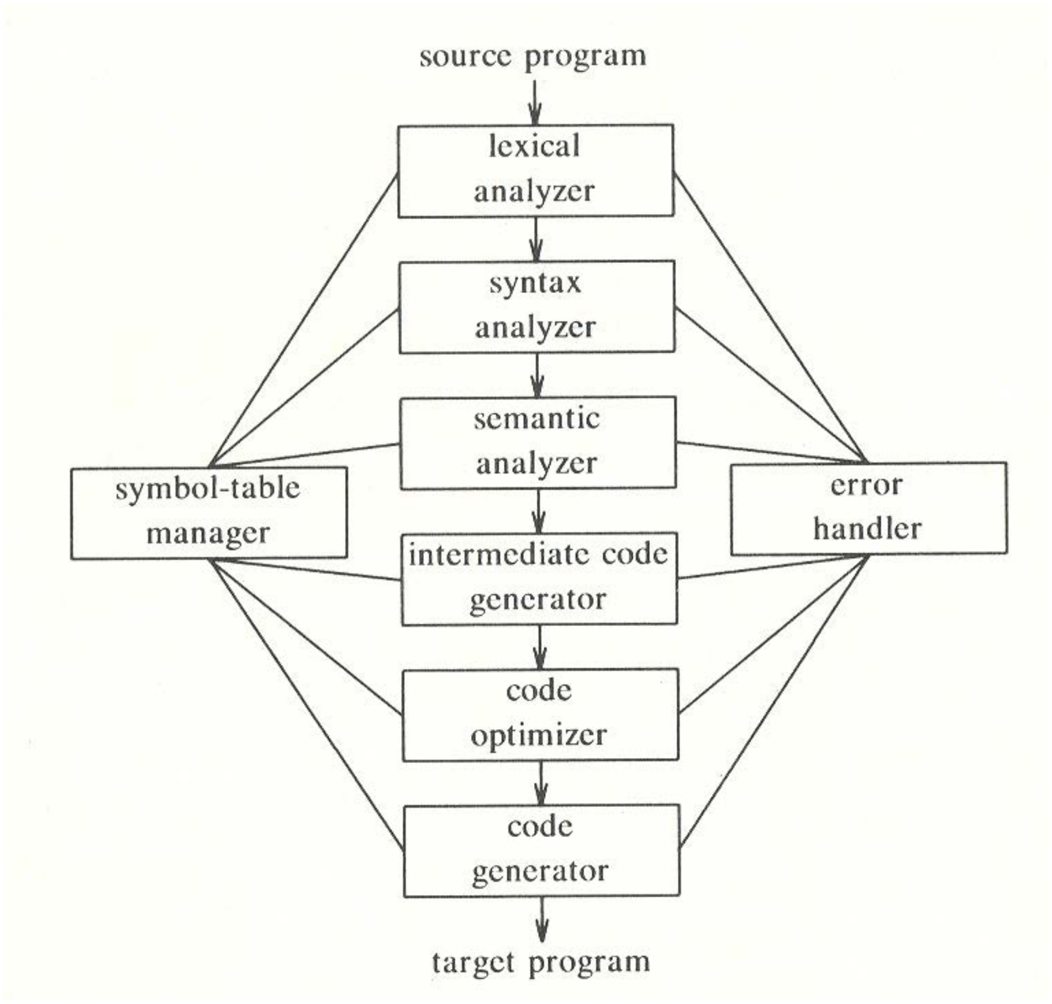
CSCI 48400

An Overview of Compiler Theory

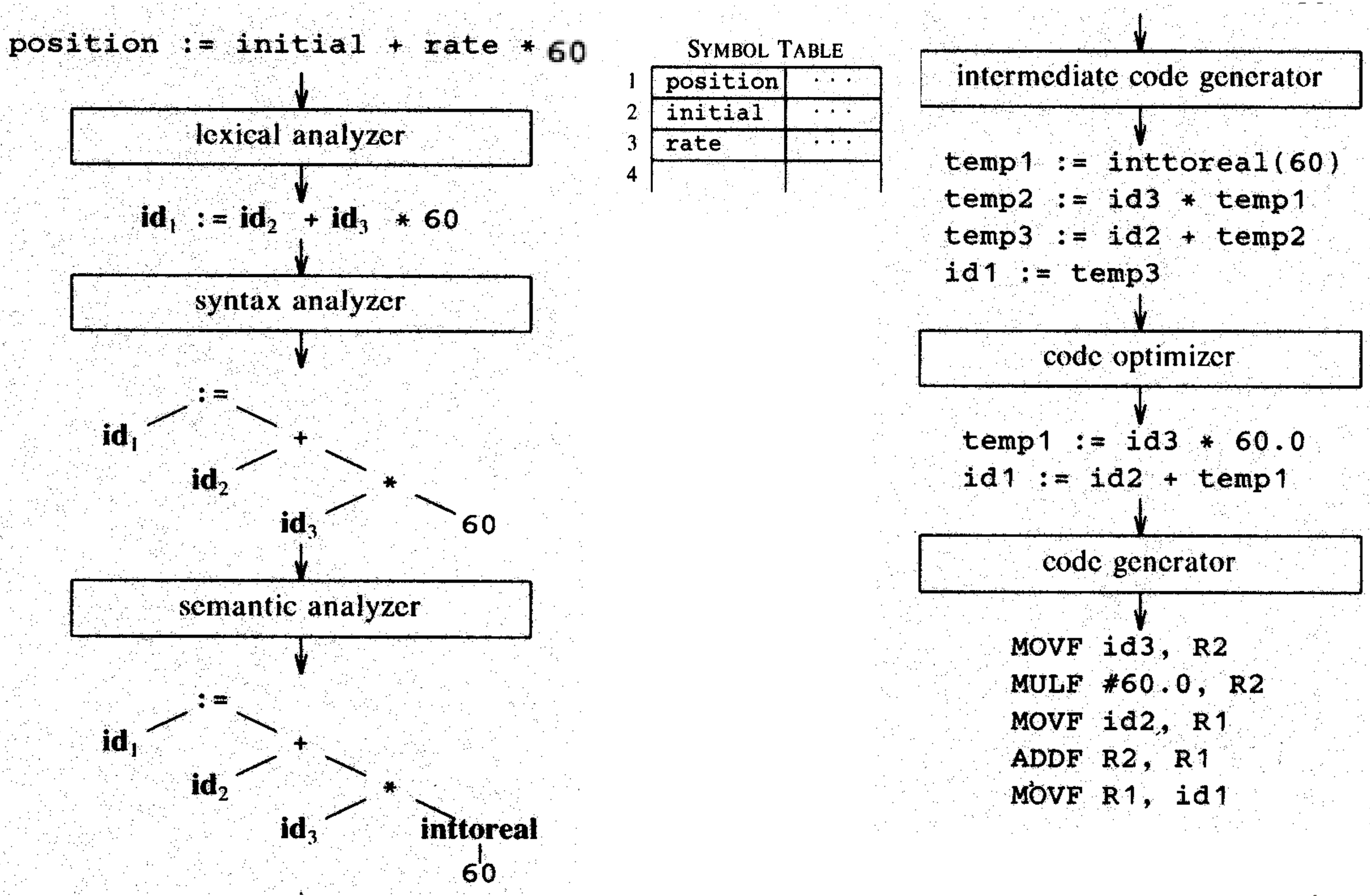
What do we want to do?

- Look at
  - Theory ideas from CSCI 48400
  - Basic compiler ideas
- To see how
  - They work together
  - Enable the construction of real-world compilers

Steps in the compilation process



A very brief example



```
position := initial + rate * 60
```

id_1 := id_2 + id_3 * 60

SYMBOL TABLE

1	position	· · ·
2	initial	· · ·
3	rate	· · ·
4		

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Lexical Analyzer

- Its job is to
  - Identify the discrete strings (lexemes) in the stream of input symbols
  - add them (the lexemes – the actual strings) to the symbol table along with what “kind” of item (token) it is
    - Identifier, arithmetic operator, integer, etc.

How is this done?

- Delimiters play a big role in separating lexemes (whitespace: space, tab, newline)
- You do this when you read
  - In the above sentence, your brain sees blanks as word delimiters
  - "You", "do", "this" etc.

Regular expressions

- Kinds of tokens are defined by regular expressions
- ab*(a | aa) means
  - A single a followed by 0 or more b's followed by 1 or 2 a's
- Concatenation, repetition, or
- Example strings that match (are in the language of) this regular expression
  - aa, aba, aaa, abbbba, abbbaa, ...

Simple Programming Language Tokens

- Alphabet = {f,i,y,0,1,2,*,+,w}
- Letter = f|i|y
- Delimiter = w
- Digit = 0|1|2
- Operator = *|+
- Int = digit(digit)*   [2, 2001]
- Id = letter(letter|digit)* [yy3f02]
- Reserved word = if

Recognizer

- The lexical analyzer must be able to recognize the various kinds of tokens
- How is this done?
- Using a finite state automaton (machine)

What is a FSA?

- A machine that has a finite number of states and a finite input alphabet
- At any moment machine is in a given state, looking at an input symbol
- Machine reads the input symbol, transitions to a new state (or same state)
- It "recognizes" a string if it's in a final state at the end of reading that string

Theorem

- Any language that can be defined by a regular expression has a finite-state machine recognizer.

DFA, NFA

- **Deterministic** fsa (dfa)
  - Every state has one and only one transition for each symbol in the alphabet.
- **Nondeterministic** fsa (nfa)
  - A given state may have
    - no transition for a given symbol
    - multiple transitions for a given symbol
    - a transition on the empty string [no input symbol consumed]
    - recognizes a string if *some* path leads to a final state at the end of reading that string, even if other paths do not lead to a final state

