nullable

nullable을 확인하는 알고리즘: 처음에는 모든 non-terminals이 non-nullable이라고 가정한 다음, 더 이상 change가 없을 때까지 production rule을 repeatedly apply함.

Computing FIRST

1. X가 terminal이면 FIRST(X)에 X 추가
2. X -> ε 이면 FIRST(X)에 ε 추가
3. X가 nonterminal이고, X -> Y1Y2 ... Yk일 때,
   FIRST(Yj) (j=1, ..., i-1)에 ε이 포함되어있고, FIRST(Yi)에 a가 있다면 FIRST(X)에 a 추가
4. FIRST(Y1Y2 ... Yk)에 ε이 있다면 FIRST(X)에도 ε이 포함됨

example)

S -> ES'

S' -> ε | +S

E -> num | (S)

apply rule 1: FIRST(num) = {num}, FIRST(+) = {+}, FIRST(( ) = { (}, FIRST( )) = {) }

apply rule 2: FIRST(ε) = {ε}

apply rule 3: FIRST(S) = FIRST(E) = { }

FIRST(S') = FIRST(ε) + FIRST(+) = {ε, +}

FIRST(E) = FIRST(num) + FIRST(( ) = {num, (}

rule 3 again => FIRST(S) = FIRST(E) = {num, (}

FIRST(S') = {ε, +}

FIRST(E) = {num, (}

Computing FOLLOW

1. S가 start symbol이면 FOLLOW(S)에 $ 추가
2. A -> αBβ이면 ε이 아닌 모든 FIRST(β)를 FOLLOW(S)에 추가
3. A -> αB이거나 FIRST(β)에 αBβ 와 ε이 포함된다면 FOLLOW(B)에 FOLLOW(A) 추가

example)

S -> ES′ FIRST(S) = {num, (}

S′ -> ε | +S FIRST(S′) = {ε, +}

E -> num | (S) FIRST(E) = {num, (}

apply rule 1: FOL(S) = {$}

apply rule 2: S -> ES′ FOL(E) += {FIRST(S′) - ε} = {+}

S′ -> ε | +S –

E -> num | (S) FOL(S) += {FIRST(')') - ε} = {$,) }

apply rule 3: S -> ES′ FOL(E) += FOL(S) = {+, $,) }

FOL(S′) += FOL(S) = {$,) }

Putting it all together

FIRST(S) = {num, (} FOLLOW(S) = {$,) }

FIRST(S′) = { ε, +} FOLLOW(S′) = {$,) }

FIRST(E) = {num, (} FOLLOW(E) = {+,), $}

- X -> β production이 있을 때
- X row에서 FIRST(β)에 포함된 symbol에 ->β를 추가
- β가 nullable이라면, FOLLOW(X)에 ->β 추가

	num	+	(	)	$
S	ES’		ES’		
S’		+S		ε	ε
E	num		(S)		

Ambiguous grammars

parse table 상에서 하나의 cell에 2개 이상의 production rule이 존재하는 경우

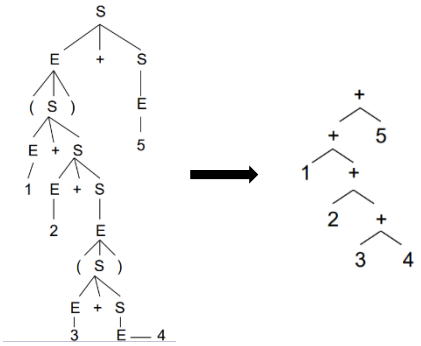
S -> S+S | S*S | num

FIRST(S+S) = FIRST(S*S) = FIRST(num) = {num}

=> row S의 num cell에 S+S, S*S, num이 모두 들어갈 수 있음

Creating the Abstract Syntax Tree (AST)

parse tree에서 불필요한 부분을 제외해서 AST를 만듦



Left vs Right associativity

"1+2+3+4"

Right recursion: S -> E+S | E (1+(2+(3+4)))

E -> num

Left recursion: S -> S+E | E (((1+2)+3)+4)

