Chapter 1

Parsing

The lexer takes a string of characters as input and transforms it into a string of tokens. A parser takes the string of tokens as input, and produces a parse tree. Sometimes the parse tree is just implicit and is not actually built. Also, some compilers finish the two tasks in one phase.

1.1 Context free grammars

1.1.1 Introduction

Not all strings of tokens are valid programs, which calls for a language for describing valid strings of tokens as well as a method to distinguish valid strings of tokens from invalid ones.

Programming languages have recursive structures: an **expression** is often made up of other expressions. **Context free grammars** are a natural notation for such structure.

A context free grammar consists of

- a set of terminals (T)
- a set of non-terminals (N)
- a start symbol ($S \in N$)
- a set of productions (P)

A production is a relation of symbols:

$$X \to Y_1 Y_2 \dots Y_n \tag{1.1}$$

in which $X \in N$, and $Y_i \in T \cup N \cup \{\varepsilon\}$.

As an example, the language $\{(^i)^i\}, i = 0, \dots, N$ can be expressed by the CFG with $N = \{S\}, T = \{(,)\}$ and productions $\{S \to (S), S \to \varepsilon\}$.

Productions can be regarded as substitution rules. Terminals are so-called because there is no rule to replace them. Once generated, they are permanent. Terminals ought to be tokens of the programming language. Let G be a CFG with start symbol S. The language $L(G)$ of G is

$$\{a_1 \dots a_n | \forall a_i \in T, S \xrightarrow{*} a_1 \dots a_n\} \tag{1.2}$$

in which $S \xrightarrow{*} a_1 \dots a_n$ means that S can be rewritten into $a_1 \dots a_n$ in a few steps with the substitution rules defined by the productions.

In the definition of production, no precedence between operators or associativity of operator is assumed. For example, the grammar

$$E \to E + E | E - E | E * E | E/E | (E) | int \tag{1.3}$$

defines the normal $+ - \times \div$ operations of integers, but do not assume the precedence of $\times \div$ over $+-$, or the normal left associativity of these operators.

1.1.2 Derivations

With the help of CFG, we are able to figure out whether a string of tokens belongs to a language. However, we would also like to know the structure of the string, i.e. the parse tree, which calls for the help of derivations.

A sequence of productions is called a derivation. It can be drawn as a tree. The start symbol is the tree's root, and for each production $X \to Y_1 Y_2 \dots Y_n$, $Y_1 \dots Y_n$ is added as children of X. Such a tree drawn to describe an expression is called the **parse tree** of the expression.

A parse tree has terminals at the leaves and non-terminals at the interior nodes. An in-order traversal of the leaves is the original input. The parse tree shows the association of operations, while the input string does not.

According to the order of the non-terminals being replaced, a derivation can be left-most or right-most, or of a random order, which is rarely used. Both the left-most and the right-most derivation have the same parse tree.

1.1.3 Ambiguity

A grammar is **ambiguous** if it has more than one parse tree for some string. Equivantly, there is more than one left-most (right-most) derivation for this string. Ambiguity is a problem we strive to avoid. There are two solutions to the problem: either we directly rewrite the grammar unambiguously, or we enforce precedences on operations (e.g. * over +).

Here we give an example of rewriting grammar. The grammar

$$\mathrm{E} \to \mathrm{E} * \mathrm{E} | \mathrm{E} + \mathrm{E} | (\mathrm{E}) | id \tag{1.4}$$

is ambiguous. 1 + 2 + 3 can be generated by {1 + 2} + 3 or 1 + {2 + 3}. It can be written as

$$\begin{aligned} &\mathrm{E} \to \mathrm{E}' + \mathrm{E} \\ &\mathrm{E}' \to id * \mathrm{E}' | id | (\mathrm{E}) * \mathrm{E}' | (\mathrm{E}), \end{aligned} \tag{1.5}$$

which is no longer ambiguous. In the new grammar, E can generate sum(+), while E' can generate product(*). {1 + 2} + 3 is no longer legal in the new grammar.