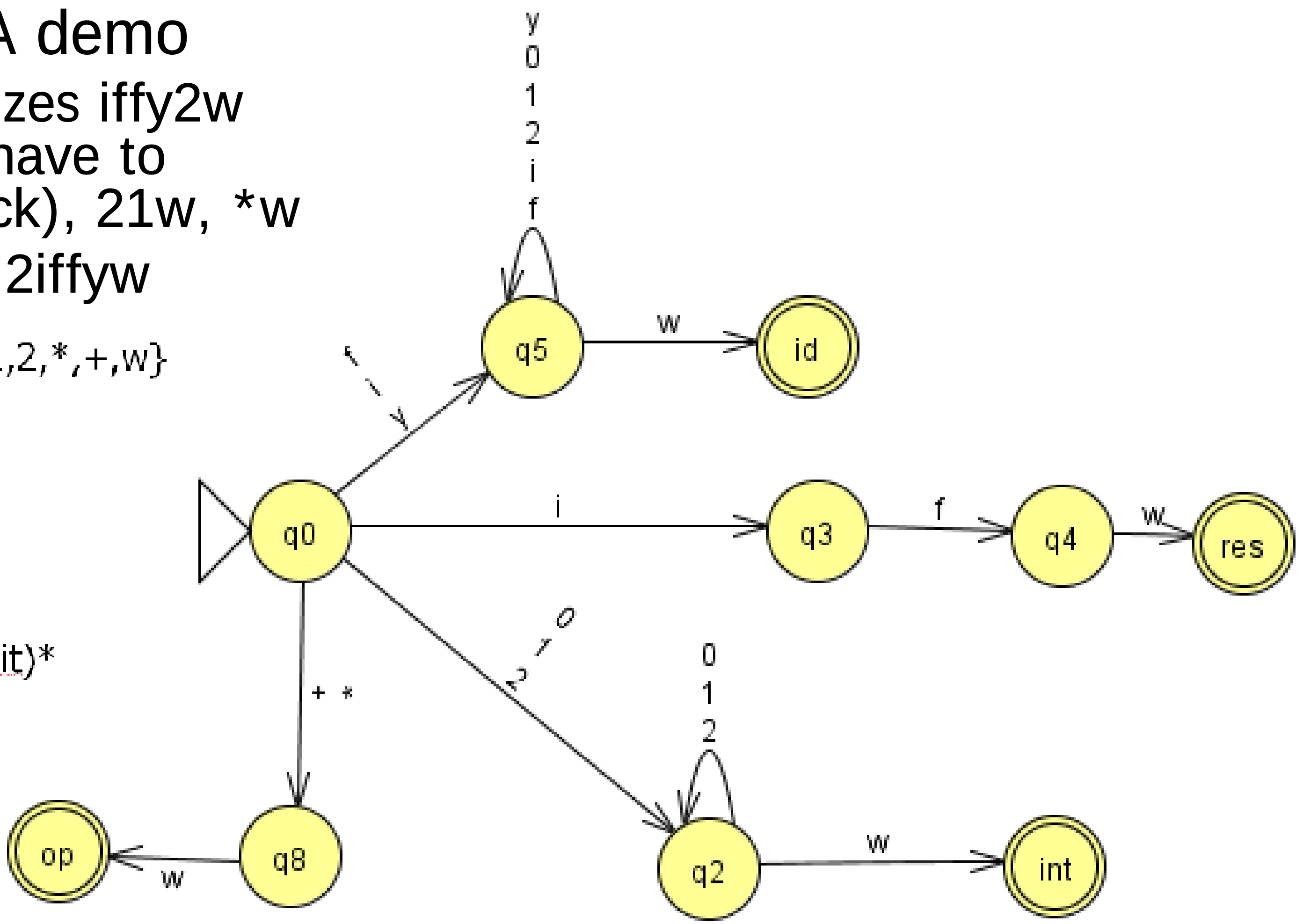
Nondeterminism

- For a word in the language of an nfa
  - Choice of moves, one of which is "correct" (leads to a final state)
  - Think of taking actions in parallel (spawn parallel machines, one of which gets to a final state)
  - Or of starting down a path and having to backtrack
  - You want to avoid this (i.e., the backtracking)

FSA for our language-sort of

- JFLAP FSA demo
  - Recognizes iffy2w (might have to backtrack), 21w, *w
  - Rejects 2iffyw

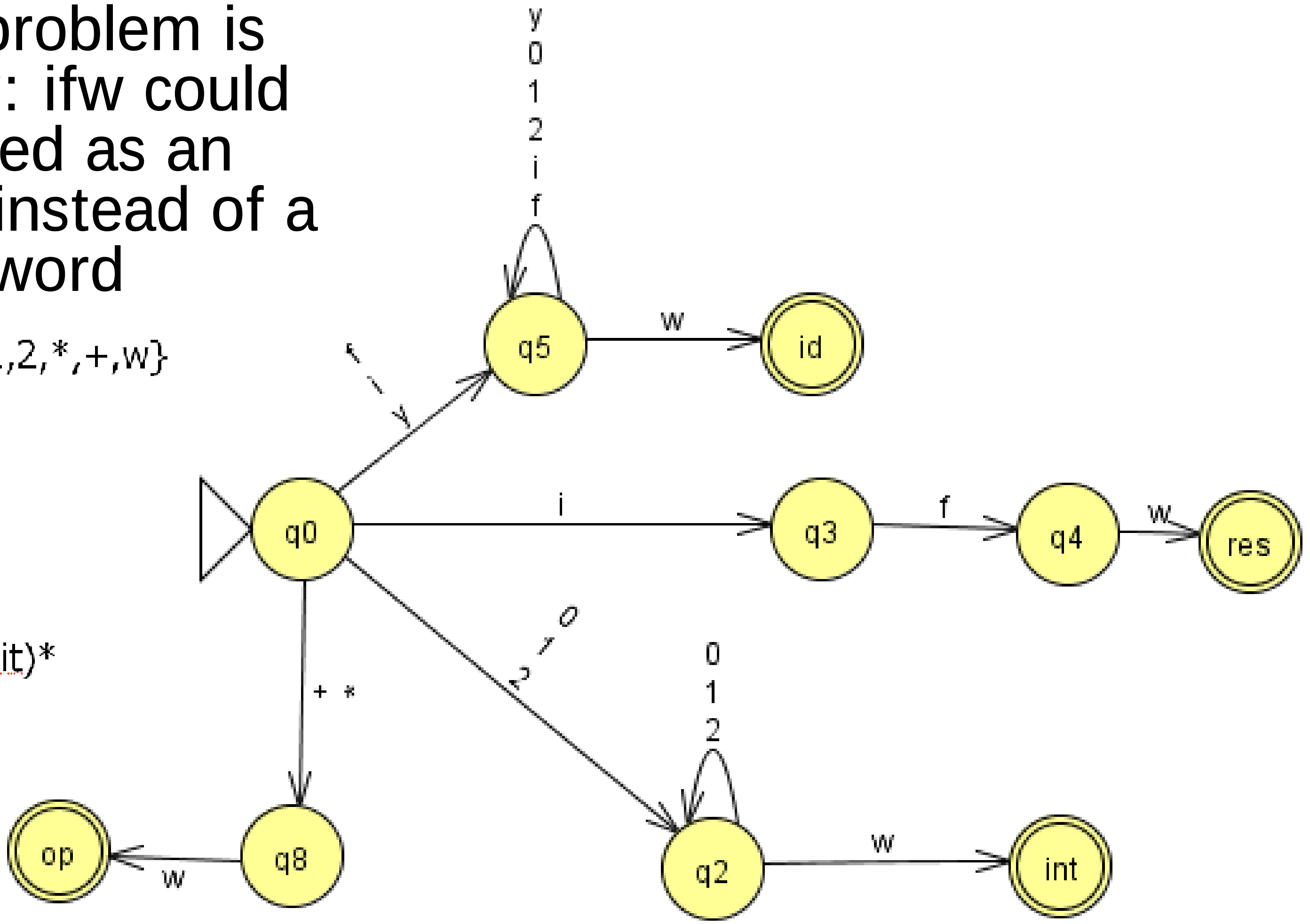
Alphabet = {f,i,y,0,1,2,*,+,w}
Letter = f|i|y
Delimiter = w
Digit = 0|1|2
Operator = *|+
Int = digit(digit)*
Id = letter(letter|digit)*
Reserved word = if



FSA for our language-sort of, 2

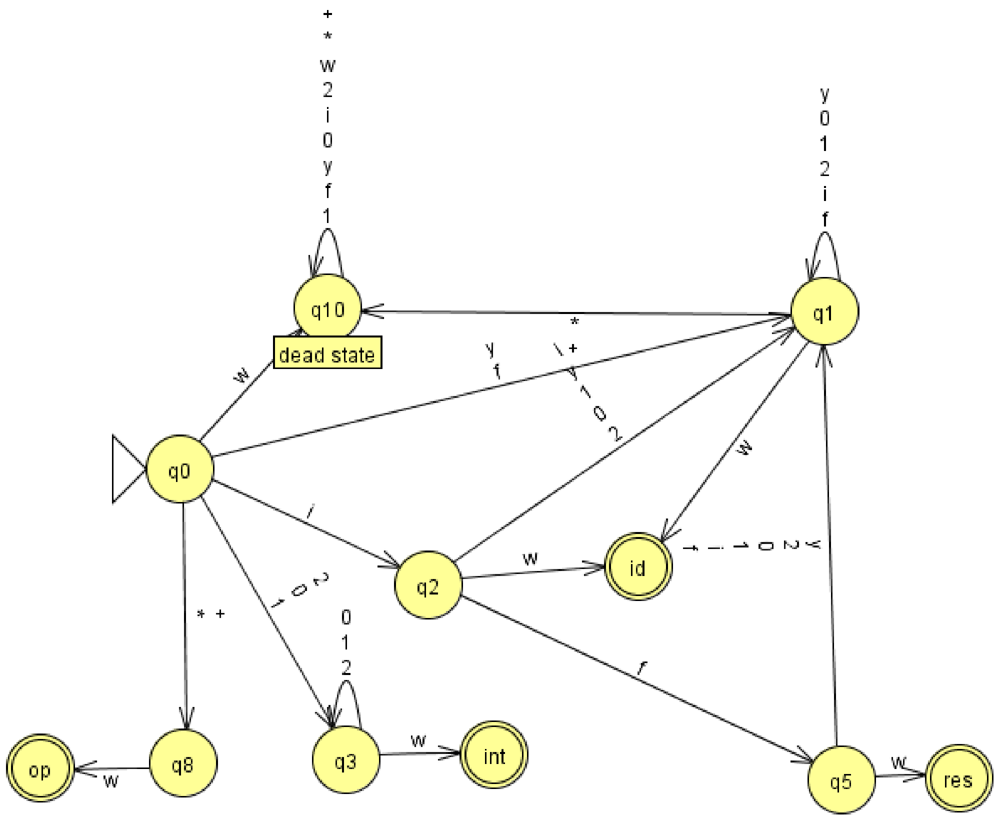
- The real problem is ambiguity: ifw could be classified as an identifier instead of a reserved word

Alphabet = {f,i,y,0,1,2,*,+,w}
Letter = f|i|y
Delimiter = w
Digit = 0|1|2
Operator = *|+
Int = digit(digit)*
Id = letter(letter|digit)*
Reserved word = if