E -> num

Left-recursive grammars

left-recursive grammar를 parsing할 때 recursion을 언제 끝내야 할 지 알 수 없음

-> top-down parser에서 제대로 동작하지 않음 (LL(1) grammar가 아님)

Eliminate left recursion

새로운 nonterminal X′을 만들고 알파와 베타의 순서를 바꿈

기존 grammar:

X -> Xα1 | ... | Xαm

X -> β1 | ... | βn

변경된 grammar:

X -> β1X′ | ... | βnX′

X′ -> α1X′ | ... | αmX′ | ε

Creating an LL(1) grammar (summary)

Start with a left-recursive grammar

S -> S+E | E

apply left-recursion elimination algorithm

S -> ES′

S′ -> +ES′ | ε

Start with right-recursive grammar

S -> E+S | E

apply left-factoring to eliminate common prefixes

S -> ES′

S′ -> +S | ε

[Syntax analysis 2_1]

Bottom-Up parsing

top-down parsing보다 좀 더 powerful한 parsing technology

LR grammar를 이용해서 parsing 수행

- left-recursive grammars (대부분의 programming language는 left-recursive이기 때문에 coverage가 높음)

Shift-reduce parser

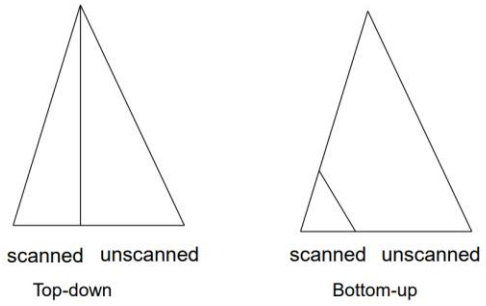
Right-most derivation

- token에서 시작해서 production rule을 반대로 수행해 start symbol로 끝남 (backward)
- RHS of production -> replace by LHS
- E->num 이라는 production rule이 있을 때 num을 E로 바꾸는 방식

(1+2+(3+4))+5 ← (E+2+(3+4))+5
← (S+2+(3+4))+5 ← (S+E+(3+4))+5
← (S+(3+4))+5 ← (S+(E+4))+5 ← (S+(S+4))+5
← (S+(S+E))+5 ← (S+(S))+5 ← (S+E)+5 ← (S)+5
← E+5 ← S+E ← S

장점: (1+2+(3+4))+5가 있을 때 1을 보고 +, 2까지 본 다음에 production rule(S+E)을 선택할 수 있음. (== postpone the selection of productions until more of the input is scanned)

Top-Down vs Bottom-Up



Terminology: LL vs LR

token stream을 왼쪽부터 보는 것은 동일하나, derivation 방향이 다름.

LL(k)	LR(k)
- Left-to-right scan of input - Left-most derivation - k symbol lookahead - [Top-down or predictive] parsing or LL parser - performs pre-order traversal of parse tree	- Left-to-right scan of input - Right-most derivation - k symbol lookahead - [Bottom-up or shift-reduce] parsing or LR parser - performs post-order traversal of parse tree

Shift-reduce parsing

shift와 reduce를 통해서 parsing 수행

현재의 parser state(stack of terminals and non-terminals. grows to the right)를 가지고 있어야함.

shift/reduce를 사용해 stack에 terminal과 non-terminal을 넣었다 뺐다 반복하며 parsing을 수행하다가, stack에 unconsumed input이 없고 S(start symbol)가 나오면 parsing 성공

Derivation step	stack	Unconsumed input
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(E+2+(3+4))+5 ←	(E	+2+(3+4))+5
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5
(S+E+(3+4))+5 ←	(S+E	+(3+4))+5
...		

shift: 지금까지 보지 않았던 token들 중 가장 왼쪽의 token(lookahead token)을 stack에 넣음

ex) shift 1: stack에 1을 넣음

reduce: stack의 맨 위에서부터 어떤 symbol β가 있을 때, 베타를 non-terminal symbol X (corresponding to the production X ->β)로 바꿈.

