EGRE 591 Compiler Construction

Dr. Dan Resler
Dept. of Electrical & Computer Engineering
Virginia Commonwealth University

August 2019

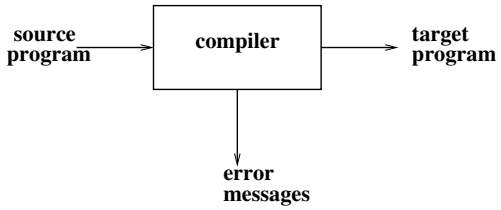
Optional Reference Book

Compilers: Principles, Techniques, and Tools (2nd Edition)
by Aho/Lam/Sethi/Ullman (Addison Wesley: 2006)
ISBN-13: 978-0321-48681-3

`http://tinyurl.com/y8ldtnr`

Definition

A **compiler** is a program that reads a program written in one language and translates it into a logically equivalent program in another language.



- Compilers usually translate human-readable computer programs into machine-readable programs
- Therefore they allow programs to be *portable* across machines
- The term compiler was coined in the early 50s by Grace Murray Hopper
- Prior to the late 1950s, many computer scientists considered compilers impossible to write
- First compiler: FORTRAN (took 18 staff-years to implement)

What Compilers Do

- Compilers can be distinguished in at least two ways:
  - by the kind of machine code they generate
    - pure machine code
    - augmented machine code
    - virtual machine code
  - by the format of the target code they generate
    - assembly language
    - relocatable binary
    - absolute binary

Interpreters

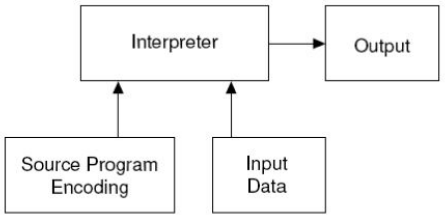


Figure 1.3: An interpreter.

Syntax & Semantics

Defintion

The definition of the **syntax** of a programming language specifices its *form* or structure; the specification of a programming language's **semantics** defines the *meaning* of the program

- Syntax is almost-universally defined using **contex-free grammars** (CFGs)
- CFGs cannot define all aspects of a program, however
- Semantics of programming languages are commonly defined into two classes: **static** & **runtime** semantics

Static Semantics

Definition

The **static semantics** of a language provide a set of rules that specify which syntactically legal programs are actually valid

- For example, in Java the statement
  ```
  a = b + c;
  ```
  is valid only if variables `a`, `b`, and `c` are defined and are of the appropriate type
- Static semantics can be specified formally or informally
- Checked at compile-time

Runtime Semantics

Definition

The **runtime semantics** of a language define what a valid program actually computes (or does) when executed

- Can be specified formally or informally
- Defining formal definitions of runtime semantics is outside the scope of this course

Organization of a Compiler

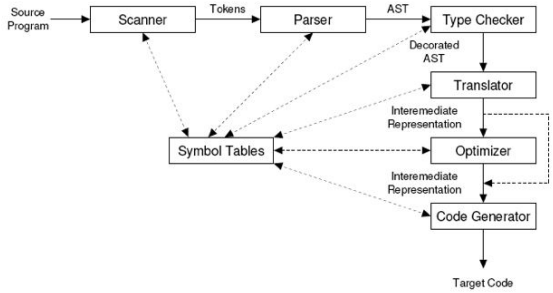
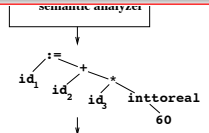


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

An Example

SYMBOL TABLE

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



intermediate code generator

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

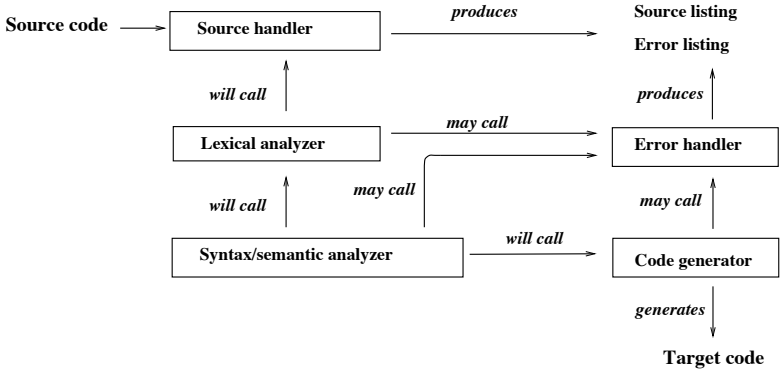
code optimizer

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

code generator

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

The Interleaved Approach



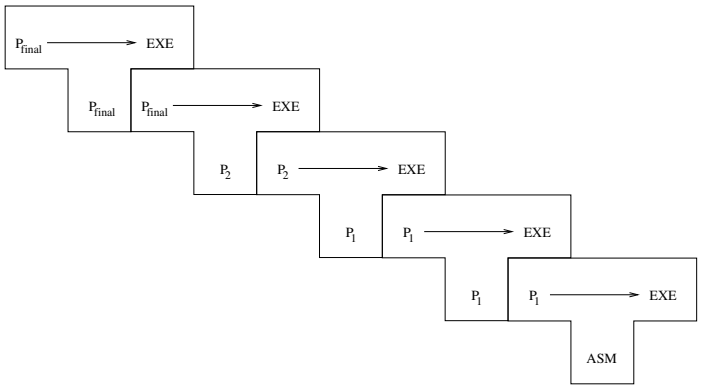
Grouping of Phases

- Often divided into front and back ends
- Front end (analysis): phases that depend primarily on the source language (scanner/parser/intermediate code generator)
- Back end (synthesis): phases that depend primarily on the target language (code optimization & generation)
- It is fairly routine to simply rewrite the back end of a compiler to target it for a new machine
- The back end of a compiler need not generate code for the machine the compiler is running on; a compiler that runs on one machine and generates code for a different machine is called a **cross-compiler**

Approaches to Compiler Development

- hand written vs. machine generated
- Writing a compiler by hand is error prone & tedious, but it often produces more efficient compilers
- Compiler generators (often referred to as *compiler compilers*) generate compilers from formal specifications:
  - *regular expressions* are used to specify lexical analyzers
  - *context free grammars* are used to specify parsers
- A compiler that runs on one computer but produces object code for a different type of computer is called a *cross-compiler*. Cross-compilers are often used to port compilers (and other software) to different architectures.
- It is possible to write a compiler for language L *in* language L using a process called *bootstrapping*

Bootstrapping a Compiler



- Scanners are often called **lexical analyzers** or **lexers**
- The primary function of a scanner is to transform a *character stream* into a *token stream*

Definition

A **token** is a sequence of characters that can be treated as a unit in the grammar of a programming language

- Example:
  Input character stream (Java):
  `if (a < 100) answer = a * RATE;`
  Output token stream:
  IF LPAREN ID(a) REL_OP(lessThan) NUMBER(100) RPAREN ID(answer) ASSIGN_OP ID(a) BIN_OP(times) ID(RATE) SEMICOLON

Definition

The character sequence forming a token is referred to as the **lexeme** for the token, e.g. for the identifier `counter` the token class would be `ID` and the lexeme "`counter`"

- Usually one token per keyword in a language (also called *reserved words*)
- Token classes (e.g. ID, REL_OP, BIN_OP) usually have *semantic values* (also called *attributes*) attached to them

tokens $\xrightarrow{\text{influence}}$ **parsing decisions**

attributes $\xrightarrow{\text{influence}}$ **translation of tokens (code generation)**

- Other tasks of scanners:
  - removing white space and comments
  - correlating error messages from compiler with source
  - producing a listing
  - calling or invoking preprocessors
- Scanners are often generated automatically (by a program called a **scanner generator**) from a formal specification of the tokens

- Few program errors are discernible by the scanner
- Example:

```
fi (n == 13) System.out.print(n);
```

- Can recognize:
  - illegal characters (e.g. % is never allowed in standard Pascal)
  - ill-formed tokens (e.g. in Pascal, `.6` is an illegal floating point literal)
- Lexical errors often result in a warning message with the offending character(s) simply ignored

- Tokens can be described using natural language, e.g.

> *Java* identifiers . . . *must start with a letter – including any currency symbol or connecting punctuation (such as _) – followed by letters, digits, or both.*

- However, using natural language to describe tokens can be tedious and error-prone. In addition, it is not always obvious how to implement a machine to recognize tokens specified in this manner.
- Therefore scanners are specified using the formal language of **regular expressions** and implemented using **deterministic finite automata**, with mathematics to connect the two

Definition

A set of strings defined by a regular expression is called a **regular set**

Definition

- Regular expressions are defined by a **formal language**

Definition

A **formal language** is a set of strings defined by a **metalanguage**; a metalanguage is a language used to describe other languages

Formal Language Definitions

Definition

- A **symbol** is an atomic entity, represented by a character, or sometimes by a reserved word or keyword
- An **alphabet** or **vocabulary** A is a non-empty finite set of symbols.
- A **word** or **string** 'over' an alphabet A is a sequence $\sigma = a_1 a_2 \ldots a_n$ of symbols from the alphabet.
- The **null string** or **empty word** is a string of length zero usually denoted by ϵ (epsilon) (Fischer et al. use λ (lambda)).
- The set of all strings over an alphabet A is denoted A^*.
- The **language** L over alphabet A is a subset of A^*. Note that this involves no concept of meaning.

Formal Language Definitions

Definition

- A **grammar** is essentially a set of rules for choosing the subsets of A^* in which we are interested; it has four components:
  - N – a set of **non-terminal** symbols
  - T – a set of **terminal** symbols
  - S – a special **goal** or **start** symbol
  - P – a set of **productions**
- The set N is often spoken of as denoting the **syntactic classes** of the grammar. The union of sets N and T denotes the **vocabulary** V of the *grammar*.

Operations on Languages

Operation	Notation	Definition
union of L and M	$L \cup M$	$L \cup M = \{s \mid s \in L \vee s \in M\}$
concatenation of L and M	LM	$LM = \{st \mid s \in L \wedge t \in M\}$
exponentiation of L	L^i	$L^i = \begin{cases} \epsilon, & \text{if } i = 0 \\ L^{i-1}L, & \text{if } i > 0 \end{cases}$
Kleene closure of L	L^*	$L^* = \cup_{i=0}^{\infty} L^i$
positive closure of L	L^+	$L^+ = \cup_{i=1}^{\infty} L^i$

Operations on Languages

- Example:
  $L = \{A, B, \ldots, Z, a, b, \ldots, z\}$ $D = \{0, 1, \ldots, 9\}$
  1. $L \cup D$ is the set of letters and digits
  2. LD is the set of strings consisting of a letter followed by a digit
  3. L^4 is the set of all four-letter strings over L
  4. L^* is the set of all strings of letters over L, including ϵ
  5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter
  6. D^+ is the set of all strings of one or more digits
- Since a symbol is a string of length one, L & D can also be thought of as finite languages
- Operations on languages can be thought of as creating new languages

Overview

Definition

A **regular expression** is a formal expression that is

a) a single character in the alphabet Σ

b) the empty string ϵ

c) the empty set $\{\}$

d) any string derived by a finite number of applications of the union, concatenation, or closure operations on any sets of strings in Σ^*

Overview

- Therefore a regular expression is built up out of simpler regular expressions according to a set of rules
- A regular expression is a set of strings L over an alphabet such that L can be defined by a regular expression
- Regular expression r is said to denote language $L(r)$
- Additional metacharacters often used in regular expressions:
  - | (vertical line) means 'or'
  - · (dot) sometimes used for concatentation
  - ? (question mark) denotes 0 or 1 occurrence
  - * (star) denotes 0 or more occurrences
  - ( ) (parenthesis) used to group terms
  - [ ] (brackets) denote character classes
  - not(X) denotes all characters except X
  - "?" quotes used to delimit metacharacter literals

Examples

a	$\{a\}$ (the language containing just the string a)
$a \mid b$	$\{a, b\}$ (a language containing 2 strings, a and b)
a"|"b	$\{a \mid b\}$ (a language containing just the string $a \mid b$)
$(a \mid b) \cdot a$	$\{aa, ba\}$
$(a \mid b)a$	$\{aa, ba\}$
$a?$	$a \mid \epsilon$
$(a \cdot b) \mid \epsilon$	{"",ab}, equivalent to $(a \cdot b)?$
$[a, b, c]$	$a \mid b \mid c$
$[A–Za–z]$	$A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z$
$((a \mid b)a)^*$	{"", aa, ba, $aaaa$, $baaa$, $aaba$, $baba$, $aaaaaa$, ... }
$(0 \mid 1)^* \cdot 0$	zero & all binary numbers that are multiples of 2
a(not(a))a	all 3 character strings beginning & ending with a except aaa
$b^*(abb^*)^*a?$	strings of a's and b's with no consecutive a's

Examples from Programming Languages

- the keyword `while`:
  ```
  while
  ```
- identifiers that begin with a letter followed by 0 or more letters or digits:
  ```
  [A-Za-z][A-za-z0-9]*
  ```
- a literal floating point number that must begin with a digit:
  ```
  [-]?[0-9]+"."[0-9]+
  ```
- a Java comment (where EOL denotes end-of-line):
  ```
  //(not(EOL))*EOL
  ```
- a comment delimited by @@ marks which allows single @'s within the comment body:
  ```
  @@((@not(@))|not(@))*@@
  ```

Limitations of Regular Expressions

- Many languages cannot be described by regular expressions; for example: $L = \{a^n b^n \mid n \geq 1\}$
- Regular expressions can only be used to denote a fixed number of repetitions *or* an unspecified number—two arbitrary number of repetitions cannot be compared to see if they are equal

Specifying Regular Expressions for Scanner Generators

```
if                                  {return IF;}
[a-z][a-z0-9]*                      {return ID;}
[0-9]+                              {return NUM;}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+) {return REAL;}
("--"[a-z]*\n)|(" "|\n|\t)+     {  /* do nothing */ }
.                                   {error();}
```

FIGURE 2.2. Regular expressions for some tokens.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Overview

Definition

A **recognizer** for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language, "no" otherwise

Definition

- **Finite automata** are the (theoretical or real) machines that recognize regular expressions
- A finite automata consists of:
  - a finite set of **states**, with one designated as the **start** state and one or more **final** states
  - labeled **edges** to connect states

Overview

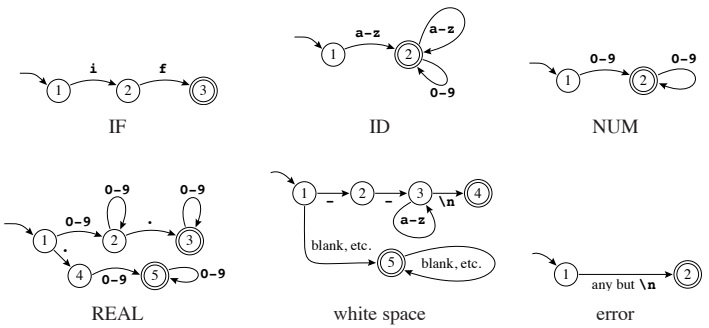


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Combining Finite Automata

```
if                                  {return IF;}
[a-z][a-z0-9]*                      {return ID;}
[0-9]+                              {return NUM;}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+) {return REAL;}
("--"[a-z]*\n)|(" "|\n|\t)+     { /* do nothing */ }
.                                   {error();}
```

FIGURE 2.2. Regular expressions for some tokens.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

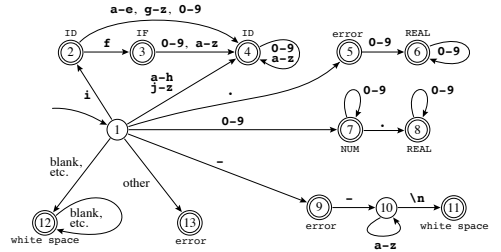


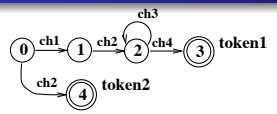
FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Token Recognition