Theorem

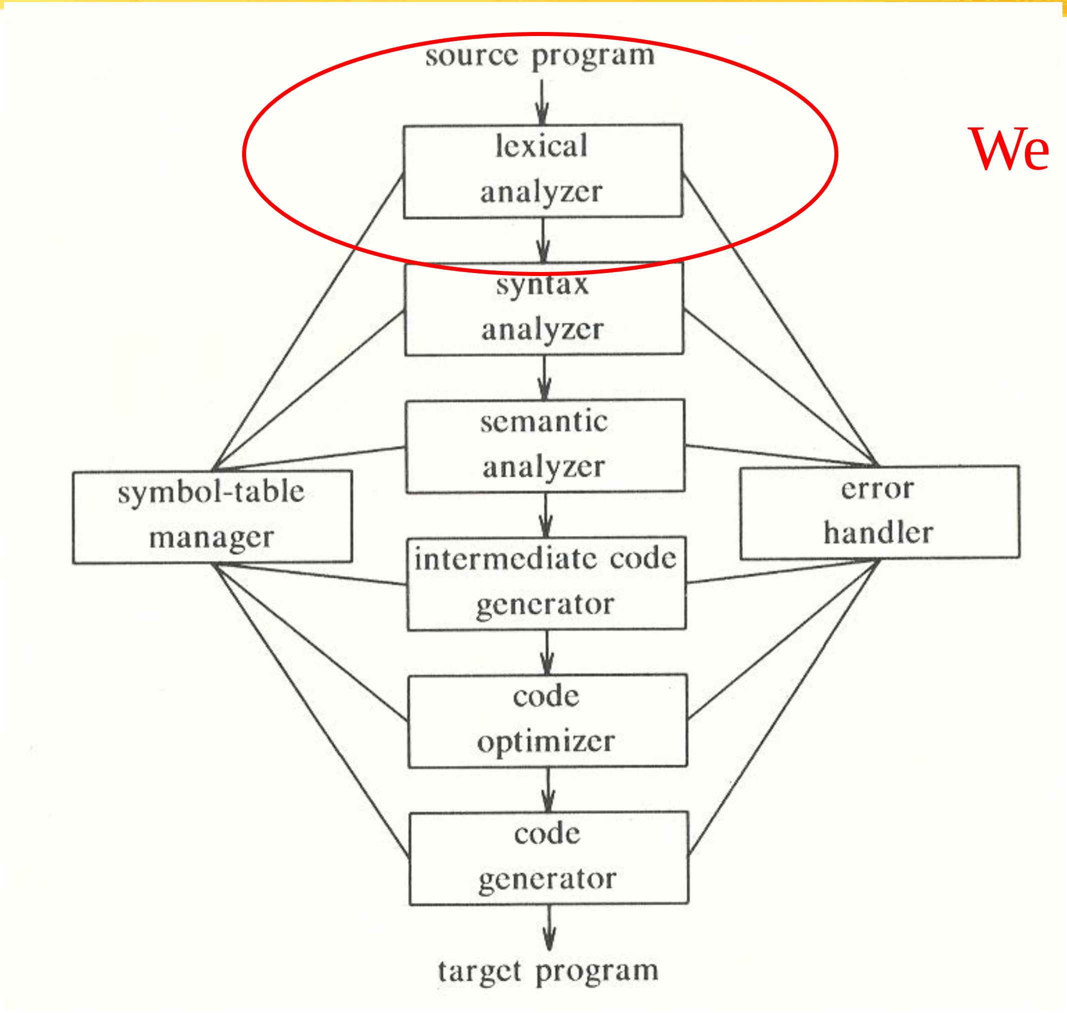
- Any nondeterministic fsa can be turned into an equivalent dfa
- JFLAP dfa for our language (many paths to dead state not shown); ifw has a unique path to a final state



At this point

- A dfa exists to recognize the lexemes and classify them as various kinds of tokens
- For each, the lexeme is put into the symbol table with an indication of the kind of token it is
- Lexical Analysis phase is complete

Steps in the compilation process



We are here

lex builds the dfa for you based on a stylized input language

Manifest Constants

```
#define      IF          308
#define      THEN        309
#define      ELSE        310
#define      ENDIF       311
```

Regular expressions

```
          /* regular expression definitions */

comment         "//".*
delimiter       [ \t\n]
whitespace      {delimiter}+
letter          [a-zA-Z]
digit           [0-9]
identifier      {letter}({letter}|{digit})*
integer         {digit}+
ascii_char      [^\"\n]
escaped_char    \\n
text_string      ({ascii_char}|{escaped_char})*
text            \"{text_string}\"
```

Code (simple) similar to that generated by lex based on this information.

```
case digit;
    addChar();
    getChar();
    while (charClass == digit) {
        addChar();
        getChar();
    }
return INT;
break;
```

(this is part of the dfa)

lex, 2

Pattern / action statements (what to send back to the syntax analyzer)

```
%%

{comment}     {             /* ignore comments */              }
{whitespace}  {             /* ignore whitespace */            }
{integer}     {if(read_sym(yytext, n_scope, 'L') < 0)
                Flag = create(yytext, n_scope, 'I', 'L',
                              atoi(yytext)); return(INTEGER);}
{text}        {if(read_sym(yytext, n_scope, 'L') < 0)
                Flag = create(yytext, n_scope, 'S', 'L', -1);
                          return(TEXT);    }
funct         { return(FUNCT);                                 }
"="           { return(ASSGN);                                 }
print         { return(PRINT);                                 }
input         { return(INPUT);                                 }
return        { return(RETURN);                                }
if            { return(IF);                                    }
then          { return(THEN);                                  }
else          { return(ELSE);                                  }
endif         { return(ENDIF);                                 }
while         { return(WHILE);                                 }
do            { return(DO);                                    }
enddo         { return(ENDDO);                                 }
int           { return(INT);                                   }

end           {return(E_O_F);  /* to shut down test */         }

{identifier}   { if(read_sym(yytext, n_scope, 'I') < 0)
                Flag = create(yytext, n_scope, 'I', 'I', -1);
                 return(IDENTIFIER);                             }

.             { return(yytext[0]);    /* pass back single characters */ }
%%
```

⌘ **If the integer discovered is not in the symbol table, enter it now, then tell the syntax analyzer what token was just found.**

The action between lexical analyzer and syntax analyzer is a series of back-and-forth communication, it's not that all lexical analysis for the whole program gets done first.

Syntax Analyzer

- The syntax analyzer must see that the strings of tokens satisfy the rules of formal grammar that define the programming language.

Formal Grammar

- A grammar consists of variables (things that can be replaced) and terminals (things that are not replaced)
- There are productions of the form
  - $\alpha \rightarrow \beta$
  - whenever α is encountered, can be replaced by β
- There is a start variable

Language of a grammar

- Set of all strings of terminals that can be derived from start symbol using productions in the grammar

Example Grammar G

- Variables = S, A, B Terminals = a, b
- Productions :
  - $S \rightarrow AB$
  - $A \rightarrow aA \mid B$
  - $B \rightarrow b$
- A derivation:

  $S \rightarrow AB \rightarrow aAB \rightarrow aaAB \rightarrow aaaAB \rightarrow aaaBB \rightarrow aaabB \rightarrow aaabb$
- So aaabb belongs to L(G)
- We can see that L(G) = a*bb
  - Not always easy

Classes of Grammars

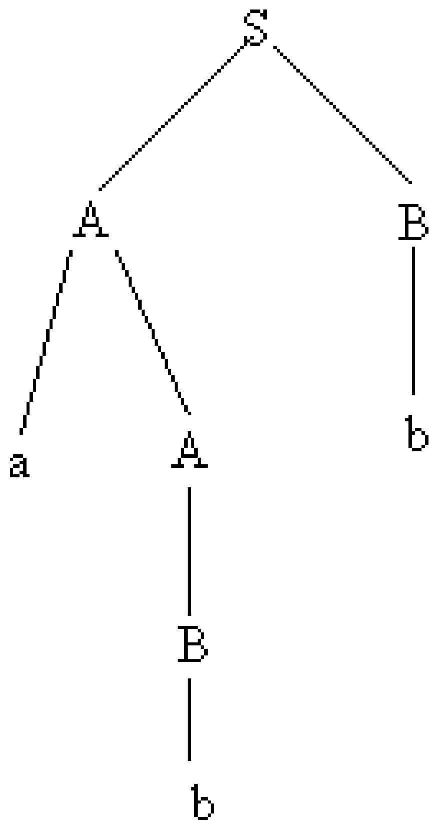
- Depends on what kinds of productions they have
- Context-free grammars
  - Each production has only a single variable on the left side, hence substitution is "context-free"
  - This is the type of grammar used for programming languages

Syntax Analyzer

- Given a string of tokens (the output of the lexical analyzer) and a grammar G
- "Parse" the string to see if it belongs to L(G)
- The overall job of the syntax analyzer is to consider the program as a whole as a string of tokens (sentence in the grammar)
- But this is accomplished piece by piece

Parse Tree for abb

- $S \to AB$
- $A \to aA \mid B$
- $B \to b$



Two possibilities

- Top down parsing (derivation)
  - Start with the start symbol, work down the tree to the string in question (if possible)
- Bottom-up parsing (reduction)
  - Start with the string in question, work up the tree to the start symbol (if possible)
- Compiler must do this algorithmically

Recognizer

- The syntax analyzer must be able to recognize legitimate strings in the language
- How is this done?
- Using a pushdown automaton

What is a pda?

- A machine that has a finite number of states and a finite input alphabet and a stack
- At any moment machine is in a given state, looking at an input symbol and a top-of-stack symbol
- Machine reads the input symbol, transitions to a new state (or same state) and either pops the top stack symbol or replaces it with something
- It “recognizes” a string if it’s in a final state at the end of reading that string

Theorem

- Any language that can be generated by a context-free grammar can be recognized by a pda. (Our Theorem 7.1)

  This is a key point in all of this. Without this “compilers” would have to be hand-crafted.

Derivations (top down)

- $E \rightarrow E+E \mid E \rightarrow E*E \mid E \rightarrow i$
- $i*i+i$ is in L(G)
- Leftmost (LL) Rightmost

$E \rightarrow E+E$		$E \rightarrow E+E$
$\rightarrow E*E+E$		$\rightarrow E+i$
$\rightarrow i*E+E$		$\rightarrow E*E+i$
$\rightarrow i*i+E$		$\rightarrow E*i+i$
$\rightarrow i*i+i$	(end with the “code”)	$\rightarrow i*i+i$

- LL means scan Left to right, substitute Leftmost

Reduction (bottom up)

- (Here is the grammar again)
  $E \rightarrow E+E \mid E \rightarrow E*E \mid E \rightarrow i$
- This is LR

  i*i+i   (start with the “code”)

  E*i+i

  E*E+i

  E+i

  E+E

  E   (end with the start symbol)
- The "R" meaning that this is the reverse of rightmost top-down (see previous slide)

Predictive parsing

- Top-down parsing – what production to use?
- We want LL(k) for some k – looking at the next k symbols of the input string (part of the right side of a production) determines the production

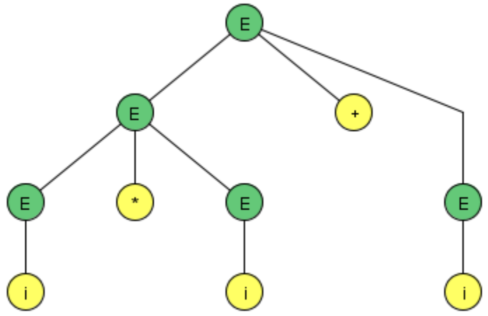
Ambiguity

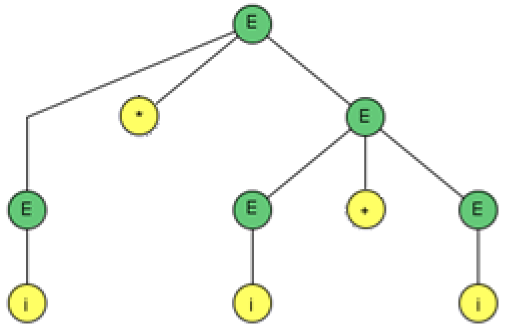
- But our grammar is ambiguous: i*i+i
  - E → E+E | E →E*E | E → i

This one is correct by

operator precedence,

(which the grammar knows nothing about)





Ambiguity, 2

- Ambiguity $\longrightarrow$ G is not LL(1)

  (And in fact not LL(k) for any k)
- The converse, not LL(1) $\longrightarrow$ ambiguity,

  is false
- G is S $\longrightarrow$ aS | ab, L = a*ab
- G is not LL(1) but it is not ambiguous

  Looking at an a, you can't tell which production to use; it is LL(2).

Foiled!