ex) reduce S -> S+E: stack의 맨 위에서부터 S+E를 검출해서 S+E를 빼고 S를 넣음

Example) S -> S+E | E

E -> num | (S) <- left-recursive한 CFG

derivation	stack	input stream	action
(1+2+(3+4))+5		(1+2+(3+4))+5	shift
(1+2+(3+4))+5	(	1+2+(3+4))+5	shift
(1+2+(3+4))+5	(1	+2+(3+4))+5	reduce E→ num
(E+2+(3+4))+5	(E	+2+(3+4))+5	reduce S→ E
(S+2+(3+4))+5	(S	+2+(3+4))+5	shift
(S+2+(3+4))+5	(S+	2+(3+4))+5	shift
(S+2+(3+4))+5	(S+2	+(3+4))+5	reduce E→ num
(S+E+(3+4))+5	(S+E	+(3+4))+5	reduce S → S+E
(S+(3+4))+5	(S	+(3+4))+5	shift
(S+(3+4))+5	(S+	(3+4))+5	shift
(S+(3+4))+5	(S+(	3+4))+5	shift
(S+(3+4))+5	(S+(3	+4))+5	reduce E→ num
...			

Potential problems

shift와 reduce 중 어떤 production을 apply할지 결정해야함

또는, reduce rule이 여러 개가 있을 때 어떤 rule을 적용해야할지 결정

Action selection problem

stack에 β가 있고, look-ahead symbol이 b일 때 가능한 선택지

- Shift b onto the stack (making βb)
- Reduce X->γ assuming that the stack has the form $\beta = \alpha\gamma$ (making αX)

stack에 $\alpha\gamma$가 있을 때, stack prefix α에 depending해 reduction X-> γ를 수행해야하는가

- γ의 length에 따라 α가 달라지기 때문에 주의해야함

LR parsing engine

Basic mechanism

- use a set of parser states
- use stack with alternating symbols and states
- use parsing table to:

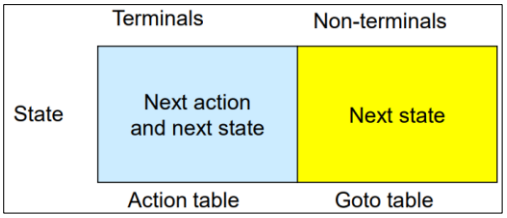
  Determine what action to apply (shift/reduce)

  Determine next state

* parser state: 현재 stack의 상태. (stack에 많은 non-terminal과 terminal이 들어가있을 때 현재의 parser state를 지정해주면 거기에 맞춰 action을 취해 제대로 parsing을 할 수 있음). symbol과 state를 동시에 넣음으로써 현재 stack의 상황을 알려줌

The parser actions can be precisely determined from the table

LR parsing table



Algorithm: look at entry for current state S and input terminal C

- If Table[S,C] = s(S') then shift:

  push(C), push(S')
- If Table[S,C] = X->α then reduce:

  pop(2*|α|), S' = top(), push(X), push(Table[S',X])

  -> α에 있는 non-terminal과 state를 함께 빼야하기 때문에 pop(2*|α|)

LR parsing table example

State	Input terminal					Non-terminals	
	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

LR(k) grammars

Left-to-right scanning, Right-most derivation, k lookahead characters

Building LR(0) parsing tables

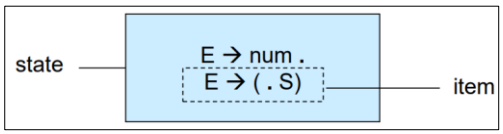
Parsing table build

- Define states of the parser
- Build a DFA to describe transitions between states
- Use the DFA to build the parsing table

Each LR(0) state is a set of LR(0) items

- LR(0) item: X → $\alpha.\beta$ where X → $\alpha\beta$ is a production in the grammar
- LR(0) items keep track of the progress on all of the possible upcoming productions
- The item X → $\alpha.\beta$ abstracts the fact that the parser already matched the string α at the top of the stack

