Crafting a Compiler with C (V)

資科系
林偉川

Scanner generator

- Limit the effort in building a scanner to specify which tokens the scanner is to recognize
- Some generators do not produce an entire scanner; rather, they generate tables that can be used with a standard driver program
- Programming a scanner generator is an example of nonprocedural programming

Nonprocedural V. S. Procedural

- Procedural programming should tell computer every detailed procedures includes what to do and how to do
- Nonprocedural programming just tell what to do not how to do ➔ SQL
- Nonprocedural programming is most successful in limited domains such as scanner generator (Lex, ScanGen) or 4GL
- Regular expression notation is very suited to the formal definition of tokens

3

Regular Expression

- Regular set start with a finite character set ➔ vocabulary denoted by V is the character set used to form tokens
- Meta-characters such as Kleene Closure (*➔zero or more)
- Regular expressions are defined as followed:
  - ∅ is a regular expression denoting the empty set
  - λ is a regular expression denoting the set that contains only the empty string
  - String S is a regular expression denoting a set containing only s. If s contains meta-characters, s can be avoid ambiguity
  - If A and B are regular expression, then A|B, AB and A* are also regular expression

4

Three standard operators

- P^+ ➔ $P^*=(P^+ \mid \lambda)$ and $P^+=PP^*$ ➔ one or more strings
- Not(A) denotes (V-A), if S is a set of strings, Not(S) denotes (V*-S)
- A^k represents all strings formed by catenation k strings from A A^k = (AAA…) (k copies)

5

Using RE to express tokens

- D=(0|1|…|9)
- L=(A|B|…|Z)
- A comment begins with --, end with Eol (End of line) can be defined as

  Comment= -- Not(Eol)* Eol
  Comment= ## ((#|λ)Not(#))* ##
- A fixed decimal literal can be defined as

  Lit=D^+ . D^+
- A identifier composed of letters, digits and underscores can be defined as

  ID=L (L | D)* (_ (L | D)$^+$)*

6

Cannot be expressed by RE

- { $[^i]^i$ | i >=1} is a well-known set that is not regular ➔balance brackets
- Palindrome such as aaaaXbbbb
- Matching parenthesis
- The power of RE is equivalence to FA
- They are easily handled by CFG
- CFG are a more powerful descriptive mechanism than RE. RE is quite adequate for specifying token-level syntax.

7

Finite Automata

- FA can be used to recognize the tokens specified by a regular expression
- FA consists of: FA={s, S, T, F}
  - A finite set of states ➔ S
  - A set of transitions (moves) ➔ T
  - A special start state ➔ s
  - A set of final or accepting states ➔ F

8

Finite Automata

- Start at the start state and the input character decides the transition from current state to the next.
- If we finish in a final state, the sequence of characters read is a valid token, otherwise, it is not a valid token
- If an FA always has an unique transition, the FA is deterministic (DFA) and DFA are often used to drive a scanner