- For our example grammar, there is no hope for predictive parsing.
- But we can modify the grammar.

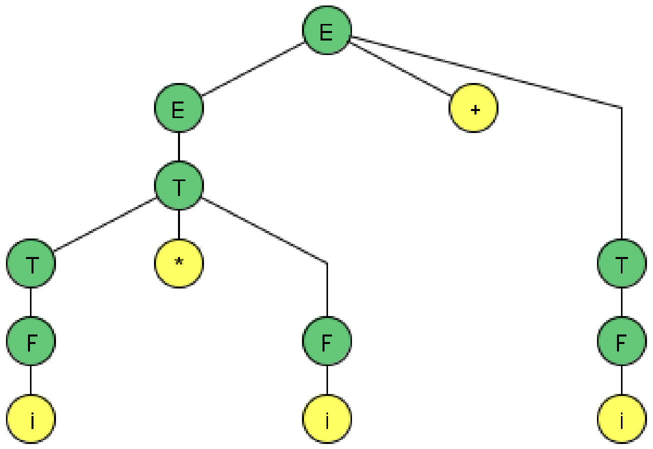
JFLAP parse tree

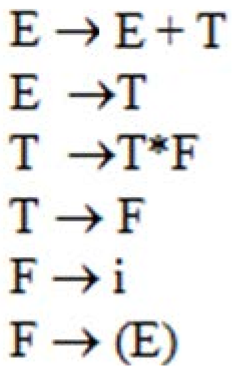
- Expand the grammar
- Derive i*i+i

This is “the” typical example

- Now no ambiguity

This is “the” example grammar in compiler theory
Expression, Term (involved with +), Factor (involved with *)

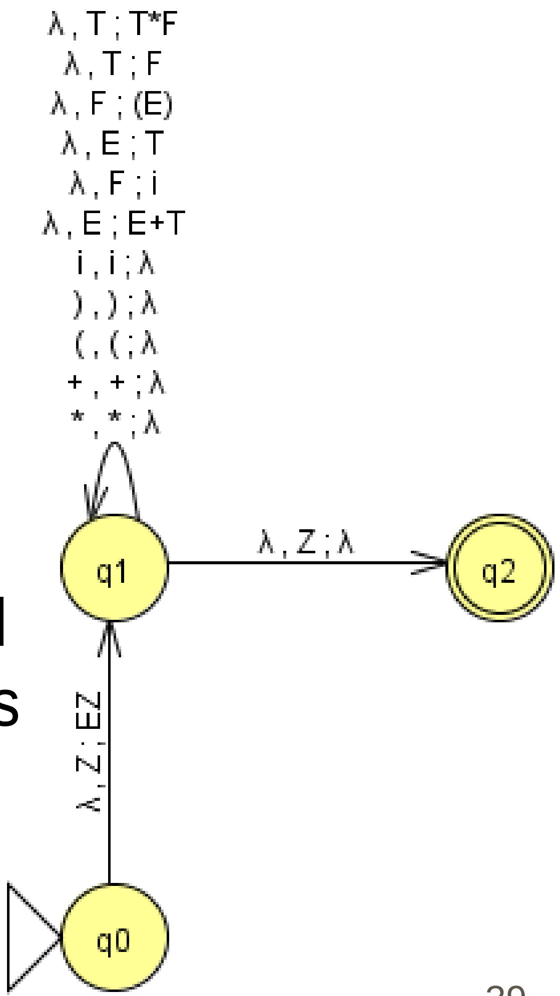




$E \to E + T$
$E \to T$
$T \to T{*}F$
$T \to F$
$F \to i$
$F \to (E)$

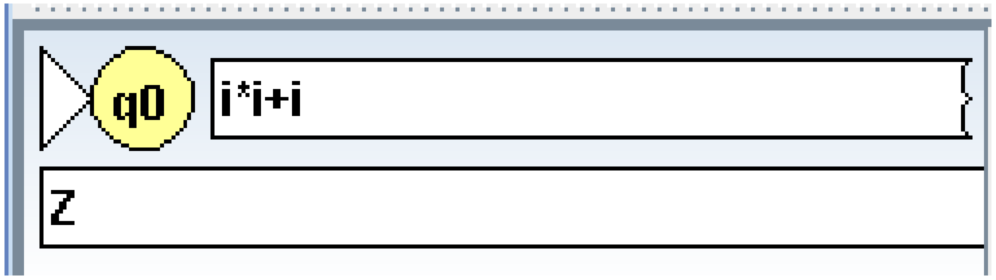
JFLAP LL pda (top down)

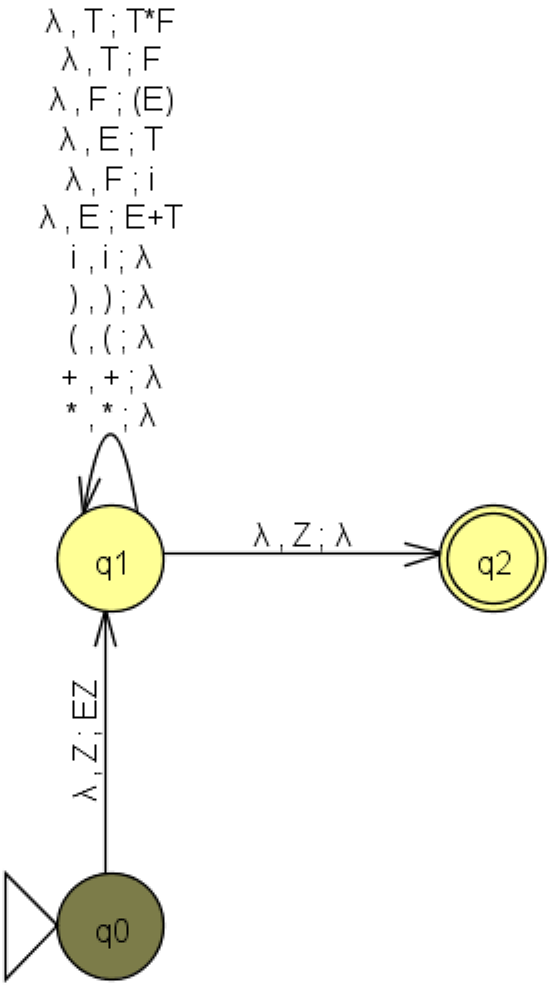
- Start state, final state
- (input, stack top; stack action)
- q0 transition ignores input symbol, puts start symbol on stack
- The top 6 q1 transitions represent the 6 productions in G
  - For example, replace E with E + T
- Bottom 5 q1 transitions pop the stack when it matches input terminal
- Transition to final state when stack is empty



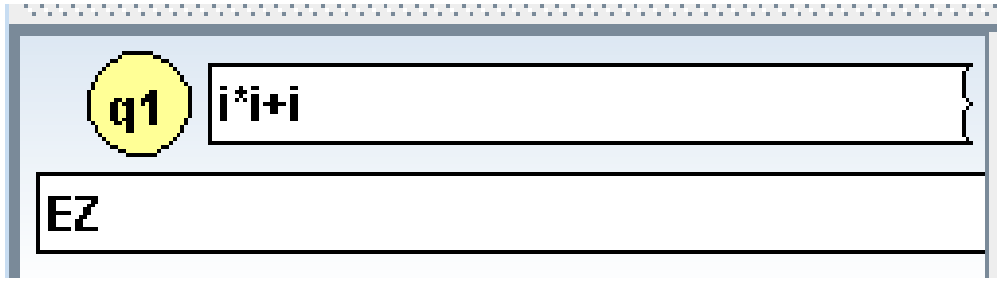
JFLAP LL pda Demo

- Run i*i+i
- Paths proliferate wildly
- If you stare hard enough you can pick out the one “right” path
- Paths come to dead ends if no move possible