Example LR(0) state



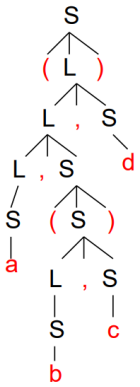
- Substring before "." is already on the stack
  -> beginnings of possible γ's to be reduced
- Substring after "." is what we might see next

LR(0) grammar

Nested lists S -> (L) | id

L -> S | L,S

Parse tree for (a, (b,c), d)



Start state and closure

Start state

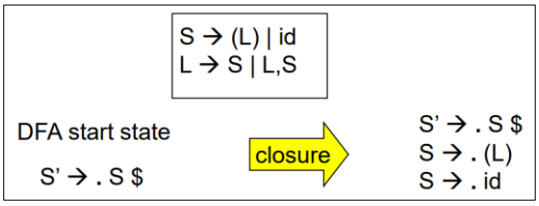
- Augment grammar with production: S' -> S$
- Start state of DFA has empty stack: S' -> .S$

Closure of a parser state

- Start with Closure(S) = S
- Then for each item in S
  - X → $\alpha.Y\beta$
  - Add items for all the productions Y → γ to the closure of S: Y → $.\gamma$

(어떠한 S가 있을 때 그 state를 closure에 넣고, 가능한 모든 production을 넣음)

Closure



어떤 state가 다음 번에 reduce될 수 있는 가능한 모든 production의 set

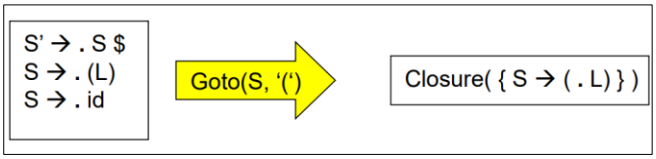
- Closure of a parser state S:
  - Start with Closure(S) = S
  - Then for each item in S:

    X → $\alpha.Y\beta$

    Add items for all the productions Y → γ to the closure of S: Y → $.\gamma$

The goto operation

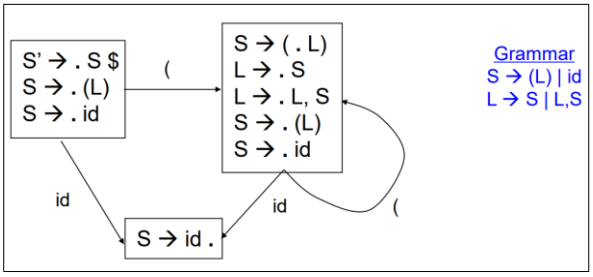
describes transitions between parser states, which are sets of items



Algorithm: for state S and a symbol Y

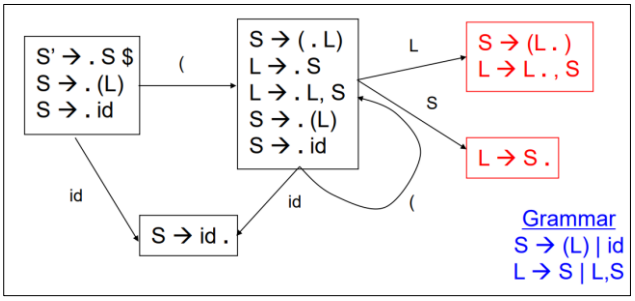
- If the item [X → $\alpha.Y\beta$] is in I, then
- Goto(I, y) = Closure([X → $\alpha Y.\beta$])

Goto: Terminal symbols



dot 바로 뒤에 input symbol이 있는 item을 포함해서 new state에서도 closure를 전부 찾음

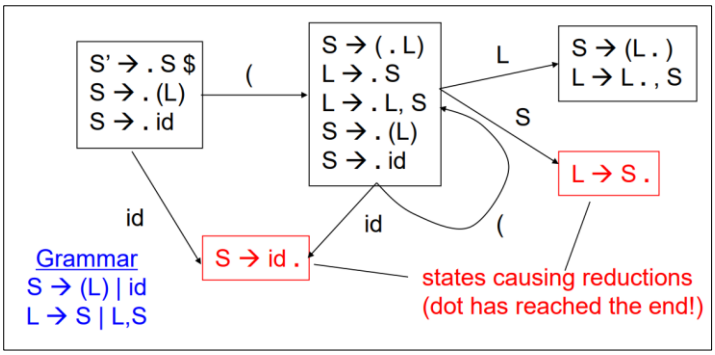
Goto: Non-terminal symbols



non-terminal에 대해서도 동일한 algorithm으로 transition.