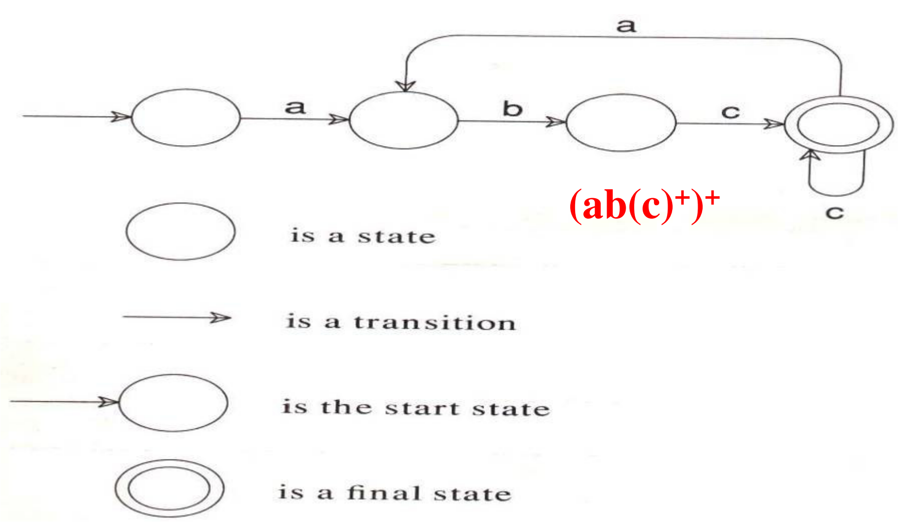
9

DFA

- A DFA is represented in a computer by a transition table
- A transition table T is indexed by a DFA state and a vocabulary symbol
- Table entries are either a DFA state or an error flag. Error entries are blank
- If we are in state s, and read character c, then T[s][c] will be the next state we visit, or it is an error flag indicating that c cannot be part of the current token

10

Finite automata example



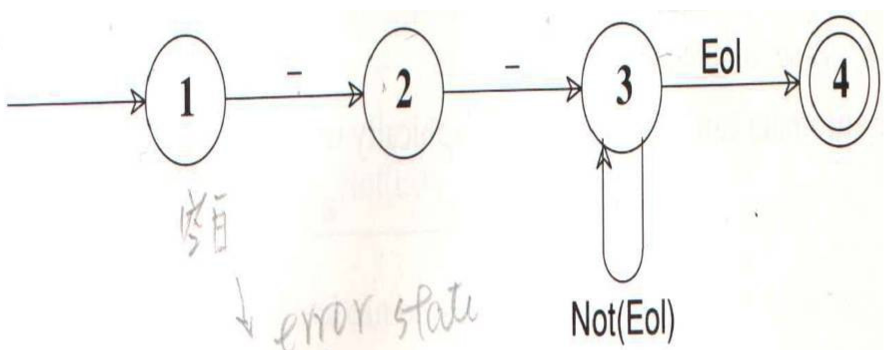
DFA Transition Table

state	character		
	a	b	c
1	2		
2		3	
3			4
4	2		4

DFA example

- Regular expression : Comment= -- Not(Eol)* Eol

The corresponding DFA is followed:



DFA Transition Table

state	character				
	-	Eol	a	b	…
1	2				
2	3				
3	3	4	3	3	3
4					

14

Scanner

- Any RE can be translated into a DFA that accepts the set of stings denoted by the regular expression
- The transition can be done
  - Automatically by a scanner generator
  - Manually by a programmer
- A DFA can be implemented as either a transition table interpreted by a driver program or directly into the control logic of a program

15

Scanner

- For the example of the comment:

  Regular expression : -- Not(Eol)*Eol
- The first form is used with a scanner generator and is language independent. This form is a simple driver that can scan any token if the transition table is properly set
- The second is produced by hand and the token is scanned and hardwired into the code

16

Scanner use the transition table

```
state=initial_state;  // -- Not(Eol)*Eol
while (true) {
  next_state=T[state][current_char];
  if (next_state == ERROR) break;
  state=next_state;
  if (current_char == EOF) break;
  current_char=getchar();
}
if (is_final_state(state)) // process valid token
else lexical_error(current_char);
```

17

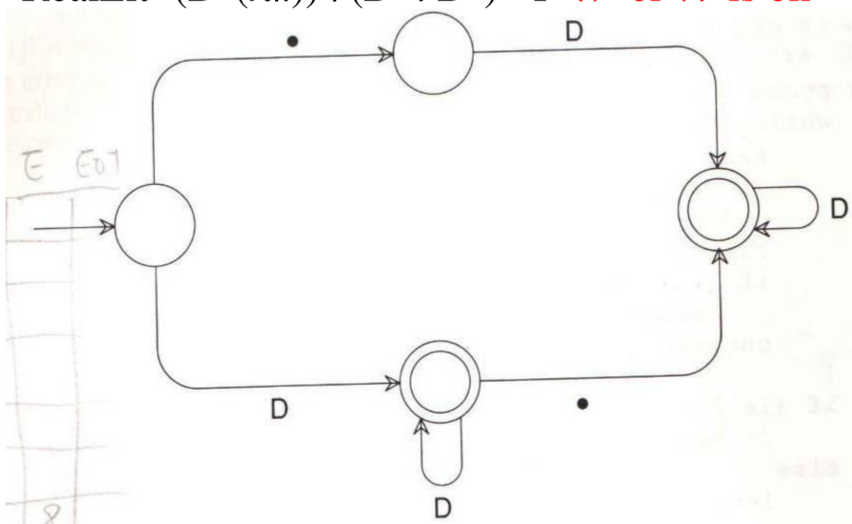
Scanner use fixed token definition

```
if (current_char == '-') {// -- Not(Eol)*Eol
   current_char=getchar();
   if (current_char == '-') // hardwired token
      do  { current_char=getchar(); }
      while (current_char != '\n');
   else {
     ungetc(current_char, stdin);
     lexical_error(current_char);
    }
 }
else lexical_error(current_char);
```