Last Final	Current State	Current Input	Accept Action
0	1	|if --not-a-com	
2	2	i|f --not-a-com	
3	3	if| --not-a-com	
3	0	if| --not-a-com	*return* IF
0	1	if| --not-a-com	
12	12	if| --not-a-com	
12	0	if| --not-a-com	*found white space; resume*
0	1	if |--not-a-com	
9	9	if |-–not-a-com	
9	10	if |-–not-a-com	
9	10	if |-–not-a-com	
9	10	if |-–not-a-com	
9	10	if |-–not-a-com	
9	0	if |-–not-a-com	*error, illegal token '–'; resume*
0	1	if --not-a-com	
9	9	if -|-not-a-com	
9	0	if -|-not-a-com	*error, illegal token '–'; resume*

FIGURE 2.5. The automaton of Figure 2.4 recognizes several tokens. The symbol | indicates the input position at each successive call to the lexical analyzer, the symbol ⊥ indicates the current position of the automaton, and ⊤ indicates the most recent position in which the recognizer was in a final state.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Implementing a Finite State Automata by Hand




```
state = 0; lexeme = ""; currChar = getChar();
while not finished do
 case state of
   0: if currChar = ch1 then
        concatenate currChar to lexeme
        state = 1; currChar = getChar()
      elseif currChar = ch2 then
        concatenate currChar to lexeme
        return token2  // state 4
      else error
   1: if currChar = ch2 then
        concatenate currChar to lexeme
        state = 2; currChar = getChar()
      else error
   2: if currChar = ch3 then
        concatenate currChar to lexeme
        currChar = getChar()
      elseif currChar = ch4 then
        concatenate currChar to lexeme
        return token1  // state 3
      else error
 endcase
endwhile
```

Types of Finite Automata

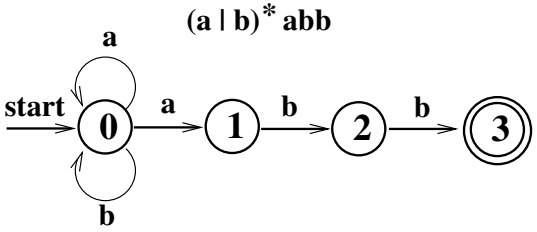
- Two types of finite automata:
  - non-deterministic (NFA)
  - deterministic (DFA)
- With NFAs, more than one transition may be possible on the same input symbol
- Both are capable of recognizing regular sets
- DFA faster, bigger than NFA
- Essentially only DFAs are used to implement scanners

Non-deterministic Finite Automata

Definition

- A **non-determinitic finite automaton** is a mathematical model that consists of
  1. a set of states S
  2. a set of input symbols Σ
  3. a transition function *move* that maps state-symbol pairs to sets of states
  4. a state s_0 designated as the *start* (or *initial*) state
  5. a set of states F designated as *accepting* (or *final*) states

Non-deterministic Finite Automata



	INPUT SYMBOL	
STATE	*a*	*b*
0	{0, 1}	{0}
1	–	{2}
2	–	{3}

Non-deterministic Finite Automata

- An NFA *accepts* an input string x if and only if there is some path in the transition graph from the start to some accepting state such that the edges along the path spell out x
- A path can be represented by a sequence of state transitions called *moves*; the following diagram shows the moves made in accepting string *aabb* using the NFA on the previous slide:

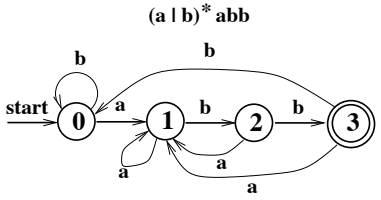
$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

- NFAs allow ϵ-transitions
- The *language defined* by an NFA is the set of input strings it accepts

Deterministic Finite Automata

Definition

- A **deterministic finite automaton** is a special case of a non-deterministic finite automaton in which
  1. no state has an ϵ-transition (a transition on input ϵ)
  2. for each state s and input symbol a, there is at most one edge labeled a leaving s



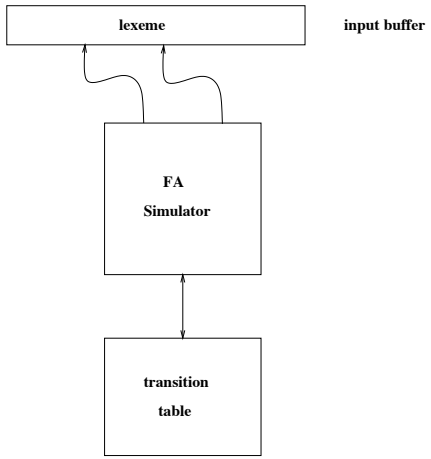
	INPUT SYMBOL	
STATE	*a*	*b*
0	{1}	{0}
1	{1}	{2}
2	{1}	{3}
3	{1}	{0}

DFA Simulator

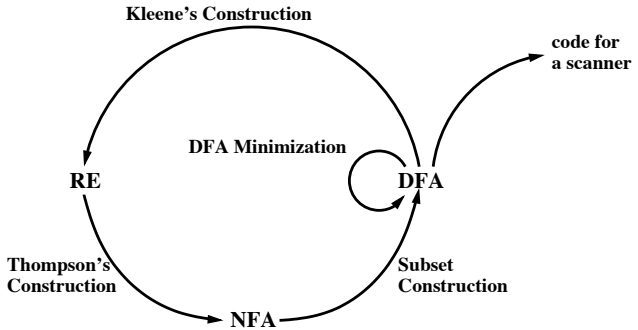
- Definition: **edge(*s*,*c*)** – the set of states reachable by following a *single* edge with label *c* from state *s*
- DFA simulation algorithm (given a start state of s_1, the input string $c_1, \ldots, c_k$, and the set of accepting states F):

$d \leftarrow \{s_1\}$
for $i \leftarrow 1$ to k **do**
  $d \leftarrow$ **edge(**d, c_i**)**
endfor
if $d \subseteq F$ **then**
  return "yes"
else
  return "no"
endif

Using the Finite Automata Model



Automatic Generation of Scanners from Regular Expressions[1]



[1] Diagram is from *Engineering a Compiler* by Cooper & Torczon, Morgan Kaufmann, 2004, pg. 45

Automatic Generation of Scanners from Regular Expressions


```
liberty:~/toy/Parser/> java JLex.Main toy.lex
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 198 states.
Working on character classes.:::::.::::::..:....:.::::.:.:.
::::::..:::.:..:.::.....:.................:....::..:.....:
NFA has 40 distinct character classes.
Creating DFA transition table.
Working on DFA states.................................
................................................
Minimizing DFA transition table.
83 states after removal of redundant states.
Outputting lexical analyzer code.
liberty:~/toy/Parser/>
```

Converting a Regular Expression to an NFA

- Useful because it is easy to convert a regular expression to an NFA
- Algorithm referred to as *Thompson's construction*
- Each individual regular expression is translated into an NFA using a standard template, then templates are combined

Converting a Regular Expression to an NFA

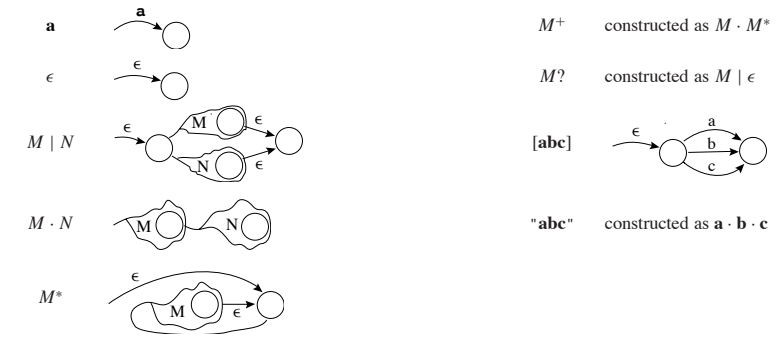


FIGURE 2.6. Translation of regular expressions to NFAs.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Converting a Regular Expression to an NFA

```
if                                  {return IF;}
[a-z][a-z0-9]*                      {return ID;}
[0-9]+                              {return NUM;}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+) {return REAL;}
("--"[a-z]*\n)|(" "|\n|\t)+         { /* do nothing */ }
.                                   {error();}
```

FIGURE 2.2. Regular expressions for some tokens.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

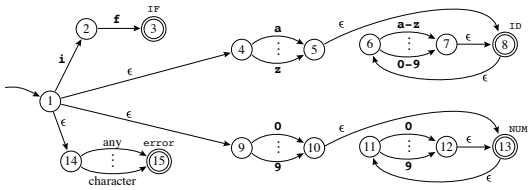
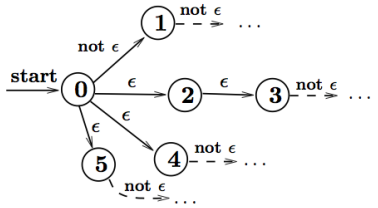


FIGURE 2.7. Four regular expressions translated to an NFA.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Converting an NFA to a DFA

- In the transition table for an NFA, each entry is a set of states; in the transition table for a DFA, each entry is a single state
- The general idea behind NFA-to-DFA translation is that each DFA state corresponds to a set of NFA states
- Process called **subset construction**
- Definition: **ϵ-closure(S)** – the set of NFA sets reachable from NFA state S (or set of states S) on ϵ-transitions alone



Converting an NFA to a DFA

- Formally, ϵ-closure(S) can be defined as the smallest set T such that

$$T = S \cup \left(\bigcup_{s \in T} \textbf{edge}(s, \epsilon) \right)$$

- Algorithm for calculating T:

$T \leftarrow S$
repeat
  $T' \leftarrow T$
  $T \leftarrow T' \cup (\bigcup_{s \in T'} \textbf{edge}(s, \epsilon))$
until $T = T'$

Converting an NFA to a DFA

- Remember that the DFA generated from our NFA-to-DFA algorithm will (potentially) have single states representing multiple NFA states:

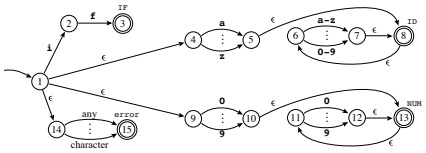


FIGURE 2.7. Four regular expressions translated to an NFA. From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

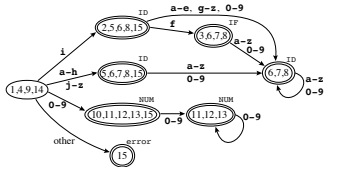


FIGURE 2.8. NFA converted to DFA. From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

Converting an NFA to a DFA

- Definition: **DFAedge(*d*,*c*)** – given the generated DFA state d will return the new DFA state reached (which represents a set of NFA states) on input c
- More formally:

$$\textbf{DFAedge}(d, c) = \epsilon\textbf{–closure} \left(\bigcup_{s \in d} \textbf{edge}(s, c) \right)$$

Converting an NFA to a DFA

- NFA simulation algorithm (given an NFA start state of s_1, the input string $c_1, \ldots, c_k$, and the set F of accepting states):

```
d ← ϵ-closure({s1})
for i ← 1 to k do
  d ←DFAedge(d, ci)
endfor
if (∃e | (e ∈ d ∧ e ∈ F)) then
  return "yes"
else
  return "no"
endif
```

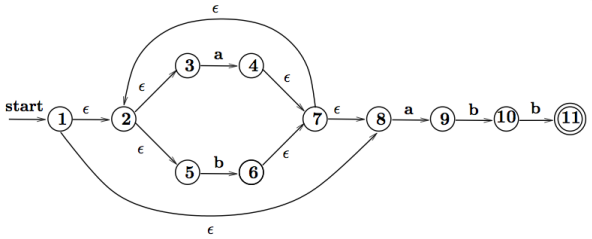
- Manipulating sets in real time each time a character is inputted is prohibitively expensive; however, it is possible to do all of the sets-of-states calculations in advance

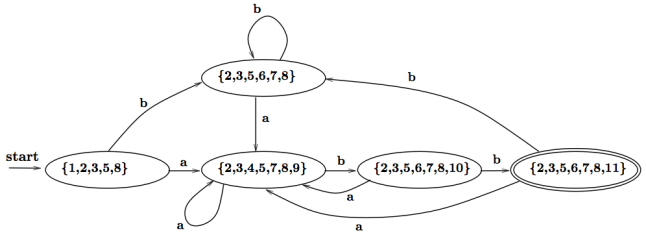
Converting an NFA to a DFA

- NFA → DFA constuction algorithm (assume that **DFAedge**(d_i,c) = d_j if there is an edge from d_i to d_j labeled with c, the NFA start state is 1, and that Σ is the alphabet):

```
states[1] ← {}; states[1] ← ϵ-closure(s1)
p ← 1; j ← 1
while j ≤ p do
  foreach c ∈ Σ do
    e ← DFAedge(states[j],c)
    if e = states[i] for some i ≤ p
     then trans[j,c] ← i
     else p ← p + 1
          states[p] ← e
          trans[j,c] ← p
    endif
  endfor
  j ← j + 1
endwhile
```


Converting an NFA to a DFA





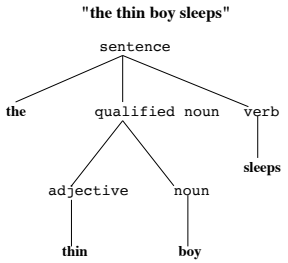
Syntax Analysis

Definition

- The syntax of programming languages can be described by **grammars**
  - A grammar is a set of rules for *choosing* subsets of a language
  - Or it can be thought of a set of rules for *generating* subsets of a language
- The process of determining whether a sentence is legal for a given language is called **parsing**

Syntax Analysis

sentence	→	**the** qualified_noun verb | pronoun verb
qualified_noun	→	adjective noun
noun	→	**man** | **girl** | **boy** | **lecturer**
pronoun	→	**he** | **she**
verb	→	**drinks** | **sleeps** | **mystifies**
adjective	→	**tall** | **thin** | **thirsty**



Components of a Grammar

- Remember that we previously defined a grammar as:
  - N – a set of **non-terminal** symbols
  - T – a set of **terminal** symbols
  - S – a special **goal** or **start** symbol
  - P – a set of **productions**
- For the grammar on the previous slide:
  - N = { sentence,qualified_noun,noun,pronoun,verb,adjective }
  - T = { **the**,**man**,**girl**,**boy**,**lecturer**, . . . ,**tall**,**thin**,**thirsty** }
  - S = sentence
  - P = { 'sentence → **the** qualified_noun verb', 'sentence → pronoun verb', 'qualified_noun → adjective noun', 'noun → **man**', 'noun → **girl**', . . . , 'adjective → **thin**', 'adjective → **thirsty** }

Backus-Naur Form (BNF)

Metasymbol	Meaning
→ or : : =	"is defined as"
|	"or"
< > or special notation	delimiters/indicators for non-terminals
non-delimited text or special notation	terminals
ϵ	empty right-hand side

- For example:

$A \rightarrow B$ **.**
$B \rightarrow$ **x** | **(** C **)** | **[** B **]** | ϵ
$C \rightarrow B\,D$
$D \rightarrow$ **+** $B\,D$ | ϵ

<A> ::= <B> .
<B> ::= x | (<C>) | [<B>] | ϵ
<C> ::= <B> <D>
<D> ::= + <B> <D> | ϵ

BNF vs. EBNF

- BNF:

$A \rightarrow B$ **.**
$B \rightarrow$ **x** | **(** C **)** | **[** B **]** | ϵ
$C \rightarrow B\,D$
$D \rightarrow$ **+** $B\,D$ | ϵ

- EBNF (Extended BNF):