stack에 들어갈 look ahead symbol은 항상 terminal인데, reduce action은 stack에서 문자열을 빼서 하나의 non-terminal을 만들고 nonterminal과 top을 봐서 next state를 만듦 => nonterminal symbol도 input이 될 수 있음

Applying reduce actions



reduce: stack의 맨 위에서 몇 개의 terminal과 non-terminal을 보고 가능한 production rule을 찾아서 반대로 적용해 새로운 input non-terminal을 만들어냄

reduce action 수행을 위해서는 stack에 어떠한 production rule의 RHS가 모두 stack 맨 위에 있어야함. (“.”이 가장 오른쪽에 있어야함)

Reductions

On reducing X → β with stack $\alpha\beta$

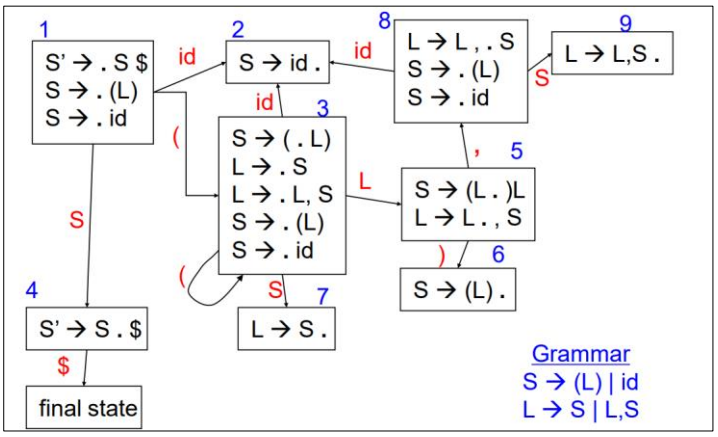
- Pop β off stack, revealing prefix α and state
- Take single step in DFA from top state
- Push X onto stack with new DFA state

(X와 현재 state로 DFA를 한 번 돌려서 새로운 DFA state를 지정하고 stack에 push)

Example)

derivation	stack	input	action
((a),b) ←	1 (3 (3	a),b)	shift, goto 2
((a),b) ←	1 (3 (3 a 2	),b)	reduce S → id
((S),b) ←	1 (3 (3 S 7	),b)	reduce L → S

Full DFA and Parsing example ((a),b)



derivation	stack	input	action
((a),b) ←	1	((a),b)	shift, goto 3
((a),b) ←	1(3	(a),b)	shift, goto 3
((a),b) ←	1(3(3	a),b)	shift, goto 2
((a),b) ←	1(3(3a2	),b)	reduce S→id
((S),b) ←	1(3(3S7	),b)	reduce L→S
((L),b) ←	1(3(3L5	),b)	shift, goto 6
((L),b) ←	1(3(3L5)6	,b)	reduce S→(L)
(S,b) ←	1(3S7	,b)	reduce L→S
(L,b) ←	1(3L5	,b)	shift, goto 8
(L,b) ←	1(3L5,8	b)	shift, goto 9
(L,b) ←	1(3L5,8b2	)	reduce S→id
(L,S) ←	1(3L8,S9	)	reduce L→L,S
(L) ←	1(3L5	)	shift, goto 6
(L) ←	1(3L5)6		reduce S→(L)
S ←	1S4	$	done

Building the parsing table

- States in the table = states in the DFA
- For transition S -> S' on terminal C:

  Table[S,C] += Shift(S')
- For transition S -> S' on non-terminal N:

  Table[S,N] += Goto(S')
- If S is a reduction state X → β then:

  Table[S,*] += Reduce(X → β)

[Syntax Analysis 2_2]

LR(0) limitations

LR(0) machine은 reduce action을 수행하는 state에 하나의 reduce action만 지정된 경우 lookahead에 관계없이 동작함.

conflict: 어떤 state에서 lookahead가 있을 때 shift 혹은 reduce 옵션이 두 개라서 무엇을 선택할지 handle할 수 없는 경우 (shift/reduce or reduce/reduce)

➔ conflict를 해결하려면 lookahead symbol을 사용해야함

A non-LR(0) grammar

Grammar for addition of numbers

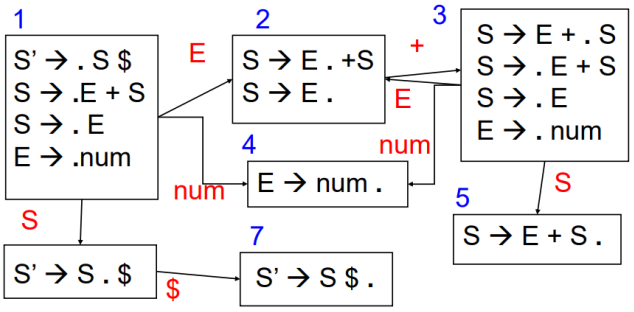
- S -> S+E | E
- E -> num

Left-associative version은 항상 LR(0)

Right-associative version

- S -> E+S | E
- E -> num

LR(0) parsing table



	num	+	$	E	S
1	s4			g2	g6
2	S→E	s3/S→E	S→E		

shift/reduce conflict

bottom-up parsing에서 conflict를 해결하려면 lookahead symbol 하나를 참조함

- SLR (Simple LR), LALR (LookAhead LR), LR(1)
- LR(1)은 가장 일반적이고 coverage가 높지만 state의 수가 늘어난다는 단점이 있음
- LALR 방법을 실제로 많이 사용함

SLR parsing

Easy extension of LR(0). LR(0)를 기반으로 간단하게 conflict를 없앰

- each reduction X → β에 대해 next symbol C를 확인
- C가 FOLLOW(X)에 포함되는 경우에만 reduction 수행

parsing table

- Adds reductions X → β only in the columns of symbols in FOLLOW(X)

  Ex) FOLLOW(S) = {$}

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		

FOLLOW(S)={$}이기 때문에 num과 +에서 S->E가 사라짐

SLR parsing table

reductions do not fill entire rows as before

	num	+	$	E	S
1	s4			g2	g6
2		s3	S→E		
3	s4			g2	g5
4		E→num	E→num		
5			S→E+S		
6			s7		
7			accept		

LR(1) parsing

recognized by a shift/reduce parser with 1 lookahead

- parser states = set of items
- LR(1) item = LR(0) item + lookahead symbol possibly following production

  LR(0) item: S -> .S+E

  LR(1) item: S -> .S+E**, +**

  현재 progress에다가, production 후에 존재할 수 있는 lookahead symbol을 추가
- Lookahead는 reduce operation에서만 impact를 가지고 (lookahead = next input일 때), shift에서는 사용되지 않음

LR(1) states

LR(1) state = set of LR(1) items

LR(1) item = (X → $\alpha.\beta$, y)

- α는 top of stack에 들어가있고, next expect to see는 β이며, 그 뒤에 y까지 올 것을 기대

Shorthand notation: (X → $\alpha.\beta$, {x1, ..., xn})

LR(1) closure

LR(1) closure operation

- Start with Closure(S) = S
- For each item in S:
  - (X → $\alpha.Y\beta$, z)
  - and for each production Y → γ, add the following item to the closure of S:

    Y → .γ, FIRST(βz)