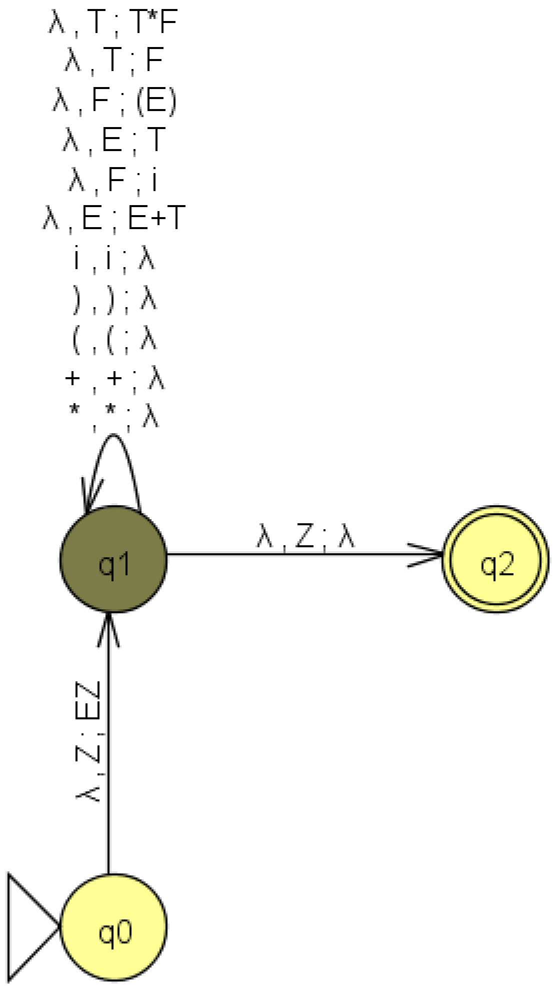


⌘Z is stack bottom symbol, need to add E

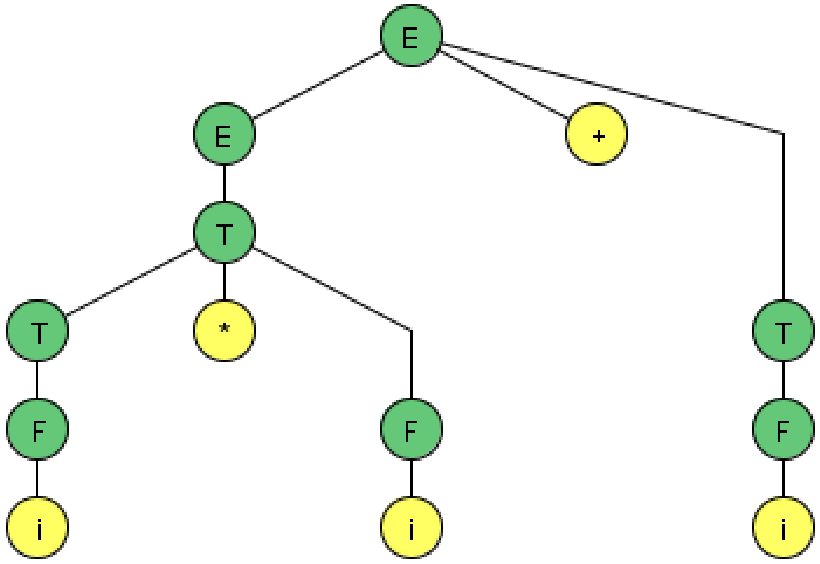


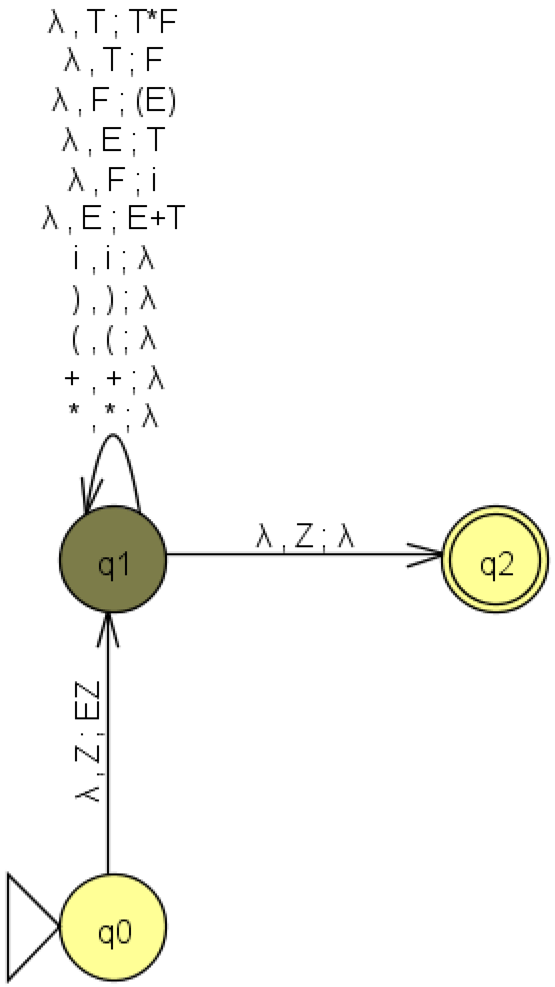
i*i+i

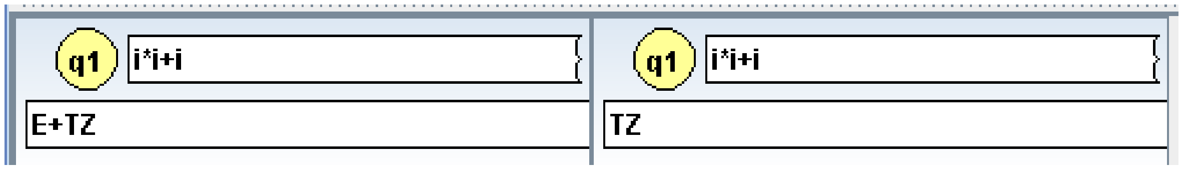
EZ



⌘E has been added to stack, now in q1







q1 i*i+i

E+TZ

q1 i*i+i

TZ

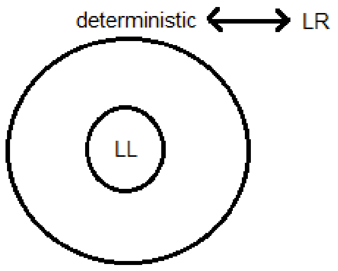
- ⌘ Multiple paths for 2 applicable productions – which is correct? No input symbols have been read yet.

Shift-reduce parsing

- This is bottom-up
- We want LR(k) for some k
  - Recognize the entire right side of a production given k input symbols [easier than LL(k)]
  - Want k to be 1
    (k = 1 means you can determine the next “step" based on the current input symbol)

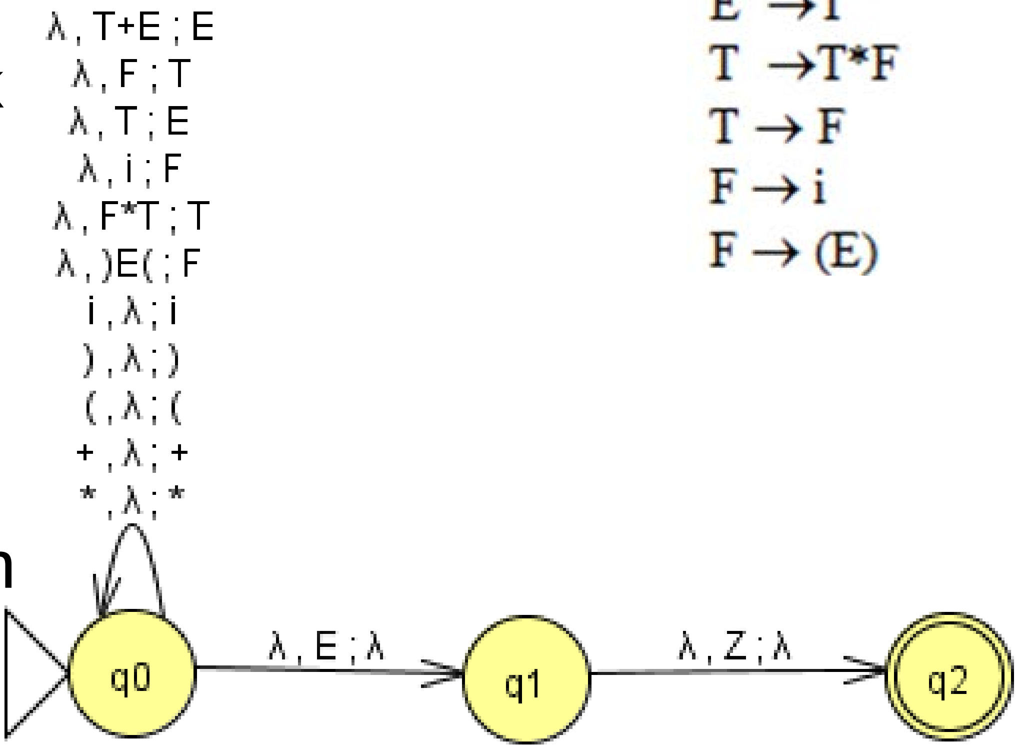
Aside

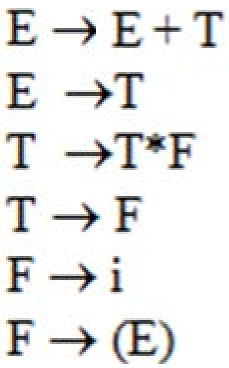
- A language that has an LL grammar is deterministic, but not conversely.
- Languages with LR grammars coincide with deterministic languages, so LR is a more powerful property
  - Some languages have LR grammars but not LL grammars



JFLAP LR pda (Bottom up)

- Bottom 5 q0 transitions put the input string on the stack, ignoring stack top
- Top 6 q0 transitions ignore input and replace **reverse** of right side of production with left side
- When E is on the stack, transition to q1 and then final state to **accept**