18

Fortran-like real literal RE & DFA

RealLit=(D⁺(λ|.)) | (D*. D⁺) ➔ .7 or 7. is ok



19

DFA Transition Table??

20

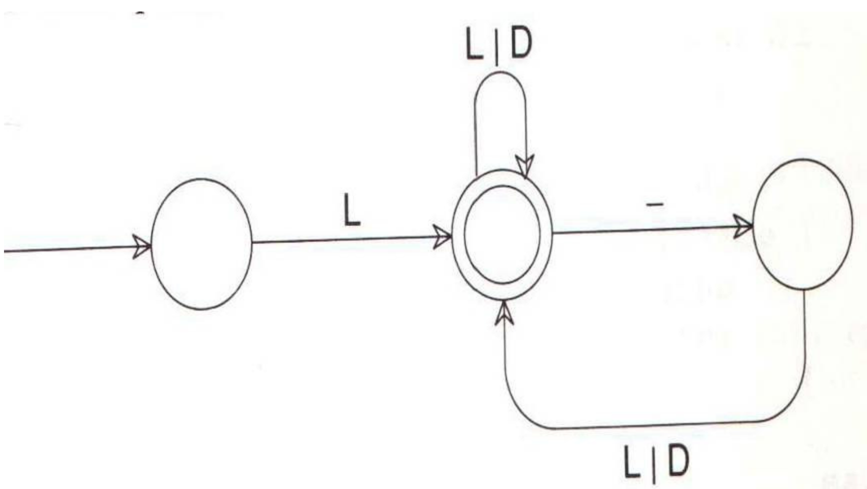
Floating point real literal

FloatingpointLit= ?? ➔ 1.7e-1, 4.123, 7.123e+12 is ok

21

Identifier

ID= L (L|D)*(_(L|D)$^+$)* L3 is ok, L_, _a, _1 ??



DFA Transition Table??

23

FA transducer

- Add an output facility to an FA and makes the FA a transducer
- As characters are read, they can be transformed and concatenated to an output string

a → Means save *a* into a token buffer

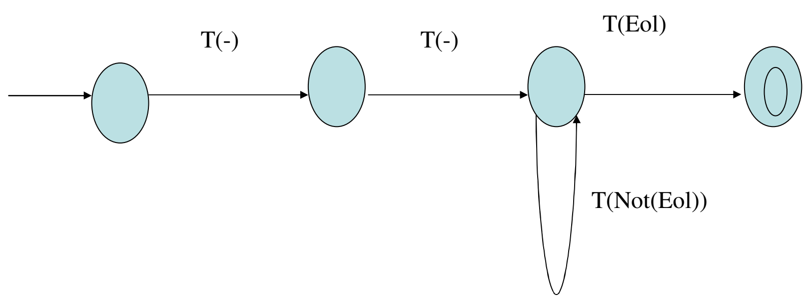
T(a) → Means don't save *a* (Toss it away)

24

For a comment transducer

- Regular expression : -- Not(Eol)*Eol

The corresponding DFA is followed:

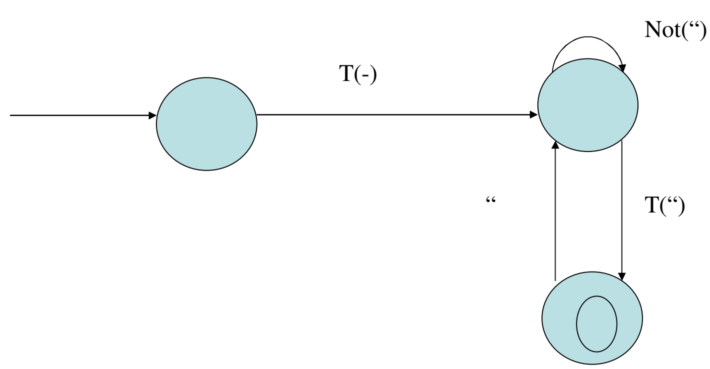


25

String example

- Regular expression : (“ (Not(“) | “”)* “)
  for a quoted strings ➔ **“”””Hi””””” ➔ “Hi”**

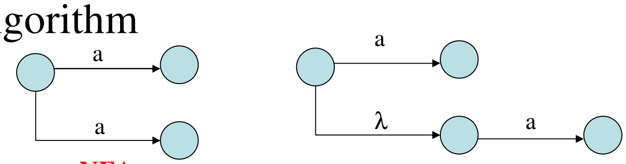
The corresponding DFA is followed:



26

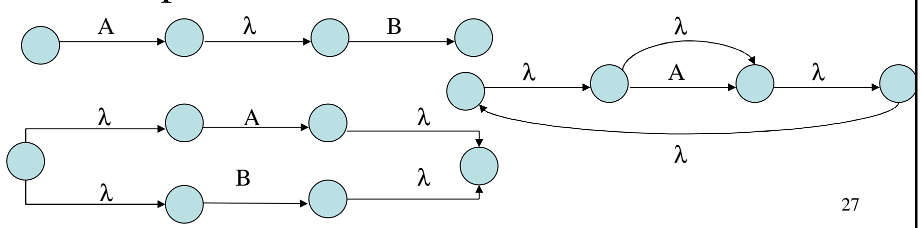
Translating RE into FA

- From RE to NFA ➔ straightforward!!!
- From NFA to DFA ➔ using subset construction algorithm



NFA NFA

- Three operator of NFA such as AB, A|B, A*



27

Subset construction

- Initial state of M (equivalent DFA) is the set of all states that N (a NFA) could be in without reading any input characters ➔ the set of states reachable from the initial state of N following only λ arrows
- Algorithm close() computes those states that can be reached following only λ transitions
- Once the start state of M is built, we begin to create successor states.

28

Subset construction

- Take any state S of M, any character c, and compute S's successor under c. S is identified with some set of N's states, $\{n_1,n_2\ldots\}$
- Find all possible successor states to $\{n_1,n_2\ldots\}$ under c, obtaining a set $\{m_1,m_2\ldots\}$

29

Subset construction

- Compute $T=close(\{m_1,m_2\ldots\})$ and T is included as a state in M, and a transition from S to T labeled with c is added to M
- Continue adding states and transition to M until all possible successors to existing states are added
- Because each state corresponds to a subset of N's states, the process of adding new states to M should eventually terminate

30

Algorithm of collect λ state to become a state

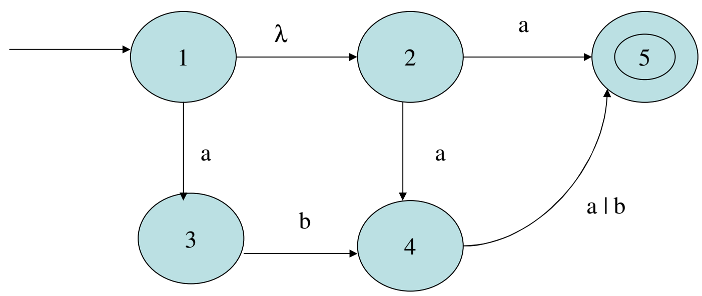
/* add to S all states reachable from it using only
 * λ transitions of N (a NFA) */

```
void close(set_of_fa_states *S) {
  while (there is a state x in S and a state y not in S
    such that x → y using λ transitions)
        add y to S;
}
```