Consider the grammar

$$\begin{aligned} \text{E} \rightarrow & \text{ if E then E} \\ & | \text{ if E then E else E} \\ & | \text{ OTHER} \end{aligned} \tag{1.6}$$

that decribes an "if then else" relation in which "else" is optional. It is ambiguous because the expression "if E_1 then if E_2 then E_3 else E_4" has two parse trees because the "else" could correspond to both "then"s. The parse tree of the previous expression could be either "if E_1 then {if E_2 then E_3 else E_4}" or "if E_1 then {if E_2 then E_3} else E_4"We want to rewrite the grammar so that every "else" matches the cloest "then". The new grammar is

$$\begin{aligned} \text{E} \rightarrow & \text{ MIF} \\ & | \text{ UIF} \\ \text{MIF} \rightarrow & \text{ if E then MIF else MIF} \\ & | \text{ OTHER} \\ \text{UIF} \rightarrow & \text{ if E then E} \\ & | \text{ if E then MIF else UIF} \end{aligned} \tag{1.7}$$

in which MIF means all "then"s are matched, while UIF means some "then"s are unmatched. In this new grammar, the second parse tree is no longer legal.

Unfortunately, it is impossible to automatically convert an ambiguous grammar to an unambiguous one. The manual conversion job is tedious, and the unambiguous grammar is often too complex to comprehend quickly. On the contrary, grammar with ambiguity is almost always tidy and more natural. But we need some mechanism to tackle the problem of ambiguity, which is the approach taken by most parsing tools. The most frequently used disambiguation mechanisms are precedence declarations and associativity declarations.

Note that the rewritten grammar no longer generates the same set of expressions from the point of view of semantic meaning. It only ensures that the same set of strings is generated. The grammar (1.3) can be rewritten as

$$\text{E} \rightarrow \text{E} + int | \text{E} - int | \text{E} * int | \text{E} / int | (E) | int \tag{1.8}$$

to remove the ambiguity. However, 3 * 5 + 2 - 6 / 2 will be parsed as and only as {{{3 * 5} + 2} - 6} / 2, which is against normal precedence and associativity assumptions. In order to make the grammar work correctly in the intended way, simply rewriting the grammar is far from enough. It must be combined with other measures such as precedence and associativity definitions.

1.2 Error handling

Compiler has two major tasks to complete: translating valid programs and detecting invalid ones. Different kinds of errors can be detected by different components of the compiler: lexcial errors by lexer, syntactic errors by parser and semantic errors by type checker. There are also errors that are not within the scope of the programming language, and thus remain to be found by the programmer through tests.

An error handler within the compiler is expected to

- report errors accurately and clearly;
- recover from an error quickly;
- not slow down the compilation of valid codes.

There are three different approaches to implement the error handler: panic mode, error productions and automatic local/global correction. The first two are used in current compilers.

1.2.1 Panic mode

Panic mode is the simplest and thus the most popular method. When an error is detected, it discards tokens until one with a clear role is found, and resumes the compilation there. The "clear" token it looks for is usually the terminator of a statement or a funtion. In Bison (a popular parser generator), a special terminal error is used to describe how much input to skip in case of an error. For example,

$$\mathrm{E} \to \mathrm{E} + \mathrm{E} | (\mathrm{E}) | \text{error int} | \text{error}$$

specifies that an integer is regarded as a token to resume the compilation when an error occurs.

1.2.2 Error productions

Common errors made by programmers can sometimes be predicted. For example, 5 * x is often mistakenly written as 5 x. We can add new productions covering such typos into the grammar so that the compilation can continue, and the programmer receives warnings concerning the errors. This approach has an obvious disadvantage that it compilates the grammar.

1.2.3 Automatic correction

In the early years of programming, compilation is a time-consuming process. Programmers had to spent hours or even a whole day waiting for the compilation result. So compilers were expected to automatically correct probable errors in the program so that the time would not be totally wasted due to small mistakes such as typos, and more errors could be found in a single compilation cycle.