$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

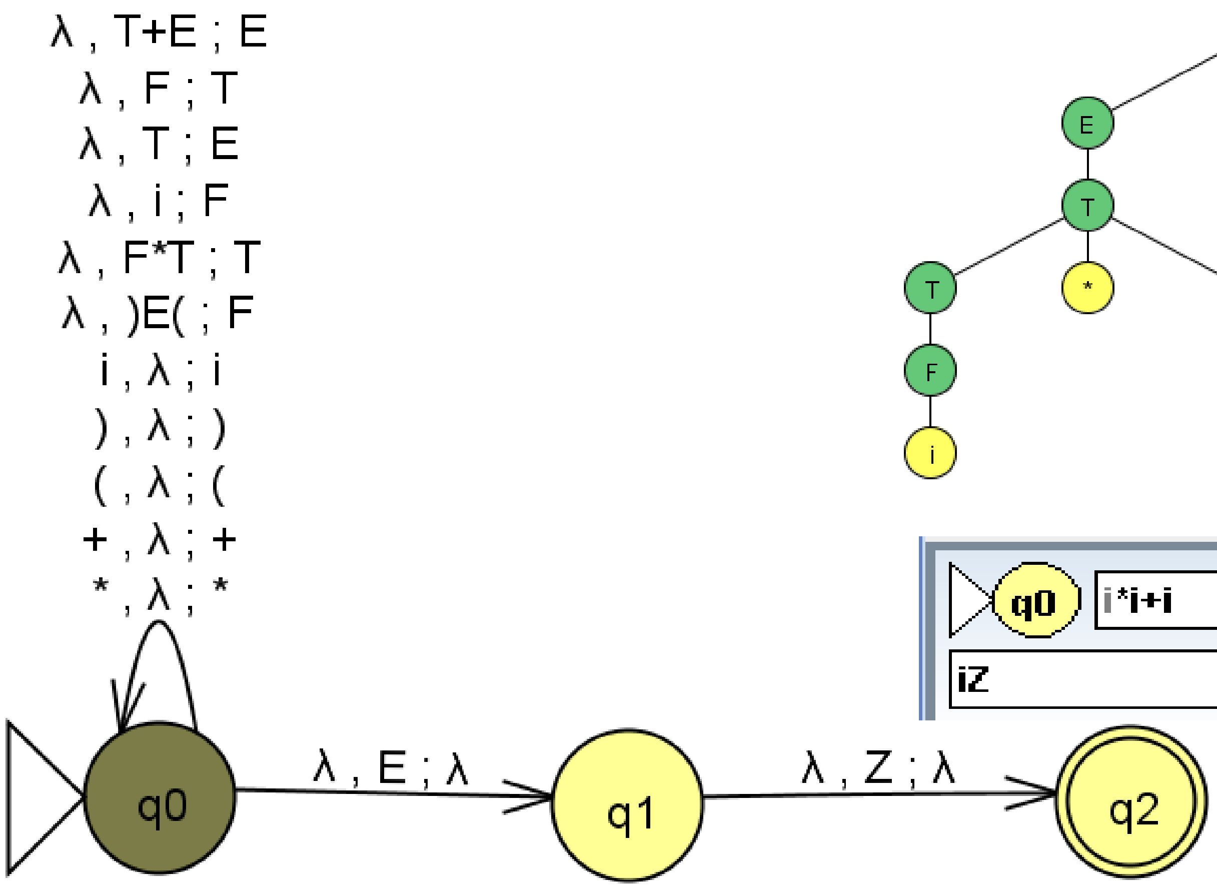
$T \rightarrow F$

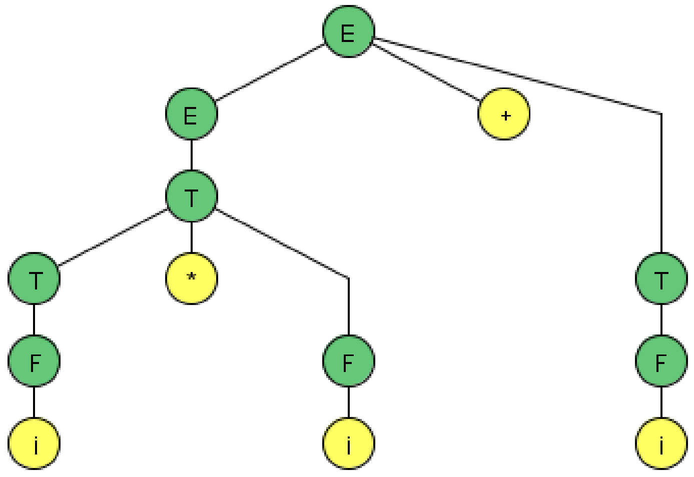
$F \rightarrow i$

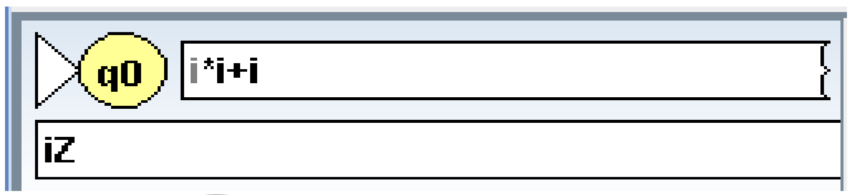
$F \rightarrow (E)$

JFLAP LR pda Demo

- Run i*i+i



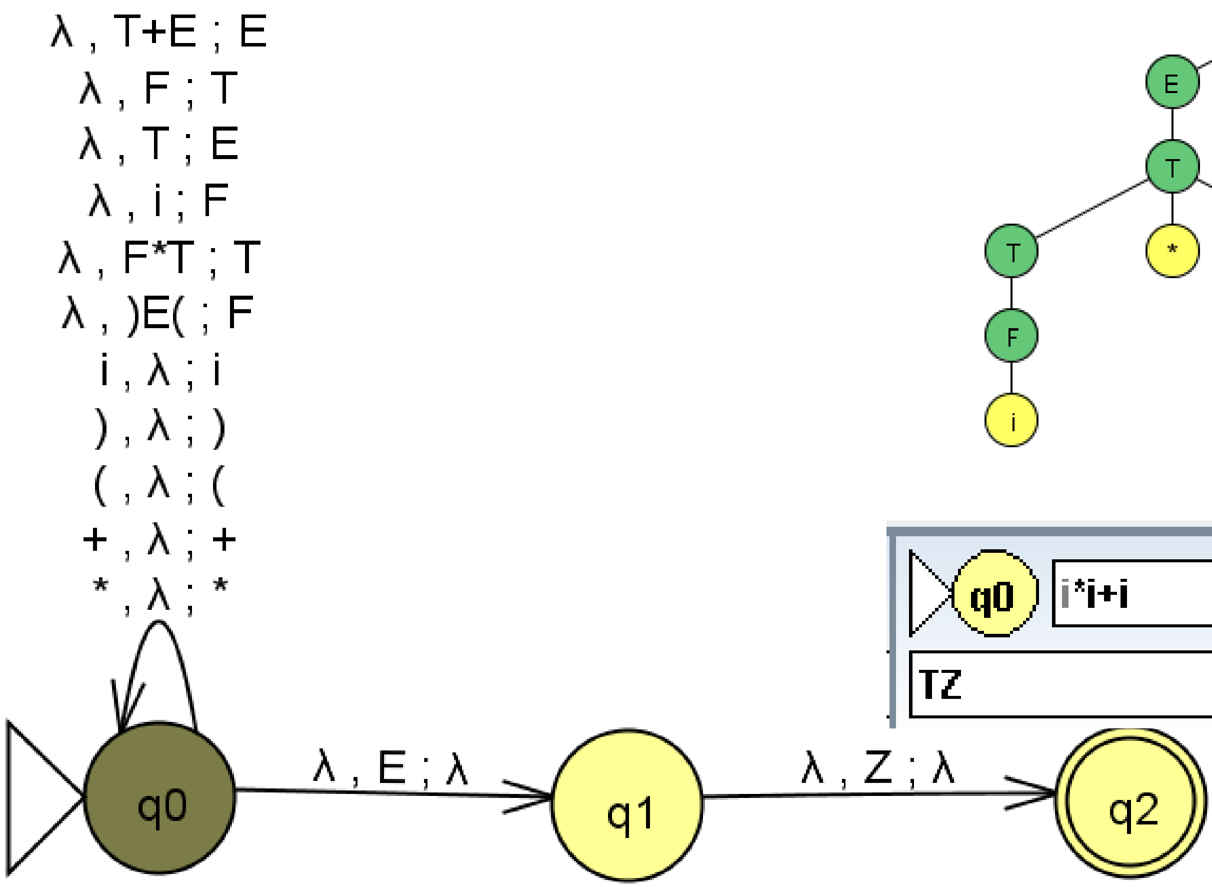


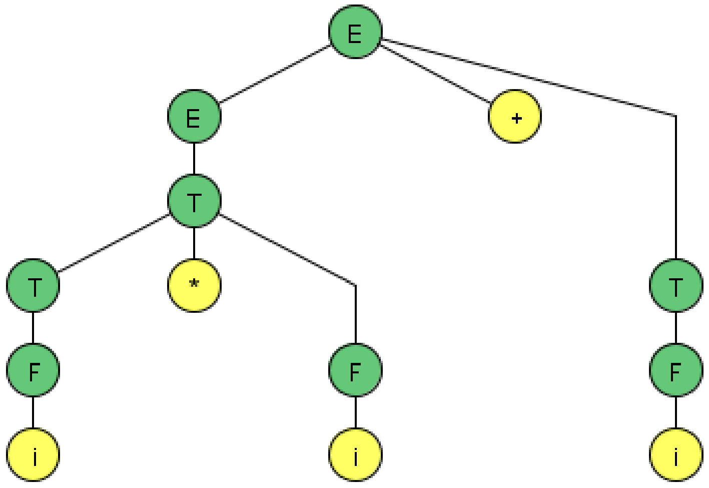


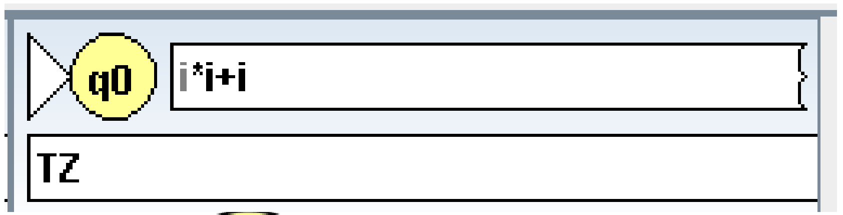
i*i+i

iZ

- ⌘ Stack the first token (supplied at this point by lexical analyzer), one input symbol has been consumed



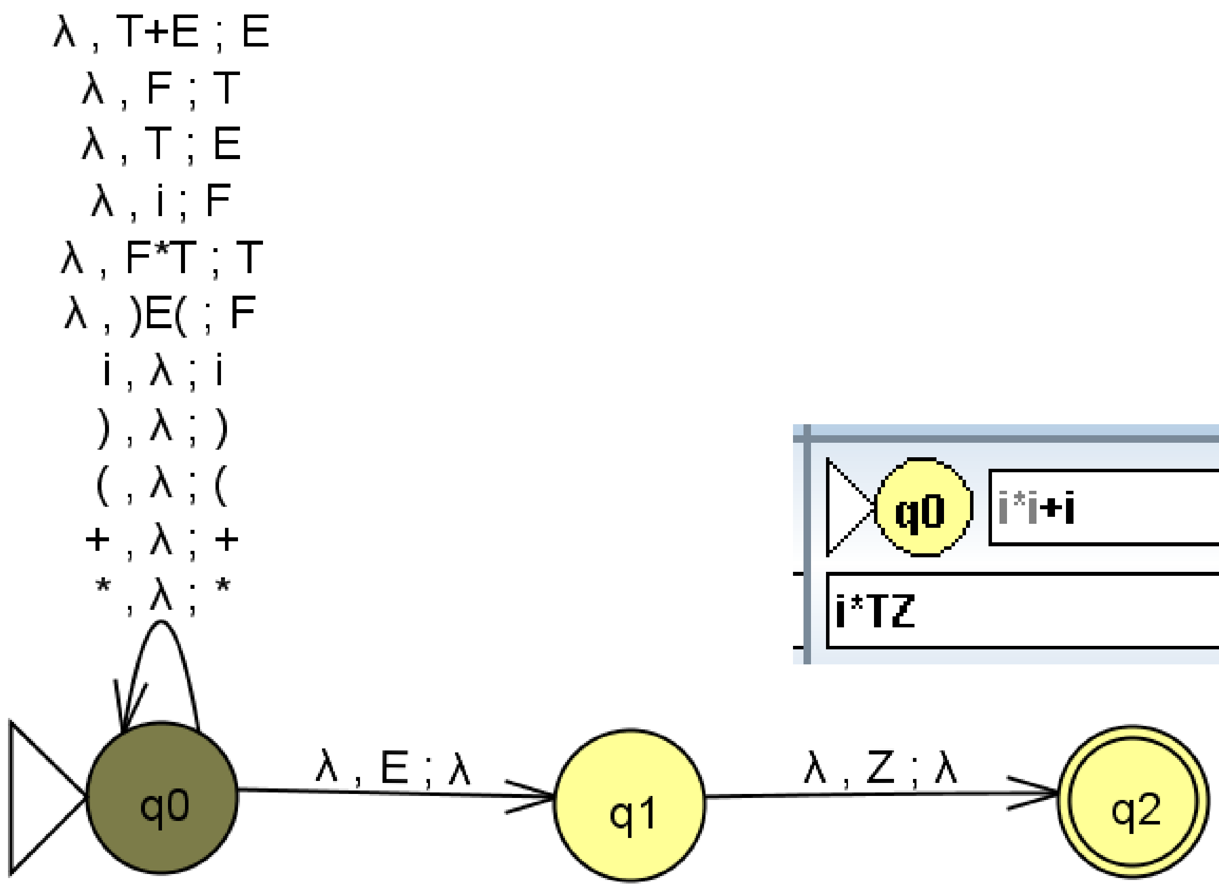


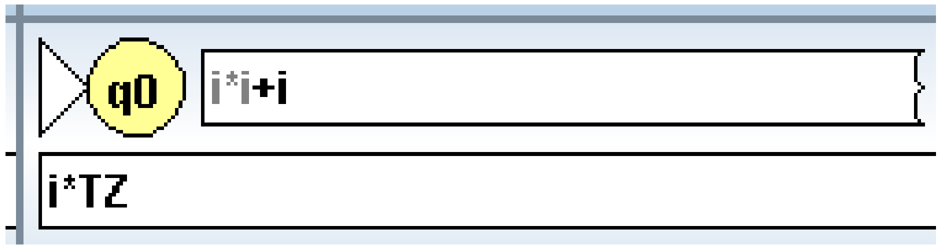

q0

i*i+i

TZ

⌘ Again, multiple paths – replace i with F or stack *. Correct path (shown) has traveled up left branch of parse tree (the last thing done in top-down rightmost derivation, so the first thing done here), i to F to T