Convert NFA to DFA

```
void make_deterministic(nondeterministic_fa N, *deterministic_fa *M)
   {
  set_of_fa_state T;   M→initial_state=SET_OF(N.initial_state);
  close(&M->initial_state); add M->initial_state to M_states;
  while (states of transitions can be added) {
     choose S in M->states and c in Alphabet;
     T=SET_OF(y in N.states such that x  →c y for some x in S);
     close(& T); if (T not in M->states) add T to M->states;
     add the transition to M->transitions: S  →c T;
  }
 M->final_state=SET_OF(S in M->states such that  N.final_state in S);
}
```

NFA題目 ➔ a | a(a|b)* | ab(a|b)*



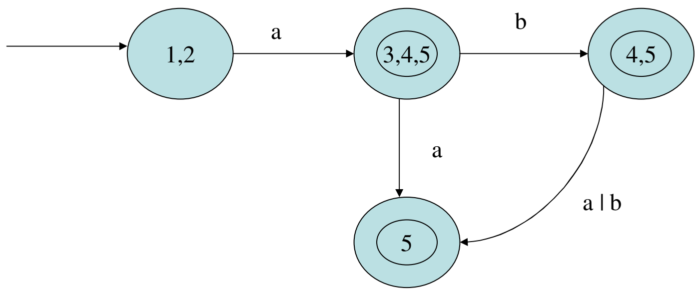
33

Convert NFA to DFA

- Start with state 1, the start state of N, and add state 2, its λ-successor➔ {1,2} as M's start state
- state 1 and 2 has no successor under b. {1,2}'s successor under a is {3,4,5}
- {3,4,5}'s successor under a is {5}, under b is {4,5}
- {4,5}'s successor under a, b is {5}
- Final state of M are those state sets that contain N's final state (5)

34

DFA



35

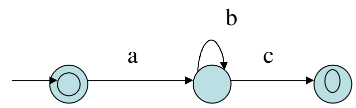
DFA V.S. NFA

- DFA is built can sometimes be much larger than the original NFA
- States of the DFA are identified with sets of NFA states. If NFA has n states, there are 2^n distinct sets of NFA states and the DFA may have 2^n states

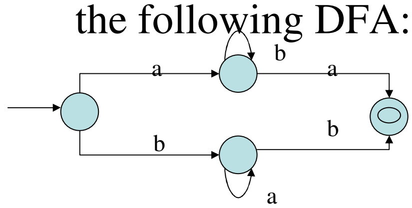
36

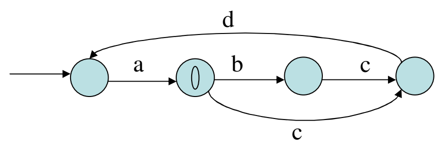
Home Work

- Write DFA that recognize the token defined by the following RE:
  - $(a \mid (bc)^{*}d)^{+}$
  - $((0|1)^{*}(2|3)^{+}) \mid 0011$
  - $(a\ \mathrm{Not}(a))^{*}aaa$



- Write RE that define the strings recognized by the following DFA:





37

Reserved words

- Most PL choose to make key words reserved. This simplifies parsing, which drives the compilation process and makes programs more readable
- Assume that in Pascal begin and end are not reserved, and some devious programmer has declared procedures name begin and end. The program can be parsed in many ways
  begin
  begin; end; end; begin;
  end

38

Reserved words

- In PL/I, key words are not reserved, but procedures are called using an explicit call key word. ➔ key word may be used as variable names: if if>0 then else=then;
- The problem with reserved words is that if they are too numerous, they may confuse inexperienced programmers who unknowingly choose an identifier name. V.S. COBOL uses many reserved words such as zero, zeros, zeroes…

39

Reserved words

- We could get a RE for non-reserved Ids by getting rid of the Nots in the expression Not (Not(Id) | begin | end | …)
- Suppose END was the only reserved word, and the alphabet (L) had only letters Nonreserved = L | (LL) | ((LLL)L^+) | ((L-'E')L^*) | (L(L-'N')L^*) | (LL(L-'D')L^*)
- A simpler solution is to treat reserved words as ordinary identifiers and use a separate table lookup to detect them ➔a hash table may be used

40

Home Work

- Write a RE that defines a Pascal-like fixed-decimal literal with no leading or trailing zeros:➔0.0, 23.01, and 1235.0 is ok but 00.00, 001.000, and 00234.1000 are not ok.?
  $(0\text{-}9)^* \mid (1\text{-}9)^+.(0\text{-}9)^+$??