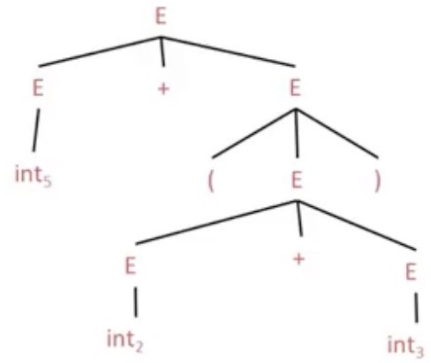

Figure 1.1: Parse tree of 5 + (2 + 3)

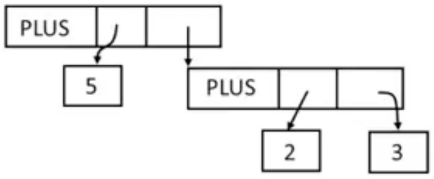

Figure 1.2: AST of 5 + (2 + 3)

This requirement makes it hard to implement a satisfactory compiler, and it also slows down the compilation of correct codes. Nowadays compilation is much more faster, and programmers tend to recompile everytime they fix an error. Thus complex error recovery becomes less compelling than a few decades ago.

1.3 Top down parsing

1.3.1 Abstract syntax tree

A parser traces the derivation of a sequence of tokens, but the rest of the compiler needs a structural representation of the program. **Abstract syntax tree**, or **AST** is the structure we use.

A parse tree traces operations of the parser and captures nesting structure of the token string being parsed. It contains a lot of verbose information, and the AST is an abstracted version that also captures the nesting structure, but is much more compact.

Consider the grammar E $\rightarrow$ int|(E)|E + E and the string 5 + (2 + 3). The parse tree and the related AST are shown in Figure 1.1 and Figure 1.2.

1.3.2 Recursive descent algorithm

Recursive descent parsing is a top-down parsing method, with which the parse tree is built from the top and from left to right. When building a parse tree, we start with the top-level non-terminal E, and try the rules of E in order. When another non-terminal is met, this process is carried out recursively. If a mis-match is found, we need to do the back tracking to find the correct rule to match.

Some rules in the illustration of the recursive descent parsing algorithm: TOKEN will be the type of tokens, and its instances will be INT (for int), OPEN (for '('), CLOSE (for ')'), TIMES (for '*'), etc; the global variable next will point to the next token in the input.

Some boolean functions need to be defined to check matches of token terminals and CFG rules. For a given token terminal tok, for the nth production of S, and for all productions of S, the functions are respectively

```
1  bool term(TOKEN tok) { return *next++ = tok; }
2  bool S_n() { ... }
3  bool S() { ... }
```

As an example, consider the grammar

$$\begin{aligned} &\mathrm{E} \rightarrow \mathrm{T} \mid \mathrm{T} + \mathrm{E} \\ &\mathrm{T} \rightarrow \mathrm{int} \mid \mathrm{int} * \mathrm{T} \mid (\mathrm{E}). \end{aligned} \tag{1.9}$$

We will have

```
1  bool E_1() { return T(); }
2  bool E_2() { return T() && term(PLUS) && E(); }
3  bool E() {
4    TOKEN *save = next;
5    return (next = save, E_1()) || (next = save, E_2());
6  }
7  bool T_1 = { return term(INT); }
8  bool T_2 = { return term(INT) && term(TIMES) && T(); }
9  bool T_3 = { return term(OPEN) && E() && term(CLOSE); }
10 bool T() {
11   TOKEN *save = next;
12   return (next = save, T_1()) ||
13        (next = save, T_2()) ||
14        (next = save, T_3());
15 }
```

To start the parser, next needs to be set to the first token, and E() should be invoked.

With the statement *save = next, the current position is saved in save. If the matching for a production succeeds, the matching for the whole non-terminal ends; otherwise next is restored with next = save, and the string continues to be matched with the next production. The use of the C++ comma operator is interesting.

Limitations

The recursive descent parsing algoruthm just presented is easy to be implemented by hand, but it is not general. Specificly for this example, consider the matching process for the string int * int. We will go from E() to T() then to $T_1()$, which returns true and leaves us with the * int segment. This segment cannot be matched, forcing us to wrongly reject the whole string.

The problem is that with this algorithm, we have no way of back tracking and trying other productions once one production has been successfully matched. There exist a more general and yet more complex implementation that solves the problem by allowing "full back tracking", which will be covered later.

Left recursion

If a grammar has the form $S \rightarrow^{+} S\alpha$ for some α, then it has a left recursion problem because the recursive descent matching process will end up in an infinite loop.