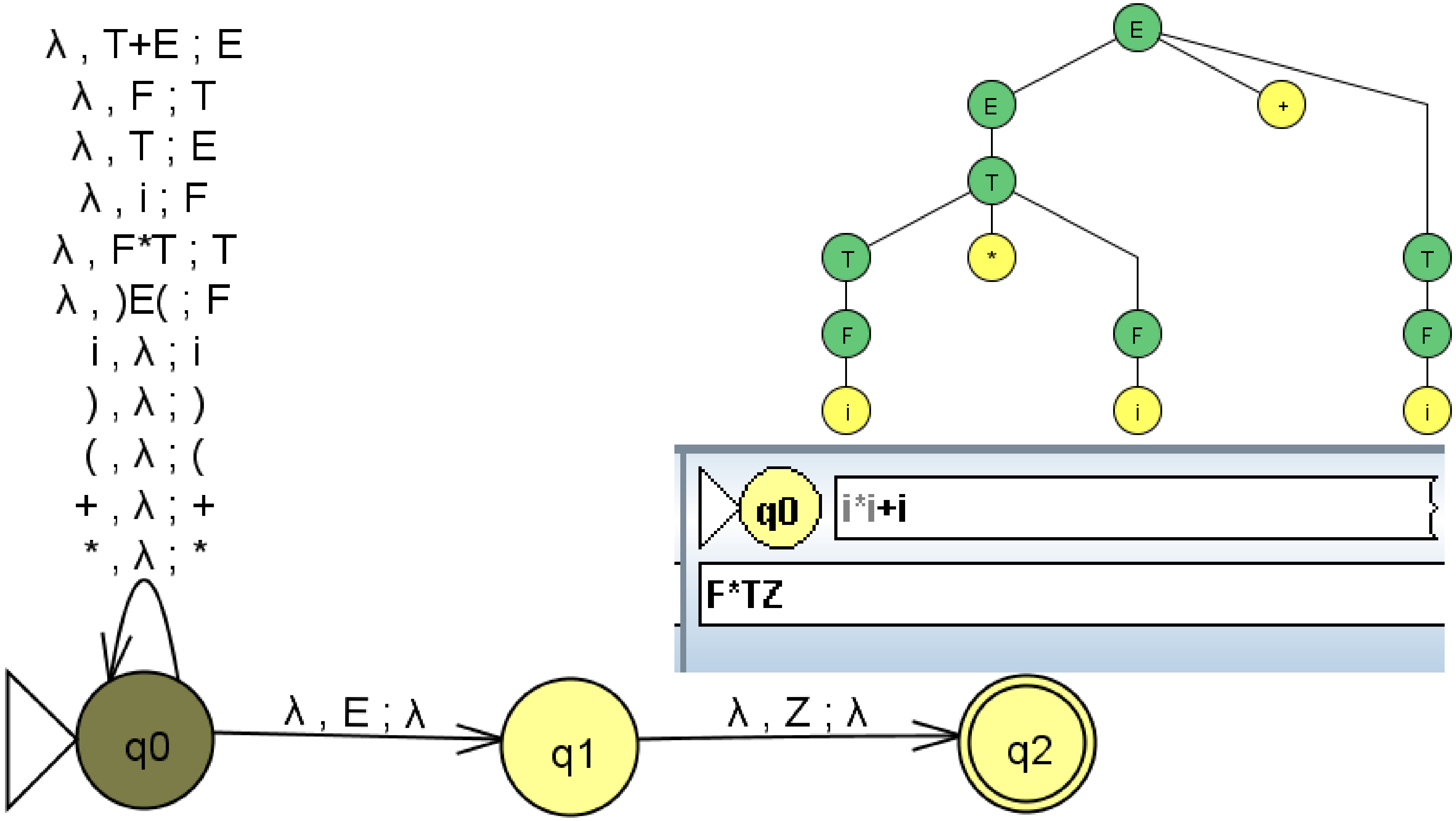




i*i+i

i*TZ

⌘ Two more input symbols consumed and stacked



q0 i*i+i

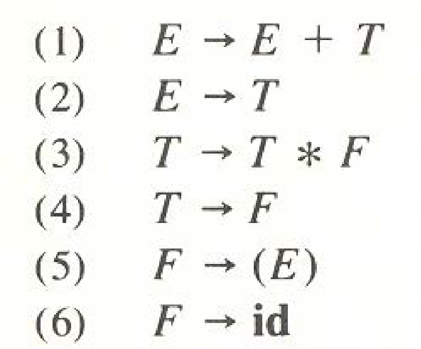
F*TZ

- Ready to replace F*T with T, eventually replace T+E with E, on to state q1 and then q2 to accept

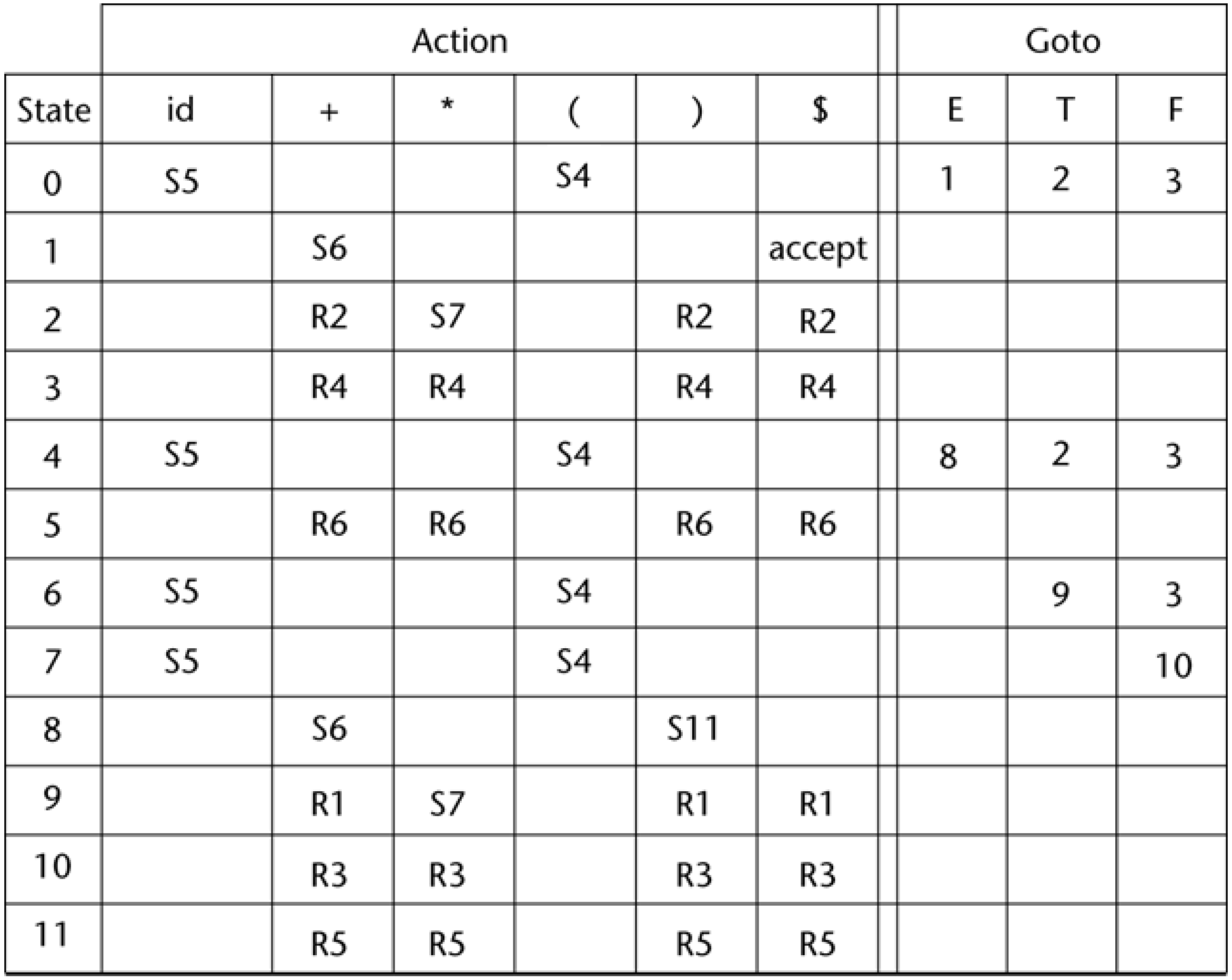
How to make LR(1)?

- So our npda did accept the word in a bottom up parse, but had many false starts.
- How can we make this LR(1)?
- By making use of the fact that * involves F while + involves T

LR Parsing Table



(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \mathbf{id}$



	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

There is an algorithm that will build the parse table for you (with k = 1)

S**n** – shift, then push state **n** on stack

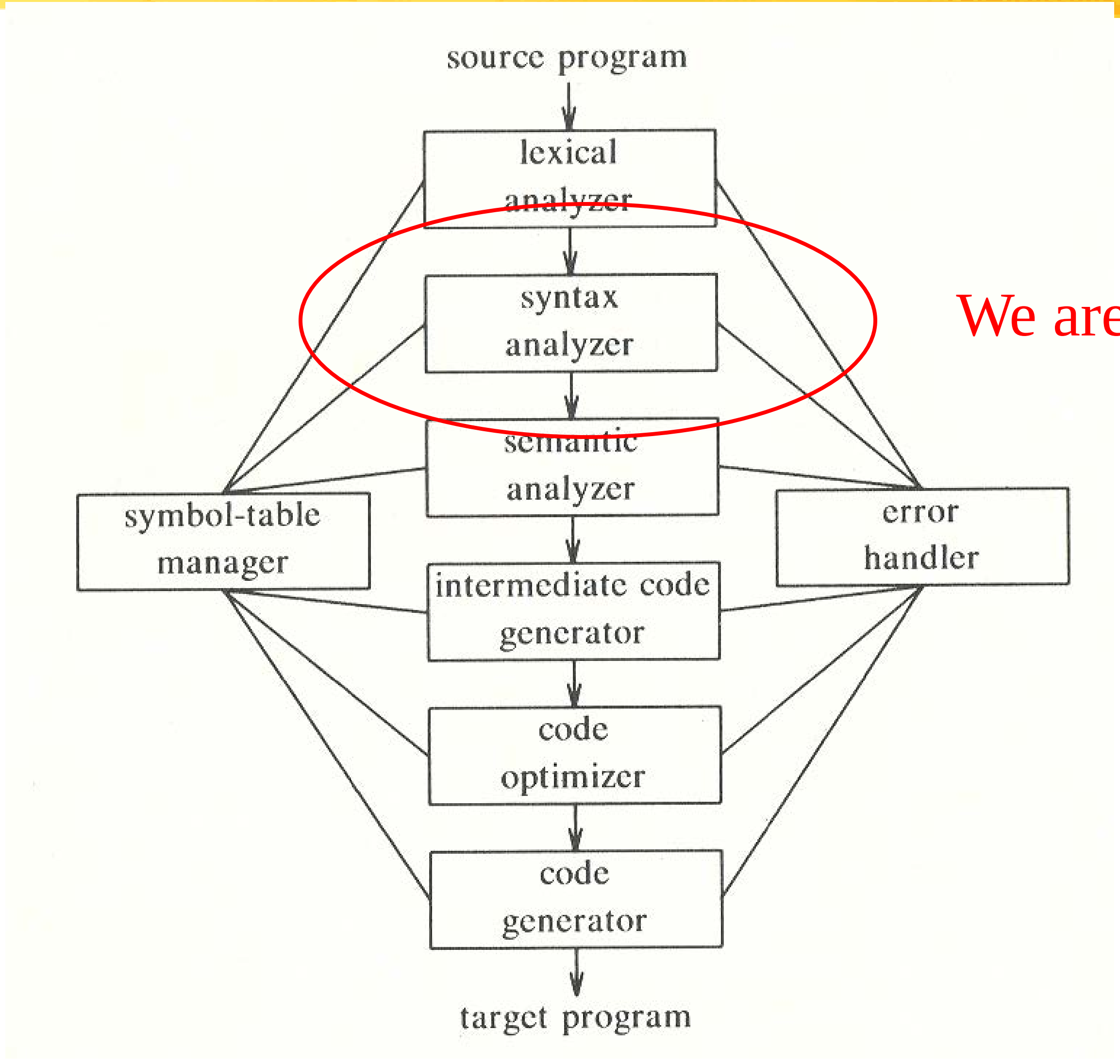
R**n** – reduce by (**n**) then goto(s,**A**) - here **A** is E | T | F - then push state **s** on stack

Result of the parse Example-actions are the substitutions we saw in LR npda stack

	Stack	Input	Action
(1)	0	**id * id + id $**	Shift 5
(2)	0 id 5	*** id + id $**	Reduce by (6) goto(0,F)
(3)	0 F 3	*** id + id $**	Reduce by (4) goto(0,T)
(4)	0 T 2	*** id + id $**	Shift 7
(5)	0 T 2 * 7	**id + id $**	Shift 5
(6)	0 T 2 * 7 id 5	**+ id $**	Reduce by (6) goto(7,F)
(7)	0 T 2 * 7 F 10	**+ id $**	Reduce by (3) goto(0,T)
(8)	0 T 2	**+ id $**	Reduce by (2) goto(0,E)
(9)	0 E 1	**+ id $**	Shift 6
(10)	0 E 1 + 6	**id $**	Shift 5
(11)	0 E 1 + 6 id 5	**$**	Reduce by (6) goto(6,F)
(12)	0 E 1 + 6 F 3	**$**	Reduce by (4) goto(6,T)
(13)	0 E 1 + 6 T 9	**$**	Reduce by (1) goto(0,E)
(14)	0 E 1	**$**	**Accept**
(15)			

On reduction, you have found the "handle" (right-hand side) of a production – if there is one, it will be on the top of the stack, else syntax error.