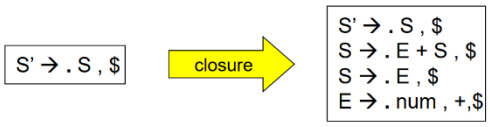
    "." 뒤에 있는 Y가 가능한 production rule이 있다면 추가함. Y-> γ production rule이 있으면, Y → .γ를 추가하고 FIRST(βz)도 추가. (Y 뒤에 나올 수 있는 FIRST가 중요하기 때문)

LR(1) start state

Start with (S' -> .S, $), then apply closure operation

Example) sum grammar

- S' -> S$
- S -> E+S | E
- E -> num

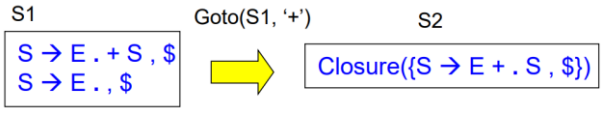


LR(1) goto operation

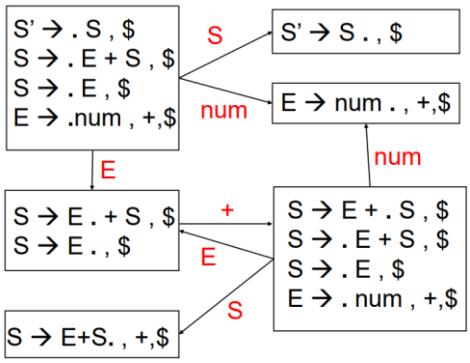
describes transitions between LR(1) states

Algorithm: for a state S and symbol Y

- If the item [X → $\alpha.Y\beta$] is in I, then
- Goto(I, Y) = Closure([X → $\alpha Y.\beta$])



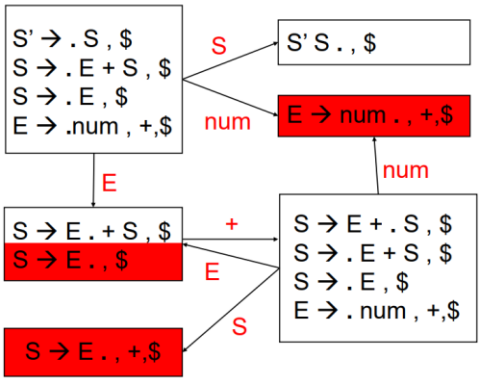
LR(1) DFA construction



Shift action은 lookahead symbol에 영향을 받지 않고, reduce action은 lookahead를 이용

LR(1) reductions

stack에 들어있는 item과, 현재 보고있는 token stream(lookahead symbol)을 보고 match될 때 reduce action 수행. 아래 그림에서 빨간색으로 marking된 것이 reductions



LR(1) parsing table construction

reduction을 제외하고는 LR(0)와 동일

- For a transition S -> S′ on terminal x:

  Table[S,x] += Shift(S′)

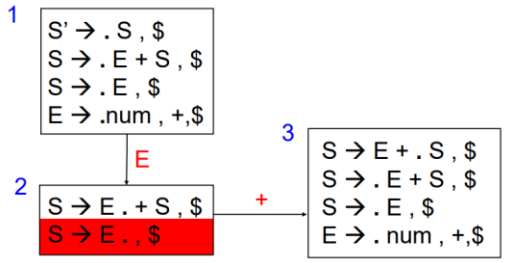
- For a transition S -> S′ on non-terminal N:

  Table[S,N] += Goto(S′)

- If I contains {(X → γ., y)} then:

  Table[I,y] += Reduce(X → γ)

LR(1) parsing table example



	+	$	E
1			g2
2	s3	S→E	

LR(0)로 하면 모든 lookahead에서 S->E production을 수행하도록 지정되어 있었기 때문에 state2에서 conflict가 발생.