Left recursion problem can be solved by writing the algorithm. The grammar $S \rightarrow S\alpha | \beta$ that generates the language $\beta\alpha*$ can be rewritten as $S \rightarrow \beta S'$, $S' \rightarrow \alpha S' | \varepsilon$. More generally, the grammar $S \rightarrow S\alpha_1 | \ldots | S\alpha_n | \beta_1 | \ldots | \beta_m$ that generates the language $(\beta_1 | \ldots | \beta_m)(\alpha_1 | \ldots | \alpha_n)*$ can be rewritten as $S \rightarrow \beta_1 S' | \ldots | \beta_m S'$, $S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon$. The general rule is to recognize the terminal rather than the non-terminal first when matching a combination of the two.

There are algorithms that carries out the elimination of left recursion automatically (in the Dragon book).

1.3.3 Predictive parsing

Predictive parsing is another top-down parsing method. It is able to "predict" which production to use by looking at the next few tokens (lookahead) and no backtracking is needed. Predictive parser accepts so-called **LL(k)** grammars which means **left-to-right left-most-derivation** grammars requiring **k** lookahead tokens. Here we will focus only on the case with k=1. Recall that in recursive descent, many choices of productions could be used at each step, and we use backtracking to undo the bad choices. But with LL(1) grammar, at most one production is available at each step.

Consider the grammar (1.9). It is hard to predict the best production to use because for T, there are two productions that start with an int, and for E, both productions start with T so that it is not clear how to make the predication. The grammar needs to be **left-factored** in order to apply predicative parsing.

The new grammar is

$$\begin{aligned} &E \rightarrow TX \\ &X \rightarrow +E \mid \epsilon \\ &T \rightarrow \text{int } Y \mid (E) \\ &Y \rightarrow * T \mid \epsilon \end{aligned} \tag{1.10}$$

A **LL(1) parsing table** can be generated according to this grammar as shown in Table 1.1. The method to generate it will be covered later. $ is the end

Table 1.1: Parsing table of grammar (1.10)

	int	*	+	(	)	$
E	TX			TX		
X			+E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ

of the input. Rows of the table represent the current leftmost non-terminal, while columns of the table represent the next input token. The entry is the rhs of the production to be used according to the combination of the leftmost non-terminal and the next input token. As an example, we have [E, int] = TX, which means that when the current non-terminal is E and the next input token is int, we should use the production E $\rightarrow$ TX. Also, [Y, +] = ϵ means that when the current non-terminal is Y and the next input token is +, the production Y $\rightarrow$ ϵ should be used, i.e. Y should be got rid of. Finally, empty entry means that there is an error.

In order to carry out predictive parsing, we need to construct a stack that records the frontier of the parse tree. It contains non-terminals to be expanded as well as terminals to be matched. The top of the stack is always the leftmost pending terminal or non-terminal. The algorithm can be illustrated by Algorithm (1.1). If the stack is empty when we reach the end of the input,

Algorithm 1.1 Predictive parsing algorithm

```
Initialize stack = <S,$> and next
repeat
    switch stack
        case <X,rest>:
            if T[X, *next++] == Y_1 ... Y_n then
                stack←< Y_1 ... Y_n rest>
            else error()
        case <t,rest>:
            if t == *next++ then
                stack←<rest>
            else error()
until stack == <  >
```

the string is accepted. If any error state is reached, the string is rejected. A step-by-step predictive parsing process of the string int * int with the grammar (1.10) is shown in Table 1.2.

Now let's discuss the construction of LL1 parsing tables. For non-terminal A, production A$\rightarrow$ α and terminal t, we will have T[A,t] = α in and only in two

Table 1.2: Predictive parsing of int * int

Stack	Input	Action
E$	int * int$	TX
TX$	int * int$	int Y
intYX$	int * int$	terminal
YX$	* int$	* T
*TX$	* int$	terminal
TX$	int$	int Y
intYX$	int$	terminal
YX$	$	ϵ
X$	$	ϵ
$	$	accept

cases:

1. When $\alpha \xrightarrow{*} t\beta$, i.e. α can derive a t at the beginning. We say $t \in \text{First}(\alpha)$.
2. Otherwise when $\alpha \xrightarrow{*} \epsilon$ and $S \xrightarrow{*} \beta A t \delta$. We say $t \in \text{Follow}(\alpha)$.