41

Listing source code

- Languages like C have elaborate macro definition and expansion facilities that are typically handled by a preprocessing phase prior to scanning and parsing (#define m(k) k*6+4
- Some languages like C and PL/I include conditional compilation directives that control whether statements are compiled or ignored (#ifdef ?? in a makefile)

42

Listing source code

- Usually these directives have the general form of an if statement, and a conditional expression will be parsed and evaluated. Tokens following the expression will be passed to the parser or ignored until an end if delimiter is reached
- Another possible function of a scanner is to list source lines. The most obvious way to produce a source listing is to echo characters as they are read, using end of line conditions to terminate a line increment line counters

43

Listing source code

- Error messages may need to be printed, and these should be written after the source line, with pointers to the offending symbol
- A source line may need to be edited before it is written. This involves inserting or deleting symbols for error repair, replacing symbols because of macro preprocessing, and reformatting symbols to pretty-print a program

44

Listing source code

- It is best to build output lines incrementally as tokens are scanned. The token image placed in the output buffer may not be an extra image of the token that was scanned, depending on error repair, pretty-printing, case conversion.
- If a token cannot fit on an output line, the line is written and the buffer is cleared

45

Listing source code

- At each token is returned by the scanner, its position in the output line buffer should be included. If an error involving the token is noted, this position marker is used to point to the token. Error message themselves are buffered and normally printed immediately after the corresponding output buffer is written

46

Listing source code

- In some cases, an error may not be detected until long after the line containing the error has been processed➔ goto to an undefined label should have the error message "undefined label in statement 101"
- In languages that freely allow forward references, delayed error may be numerous. ➔ declaration of objects after they are referenced

47

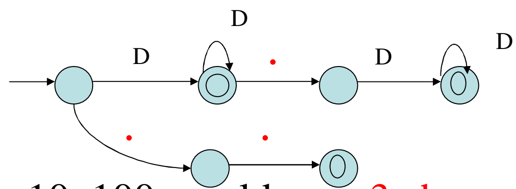
Multi-character lookahead

- We can generalize FAs to look ahead beyond the next input character and it is important for implementing FORTRAN.
  DO 10 I=1,100 is the beginning of a loop
  DO10I=1.100 is an assignment to the variable DO10I (blanks are significant in FORTRAN)
- A FORTRAN scanner can determine whether the O is the last character of a DO token only after reading as far as the comma or period DO 10 I=1.100 ➔the error was not detected until run time and made the rocket deviated

48

Multi-character lookahead

- To scan 10..100 in Pascal and Ada, we need 2 character look-ahead after the 10



- Given 10..100 would scan 3 characters and stop in a non-final state. If we stop reading in a nonfinal state, we can back up along accepted characters until a final state is found.
- Characters we back up over are rescanned to form later tokens.

49

Multi-character lookahead

- If no final state is reached during backup, we have a lexical error and invoke lexical error recovery.
- In Pascal or Ada, never have more than two-character lookahead, which simplifies buffering characters to be rescanned.
- Multiple character lookahead may also be a consideration in scanning invalid programs. ➔ 12.3e+q is an invalid token➔ the scanner could be backed up to produce 4 tokens (12.3, e, +, q) is invalid, the parser will detect a syntax error when it processes the sequence

50

Multi-character lookahead

- Whether to consider this a lexical error or a syntax error (or both) is irrelevant.
- Build a scanner that can perform general backup is easy. As each character is scanned, it is buffered, and a flag is set indicating whether the character sequence scanned so far is a valid token. If we reach a situation in which we are not in a final state and cannot scan any more characters, backup is invoked. We extract characters from the right end of the buffer and queue them for rescanning