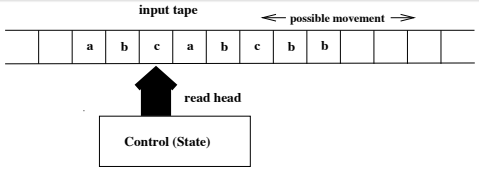
$A \rightarrow B$ **.**
$B \rightarrow$ [**x** | **(** C **)** | **[** B **]**]
$C \rightarrow B\,D$
$D \rightarrow$ { **+** B }

- In addition EBNF often delimits tokens with double quotes, e.g. "x"

Chomsky's Heirarchy

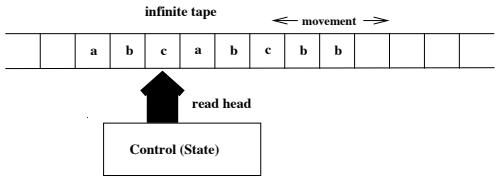
Class	Grammar	Recognizer
0	**Unrestricted** $\alpha \rightarrow \beta$ (where α contains a nonterminal)	Turing Machine
1	**Context-sensitive** $\alpha B \gamma \rightarrow \alpha \beta \gamma$	Linear-bounded Automaton
2	**Context-free** $A \rightarrow \alpha$	Push-down Automaton
3	**Regular** $A \rightarrow a \mid aB \mid \epsilon$	Finite-state Automaton

Grammars and Recognizers



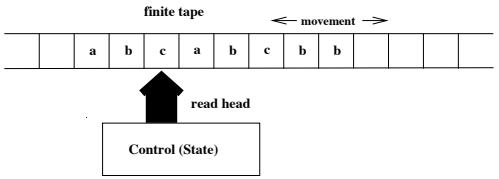
- A basic automaton consists of:
  - a control mechanism with a finite number of *states*
  - a possibly infinite *tape* which may be read and advanced (plus possibly moved in either direction and written to)
  - A finite *alphabet*
  - A *start state* and a series of *transitions* to other states upon processing of the tape
  - Initial *input* represented by characters on the tape
  - Ideally it will have transitions to *halt* either upon it recognizing a legal string or upon it being *blocked* by illegal input
  - Our recognizers will be variations of this machine

Unrestricted Languages



- Unrestricted automatons model Turing machines
- Infintely long read/write tape moveable in both directions with a start but no end
- Can compute all programs

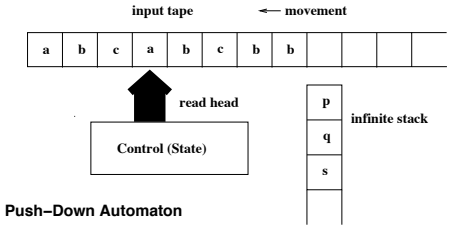
Context-Sensitive Languages



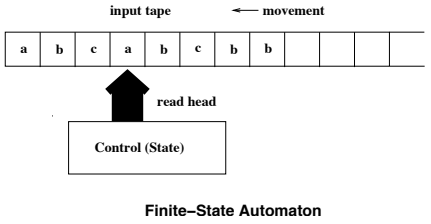
- Linear bounded automatons recognize context-sensitive languages
- Finite read/write tape moveable in both directions
- Essentially a Turing machine running on a finite machine

Context-Free Languages

- A language that can be generated by a context-free grammar is said to be a *context-free language*
- Recognizer for context-free languages: *push-down automata*



Regular Languages



- Finite state automatons recognize regular languages
- A regular grammar defines a regular set
- Notice that a finite state automaton is a push-down automaton without the stack

Derivations

- Sentences can be **derived** from a grammar through a series of rewrite rules that specify the method for expanding non-terminals

1 $S \rightarrow S\ ;\ S$	4 $E \rightarrow \text{id}$	8 $L \rightarrow E$
2 $S \rightarrow \text{id} := E$	5 $E \rightarrow \text{num}$	9 $L \rightarrow L\ ,\ E$
3 $S \rightarrow \text{print}\ (\ L\ )$	6 $E \rightarrow E + E$	
	7 $E \rightarrow (\ S\ ,\ E\ )$	

GRAMMAR 3.1. A syntax for straight-line programs.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

$\underline{S}$
$S\ ;\ \underline{S}$
$\underline{S}\ ;\ \text{id} := E$
$\text{id} := \underline{E}\ ;\ \text{id} := E$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := E + \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E} + (S\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\underline{S}\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \underline{E}\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := E + E\ ,\ \underline{E}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \underline{E} + E\ ,\ \text{id}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \text{num} + \underline{E}\ ,\ \text{id}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \text{num} + \text{num}\ ,\ \text{id}\ )$

DERIVATION 3.2.
From *Modern Compiler Implementation in Java,*

Sentential Forms

Definition

A **sentential form** is the goal or start symbol or any string that can be derived from it

- A string is said to be a *sentential form of a grammar* if it can be derived from it

Left-most vs. Right-most Derivations

1 $S \rightarrow S\ ;\ S$	4 $E \rightarrow \text{id}$	8 $L \rightarrow E$
2 $S \rightarrow \text{id} := E$	5 $E \rightarrow \text{num}$	9 $L \rightarrow L\ ,\ E$
3 $S \rightarrow \text{print}\ (\ L\ )$	6 $E \rightarrow E + E$	
	7 $E \rightarrow (S\ ,\ E\ )$	

GRAMMAR 3.1. A syntax for straight-line programs.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

- For the sentence '`id := (id := num, id); print(id)`':
  - **left-most:**

S ⇒ **S** ; S ⇒ id := **E** ; S ⇒ id := (**S** , E) ; S ⇒
id := (id := **E** ; E) ; S ⇒ id := (id := num ; **E**) ; S ⇒
id := (id := num ; id) ; **S** ⇒ id := (id := num ; id) ; print (**L**) ⇒
id := (id := num ; id) ; print (**E**) ⇒ id := (id := num ; id) ; print (id)

  - **right-most:**

S ⇒ S ; **S** ⇒ S ; print (**L**) ⇒ S ; print (**E**) ⇒ **S** ; print (id) ⇒
id := **E**; print (id) ⇒ id := (S , **E**); print (id) ⇒
id := (**S** , id); print (id) ⇒ id := (id := **E**, id); print (id) ⇒
id := (id := num, id); print (id)

Parse Trees

- Essentially a graph of (or representing) a derivation
- Two different derivations can have the same parse tree

$\underline{S}$
$S\ ;\ \underline{S}$
$\underline{S}\ ;\ \text{id} := E$
$\text{id} := \underline{E}\ ;\ \text{id} := E$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := E + \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E} + (S\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\underline{S}\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \underline{E}\ ,\ E\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := E + E\ ,\ \underline{E}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \underline{E} + E\ ,\ \text{id}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \text{num} + \underline{E}\ ,\ \text{id}\ )$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \text{num} + \text{num}\ ,\ \text{id}\ )$

DERIVATION 3.2.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

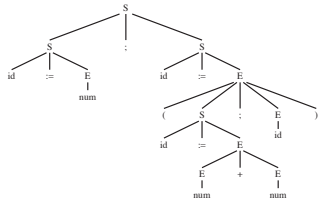


FIGURE 3.3. Parse tree.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Abiguous Grammars

- Every parse tree has associated with it a unique derivation
- However, not every sentence necessarily has only one possible parse tree
- A grammar that can derive more than one parse tree for a given sentence is said to be *ambiguous*

$E \rightarrow \text{id}$
$E \rightarrow \text{num}$
$E \rightarrow E * E$
$E \rightarrow E\ /\ E$
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow (\ E\ )$

GRAMMAR 3.5.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

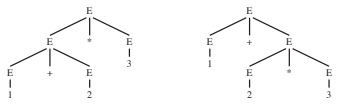
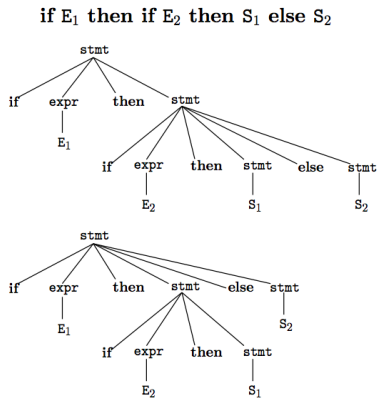


FIGURE 3.7. Two parse trees for the sentence 1+2*3 in Grammar 3.5.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Eliminating Ambiguity

- Sometimes a grammar can be re-written to eliminate ambiguity
- Example: the 'dangling else'

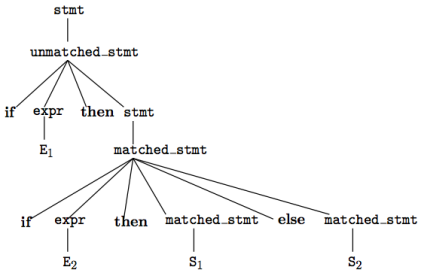
stmt → **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| . . .



Eliminating Ambiguity

stmt → matched_stmt
| unmatched_stmt
matched_stmt → **if** expr **then** matched_stmt **else** matched_stmt
| . . .
unmatched_stmt → **if** expr **then** stmt
| **if** expr **then** matched_stmt **else** unmatched_stmt

if E_1 **then if** E_2 **then** S_1 **else** S_2



Syntax Error Recovery

- Simplest: parsing halts on first error
- Often can do better by restoring parser to some state where parsing can continue
- Strategies:

  Panic-mode recovery – discard tokens until some synchronizing token if found

  Phrase-level recovery – parser corrects token stream by changing or inserting tokens

  Error productions – parser writer anticipates certain errors and inserts error productions into grammar

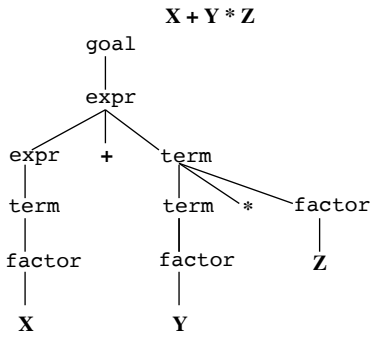
  Global correction – analyze and modify entire program with purpose of altering it to be syntacticly correct
- Can use combinations of strategies

Top-down vs. Bottom-up Parsing

- **top-down**: start from the goal symbol and try to generate the sentence by substituting the *expanding* non-terminals at each step
- **bottom-up**: start from the input sentence and make *reductions* to non-terminals until you end up with the goal symbol
- top-down parsing is associated with left-most derivations, bottom-up with right-most

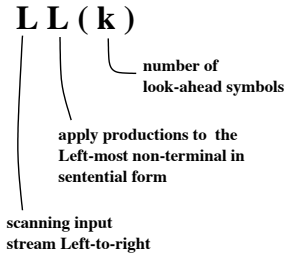
Top-down vs. Bottom-up Parsing

`goal` → `expr`
`expr` → `term` | `expr` **+** `term`
`term` → `factor` | `term` * `factor`
`factor` → **X** | **Y** | **Z**



LL(k) Parsers

- LL(k) parsers parse LL(k) languages
- (usually deterministic) top-down parsers



- Most common type: LL(1)
- Informally, a language is LL(k) if there is enough information to choose the correct expansion of a non-terminal at any point in the parse given the current k look-ahead symbols
- Not all LL languages can be parsed with LL(1) parsers

LL(k) Parsers

$A \rightarrow \mathbf{x}B \mid \mathbf{x}C$
$B \rightarrow \mathbf{x}B \mid \mathbf{y}$
$C \rightarrow \mathbf{x}C \mid \mathbf{z}$

Sentential Form	Input String	
S = A	xxxz	← try A → xB
xB	xxxz	
~~x~~B	~~x~~xxz	
xB	xxz	
~~x~~B	~~x~~xz	
xB	xz	
~~x~~B	~~x~~z	
xB	z	
y	z	?

- Solution: allow parser to backtrack
- Very expensive
- Need determinsitic parsers, one where we can be sure which production to apply at any given moment in the parse

Predictive Parsers

Definition

A **predictive parser** is a deterministic LL(k) parser

- With predictive parsing, the inability to precede with the parsing of a sentence indicates an error (i.e. the sentence is not a legal sentence for the language)
- Most commonly used predictive parsers: **recursive descent parsers**

Overview

- **Recursive descent parsers** execute a set of recursive procedures to parse the input
  - a parsing procedure P is associated with each nonterminal A
  - procedure P is responsible for accomplishing one step of a derivation by choosing and applying one of A's productions (i.e. it recognizes the language subphrase associated with A)
  - the parser chooses the appropriate production for A by inspecting the next k tokens in the input stream
  - the **predict set** for production $A \rightarrow \alpha$ is the set of tokens that trigger application of that production
  - the predict set for $A \rightarrow \alpha$ is determined primarialy by α, the right-hand side (RHS) of the production

Overview

$A \rightarrow B$ **.**
$B \rightarrow$ **x** | **(** C **)** | **[** B **]** | ϵ
$C \rightarrow B\,D$
$D \rightarrow$ **+** $B\,D$ | ϵ

- A recursive descent parser for the above language would require 4 procedures
- Note that if we are recognizing subphrase B, if the next token is:

x	expand using production $B \rightarrow$ **x**
(	expand using production $B \rightarrow$ **(** C **)**
[	expand using production $B \rightarrow$ **[** B **]**
.)] +	expand using production $B \rightarrow \epsilon$

- Therefore the predict set for production B: { **x**,**(**,**[**,**.**,**)**,**]**,**+** }

Overview

$S \rightarrow$ if E then S else S
$S \rightarrow$ begin $S\ L$
$S \rightarrow$ print E

$L \rightarrow$ end
$L \rightarrow$; $S\ L$

$E \rightarrow$ num = num

GRAMMAR 3.11.
From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

```
final int IF=1, THEN=2, ELSE=3, BEGIN=4, END=5, PRINT=6,
 SEMI=7, NUM=8, EQ=9;

int tok = getToken();

void advance() {tok = getToken();}
void eat(int t) {if (tok==t) advance(); else error();}

void S() { switch(tok) {
            case IF:    eat(IF); E(); eat(THEN); S();
                        eat(ELSE); S(); break;
            case BEGIN: eat(BEGIN); S(); L(); break;
            case PRINT: eat(PRINT); E(); break;
            default:    error();
           } }

void L() { switch(tok) {
            case END:   eat(END); break;
            case SEMI:  eat(SEMI); S(); L(); break;
            default:    error();
           } }

void E() {eat(NUM); eat(EQ); eat(NUM); }
```


Overview

$S \rightarrow E\,\$$

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$

$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$

$F \rightarrow$ id
$F \rightarrow$ num
$F \rightarrow (\ E\ )$

GRAMMAR 3.10.
From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

```
void S() { E(); eat(EOF); }
void E() { switch(tok) {
            case ?:  E(); eat(PLUS); T(); break;
            case ?:  E(); eat(MINUS); T(); break;
            case ?:  T(); break;
            default:  error();
           } }
void T() { switch(tok) {
            case ?:  T(); eat(TIMES); F(); break;
            case ?:  T(); eat(DIV); F(); break;
            case ?:  F(); break;
            default:  error();
           } }
```

- Recursive-descent parsing works only on grammars where the *first terminal symbol* of each subexpression provides enough information to choose which production to use

Overview

- Recursive-descent parsing is so powerful and easy to implement that we'll consider the restrictions one would have to put on a grammar to allow one to use the recursive-descent algorithm
- Problems arise when we have alternative productions for the same non-terminal where each production has the same initial symbol, e.g.