First set

The formal definition of First(X) is

$$\text{First}(X) = \{t | X \xrightarrow{*} t\alpha\} \cup \{\epsilon | X \xrightarrow{*} \epsilon\}. \tag{1.11}$$

The algorithm to calculate First(X) is

1. For all terminal t, First(t) = {t}.
2. $\epsilon \in$ First(X) if $X \xrightarrow{*} \epsilon$, or if $X \xrightarrow{*} A_1 \dots A_n$ and $\epsilon \in \bigcap_j \text{First}(A_j)$.
3. $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \xrightarrow{*} A_1 \dots A_n \alpha$ and $\epsilon \in \bigcap_j \text{First}(A_j)$.

For grammar (1.10), we have

- For terminals, First(t) = t, t = +, *, (,), int.
- First(T) = {int, (}
- First(E) = {int, (}
- First(X) = {+, ϵ}
- First(Y) = {*, ϵ}

Follow set

The formal definition of Follow(X) is

$$\text{Follow}(X) = \{t | S \xrightarrow{*} \beta X t \delta\}. \tag{1.12}$$

The algorithm to calculate Follow(X) is

1. \$ $\in$ Follow(S)
2. For production $A \to \alpha X \beta$, First(β) $- \{\epsilon\} \subseteq$ Follow(X).
3. For production $A \to \alpha X \beta$ in which $\epsilon \in$ First(β), Follow(A) $\subseteq$ Follow(X).

For grammar (1.10), we have

- Follow(E) = {\$,)}
- Follow(X) = {\$,)}
- Follow(T) = {+,\$,)}
- Follow(Y) = {+,\$,)}
- Follow('(') = {int, (}
- Follow(')') = {+,\$,)}
- Follow(int) = {*,+,\$,)}
- Follow(+) = {int, (}
- Follow(*) = {int, (}

Parsing table

To build a parsing table, for each production $A \to \alpha$, we need to do:

1. For each terminal $t \in$ First(α), T[A, t] = α.
2. If $\epsilon \in$ First(α), for each terminal $t \in$ Follow(A), T[A, t] = α.
3. If $\epsilon \in$ First(α) and \$ $\in$ Follow(A), T[A, \$] = α.

If any entry of the parsing table is multiple defined, then the grammar is nor LL(1). In particular, if the grammar is

- not left factored
- left recursive
- ambiguous
- ...

then it is not LL(1). Actually most programming language CFGs are not LL(1). LL(1) grammar is too week to describe all the features required in these languages.

1.4 Bottom-up parsing

Bottom-up parsing is more general than deterministic top-down parsing. Actually it is as efficient, and builds on all the ideas we have discussed in top-down parsing. Bottom-up parsing is the preferable method in reality. Bottom-up parsing does not require left-factored grammer, thus we can revert to the natural grammar (1.9) in the following discussion. Nonetheless, bottom-up parsers do not deal with ambiguous grammars, thus we still have to enforce precedence and associativity rules.

Bottom-up parsing reduces a string to the start symbol by inverting productions. As an example, consider the string int * int + int. It can be reduced to the start symbol E via the following path:

int * int + int	T→int
int * T + int	T→int * T
T + int	T→ int
T + T	E→T
T + E	E→T + E
E	

Obviously, the left column is the rightmost derivation of the string written in reverse. This is actually always true for bottom up parsers. **Bottom-up parser traces a rightmost derivation in reverse.** It builds the parse tree from its leaves up towards the root, by combining smaller parse trees into larger ones.

1.4.1 Shift reduce parsing

Suppose $\alpha\beta\omega$ is a step of a bottom-up parse, and the next reduction to apply is $X \to \beta$. Since $\alpha X\omega \to \alpha\beta\omega$ is a step in a rightmost derivation, we can be sure that ω is a string of terminals. This inspires us of the idea of shift-reduce parsing. The string is split into two substrings, the right one unexamined by the parser, and the left one containing terminals and non-terminals. The dividing point is marked by a | sign. Two actions are needed to carry out bottom-up parsing: **Shift** and **Reduce**. Shift means moving | one place to the right, i.e. shifting one terminal to the left substring. Reduce means applying an inverse production at the right end of the left substring.

The left substring in shift-reduce string can be implemented by a stack, with the top of the stack being the | sign. Each shift action pushes a terminal on the stack. Each reduce action pops symbols (rhs of the production rule, terminals and nonterminals) out of the stack, and pushes a nonterminal on the stack.