Steps in the compilation process



We are here

yacc – builds the LR parser for you

Statement production

```
statement          : assgn_statement
                   | return_statement
                   | print_statement
                   | input_statement
                   | null_statement
                   | if_statement
                   | while_statement
                   | block
                   | error
                     {
                       error("Bad statement syntax.");
                       $$ = Null;
                     }
                   ;
```

Expression Production

Code generators

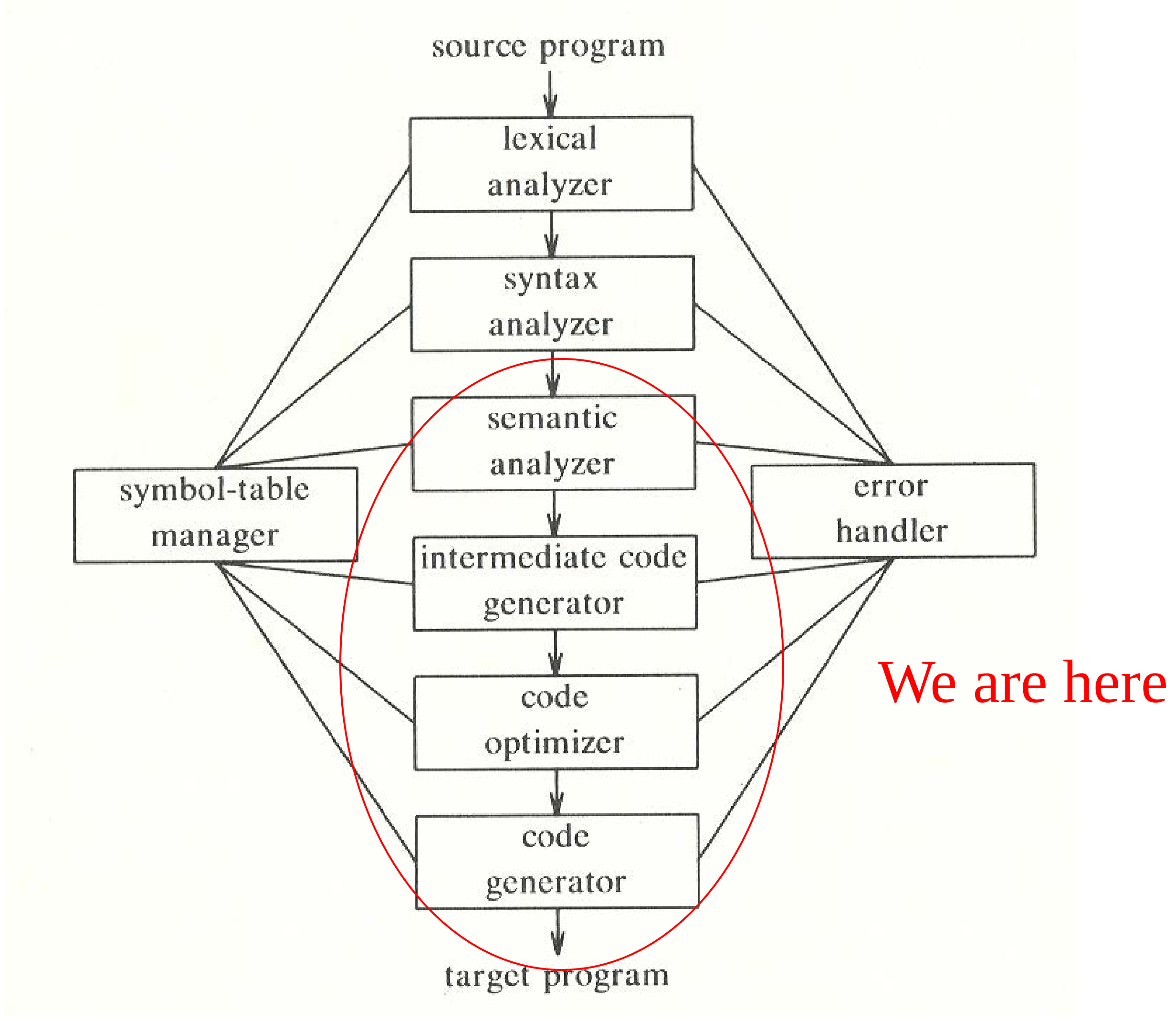
```
expression       : expression '+' expression
                   { $$ = do_bin(TAC_ADD, $1, $3); }
                 | expression '-' expression
                   { $$ = do_bin(TAC_SUB, $1, $3); }
                 | expression '*' expression
                   { $$ = do_bin(TAC_MUL, $1, $3); }
                 | expression '/' expression
                   { $$ = do_bin(TAC_DIV, $1, $3); }
                 | '(' expression ')'
                   { $$ = $2; }
                 | INTEGER
                   {
                     $$ = make_enode(Null, $1, Null);
                   }
                 | IDENTIFIER
                   {
                    if(GetUsage($1) != 'I')
                       {
                        error("Undelcared identifier in expression");
                        $$ = make_enode(Null,create("0", num_scopes, 'I', 'L', Null),
                                                Null);
                       }
                     else
                        $$ = make_enode(Null, $1, Null);
                   }
                 | IDENTIFIER '(' argument_list ')'
                   { $$ = do_fnap($1, $3); }
                 | error
                   {
                     error("Bad expression syntax.");
                     $$ = make_enode(Null, Null, Null);
                   }
                 ;
```

Code for syntax analysis is much more difficult to write than code for lexical analysis

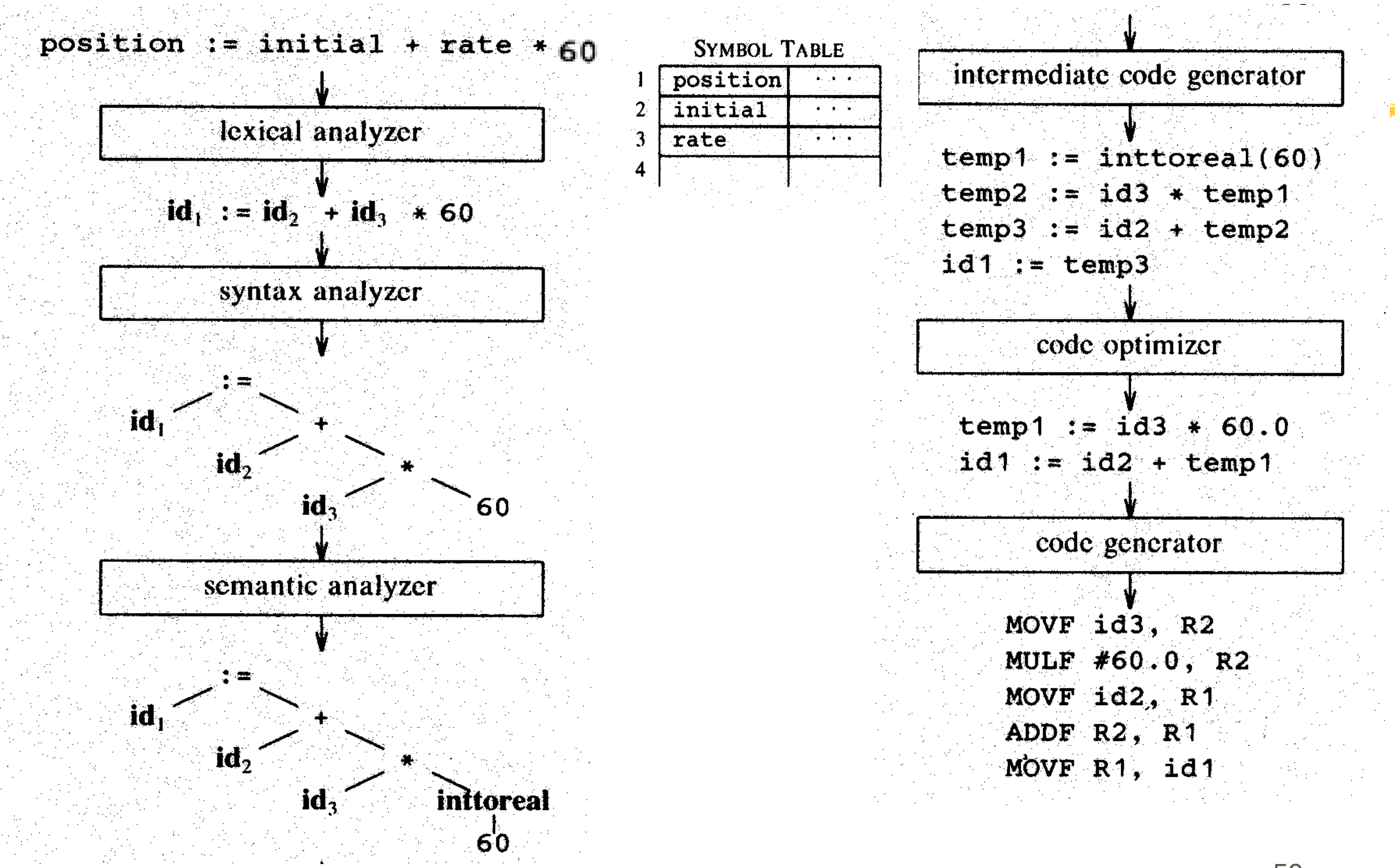
Then what?

- Semantic checks (e.g., type checking)
- Intermediate code generation
- Optimization
- Machine / Assembly language code generation

Steps in the process



A very brief example



```
position := initial + rate * 60
```

$id_1 := id_2 + id_3 * 60$

SYMBOL TABLE

1	position	···
2	initial	···
3	rate	···
4		

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

What course is this?

- Compiler theory
  - Cover some of this in CSCI 35500
  - CSCI 50200 is the compiler theory course
    - Spring 2016