A $\rightarrow$ **x**B | **x**C
B $\rightarrow$ **x**B | **y**
C $\rightarrow$ **x**C | **z**

- It is possible to evaluate a grammar and determine if it can be parsed using recursive-descent parsing; to do so one must examine the FIRST and FOLLOW sets for grammar

LL(1) Grammars for Predictive Parsing

Rule 1

Given a production

$$A \rightarrow \xi_1 \mid \xi_2 \mid \ldots \xi_n$$

the set of initial terminal symbols of all sentences that can be generated from each of the ξ_k's must be disjoint, that is

$$\text{FIRST}(\xi_j) \cap \text{FIRST}(\xi_k) = \phi \quad \text{for all } j \neq k$$

where the set FIRST(ξ_k) is the set of all terminal symbols that can appear in the first position of sentences derived from ξ_k. That is

$$a \in \text{FIRST}(\xi_k) \quad \text{if } \xi_k \rightarrow a\zeta$$

LL(1) Grammars for Predictive Parsing

Rule 2

For every non-terminal A which can generate the null string, the set FIRST(A) of the initial symbols which can be derived by using any alternative productions for A must be disjoint from the set FOLLOW(A) of symbols that may follow any sequence generated from A, that is

$$\text{FIRST}(A) \cap \text{FOLLOW}(A) = \phi$$

where the set FOLLOW(A) is computed by considering every production P_k of the form

$$P_k \rightarrow \xi_k A \zeta_k$$

and forming the sets FIRST(ζ_k), when

$$\text{FOLLOW}(A) = \text{FIRST}(\zeta_1) \cup \text{FIRST}(\zeta_2) \cup \ldots \cup \text{FIRST}(\zeta_n)$$

with the addition that if at least one ζ_k is capable of generating the null string, then the set of FOLLOW(P_k) has to be included in the set FOLLOW(A) as well.

LL(1) Grammars for Predictive Parsing

- when a grammar rule A can derive the empty set (i.e. $A \rightarrow \epsilon$),

$$predict(A) = FIRST(A) \cup FOLLOW(A)$$

- however, when a grammar rule A *does not* derive the empty set, $predict(A) = FIRST(A)$

LL(1) Grammars for Predictive Parsing

$S \rightarrow A\,C_1\,\$$
$C \rightarrow c \mid \epsilon$
$A \rightarrow a\,B\,C\,d \mid B\,Q$
$B \rightarrow b\,B \mid \epsilon$
$Q \rightarrow q \mid \epsilon$

FIRST & FOLLOW sets?

- FIRST(C) = { c }; FIRST(B) = { b }; FIRST(Q) = { q }
- FOLLOW(A) = FIRST(C) $\cup$ FOLLOW(C_1) = { c,$ }
- FIRST(A) = { a } $\cup$ FIRST($B\,Q$) = { a,b,q }
- FIRST(S) = FIRST($A\,C_1\,\$$) = { a,b,c,q,$ }
- FOLLOW(C) = { d,$ }
- FOLLOW(B) = FIRST(C) $\cup$ FOLLOW(C) $\cup$ FIRST(Q) $\cup$ FOLLOW(A) = { c,d,q,$ }
- FOLLOW(Q) = FOLLOW(A) = { c,$ } Is it LL(1)?

LL(1) Grammars for Predictive Parsing

```
1 S → A C $
2 C → c
3   | λ
4 A → a B C d
5   | B Q
6 B → b B
7   | λ
8 Q → q
9   | λ
```

Figure 5.2: A CFGs.

Rule Number	A	$X_1 \ldots X_m$	First($X_1 \ldots X_m$)	Derives Empty?	Follow(A)	Answer
1	S	A C $	a,b,q,c,$	No		a,b,q,c,$
2	C	c	c	No		c
3		λ		Yes	d,$	d,$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,$	b,q,c,$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,$	q,c,d,$
8	Q	q	q	No		q
9		λ		Yes	c,$	c,$

Figure 5.3: Predict calculation for the grammar of Figure 5.2.

LL(1) Grammars for Predictive Parsing

- Problem: left recursion

$S \to E\,\$$		
	$T \to T * F$	$F \to \text{id}$
$E \to E + T$	$T \to T / F$	$F \to \text{num}$
$E \to E - T$	$T \to F$	$F \to (\,E\,)$
$E \to T$		

GRAMMAR 3.10. From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

LL(1) Grammars for Predictive Parsing

- Eliminating left recursion:

  $A \to A\alpha \mid \beta$

  is transformed into

  $A \to \beta A'$
  $A' \to \alpha A' \mid \epsilon$

- Note the right recursion is OK
- Generally:

  $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

  where no β_i begins with an A.
  This becomes:

  $A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
  $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

$S \to E\,\$$		
	$T \to T * F$	$F \to \text{id}$
$E \to E + T$	$T \to T / F$	$F \to \text{num}$
$E \to E - T$	$T \to F$	$F \to (\,E\,)$
$E \to T$		

GRAMMAR 3.10. From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

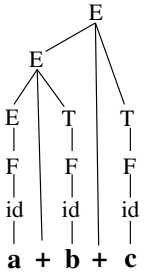
$\underbrace{E}_{A} \to \underbrace{E}_{A}\underbrace{+T}_{\alpha_1} \mid \underbrace{E}_{A}\underbrace{-T}_{\alpha_2} \mid \underbrace{T}_{\beta}$

$E \to T\,E'$
$E' \to +T\,E' \mid -T\,E' \mid \epsilon$

LL(1) Grammars for Predictive Parsing

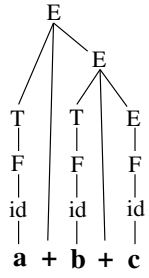
- left recursive:

$E \to T \mid E + T \mid E - T$
$T \to F \mid T * F \mid T / F$
$F \to \textbf{id} \mid \textbf{num} \mid (\,E\,)$



- right recursive:

$E \to T \mid T + E \mid T - E$
$T \to F \mid F * T \mid F / T$
$F \to \textbf{id} \mid \textbf{num} \mid (\,E\,)$



LL(1) Grammars for Predictive Parsing

- The problem – the following grammar is not predictive LL(1):

  $S \rightarrow$ if E then S else S

  $S \rightarrow$ if E then S

- Solution – **left factor** the grammar:

  $S \rightarrow$ if E then $S\ X$

  $X \rightarrow$ else $S \mid \epsilon$

Writing Recursive-Descent Parsers[2]

- Need to write a procedure for each production in the LL grammar; this procedure usually is named the same as the non-terminal
- Procedure analyzes a sequence of symbols defined by the RHS of the production and verifies that they are correct for the given grammar, e.g. given a non-terminal S and a look-ahead buffer of SYM, procedure S will:
  - (a) assume that S has been called with SYM $\in$ FIRST(S)
  - (b) will then parse a set of symbols that can be derived from production S
  - (c) will relinguish parsing assuming that SYM $\in$ FOLLOW(S)

[2] much of this section is from *Programming Language Translation* by Pat Terry, Addison-Wesley, 1986

Writing Recursive-Descent Parsers

- Shell of parsing procedure:

```
procedure S;
  begin
    (* assert SYM ∈ FIRST(S) *)
    T(string)
    (* assert SYM ∈ FOLLOW(S) *)
  end
```

where the transformation T(string) is governed by the following rules:

1. if the production expands to a single terminal:

   T(terminal) $\Rightarrow$

   if SYM = terminal
   then get next symbol SYM
   else report error

   e.g. Given S $\rightarrow$ a ,

```
if SYM = token_a
  then SYM = getNextToken()
  else reportError();
```

Writing Recursive-Descent Parsers

2. if the production expands to a single non-terminal:

   T(single_non-terminal<A>) $\Rightarrow$ A

   e.g. Given S $\rightarrow$ A, just call the procedure A

3. if the production expands into a number of alternative forms:

   T($\alpha_1 \mid \alpha_2 \mid \ldots \alpha_n$) $\Rightarrow$

   case SYM of
   FIRST(α_1): T(α_1);
   FIRST(α_2): T(α_1);
   ...
   FIRST(α_n): T(α_n);
   FOLLOW(S): (* do nothing *)
   end;

   e.g. Given S $\rightarrow$ aB | bC | ϵ ,

```
case SYM of
  token_a: begin SYM = getNextToken(); B; end;
  token_b: begin SYM = getNextToken(); C; end;
end;
```

Writing Recursive-Descent Parsers

4. if the production allows for possible repetition of the RHS:

   T($\{\alpha\}$) $\Rightarrow$ **while** SYM $\in$ FIRST(α) **do** T(α)

   e.g. Given S $\rightarrow$ {aB},

```
while SYM = token_a do
 begin
  SYM = getNextToken(); B;
 end;
```

5. if the production generates a sequence of terminals and non-terminals:

   T($\alpha_1\ \alpha_2\ \ldots\ \alpha_n$) $\Rightarrow$ **begin** T(α_1); T(α_2); ... T(α_n); **end**

   e.g. Given S $\rightarrow$ a {+ A} BC,

```
if SYM = token_a then
 begin
  SYM = getNextToken();
  while SYM = token_plus do
    begin SYM = getNextToken(); A; end;
  B; C;
 end;
else reportError();
```

Example: A Simple Recursive-Descent Parser

- Grammar:

  A $\rightarrow$ B **.**
  B $\rightarrow$ **x** | **(** C **)** | **[** B **]** | ϵ
  C $\rightarrow$ B D
  D $\rightarrow$ { **+** B }

```
// Author: D. Resler 4/2004
//  modified from "Programming Language Translation"
//  by Pat Terry (Addison- Wesley, 1986)

import java.io.*;

public class RecursiveDescent {

  private static char sym;
  private static String buffer;
  private static boolean debug = true;

  public static void main(String [] args) throws Exception {
    getSourceString();
    sym = getNextToken(); A();
    System.out.println("Successful parse");
  }
```

Example: A Simple Recursive-Descent Parser

```
private static void getSourceString() throws IOException {
  BufferedReader stdin =
     new BufferedReader( new InputStreamReader(System.in) );
  System.out.print("input: ");
  buffer = stdin.readLine()+"$"; // $ represents end-of-stream
}

private static int currentSym = 0;
private static char getNextToken() {
  char ch = buffer.charAt(currentSym++);
  if (debug)
    System.out.println("[SCANNER] "+ch);
  return ch;
}

private static void load_sym() { sym = getNextToken(); }

private static void accept(char terminal) throws Exception {
  if (sym == terminal)
    load_sym();
  else
    throw new Exception("syntax error");
}

private static void entering(String production) {
  if (debug)
    System.out.println("[PARSER] entering " + production);
}
```

Example: A Simple Recursive-Descent Parser

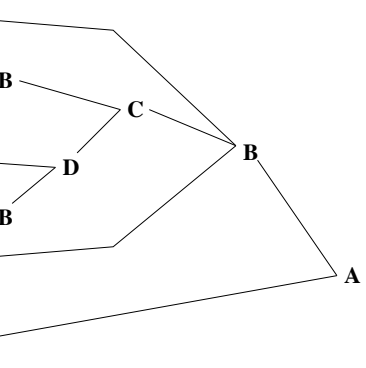
```
private static void exiting(String production) {
  if (debug)
    System.out.println("[PARSER] exiting " + production);
}

private static void A() throws Exception {
  entering("A"); B(); accept('.'); exiting("A");
}

private static void B() throws Exception {
  entering("B");
  switch (sym) {
    case 'x': load_sym(); break;
    case '(': load_sym(); C(); accept(')'); break;
    case '[': load_sym(); B(); accept(']'); break;
    case ')': case ']': case '+': case '.':
        /* followers of B - do nothing */
        break;
    default: throw new Exception("syntax error");
  }
  exiting("B");
}

private static void C() throws Exception {
  entering("C");
  B(); D();
  exiting("C");
}
```

A $\rightarrow$ B **.**
B $\rightarrow$ **x** | **(** C **)** | **[** B **]** | ϵ
C $\rightarrow$ B D
D $\rightarrow$ { **+** B }

```
private static void D() throws Exception {
  entering("D");
  while (sym == '+') {
    load_sym(); B();
  }
  exiting("D");
}
}
```

Example: A Simple Recursive-Descent Parser

```
input: (x+x).
[SCANNER] (
[PARSER] entering A
[PARSER] entering B
[SCANNER] x
[PARSER] entering C
[PARSER] entering B
[SCANNER] +
[PARSER] exiting B
[PARSER] entering D
[SCANNER] x
[PARSER] entering B
[SCANNER] )
[PARSER] exiting B
[PARSER] exiting D
[PARSER] exiting C
[SCANNER] .
[PARSER] exiting B
[SCANNER] $
[PARSER] exiting A
Successful parse
```



Error Recovery in Recursive-Descent Parsers

- Remember that when a procedure A that recognizes language phrase $A \rightarrow \alpha$ is called, the lookahead buffer should contain a symbol from FIRST(α)
- Also, when you leave procedure A, the lookahead buffer should contain a symbol from FOLLOW(A)
- Possible strategy:

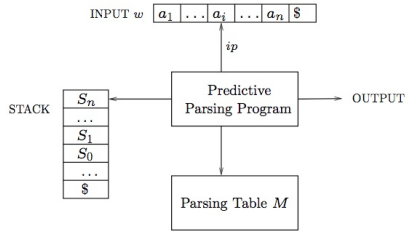
```
procedure A;
begin
 if not lookahead in FIRST(A)
   then report error and SKIPTO(FIRST(A) ∪ FOLLOW(A))
 if lookahead in FIRST(A)
   then parse phrase A
        if not lookahead in FOLLOW(A)
          then report an error and SKIPTO(FOLLOW(A))
end;
```

- This often skips too many symbols, however (why?)

Error Recovery in Recursive-Descent Parsers

- To avoid skipping too many symbols, the parser can pass a set FOLLOWERS to procedure A that contains all of the symbols you can skip to upon an error
- FOLLOWERS is constructed to include:
  1. the set FOLLOW(A)
  2. symbols that have been passed as FOLLOWERS to the procedure(s) that called A
  3. so-called *beacon* symbols that should never be skipped

Table-driven Predictive Parsing



(a = current input symbol, ip = instruction pointer)

set ip to the first symbol in w; push start non-terminal onto the stack
set X to the top of stack symbol
while ($X \neq \$$) /* stack is not empty */
  if (X is a) pop the stack and advance ip
  else if (X is a terminal) *error()*
  else if ($M[X, a]$ is an error entry) *error()*
  else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
    output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$
    pop the stack
    push Y_k,Y_{k-1},. . .,Y_1 onto the stack, with Y_1 on top
  }
  set X to the top of stack symbol
}

Table-driven Predictive Parsing

- Grammar:

$E \rightarrow T\,E'$
$E' \rightarrow +\,T\,E' \mid \epsilon$
$T \rightarrow F\,T'$
$T' \rightarrow *\,F\,T' \mid \epsilon$
$F \rightarrow (\,E\,) \mid \mathbf{id}$

```
set ip to the first symbol in w;
push start non-terminal onto the stack
set X to the top of stack symbol
while ( X ≠ $ ) /* stack is not empty */
  if ( X is a ) pop the stack and advance ip
  else if ( X is a terminal ) error()
  else if ( M[X, a] is an error entry ) error()
  else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
      output the production X → Y1Y2 ··· Yk
      pop the stack
      push Yk,Yk−1,. . .,Y1 onto the stack, with Y1 on top
  }
  set X to the top of stack symbol
}
```

- Parse table:

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

Table-driven Predictive Parsing

- Parse of '**id** + **id** ∗ **id**':

MATCHED	STACK	INPUT	ACTION
	$E\$$	id + id ∗ id$	
	$TE'\$$	id + id ∗ id$	output $E \rightarrow TE'$
	$FT'E'\$$	id + id ∗ id$	output $T \rightarrow FT'$
	id $T'E'\$$	id + id ∗ id$	output $F \rightarrow$id
id	$T'E'\$$	+ id ∗ id$	match id
id	$E'\$$	+ id ∗ id$	output $T' \rightarrow \epsilon$
id	$+TE'\$$	+ id ∗ id$	output $E' \rightarrow +TE'$
id +	$TE'\$$	id ∗ id$	match +
id +	$FT'E'\$$	id ∗ id$	output $T \rightarrow FT'$
id +	id $T'E'\$$	id ∗ id$	output $F \rightarrow$ id
id + id	$T'E'\$$	∗ id$	match id
id + id	$*FT'E'\$$	∗ id$	output $T' \rightarrow *FT'$
id + id ∗	$FT'E'\$$	id$	match ∗
id + id ∗	id $T'E'\$$	id$	output $F \rightarrow$ id
id + id ∗ id	$T'E'\$$	$	match id
id + id ∗ id	$E'\$$	$	output $T' \rightarrow \epsilon$
id + id ∗ id	$\$$	$	output $E' \rightarrow \epsilon$