In a given state, more than one action might lead to a valid parse. If it is legal to shift or reduce, there is a **shift/reduce** conflict. If there are two legal reduces, then there is a **reduce/reduce** conflict. Reduce/reduce conflicts are always bad, indicating some serious problem of the grammar. Shift/reduce conflicts can usually be removed by precedence definitions.

1.4.2 Handle & viable predixes

Reducing whenever we meet a rhs of production will probably cause incorrect results. For example, we cannot reduce int * int to T * int after the first shift according to the grammar (1.9). We should only reduce when its result can still be reduced to the start symbol. A **handle** is a reduction that also allows further reductions back to the start symbol. For a rightmost derivation $S \overset{*}{\rightarrow} \alpha X\omega \rightarrow \alpha\beta\omega$, we say $\alpha\beta$ is a handle of $\alpha\beta\omega$. **In shift-reduce parsing, handles appear only at the top of the stack (never inside).** Handles are never to the left of the rightmost non-terminal. Thus shift-reduce moves are sufficient, and the | sign never has to move left. Bottom-up parsing algorithms are based on recognizing handles.

Recognizing handles

Unfortunately there exists no known efficient algorithm to recognize handles. Nonetheless, there are good heuristics for guessing handles, and for some fairly large classes of CFGs, these heuristics always identify the handles correctly. Figure 1.3 illustrates the relationship of different kinds of CFGs. Most of the

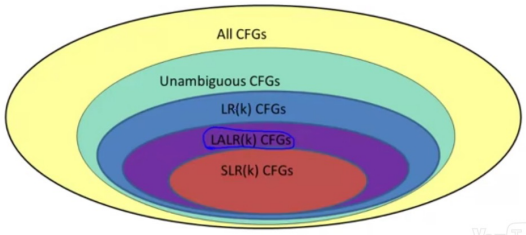

Figure 1.3: Relationship of different CFGs

time we will talk about SLR(k) CFGs.

It is not obvious how to detect handles. We should always keep in mind that at each step, the parser sees only the stack and not the entire input, which is the starting point of the whole discussion. First we define a **viable prefix**. α is a viable prefix if there is an ω such that $\alpha|\omega$ is a state of a shift-reduce parser. Here, α is the visible stack, while ω is the rest of the input. A viable prefix is a string that does not extend past the right end of the handle. It is called "viable" because it is a prefix of the handle. As long as the parser has viable prefixes on the stack, it means that no parsing error has been detected.

For any grammar, the set of viable prefixes is a regular language. As a result, the set of viable prefixes can be recognized by a finite automaton. We will shown how to compute the automata that accept viable prefixes.

First we introduce the idea of an **item**. An item is a production with a $\cdot$[1] somewhere in the rhs. For example, the production T→(E) produces 4 items:

[1] In the lectures . is used. I use $\cdot$(LaTeX\cdot) to avoid confusion with period.

T→ ·(E), T→(·E), T→(E·) and T→(E)·.For ϵ-productions, the only item for X→ ϵ is X→ ·. Items are usually called the LR(0) items. They provide a description of intermediate steps of shirt-reduce parsing. Consider the input (int) for grammar (1.9). (E|) is a state of a shift-reduce parse for it. (E is a prefix of the rhs of the production T→(E), and it is to be reduced if a) is recognized after the next shift. Item T→(E·) decribes such situation: we have seen (E and hope to see).

The stack contains actually many prefixes of rhses of productions:

$$\text{Prefix}_1\text{Prefix}_2 \dots \text{Prefix}_{n-1}\text{Prefix}_n$$

Let Prefix_i be a prefix of rhs of $X_i \to \alpha_i$. Prefix_i will eventually reduce to X_i. In order that the parsing can continue, the missing part of α_{i-1} must start with X_i, i.e. there must exist a production $X_{i-1} \to \text{Prefix}_{i-1}X_i\beta$ for some β. Recursively $\text{Prefix}_{k+1} \dots \text{Prefix}_n$ eventually reduces to the missing part of α_k.