Multi-character lookahead

- The process continues until we reach a prefix of the scanned characters flagged as a valid token. This token is returned by the scanner. If no prefix is flagged as valid, we have a lexical error
- An example of scanning with backup for 12.3e+q is shown as listed table. This table shows how the buffer is built and flags are set
- When q is scanned backup is invoked

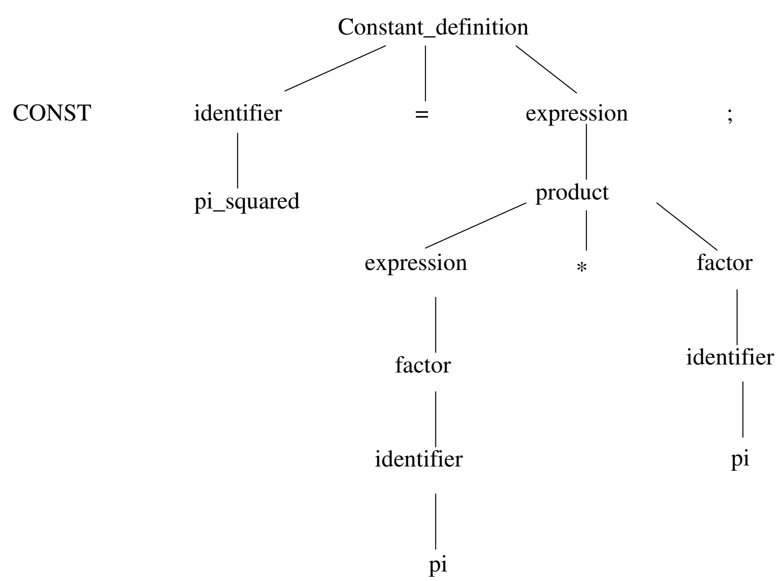
Buffered token	Token flag
1	Integer literal
12	Integer literal
12.	Invalid
12.3	Real literal
12.3e	Invalid
12.3e+	Invalid

From program text to abstract syntax tree

- Grammar for constant definition is:
  constant_definition→CONST <id>=<expression>;
- The grammar rule for <expression> is shown:
  <expression> → <product> | <factor>
  <product> → <expression> * <factor>
  <factor> → <number> | <identifier> (<id>)
- The actual syntax tree for
  CONST pi_squared = pi * pi;

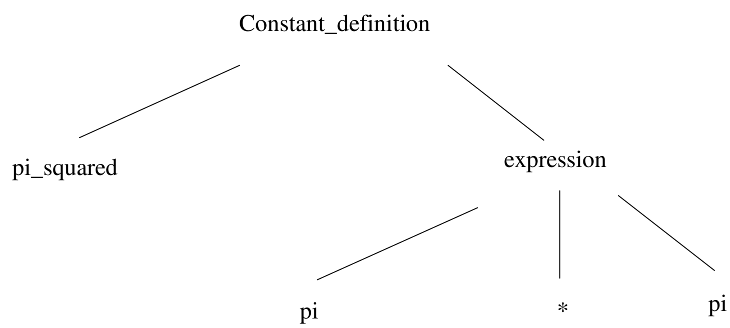
53

Actual syntax tree for a constant



54

Abstract syntax tree for a constant



55

Simplification of actual to abstract syntax tree

- Tokens CONST, =, and ; serve only to alert the reader and the parser to the presence of the constant definition, and need not be retained for further processing
- The semantics of identifier, expression, and factor are trivial and need not be recorded

56

Simplification of actual to abstract syntax tree

- Nodes for constant_definition can be implemented in the compiler as records with two fields:
  struct constant_definition {
   Identifier *CD_idf;
   Expression *CD_expr;
  }