Error Recovery in Table-driven Predictive Parsers[3]

- A heuristic is used to determine a set of synchronizing tokens for productions
- New parse table for our example (where sync = set of sync tokens):

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	sync	sync
E'		$E \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	sync		$T \rightarrow FT'$	sync	sync
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**	sync	sync	$F \rightarrow (E)$	sync	sync

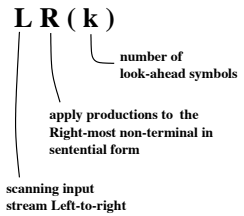
- When the parser looks up table entry $M[A, a]$:
  - if it is blank: input symbol a is skipped
  - if it is sync: nonterminals on top of stack are popped until token on top of stack is in sync set; then if the token on the top of the stack does not match a, pop the token from the stack as before
- Another popular method: call error handling routine whenever error entry reached in table

Properties of LL(1) Parsers

- a correct, left-most parse is constructed
- all grammars in the LL(1) class are unambiguous
- all table-driven LL(1) parsers operate in linear time and space with respect to the length of the parsed input

LR Parsing

- The fundamental concern of LL parsers is which production to choose in expanding a given nonterminal
- Weakness of LL(k) parsers: must predict which production to use having seen only k tokens of the right-hand-side (RHS)
- More powerful technique: LR(k) bottom-up parsing



- Look-ahead k assumed to be 1 if not explicitly stated

LR Parsing

- LR parsers:
  - begin with the parse tree's leaves and move towards it root
  - trace a rightmost derivation (in reverse)
  - uses grammar rules to replace a rule's right-hand side with its left hand side
- LR parsers can be constructed to recognize virtually all programming language constructs
- The class of grammars that can be parsed using LR parsing is a proper superset of the class of grammars that can be parsed with LL predictive parsing
- Principle drawback: difficult to construct LR parsers by hand
- Therefore usually done using a LR parser generator (a 'compiler-compiler')

LR Parsing

- Also called **shift-reduce** parsing
- The parser has a **stack** and an **input**. The first k tokens of the input are the **look-ahead**. Based on the contents of the stack and the look-ahead, the parser performs one of 2 possible actions:

  Shift push the first input token onto the stack

  Reduce choose a grammar rule $X \rightarrow \alpha\beta\gamma$; pop γ, β, α from the stack; push X onto the stack
- Initially: stack is empty, parser is at beginning of input
- Parser *accepts* string when end-of-file marker ($) is shifted and there is no more input

LR Parsing

$S \rightarrow E\,\$$

$E \rightarrow E + T$ | $T \rightarrow T * F$ | $F \rightarrow \text{id}$
$E \rightarrow E - T$ | $T \rightarrow T / F$ | $F \rightarrow \text{num}$
$E \rightarrow T$ | $T \rightarrow F$ | $F \rightarrow (\,E\,)$

GRAMMAR 3.10.
From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$:=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
1 id$_4$:=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 :$_3$	b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$	:= c + (d := 5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$	c + (d := 5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ id$_{20}$	+ (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ id
1 S_2 :$_3$ id$_4$:=$_6$ E_{11}	+ (d := 5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8	d := 5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$	:= 5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$	5 + 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ num$_{10}$	+ 6 , d) $	*reduce* $E \rightarrow$ num
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11}	+ 6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$	6 , d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$ num$_{10}$	, d) $	*reduce* $E \rightarrow$ num
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	*reduce* $E \rightarrow E + E$
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11}	, d) $	*reduce* $S \rightarrow$ id:=E
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12}	, d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 $S_{12\,,18}$	d) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 $S_{12\,,18}$ id$_4$	) $	*reduce* $E \rightarrow$ id
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 $S_{12\,,18}$ E_{21}	) $	*shift*
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 $S_{12\,,18}$ E_{21})$_{22}$	$	*reduce* $E \rightarrow (S; E)$
1 S_2 :$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	$	*reduce* $E \rightarrow E + E$
1 S_2 :$_3$ id$_4$:=$_6$ E_{11}	$	*reduce* $S \rightarrow$ id:=E
1 S_2 :$_3$ S_5	$	*reduce* $S \rightarrow S; S$
1 S_2	$	*accept*

FIGURE 3.18. Shift-reduce parse of a sentence. Numeric subscripts in the *Stack* are DFA state numbers; see Table 3.19.
From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

LR Parsing Engine

- How does the LR parser know when to shift and when to reduce? By using a DFA!
- DFA applied to stack, not input
- Edges of the DFA are labeled by the symbols (terminals and nonterminals) that can appear on the stack

LR Parsing Engine

	id	num	print	:	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9												g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

TABLE 3.19. LR parsing table for Grammar 3.1.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

1. $S \rightarrow S\ ;\ S$
2. $S \rightarrow \text{id}\ :=\ E$
3. $S \rightarrow \text{print}\ (\ L\ )$
4. $E \rightarrow \text{id}$
5. $E \rightarrow \text{num}$
6. $E \rightarrow E\ +\ E$
7. $E \rightarrow (\ S\ ,\ E\ )$
8. $L \rightarrow E$
9. $L \rightarrow L\ ,\ E$

GRAMMAR 3.1. A syntax for straight-line programs.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

s*n* Shift into state *n*
g*n* Goto state *n*
r*k* Reduce by rule *k*
a Accept
Error (a blank entry in table)

LR Parsing Algorithm

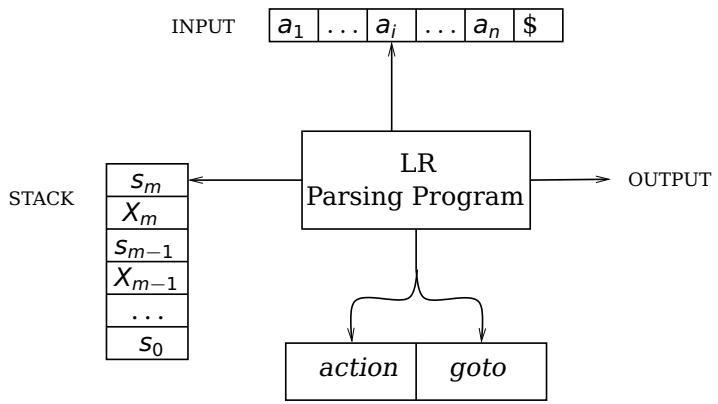
Step 1 Look up action in table using top stack state and input symbol

Step 2

If action is:	**Do:**
Shift(n)	Push n onto stack, advance input one token
Reduce(k)	1. Pop x symbols off stack, where x = number of symbols on the RHS of rule k 2. Let X be the LHS symbol of rule k; use state currently on top of stack and X to index into table to get 'goto n' 3. Push n onto stack
Accept	Stop parsing, report **success**
Error	Stop parsing, report **failure**

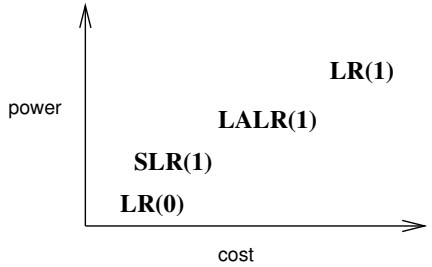
Step 3 Goto **Step 1**

Model of a LR Parser



The LR Parsing Family

LR(0)	LR parser with no lookahead
SLR(1)	*Simple* LR parser, single lookahead; LR(0) parser that uses FOLLOW sets to resolve conflicts
LR(1)	canonical LR parser, single lookahead
LALR(1)	*Lookahead* LR parser, single lookahead; optimized LR(1) parser



LR Parse Tables – The Big Picture

- Example grammar L_1 & parse table:

0: s → e 4: t → f
1: e → e + t 5: f → (e)
2: e → t 6: f → **NUM**
3: t → t ∗ f

	$	NUM	+	∗	(	)	s	e	t	f
0	–	s1	–	–	s2	–	–	g3	g4	g5
1	r6	–	r6	r6	–	r6	–	–	–	–
2	–	s1	–	–	s2	–	–	g6	g4	g5
3	acc	–	s7	–	–	–	–	–	–	–
4	r2	–	r2	s8	–	r2	–	–	–	–
5	r4	–	r4	r4	–	r4	–	–	–	–
6	–	–	s7	–	–	s9	–	–	–	–
7	–	s1	–	–	s2	–	–	–	g10	g5
8	–	s1	–	–	s2	–	–	–	–	g11
9	r5	–	r5	r5	–	r5	–	–	–	–
10	r1	–	r1	s8	–	r1	–	–	–	–
11	r3	–	r3	r3	–	r3	–	–	–	–

LR Parse Tables – The Big Picture

0: s → e 4: t → f
1: e → e + t 5: f → (e)
2: e → t 6: f → **NUM**
3: t → t ∗ f

	$	NUM	+	∗	(	)	s	e	t	f
0	–	s1	–	–	s2	–	–	g3	g4	g5
1	r6	–	r6	r6	–	r6	–	–	–	–
2	–	s1	–	–	s2	–	–	g6	g4	g5
3	acc	–	s7	–	–	–	–	–	–	–
4	r2	–	r2	s8	–	r2	–	–	–	–
5	r4	–	r4	r4	–	r4	–	–	–	–
6	–	–	s7	–	–	s9	–	–	–	–
7	–	s1	–	–	s2	–	–	–	g10	g5
8	–	s1	–	–	s2	–	–	–	–	g11
9	r5	–	r5	r5	–	r5	–	–	–	–
10	r1	–	r1	s8	–	r1	–	–	–	–
11	r3	–	r3	r3	–	r3	–	–	–	–

	Stack		Input	Next Action
	States	Symbols		
(1)	0		1 ∗ (2 + 3) $	shift NUM
(2)	0 1	NUM	∗ (2 + 3) $	reduce by 6
(3)	0	f	∗ (2 + 3) $	goto 5
(4)	0 5		∗ (2 + 3) $	reduce 4
(5)	0	t	∗ (2 + 3) $	goto 4
(6)	0 4		∗ (2 + 3) $	shift *
(7)	0 4 8	t *	(2 + 3) $	shift (
(8)	0 4 8 2	t * (	2 + 3) $	shift NUM
(9)	0 4 8 2 1	t * (NUM	+ 3) $	reduce 6
(10)	0 4 8 2	t * (f	+ 3) $	goto 5
(11)	0 4 8 2 5	t * (f	+ 3) $	reduce 4
(12)	0 4 8 2	t * (t	+ 3) $	goto 4

LR Parse Tables – The Big Picture

0: s → e 4: t → f
1: e → e + t 5: f → (e)
2: e → t 6: f → **NUM**
3: t → t ∗ f

	$	NUM	+	∗	(	)	s	e	t	f
0	–	s1	–	–	s2	–	–	g3	g4	g5
1	r6	–	r6	r6	–	r6	–	–	–	–
2	–	s1	–	–	s2	–	–	g6	g4	g5
3	acc	–	s7	–	–	–	–	–	–	–
4	r2	–	r2	s8	–	r2	–	–	–	–
5	r4	–	r4	r4	–	r4	–	–	–	–
6	–	–	s7	–	–	s9	–	–	–	–
7	–	s1	–	–	s2	–	–	–	g10	g5
8	–	s1	–	–	s2	–	–	–	–	g11
9	r5	–	r5	r5	–	r5	–	–	–	–
10	r1	–	r1	s8	–	r1	–	–	–	–
11	r3	–	r3	r3	–	r3	–	–	–	–

	Stack		Input	Next Action
	States	Symbols		
(12)	0 4 8 2	t * (t	+ 3) $	goto 4
(13)	0 4 8 2 4	t * (t	+ 3) $	reduce 2
(14)	0 4 8 2	t * (e	+ 3) $	goto 6
(15)	0 4 8 2 6	t * (e	+ 3) $	shift +
(16)	0 4 8 2 6 7	t * (e +	3) $	shift NUM
(17)	0 4 8 2 6 7 1	t * (e + NUM	) $	reduce 6
(18)	0 4 8 2 6 7	t * (e + f	) $	goto 5
(19)	0 4 8 2 6 7 5	t * (e + f	) $	reduce 4
(20)	0 4 8 2 6 7	t * (e + t	) $	goto 10
(21)	0 4 8 2 6 7 10	t * (e + t	) $	reduce 1
(22)	0 4 8 2	t * (e	) $	goto 6
(23)	0 4 8 2 6	t * (e	) $	shift)

LR Parse Tables – The Big Picture

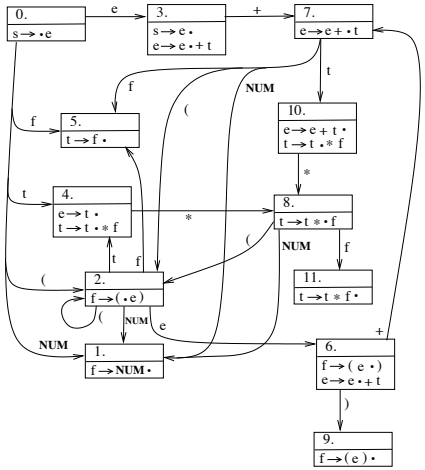
0: s → e 4: t → f
1: e → e + t 5: f → (e)
2: e → t 6: f → **NUM**
3: t → t ∗ f

	$	NUM	+	∗	(	)	s	e	t	f
0	–	s1	–	–	s2	–	–	g3	g4	g5
1	r6	–	r6	r6	–	r6	–	–	–	–
2	–	s1	–	–	s2	–	–	g6	g4	g5
3	acc	–	s7	–	–	–	–	–	–	–
4	r2	–	r2	s8	–	r2	–	–	–	–
5	r4	–	r4	r4	–	r4	–	–	–	–
6	–	–	s7	–	–	s9	–	–	–	–
7	–	s1	–	–	s2	–	–	–	g10	g5
8	–	s1	–	–	s2	–	–	–	–	g11
9	r5	–	r5	r5	–	r5	–	–	–	–
10	r1	–	r1	s8	–	r1	–	–	–	–
11	r3	–	r3	r3	–	r3	–	–	–	–

	Stack		Input	Next Action
	States	Symbols		
(23)	0 4 8 2 6	t ∗ (e	) $	shift)
(24)	0 4 8 2 6 9	t ∗ (e)	$	reduce 5
(25)	0 4 8	t ∗ f	$	goto 11
(26)	0 4 8 11	t ∗ f	$	reduce 3
(27)	0	t	$	goto 4
(28)	0 4	t	$	reduce 2
(29)	0	e	$	goto 3
(30)	0 3	e	$	**accept**

Creating LR Parse Tables – The Big Picture

- LR State-Machine S_1 for language L_1:



Creating LR Parse Tables – The Big Picture

- Relating the table to the graph:
  - Each box is a state with the state number being the row in the parse table
  - Edges are labeled with either a terminal or nonterminal (never both)
  - Edges labeled with a terminal relate to shift directives in the table
  - Use the goto portion of the table to determine the next state when traversing edges labeled with a nonterminal
- Productions at the bottom of each state are called **LR(0) items**; LR(0) items are productions augmented by a dot
- Dots are used to indicate the present position of the parse, i.e. what is on the parse stack

Creating LR Parse Tables – The Big Picture

- A dot all the way to the right of a production indicates the parser can reduce using that production
- Shifts cause the dot to move from the left of the symbol labeling the traversed edge to the right of that symbol
- Reductions first require moving backwards (against the arrows) one edge for every symbol on the right hand side of the production followed by moving to the indicated state

Creating LR Parse Tables

- How do we create an LR parse table?
- Main steps:
  1. create LR(0) item sets
  2. generate the table from items