LR(1) item은 reduction이 언제 어떠한 lookahead가 있을 때 specific하게 지정해줌

'+'일 때는 shift, '$'일 때는 reduction을 수행하도록해서 conflict가 제거됨

LALR(1) grammars

problem with LR(1): progress에 lookahead symbol까지 추가해야하기 때문에 state 수가 많아짐

LALR(1) parsing

- Constructs LR(1) DFA and then merge any 2 LR(1) states whose items are identical except lookahead
- Smaller parser tables (메모리 효율적으로 사용), theoretically less powerful than LR(1)

LALR parsers

LALR(1)

- 일반적으로 SLR과 동일한 state 개수 (much less than LR(1))
- LR(1)과 동일한 lookahead capability (much better than SLR)

LL/LR grammar summary

LL parsing tables (top-down parser)

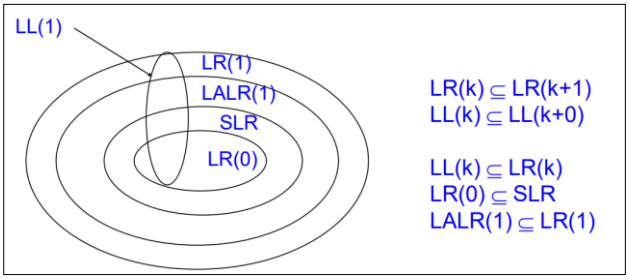
- Non-terminals x terminals -> productions
- Computed using FIRST/FOLLOW

LR parsing tables (bottom-up parser)

- LR states x terminals -> {shift/reduce}
- LR states x non-terminals -> goto
- Computed using closure/goto operations on LR states

어떤 grammar parsing table을 만들었을 때 conflict가 없는 경우 해당 grammar라고 할 수 있음

Classification of grammars



원의 범위는 coverage를 의미함

Automate the parsing process

Automate 할 수 있는 것

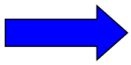
- The construction of LR parsing tables
- The construction of shift-reduce parsers based on these parsing tables

LALR(1) parser generators

- yacc, bison
- LR(1) in practice와 크게 다르지 않고, LR(1)에 비해 parsing table이 작음
- Augment LALR(1) grammar specification with declarations of precedence, associativity
- Output: LALR(1) parser program

Associativity

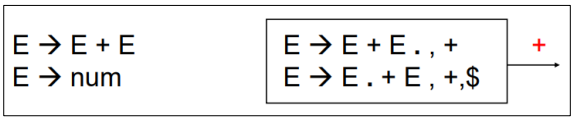
S → S + E | E
E → num



E → E + E
E → num

왼쪽 grammar는 left-recursive인데, 오른쪽은 어느쪽으로 recursive한지 알 수 없음

ambiguous grammar를 bottom-up parser(LALR parser)가 어떻게 해결할 수 있는가?



LALR parser를 만들었을 때 오른쪽처럼 만들어진다면 lookahead symbol이 "+"일 때 shift/reduce conflict가 발생함.

token stream: 1+2+3 //stack에 1+2까지 들어가있을 때 그 다음 +에서 shift/reduce conflict 발생

case1) shift: stack에 +를 추가 => 나중에 2+3을 먼저 계산하게 됨 (right-associative)

case2) reduce: 1+2를 먼저 계산하고 그 다음으로 넘어감 (left-associative)

우리가 원하는 associativity에 따라 shift와 reduce의 우선순위를 정하면 ambiguous를 해결할 수 있음.

- If an operator is left associative
  input stream보다 stack에 더 높은 precedence를 지정해 reduce가 priority를 갖도록 함
- If an operator is right associative
  stack보다 input stream에 더 높은 precedence를 지정해 shift가 priority를 갖도록 함

Precedence

E → E + E | T
T → T x T | num | (E)

E → E + E | E x E | num | (E)

오른쪽 grammar가 ambiguous한 이유: 덧셈과 곱셈 중 무엇을 먼저 수행해야하는지 알 수 없음

E → E . + E , ... E → E x E . , +	E → E + E . , x E → E . x E, ...	Shift/reduce conflict results

precedence: attach precedence indicators to terminals

input token과 parse stack의 마지막 terminal의 precedence를 비교해 shift/reduce를 결정

왼쪽 네모에서 input token이 "+"이고, stack에 있는게 'x'이므로 input token의 priority가 낮기 때문에 reduce를 먼저해서 곱셈을 먼저 수행함.