57

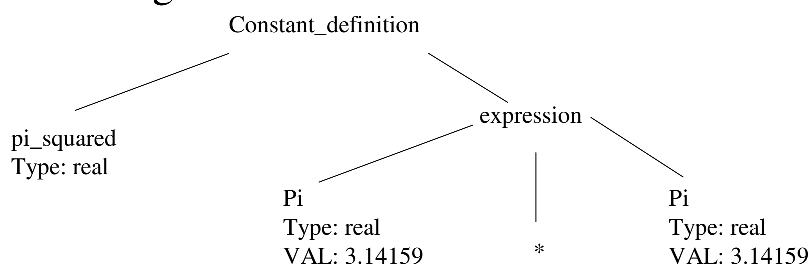
Context handling module

- Context handling module gathers information about the nodes and combines it with that of other nodes. This information serves to perform contextual checking and to assist in code generation
- The abstract syntax tree decorated with these bits of information is called annotated abstract syntax tree. The abstract syntax tree passes through many stages of "annotatedness" during compilation

58

Context handling module

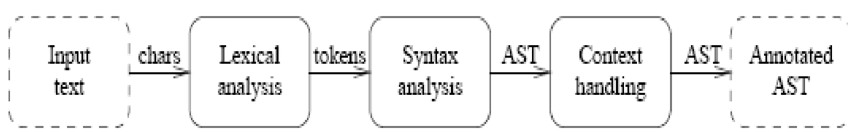
- The degree of annotatedness starts out at almost zero, straight from parsing, and continues to grow through code generation and actual memory addresses may be attached as annotations to nodes. At the end of the context handling, the AST might have the form



59

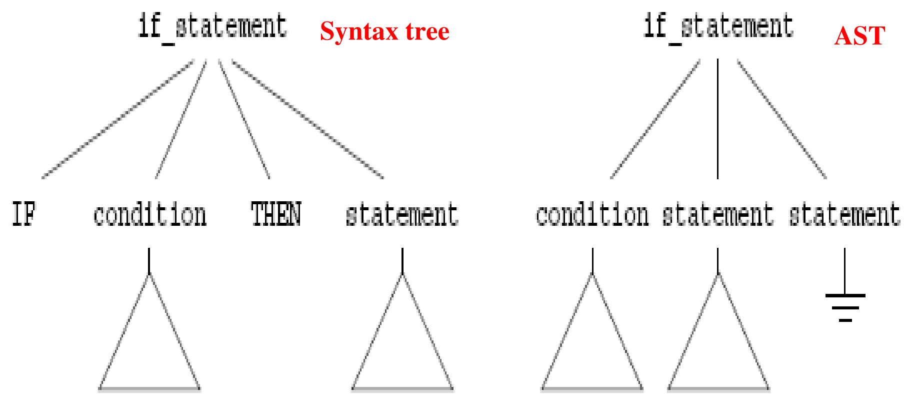
Context handling module

- Having established the annotated abstract syntax tree as the ultimate goal of the front-end, we can work our way back through the design
- To get an abstract syntax tree we need a parse tree. To get a parse tree we need a parser, which needs a stream of tokens; to get the tokens we need a lexical analyzer, which needs a stream of characters, and to get these characters we need to read them



Difference between parse tree and AST

- Combination of the node types for if-then-else and if-then into one node type if-then-else.



Lexical error recovery

- It is unreasonable to stop compilation because of a minor error, so it is necessary to try some sort of lexical error recovery
- Two approaches come to mind:
  - Delete the characters read so far and restart scanning at the next unread character (reset the scanner and begin scanning anew)
  - Delete the first character read by the scanner and resume scanning at the character following it

62

Lexical error recovery

- Can be implemented using the buffering mechanism for scanner backup
- The effect of lexical error recovery might create a syntax error, which will be detected and repaired by the parser. A good syntactic error-repair algorithm will make some reasonable repair although quite possible not correct
- If the parser has a syntactic error-repair mechanism, it can be useful to return a special warning token when a lexical error occurs

63

Lexical error recovery

- The semantic value of the warning token is the character string deleted to restart scanning. When the parser sees the warning token, it is warned that the next token is unreliable and that error repair may be required. The text that was deleted may be helpful in choosing the most appropriate repair

64

Lexical error recovery

- Catch runaway strings is to introduce an error token that represents a string terminated by an end of line rather than a quote character
- A correct quoted string, we might have
  “ (Not(“ | Eol) | “”)* “
  for a runway string we would use
  “ (Not(“ | Eol) | “”)* Eol

65