LR(0) Parsing

- LR(0) grammars are those that can be parsed looking only at the stack
- Too weak of a parsing method to be of much use
- However, generating an LR(0) parse table is the usual starting point for many LR parsers
- Producing an LR(0) table involves first building LR(0) item sets

Creating LR(0) Items

Step 1 Initialization

- Create state 0 with the start production with a **dot** at far left of the RHS
- Initial set of items in any state is called the **kernel** or **seed** items

0
s $\rightarrow \cdot$ **e**

0: s $\rightarrow$ e	4: t $\rightarrow$ f
1: e $\rightarrow$ e + t	5: f $\rightarrow$ (e)
2: e $\rightarrow$ t	6: f $\rightarrow$ **NUM**
3: t $\rightarrow$ t * f	

Creating LR(0) Items

Step 2 Close the kernel items

- Kernel items for next states are created with the **closure** operation
- Closure is performed on kernel items that have a nonterminal to the right of a dot
- Begin the closure process by adding all productions of the nonterminal that follows the dot

0
s $\rightarrow \cdot$ **e**
e $\rightarrow \cdot$ **e** + **t**
e $\rightarrow \cdot$ **t**

0: s $\rightarrow$ e	4: t $\rightarrow$ f
1: e $\rightarrow$ e + t	5: f $\rightarrow$ (e)
2: e $\rightarrow$ t	6: f $\rightarrow$ **NUM**
3: t $\rightarrow$ t * f	

Creating LR(0) Items

- Closing state 0:
  - Since the dot is to the left of *e*, add the *e* productions

0
s → ·**e**
e → ·**e** + **t**
e → ·**t**

0: s → e
1: e → e + t
2: e → t
3: t → t ∗ f
4: t → f
5: f → (e)
6: f → **NUM**

  - Note we have new items with a dot to the left of a nonterminal, so we need to add the *t* productions

0
s → ·**e**
e → ·**e** + **t**
e → ·**t**
t → ·**t** ∗ **f**
t → ·**f**

Creating LR(0) Items

- Closing state 0 *continued*:
  - Repeat this process until no more items can be added
  - Result:

0
s → ·**e**
e → ·**e** + **t**
e → ·**t**
t → ·**t** ∗ **f**
t → ·**f**
f → ·(**e**)
f → ·**NUM**

0: s → e
1: e → e + t
2: e → t
3: t → t ∗ f
4: t → f
5: f → (e)
6: f → **NUM**

Creating LR(0) Items

Step 3 Partition the kernel items

- Group together all items that have the same symbol (called the **transition** symbol) to the right of the dot
- All items with the same transition symbol form a **partition**

0
s → ·**e**
e → ·**e** + **t**
e → ·**t**
t → ·**t** ∗ **f**
t → ·**f**
f → ·(**e**)
f → ·**NUM**

Creating LR(0) Items

Step 4 Form new states and next-state transitions as necessary

- Partitions form new states *except* those with dots at the far right (all ϵ productions are part of this latter partition)
- Modify productions in the potentially new state by moving the dot one symbol to the right
- If a state already exists that has the same set of kernel items as the modifed partition, add an edge labeled with the transition symbol from the current state to the existing state

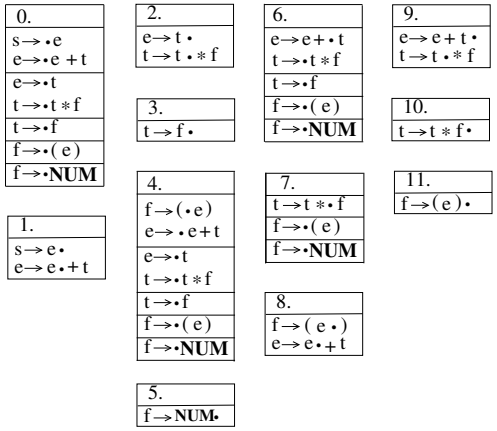
Creating LR(0) Items

Step 4 continued

- The transition symbol is the symbol you moved the dot over
- If there is no existing state with the same kernel items as a given partition, create a new state with an edge from the current state labeled with the transition symbol

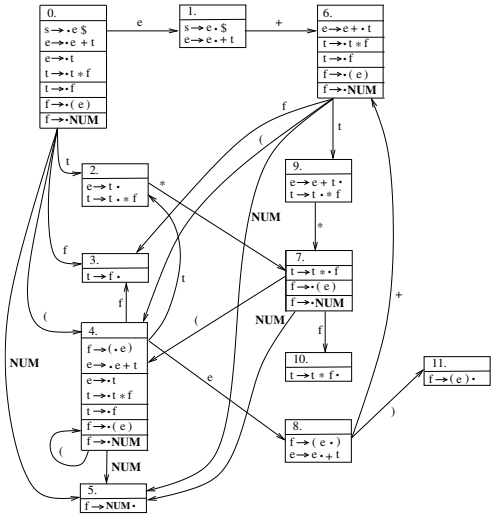
Step 5 Repeat steps 2–4 until no more new states can be added to the diagram

Creating LR(0) Items



- Item set numbering is irrelevant

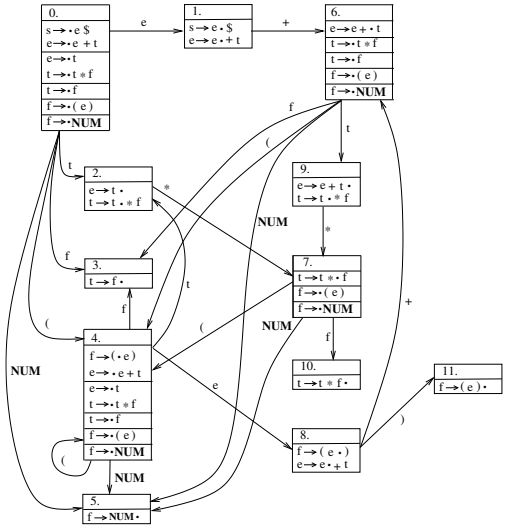
Creating LR(0) Items



Creating LR Parse Tables from Item Sets

- For each edge $I \xrightarrow{X} J$ where
  - X is a terminal: in the action table, insert **shift** J at position (I, X)
  - X is a nonterminal: in the goto table, insert **goto** J at position (I, X)
- For each state containing a production n item $A \to \gamma\cdot$, insert a **reduce** n action at (I, Y) for every token Y in FOLLOW(n)
- For each state I containing an item $S \to \gamma \cdot \$$, insert an **accept** action at $(I, \$)$

Creating LR Parse Tables from Item Sets



For each state containing a production n item
For each state containing an item $S \rightarrow \gamma \cdot \$$,
$A \rightarrow \mathbf{x}$ is in reduce actions (rule n),
insert an **accept** action at $(n, \$)$
for every terminal x in FOLLOW(A), insert **shift** j at position (i, x)

$n = 0, i = 1$ FOLLOW(0) = {$}

	$	NUM	+	*	(	)
0	–	–s5	s5	–	–	–s4	s4	–
1	–r0	r0	acc	acc	–	–s6	s6	–	–	–
2	–r2	r2	–	–r2	r2	–s7	s7	–	–r2	r
3	–r4	r4	–	–r4	r4	–r4	r4	–	–r4	r
4	–	–s5	s5	–	–	–	–	–s5	s5	–
5	–r6	r6	–	–r6	r6	–r6	r6	–	–r6	r
6	–	–s5	s5	–	–	–s4	s4	–
7	–	–s5	s5	–	–	–s4	s4	–
8	–	–	–	–s6	s6	–	–	–s11	s
9	–r1	r1	–	–r1	r1	–s7	s7	–	–r1	r
10	–r3	r3	–	–r3	r3	–r3	r3	–	–r3	r
11	–r5	r5	–	–r5	r5	–r5	r5	–	–r5	r

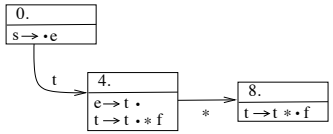
0: s → e
1: e → e + t
2: e → t
3: t → t * f
4: t → f
5: f → (e)
6: f → **NUM**

Creating LR Parse Tables from Item Sets

	$	NUM	+	*	(	)	s	e	t	f
0	–	s5	–	–	s4	–	–	g1	g2	g3
1	acc	–	s6	–	–	–	–	–	–	–
2	r2	–	r2	s7	–	r2	–	–	–	–
3	r4	–	r4	r4	–	r4	–	–	–	–
4	–	s5	–	–	s5	–	–	g8	g2	g3
5	r6	–	r6	r6	–	r6	–	–	–	–
6	–	s5	–	–	s4	–	–	–	g9	g3
7	–	s5	–	–	s4	–	–	–	–	g10
8	–	–	s6	–	–	s11	–	–	–	–
9	r1	–	r1	s7	–	r1	–	–	–	–
10	r3	–	r3	r3	–	r3	–	–	–	–
11	r5	–	r5	r5	–	r5	–	–	–	–

Problems with LR(0) State Machines

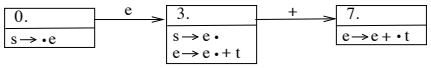
- Consider state 4 in S_1, reproduced below:



- If you are currently in state 4, should you shift or reduce?
- Such situations are called **shift/reduce conflicts**
- Can also have **reduce/reduce conflicts**—two or more productions in the same state with dots all the way to the right
- States that contain these conflicts are called **inadequate** states
- A language is considered LR(0) is its state machine has no inadequate states
- Inadequate states produce multiple entries in an LR(0) parse table

SLR(1) Grammars

- Can solve most LR(0) shift/reduce errors by considering the next token in the input stream
- In an LR(0) parser we always reduce when a state contains a production in the form of $S \rightarrow \gamma \cdot$; in an SLR parser we reduce for states that contain $S \rightarrow \gamma \cdot$ *and* the lookahead symbol is in FOLLOW(S)
- Consider state 3 in S_1, reproduced below:



  - Reduce when lookahead is in FOLLOW(s), shift otherwise
  - Works if + is not a member of FOLLOW(s)
- If all LR(0) conflicts can be resolved using this scheme, language is said to be SLR(1)

LR(1) Grammars

- Many grammars are not SLR—a FOLLOW set may contain symbols that also label an outgoing edge
- Note that a nonterminal's FOLLOW set contains *all* symbols that can follow that nonterminal in *every* context; however, we only care about the follow symbols for the current state
- This set of relevant lookahead symbols is called the **lookahead set**; typically it is a subset of the complete FOLLOW set
- An **LR(1) item** is an LR(0) item augmented with a lookahead symbol from this lookahead set
- The lookahead symbol is actually part of an item—two items with the same production and dot position but with different lookaheads are different items

Creating LR(1) Items

- LR(1) item $[s \rightarrow \alpha \cdot \beta, D]$ consists of a production, a dot, and a lookahead D
- Process same as creating LR(0) items except LR(1) items created during closure
- Initial item for L_1: $[s \rightarrow \cdot e, \$]$
- Given the following LR(1) kernel item and production,

  $$[s \rightarrow \alpha \cdot x\beta, C]$$
  $$x \rightarrow \cdot\gamma$$

  create the new LR(1) item using

  $$[x \rightarrow \cdot\gamma, \text{FIRST}(\beta, C)]$$

  where α and β can be null, and

  $$\text{FIRST}(\beta, C) = \begin{cases} \text{FIRST}(\beta) \cup C, & \text{if } \beta == \text{null} \\ \text{FIRST}(\beta), & \text{otherwise} \end{cases}$$

- Continue in the same manner as with LR(0) until no more LR(1) items can be created

LALR(1) Grammars

- Problem with LR(1) parsers: the state machine (and therefore the parse table) is typically twice the size of an equivalent LR(0) machine
- Many states in LR(1) machines differ only in their lookahead symbols; LALR parsers merge many of these states
- Though uncommon, these merges sometimes produce inadequate states
- LALR(1) parsers have the same number of states as LR(0)/SLR parsers with almost the power of LR(1) parsers
- Almost all modern programming languages are LALR(1)
- Most compiler generators produce LALR parsers

Hierarchy of Grammar Classes

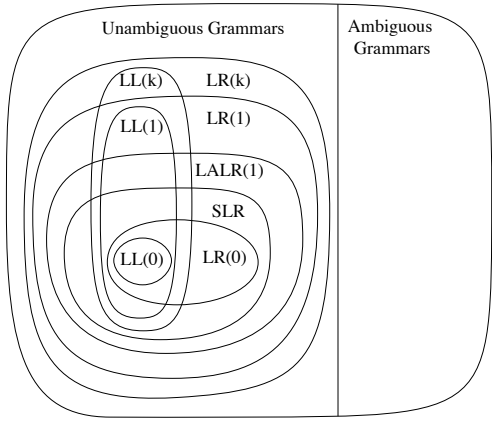


FIGURE 3.30. A hierarchy of grammar classes.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Error-recovery in LR Parsers

- Errors detected when lookup in parsing table produces error entry
- General error-recovery scheme:
  - scan down stack until a state *s* with a goto on a particular nonterminal *A* is found (might be more than one choice for *A*)
  - zero or more input symbols are then discarded until a symbol *a* is found that can legitimately follow *A*
  - parser then stacks the state GOTO(*s*,*A*) and resumes normal parsing
- This method of recovery attempts to eliminate the phrase containing the syntax error
- As in table-drive predicative parsing, can also call special error handling routines when error entry encountered

Definitions

- An attribute grammar is a context-free grammar that has been extended to provide context-sensitive information by appending attributes to some of its nonterminals
- Each distinct symbol in the grammar has associated with it a finite, possibly empty, set of attributes
  - Each attribute has a domain of possible values
  - An attribute may be assigned values from its domain during parsing
  - Attributes can be evaluated in assignments or conditions
- Two Classes of Attributes:

  Synthesized attributes: An attribute that gets it values from the attributes attached to children of its nonterminal

  Inherited attributes: An attribute that gets it values from the attributes attached to the parent (or siblings) of its nonterminal

An Example

- Consider again the context-sensitive language L:

$$L = \{a^n b^n c^n \mid n \geq 1\}$$

- No context-free grammar generates L!
- An attempt:

  string → a_seq b_seq c_seq
  a_seq → **a** | a_seq **a**
  b_seq → **b** | b_seq **b**
  c_seq → **c** | c_seq **c**

- This context-free grammar generates the language L':

$$L' = a^+ b^+ c^+ = \{a^k b^m c^n \mid k \geq 1, m \geq 1, n \geq 1\}$$

Using an Attribute Grammar

- Attach an integer synthesized attribute, *size*, to each of the nonterminals a_seq, b_seq, c_seq
- Impose a condition on the first production:

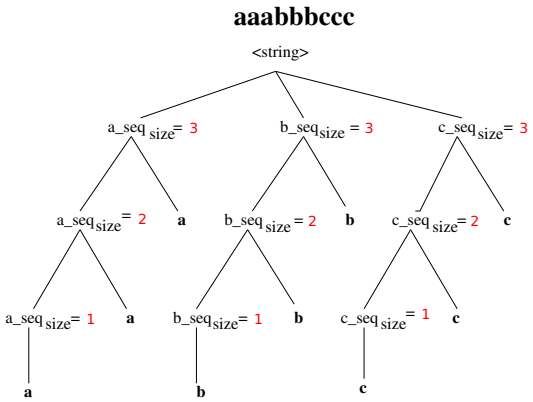
  string → $\texttt{a\_seq}_{size}$ $\texttt{b\_seq}_{size}$ $\texttt{c\_seq}_{size}$
  condition
  $\texttt{a\_seq}_{size} = \texttt{b\_seq}_{size} = \texttt{c\_seq}_{size}$

  $\texttt{a\_seq}_{1_{size}}$ → **a**
  $\texttt{a\_seq}_{1_{size}} \leftarrow 1$
  | $\texttt{a\_seq}_{2_{size}}$ **a**
  $\texttt{a\_seq}_{1_{size}} \leftarrow \texttt{a\_seq}_{2_{size}} + 1$