Consider (int * int) for grammar (1.9). (int * | int) is a state of a shift-reduce parse. We have the stack of items:

$$\begin{aligned} &\text{T} \to (\cdot\, \text{E}) \\ &\text{E} \to \cdot\, \text{T} \\ &\text{T} \to \text{int}^* \cdot \text{T} \end{aligned}$$

Recognizing viable prefixes

As concluded previously, to recognize viable prefixes, we must recognize a sequence of partial rhses of productions, where each partical rhs can eventually reduce to part of the missing suffix of its predecessor. We will build an NFA that takes the stack as an input and decides whether to accept or reject it.

1. Add a dummy production S'→S to the grammar G.
2. The NFA states are items of G, including the dummy production just added.
3. For item E→ $\alpha \cdot X\beta$, add transition

   $$\text{E} \to \alpha \cdot X\beta \xrightarrow{X} \text{E} \to \alpha X \cdot \beta$$

   Here X is either a terminal or a non-terminal. This rule extends a prefix or an rhs.
4. For item E→ $\alpha \cdot X\beta$ and production $X \to \gamma$, add transition

   $$\text{E} \to \alpha \cdot X\beta \xrightarrow{\epsilon} X \to \cdot\gamma$$

   Here X can only be non-terminals. This rule ends the current prefix and starts a new one.
5. Every state is an accepting state.

6. Start state is S'→S.

Consider grammar (1.9). After adding the dummy production, it becomes

$$\begin{aligned} &\text{S'} \rightarrow \text{E} \\ &\text{E} \rightarrow \text{T} \mid \text{T} + \text{E} \\ &\text{T} \rightarrow \text{int} \mid \text{int} * \text{T} \mid (\text{E}). \end{aligned} \tag{1.13}$$

By applying the algorithm above, we wind up with the NFA shown in Figure 1.4.

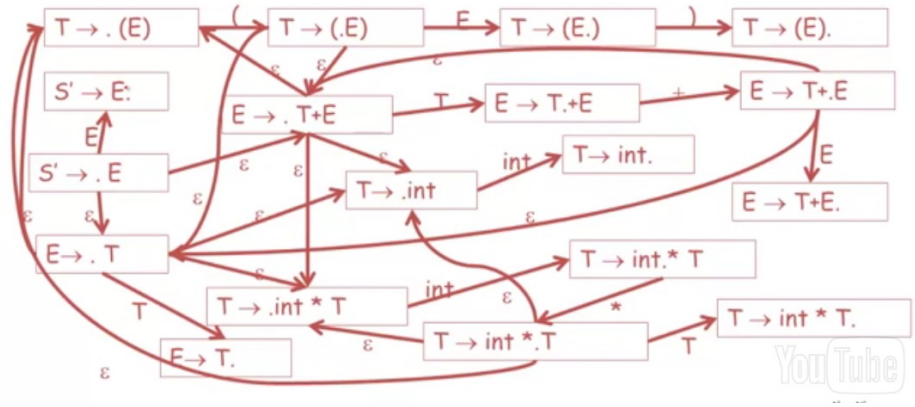

Figure 1.4: NFA of grammar (1.9)

The NFA gained using the algorithm above can be converted into a DFA. The states of the DFA are sets of items. The state of the DFA are called "canonical collections of items (or LR(0) items)"[2].

The item X→ $\beta \cdot \gamma$ is said to be **valid** for a viable prefix $\alpha\beta$ if

$$S' \xrightarrow{*} \alpha X \omega \rightarrow \alpha\beta\gamma\omega$$

by right-most derivation. After parsing $\alpha\beta$, the valid items are possible tops of the stack of items. An equivalent explanation is that for a given viable prefix α, the valid items are exactly the items in the final state of the DFA after it reads that prefix.

1.4.3 SLR parsing

In this section we will present the **SLR parsing** algorithm. SLR means "simple left-to-right rightmost".

First we will introduce a weak bottom-up parsing algorithm called LR(0) parsing. Suppose that the stack contains α and the next input is token t. The DFA on input α terminates in state s. Then the rules for shift-reduce actions are:

[2]Dragon book provides another way to define LR(0) items.

- Reduce by $X \to \beta$ if $X \to \beta\cdot \in s$
- Shift if $X \to \beta \cdot t\omega \in s$. This is equivalent to the fact that s has a trasition labeled t.