  $\texttt{b\_seq}_{1_{size}}$ → **b**
  $\texttt{b\_seq}_{1_{size}} \leftarrow 1$
  | $\texttt{b\_seq}_{2_{size}}$ **b**
  $\texttt{b\_seq}_{1_{size}} \leftarrow \texttt{a\_seq}_{2_{size}} + 1$

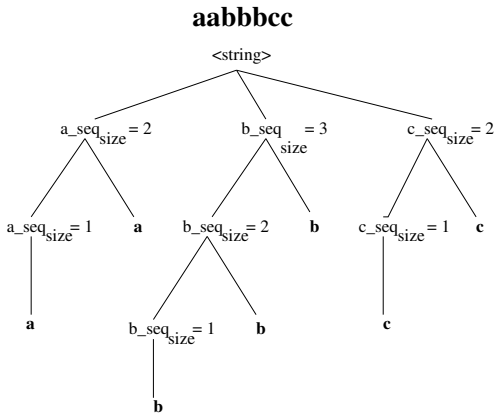
  $\texttt{c\_seq}_{1_{size}}$ → **c**
  $\texttt{c\_seq}_{1_{size}} \leftarrow 1$
  | $\texttt{c\_seq}_{2_{size}}$ **c**
  $\texttt{c\_seq}_{1_{size}} \leftarrow \texttt{c\_seq}_{2_{size}} + 1$

Decorating a Parse Tree



string $\rightarrow$ a_seq_{size} b_seq_{size} c_seq_{size}
condition
$a\_seq_{size} = b\_seq_{size} = c\_seq_{size}$
$a\_seq1_{size} \rightarrow$ **a**
$a\_seq1_{size} \leftarrow 1$
| a_seq2_{size} **a**
$a\_seq1_{size} \leftarrow a\_seq2_{size} + 1$
$b\_seq1_{size} \rightarrow$ **b**
$b\_seq1_{size} \leftarrow 1$
| b_seq2_{size} **b**
$b\_seq1_{size} \leftarrow a\_seq2_{size} + 1$
$c\_seq1_{size} \rightarrow$ **c**
$c\_seq1_{size} \leftarrow 1$
| c_seq2_{size} **c**
$c\_seq1_{size} \leftarrow c\_seq2_{size} + 1$

A problem?



string $\rightarrow$ a_seq_{size} b_seq_{size} c_seq_{size}
condition
$a\_seq_{size} = b\_seq_{size} = c\_seq_{size}$
$a\_seq1_{size} \rightarrow$ **a**
$a\_seq1_{size} \leftarrow 1$
| a_seq2_{size} **a**
$a\_seq1_{size} \leftarrow a\_seq2_{size} + 1$
$b\_seq1_{size} \rightarrow$ **b**
$b\_seq1_{size} \leftarrow 1$
| b_seq2_{size} **b**
$b\_seq1_{size} \leftarrow a\_seq2_{size} + 1$
$c\_seq1_{size} \rightarrow$ **c**
$c\_seq1_{size} \leftarrow 1$
| c_seq2_{size} **c**
$c\_seq1_{size} \leftarrow c\_seq2_{size} + 1$

Using an Inherited Attribute

- Attach a synthesized attribute *size* to a_seq and and inherited attributes *inSize* to b_seq and c_seq

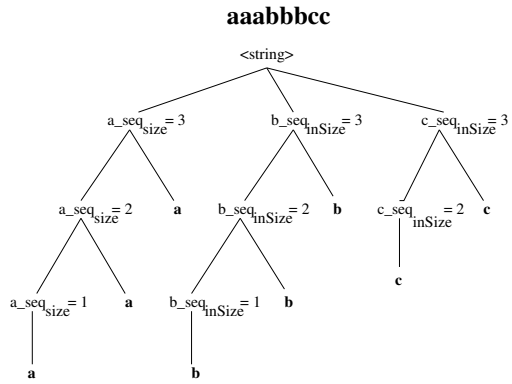
string $\rightarrow$ a_seq_{size} b_seq_{inSize} c_seq_{inSize}
$b\_seq_{inSize} \leftarrow a\_seq_{size}$
$c\_seq_{inSize} \leftarrow a\_seq_{size}$

$a\_seq_{size} \rightarrow$ **a**
$a\_seq_{size} \leftarrow 1$
| a_seq2_{size} **a**
$a\_seq_{size} \leftarrow a\_seq2_{size} + 1$

$b\_seq_{inSize} \rightarrow$ **b**
condition: $b\_seq_{inSize} = 1$
| b_seq2_{inSize} **b**
$b\_seq2_{inSize} \leftarrow b\_seq_{inSize} - 1$

$c\_seq_{inSize} \rightarrow$ **c**
condition: $c\_seq_{inSize} = 1$
| c_seq2_{inSize} **c**
$c\_seq2_{inSize} \leftarrow c\_seq_{inSize} - 1$

Using Inherited & Synthesized Attributes



string $\rightarrow$ a_seq_{size} b_seq_{inSize} c_seq_{inSize}
$b\_seq_{inSize} \leftarrow a\_seq_{size}$
$c\_seq_{inSize} \leftarrow a\_seq_{size}$

$a\_seq_{size} \rightarrow$ **a**
$a\_seq_{size} \leftarrow 1$
| a_seq2_{size} **a**
$a\_seq_{size} \leftarrow a\_seq2_{size} + 1$

$b\_seq_{inSize} \rightarrow$ **b**
condition: $b\_seq_{inSize} = 1$
| b_seq2_{inSize} **b**
$b\_seq2_{inSize} \leftarrow b\_seq_{inSize} - 1$

$c\_seq_{inSize} \rightarrow$ **c**
condition: $c\_seq_{inSize} = 1$
| c_seq2_{inSize} **c**
$c\_seq2_{inSize} \leftarrow c\_seq_{inSize} - 1$

Recognizer & Translator

- Compilers typically recognize legal programs for a given language *and* translate a program into another form
- Uses context-free grammar (CFG) to specify syntax for the recognizer
- Source program is also translated to target code as phrases are being recognized by parser
- Each grammar symbol in the CFG has a (possibly empty) set of **attributes** associated with it; these attributes usually are bound to values in the domain of the target language

Syntax-Directed Translation Definitions

- Each production in the CFG has a (possibly empty) set of **semantic rules** for computing attributes
- A **syntax-directed definition** (SDD) is a CFG plus a set of semantic rules
- An **attribute grammar** is a SDD in which the semantic rules cannot have side effects
- A SDT produces an **annotated parse tree**: a parse tree containing attributes and their values at each node
- Semantic rules that can be embedded anywhere in the right-hand side of a production are often referred to as **semantic actions**

Semantic Actions

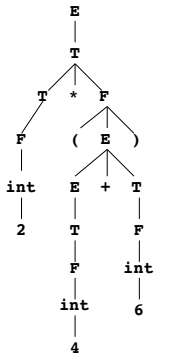
- Remember that parsers generally have two main tasks – to recognize legal sentences for a given language *and* to transform those sentences into intermediate or target code
- Semantic actions are actions performed by the parser as it recognizes phrases in the source program
- In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing actions
- In the specification file for an LR parser generator, semantic actions are fragments of program code attached to grammar productions

Example 1 – A Simple Calculator

- Goal: design a SDD that translates an arithmetic expression into its integer value, e.g. $2*(4+6)$ would result in 20.

Step 1 Design a CFG for arithmetic expressions

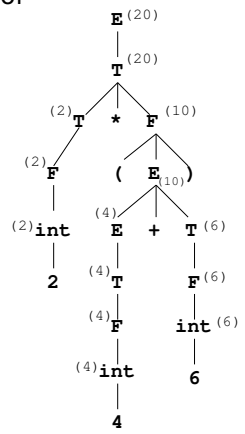
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow \textbf{int} \mid \textbf{(}\ E\ \textbf{)}$$



Example 1 – A Simple Calculator

Step 2 Determine the attributes associated with each grammar symbol

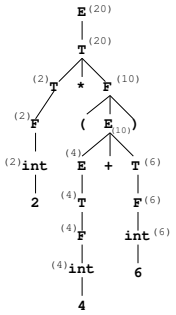
annotated parse tree



Example 1 – A Simple Calculator

Step 3 Write the semantic rules that will be used to determine the attributes.

CFG	Semantic Rules
$E_1 \rightarrow E_2$ + T	E_1.val = E_2.val + T.val
E → T	E.val = T.val
$T_1 \rightarrow T_2$ * F	T_1.val = T_2.val * F.val
T → F	T.val = F.val
F → **int**	F.val = **int**.val
F → **(** E **)**	F.val = E.val



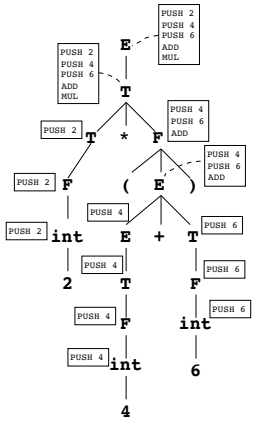
Example 2 – A Simple Translator

- Goal: Design a SDD that translates an arithmetic expression into a series of stack machine instructions, e.g.

2 * (4 + 6) ⟹

```
PUSH 2
PUSH 4
PUSH 6
ADD
MUL
```

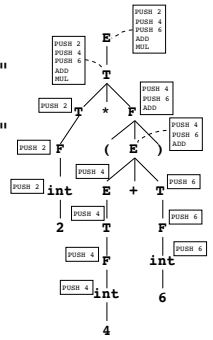
Step 1 Define the attributes



Example 2 – A Simple Translator

Step 2 Write the semantic rules that will be used to determine the attributes.

CFG	Semantic Rules
$E_1 \rightarrow E_2$ + T	E_1.code = E_2.code + T.code + " ADD"
E → T	E.code = T.code
$T_1 \rightarrow T_2$ * F	T_1.code = T_2.code + F.code + " MUL"
T → F	T.code = F.code
F → **int**	F.code = "PUSH " + **int**.val
F → **(** E **)**	F.code = E.code



Question

What parsing method (i.e. LL or LR) is being used here?

Synthesized vs. Inherited Attributes

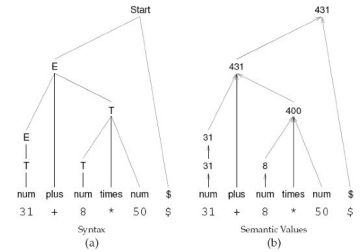


Figure 7.1: (a) Parse tree for the displayed expression; (b) Synthesized attributes transmit values up the parse tree toward the root.

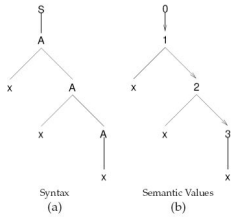


Figure 7.2: (a) Parse tree for the displayed input string; (b) Inherited attributes pass from parent to child.

Bottom-Up Syntax-Directed Translation

- Semantic actions are 'triggered' as reductions take place
- Abstractly, bottom-up parsers maintain two stacks:
  - the **syntactic (or parse) stack** containing terminal & nonterminal symbols
  - the **semantic stack** containing values associated with grammar symbols

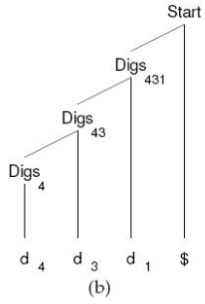
Bottom-Up Syntax-Directed Translation

1 Start $\rightarrow$ Digs_{ans} \$
call (*ans*)

2 $\text{Digs}_{up} \rightarrow \text{Digs}_{below}\ \text{d}_{next}$
$up \leftarrow below \times 10 + next$

3 | d_{first}
$up \leftarrow first$

(a)



(b)

Figure 7.3: (a) Grammar with semantic actions; (b) Parse tree and propagated semantic values for the input 4 3 1 \$.

Bottom-Up Syntax-Directed Translation

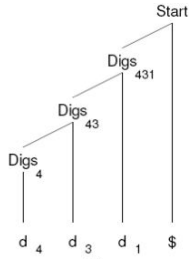
- Example for string **431**:

Sentential Form	Reduction	Action
$\text{d}_4\ \text{d}_3\ \text{d}_1\ \$$		
$\text{Digs}_4\ \text{d}_3\ \text{d}_1\ \$$	$\text{Digs}_{up} \rightarrow \text{d}_{first}$	$up \leftarrow first$
$\text{Digs}_{43}\ \text{d}_1\ \$$	$\text{Digs}_{up} \rightarrow \text{Digs}_{below}\ \text{d}_{next}$	$up \leftarrow below \times 10 + next$
$\text{Digs}_{431}\ \$$	$\text{Digs}_{up} \rightarrow \text{Digs}_{below}\ \text{d}_{next}$	$up \leftarrow below \times 10 + next$
Start	$\text{Start} \rightarrow \text{Digs}_{ans}\ \$$	**call** PRINT(*ans*431)

1 Start $\rightarrow$ Digs_{ans} \$
call (*ans*)

2 $\text{Digs}_{up} \rightarrow \text{Digs}_{below}\ \text{d}_{next}$
$up \leftarrow below \times 10 + next$

3 | d_{first}
$up \leftarrow first$

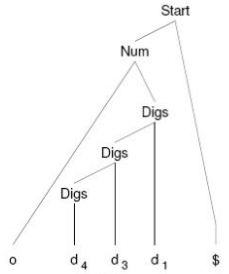


Bottom-Up Syntax-Directed Translation

- Problem: what if we wish to trigger (execute) a semantic action *before* an actual reduction takes place?

1 Start → Num $
2 Num → o Digs
3 | Digs
4 Digs → Digs d
5 | d

(a)



(b)

Figure 7.4: (a) Grammar and (b) parse tree for the input o 4 3 1 $.

Bottom-Up Syntax-Directed Translation

- Solution: use "dummy" ϵ-rules to force semantic actions

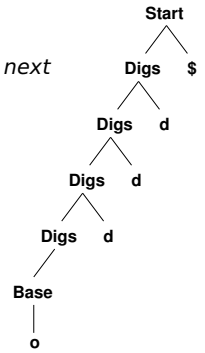
1 Start → Num_{ans} $
call (*ans*)
2 Num_{ans} → SignalOctal Digs_{octans}
ans ← *octans*
3 | SignalDecimal Digs_{decans}
ans ← *decans*
4 SignalOctal → o
base ← 8
5 SignalDecimal → λ
base ← 10
6 Digs_{up} → Digs_{below} d_{next}
$up \leftarrow below \times base + next$
7 | d_{first}
$up \leftarrow first$

Figure 7.6: Use of λ-rules to force semantic action.