LR(0) parsing has a reduce/reduce conflict if any state has two reduce items $X_1 \to \beta_1\cdot$ and $X_2 \to \beta_2\cdot$. It has a shift/reduce conflict if any state has a reduce item $X_1 \to \beta_1\cdot$ and a shift item $X_2 \to \beta_2 \cdot t\delta$.

SLR improves on LR(0) shift/reduce heuristics. As a result, fewer shift/reduce conflicts will happen. The only adjustment is on the reduce rule, which is changed to

- Reduce by $X \to \beta$ if $X \to \beta\cdot \in s$ and t$\in$Follow(X)

If any conflict still exists, then the grammar is not an SLR grammar. The handles of SLR grammars can be exactly detected by the heuristics. A lot of grammars are not SLR grammars. We can define define rules to resolve some confilicts for such grammars, which sometimes have the effect of precedence declaration.

Now we present the complete SLR parsing algorithm.

Algorithm 1.2 SLR parsing algorithm

Let M be the DFA for viable prefixes of G. and let $|x_1 \dots x_n\$$ be the initial configuration.
repeat
  Let $\alpha|\omega$ be the current configuration and run M on the current stack α.
  if M rejects α **then**
    report parsing error
  else
    M accepts α with items I, let a be the next input
    if $X \to \beta \cdot a\gamma \in I$ **then** shift
    else if $X \to \beta\cdot \in I$ **and** $a \in$Follow(X) **then** reduce
    else report parsing error
until configuration is $S|\$$

See video 8-6 for an exhaustive example of applying the algorithm.

Algorithm (1.2) can be improved because rerunning the viable predixes automaton is wasteful: reduce or shift will only change a few symbols on the top of the stack, while the rest of the stack stays the same, and the work on them is just a repeat. If the repeat can be avoided, the algorithm can run much more quickly.

The state of the automaton on each prefix can be remembered. It will be more convenient to make the stack store <Symbol, DFA state> pairs rather than just symbols. We will have a stack

$$< \text{sym}_1, \text{state}_1 > \cdots < \text{sym}_n, \text{state}_n >,$$

in which state_n is the final state of the DFA on $\text{sym}_1 \ldots \text{sym}_n$. The bottom of the stack is now <any, start>, in which any is any dummy state, while start is the start state of the DFA. The algorithm has 4 possible moves:

- Shift x. <a,x> will be pushed on the stack, in which a is the current input, while x is a DFA state.
- Reduce X$\rightarrow \alpha$. As before, pop the rhs elements off the stack and push the lhs on.
- Accept.
- Error.

We will define two parsing tables.

- Define goto[i,A] = j if $\text{state}_i \xrightarrow{A} \text{state}_j$. goto is just the transition function of the DFA.
- Define action[i,a] for each DFA state s_i and the next input terminal a.
  - If s_i contains item $X \rightarrow \alpha \cdot a\beta$ and goto[i,a] = j then action[i,a] = shift j.
  - If s_i contains item $X \rightarrow \alpha\cdot$ and a$\in$Follow(X) and X$\neq$S' then action[i,a] = reuce $X \rightarrow \alpha\cdot$.
  - If s_i contains item S'$\rightarrow$S, then action[i,\$] = accept.
  - Otherwise action[i,a] = error.

The improved algorithm is shown in Algorithm (1.3).

In practice, SLR parsing is too simple and sometimes naive. LR(1), or LALR(1), which is based on LR(1) and includes some optimisations, parsing is much more widely used. The main difference between LR(1) and SLR is that LR(1) builds lookahead into the items. An LR(1) items looks like [T$\rightarrow \cdot$int * T, \$], which means "after seeing int * T, reduce if lookahead is \$". This mechanism is more accurate that using just the follow set to decide whether to reduce.

Algorithm 1.3 Improved SLR parsing algorithm

```
Let I = w$ be the initial input
Let j = 0.
Let DFA state 1 have item S'→S.
Let stack = <dummy, 1>
repeat
    switch action[top_state(stack), I[j]]
        case Shift k:
            Push <I[j++], k> on stack
        case reduce X → α:
            Pop |A| pairs
            Push <X, goto[top_state(stack), X]> on stack
        case accept:
            Halt normally
        case error:
            Halt with an error
until configuration is S|$
```