Bottom-Up Syntax-Directed Translation

- Better solution avoiding a global variable:

Start → Digs_{ans} $
call PRINT(*ans.val*)
Digs_{up} → Digs_{below} d_{next}
up.base ← *below.base*
up.val ← *below.val* × *below.base* + *next*
| Base_{base}
up.base ← *base*
up.val ← 0
Base_{n} → o
n ← 8
| ϵ
n ← 10



Bottom-Up Syntax-Directed Translation

- Bottom-up SDDs are naturally implemented using LR parsers
- A partial implementation of the grammar on the previous slide is given on the next page using the syntax of CUPS

Bottom-Up Syntax-Directed Translation

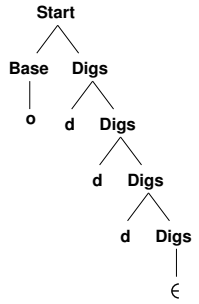
$\text{Start} \rightarrow \text{Digs}_{ans}$ \$
  call PRINT(*ans.val*)
$\text{Digs}_{up} \rightarrow \text{Digs}_{below}\ \text{d}_{next}$
  $up.base \leftarrow below.base$
  $up.val \leftarrow below.val \times below.base + next$
 | Base_{base}
  $up.base \leftarrow base$
  $up.val \leftarrow 0$
$\text{Base}_{n} \rightarrow$ o
  $n \leftarrow 8$
 | ϵ
  $n \leftarrow 10$

```
terminal Integer DIGIT;
terminal OCTAL;

non terminal start;
non terminal Integer base;
non terminal SemVal digs;

start::= digs:ans
        {: System.out.println(ans.val); :}
        ;

digs ::= digs:below DIGIT:next
        {: RESULT = new SemVal(below.base,
           below.val * below.base + next.intValue()); :}
        |
        base:b
        {: RESULT = new SemVal(b.intValue(),0); :}
        ;

base ::= OCTAL
        {: RESULT = new Integer(8); :}
        |
        {: RESULT = new Integer(10); :}
        ;
```

Bottom-Up Syntax-Directed Translation

- Though it can be awkward, bottom-up SDDs *can* be strictly implemented with LL predictive parsers
- First, rewrite the grammar:

$\text{Start} \rightarrow \text{Base}\ \text{Digs}_{ans}$
  call PRINT(*ans*)
$\text{Digs}_{up} \rightarrow \text{d}_{next}\ \text{Digs}_{below}$
  $up \leftarrow next \times multiplier + below$
  $multiplier \leftarrow multiplier \times base$
 | ϵ
  $up \leftarrow 0$
Base $\rightarrow$ o
  $base \leftarrow 8$
  $multiplier \leftarrow 1$
 | ϵ
  $base \leftarrow 10$
  $multiplier \leftarrow 1$



Bottom-Up Syntax-Directed Translation

```
import java.util.Scanner;

public class CMSC504SDTexampleTOPDOWN1 {

// assumes octal digits are legal

    private static String source;
    private static int current = 0;
    private static final char EOF = '\0'; // arbitrary

    private static char buf;
    private static int base;
    private static int multiplier;

    private static char getToken() {
        return (current < source.length() ? source.charAt(current++) : EOF);
    }

    public static void main(String [] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Input: "); source = s.nextLine();
        buf = getToken();
        start();
    }
```

Start → Base Digs
Digs → d Digs | ϵ
Base → o | ϵ

Bottom-Up Syntax-Directed Translation

```
    private static void start() {
        base();
        System.out.println(digs());
    }

    private static int digs() {
        if (buf == EOF)
            return 0;
        else if (Character.isDigit(buf)) {
            int next = ((char) buf - (char) '0');
            buf = getToken();
            int below = digs();
            int up = next * multiplier + below;
            multiplier = multiplier * base;
            return up;
        }
        else throw new RuntimeException("illegal character - "+buf);
    }

    private static void base() {
        if (buf == 'o') {
            buf = getToken(); base = 8;
        }
        else base = 10;
        multiplier = 1;
    }
}
```

Start → Base Digs
Digs → d Digs | ϵ
Base → o | ϵ

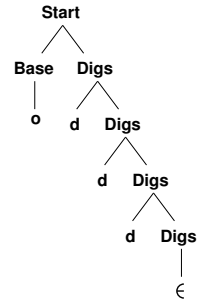
Top-Down Syntax-Directed Translation

- Top-down parsers are usually written by hand with semantic actions embedded in the code
- Recall that in a recursive-decent parser there is one subroutine for each grammar production
- Semantic values can be naturally passed via subroutine parameters
- Since values can easily be passed in *and* out via parameters, it is common in recursive-descent top-down parsers to make use of both synthesized and inherited values

Top-Down Syntax-Directed Translation

- Let's rewrite our example using both synthesized and inherited attributes (with synthesized (out) attributes in red, inherited (in) attributes in blue):

$$
\begin{aligned}
\text{Start} &\rightarrow \text{Base}_{base}\ \text{Digs}_{base,mult,ans} \\
&\quad \textbf{call}\ \text{PRINT}(ans) \\
\text{Digs}_{base,mult,up} &\rightarrow \text{d}_{next}\ \text{Digs}_{base,mult,below} \\
&\quad up \leftarrow next \times mult + below \\
&\quad mult \leftarrow mult \times base \\
&\quad \mid \epsilon \\
&\quad up \leftarrow 0 \\
&\quad mult \leftarrow 1 \\
\text{Base}_{base} &\rightarrow \text{o} \\
&\quad base \leftarrow 8 \\
&\quad \mid \epsilon \\
&\quad base \leftarrow 10
\end{aligned}
$$



- Note - no globals

Top-Down Syntax-Directed Translation

```
import java.util.Scanner;

public class CMSC504SDTexampleTOPDOWN2 {

// assumes octal digits are legal

    private static String source;
    private static int current = 0;
    private static final char EOF = '\0'; // arbitrary

    private static char buf;

    private static char getToken() {
        return (current < source.length() ? source.charAt(current++) : EOF);
    }

    public static void main(String [] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Input: "); source = s.nextLine();
        buf = getToken();
        start();
    }

    private static void start() {
        int b = base(); IntObj i = new IntObj();
        System.out.println(digs(b,i));
    }
```

Start → Base Digs
Digs → d Digs | ϵ
Base → o | ϵ

Top-Down Syntax-Directed Translation

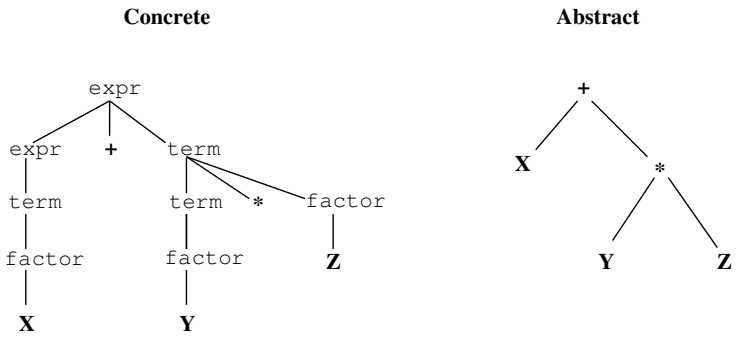
```
private static int digs(int base, IntObj mult) {
    if (buf == EOF) { // Digs --> epsilon
        mult.value = 1;
        return 0;
    } else if (Character.isDigit(buf)) {
        int next = ((char) buf - (char) '0');
        buf = getToken();
        int below = digs(base,mult);
        int up = next * mult.value + below;
        mult.value = mult.value * base;
        return up;
    }
    else throw new RuntimeException("illegal character - "+buf);
}

private static int base() {
    if (buf == 'o') {
        buf = getToken(); return 8;
    }
    else return 10;
}

private static class IntObj {
    public int value;
    public IntObj(){}; // necessary 'cause class private
}
```

Start → Base Digs
Digs → d Digs | ϵ
Base → o | ϵ

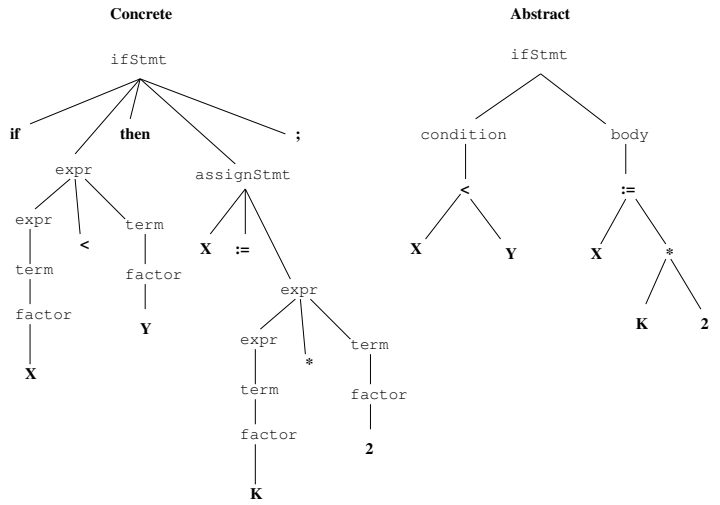
Overview

- Derivation trees from BNF grammars contain more information than is required to specify the semantics of a language
- The **abstract syntax** of a language describes the essential components of a language while ignoring inessential details
- The abstract syntax of a language can be used to create an **abstract syntax tree (AST)**
- Specifying the abstract syntax of a language means providing the possible templates for the various constructs of the language.
- Abstract syntax generally allows ambiguity since parsing has already been done
- Semantic specifications based on abstract syntax are much simpler

Abstract Syntax Trees



Abstract Syntax Trees



Concrete Syntax for Wren

<program> → **program ident is** <block>
<block> → <dec seq> **begin** <cmd-seq> **end**
<dec seq> → ϵ | <dec><dec seq>
<dec> → **var** <var list> **:** <type> **;**
<type> → **integer** | **boolean**
<var list> → <var> | **,** <var list>
<cmd seq> → <cmd> | <cmd> **;** <cmd seq>
<cmd> → **skip** | **ident :=** <expr>
| **read ident** | **write** <int expr>
| **while** <bool expr> **do** <cmd seq> **end while**
| **if** <bool expr> **then** <cmd seq> **end if**
| **if** <bool expr> **then** <cmd seq> **else** <cmd seq> **end if**
<expr> → <int expr> | <bool expr>
<int expr> → <term> | <int expr> **weakop** <term>
<term> → <element> | <term> **strongop** <element>
<element> → **number** | **ident** | **(** <int expr> **)** |– <element>
<bool expr> → <bool term> | <bool expr> **or** <bool term>
<bool term> → <bool element> | <bool term> **and** <bool element>
<bool element> → **true** | **false** | **ident** | <comparison>
| **not (** <bool expr> **)** | **(**<bool expr> **)**
<comparison> → <int expr> **relationop** <int expr>

Abstract Syntax of Wren

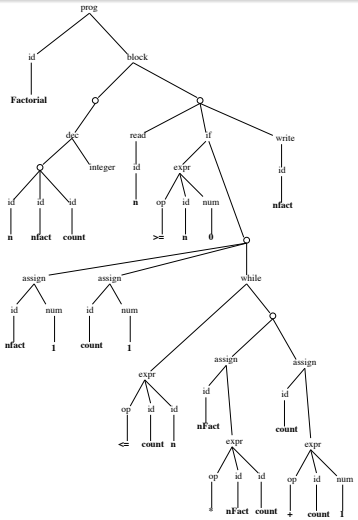
$$
\begin{aligned}
\text{Program} &\triangleq \textit{prog}(\text{Identifier, Block}) \\
\text{Block} &\triangleq \textit{block}(\text{Declaration}^{+}, \text{Command}^{+}) \\
\text{Declaration} &\triangleq \textit{dec}(\text{Identifier}^{+},\text{Type}) \\
\text{Type} &\triangleq \textit{integer} \mid \textit{boolean} \\
\text{Command} &\triangleq \textit{assign}(\text{Identifier,Expression}) \mid \textit{skip} \\
&\mid \textit{read}(\text{Identifier}) \mid \textit{write}(\text{Expression}) \\
&\mid \textit{while}(\text{Expression,Command}^{+}) \\
&\mid \textit{if}(\text{Expression,Command}^{+}) \\
&\mid \textit{ifelse}(\text{Expression,Command}^{+},\text{Command}^{+}) \\
\text{Expression} &\triangleq \text{Numeral} \mid \text{Identifier} \mid \textit{true} \mid \textit{false} \\
&\mid \textit{expr}(\text{Operator,Expression,Expression}) \\
&\mid \textit{minus}(\text{Expression}) \\
&\mid \textit{not}(\text{Expression}) \\
\text{Operator} &\triangleq + \mid - \mid * \mid / \mid \textit{or} \mid \textit{and} \\
&\mid <= \mid < \mid = \mid > \mid >= \mid <>
\end{aligned}
$$

Concrete vs. Abstract Representation

```
program Factorial is
var n,nFact,count:integer;
begin
 read n;
 if n >= 0 then
  nFact := 1; count := 1;
  while count <= n do
   nFact := nFact * count;
   count := count + 1;
  end while;
  write nFact;
 end if
end
```

```
prog(
 id(Factorial),
 block(
  [dec([id(n),id(nFact),id(count)],integer)],
  [read(id(n)),
   if(expr(op(>=),id(n),num(0)),
    [assign(id(nFact),num(1)),
     assign(id(count),num(1)),
     while(expr(op(<=),id(count),id(n)),
      [assign(id(nFact),expr(op(*),id(nFact),id(count))),
       assign(id(count),expr(op(+),id(count),num(1)))
      ]
     )
    ],
    write(id(nFact))
   )
  ]
 )
)
```


Concrete vs. Abstract Representation



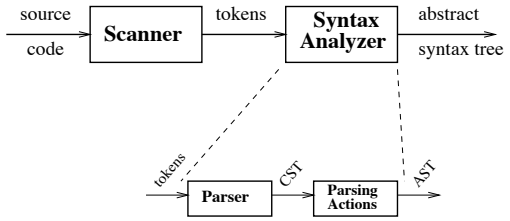
Creating the Abstract Syntax Tree

- The AST is typically constructed bottom-up; the AST data structure should therefore support tree construction from the leaves to the root
- Construction of the AST is usually done by semantic actions in the parser

Definition

Definition

A **canonical form** is the simplest or most symmetrical form to which all entities of a certain class can be reduced or transformed to without loss of generality



Overview

- Checks source program for semantic errors
- Gathers information for subsequent code-generation phase
- Main activity is usually type checking
- Relies heavily on information stored in symbol tables
- Semantic information is often specified using attribute grammars

Overview

- Used to keep track of scope and binding information for names
- Searched everytime a name is encountered in code
- Required operations at compile-time: create S.T., remove S.T., add name, locate name
- Often must search multiple symbol tables for name
- Keywords usually handled separately
- Often implemented using a hash table or “searchable stack”

Example

```
1  function f(a:int, b:int, c:int) =
2    ( print_int(a+c);
3      let var j := a+b
4          var a := "hello"
5      in print(a); print_int(j);
6      end;
7      print_int(b)
8    )
```

line 1: $\sigma_0 = \sigma_{global} \uplus \{f \mapsto \texttt{func}\}$

$\sigma_1 = \sigma_0 \uplus \{a \mapsto \texttt{int}, b \mapsto \texttt{int}, c \mapsto \texttt{int}\}$

$\uplus \equiv$ overriding union

line 2: look up a, c in σ_1

line 3,4: $\sigma_2 = \sigma_1 \uplus \{j \mapsto \texttt{int}, a \mapsto \texttt{string}\}$

line 5: look up a, j in σ_2

line 6: discard σ_2

line 7: look up b in σ_1

line 8: discard σ_1

Multiple Symbol Tables

```
structure M = struct
   structure E = struct
      val a = 5;
   end
   structure N = struct
      val b = 10
      val a = E.a + b
   end
   structure D = struct
      val d = E.a + N.a
   end
end
```

(a) An example in ML

```
package M;
class E {
    static int a = 5;
}
class N {
    static int b = 10;
    static int a = E.a + b;
}
class D {
    static int d = E.a + N.a;
}
```

(b) An example in Java

FIGURE 5.1. Several active environments at once.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

$$
\begin{aligned}
\sigma_0 &= \text{base environment} \\
\sigma_1 &= \{a \mapsto \texttt{int}\} \\
\sigma_2 &= \{E \mapsto \sigma_1\} \\
\sigma_3 &= \{b \mapsto \texttt{int}, a \mapsto \texttt{int}\} \\
\sigma_4 &= \{N \mapsto \sigma_3\} \\
\sigma_5 &= \{d \mapsto \texttt{int}\} \\
\sigma_6 &= \{D \mapsto \sigma_5\} \\
\sigma_7 &= \{M \mapsto (\sigma_2 + \sigma_4 + \sigma_6)\}
\end{aligned}
$$

Using Hash Tables

```
class Bucket {String key; Object binding; Bucket next;
        Bucket(String k, Object b, Bucket n) {key=k; binding=b; next=n;}
}

class HashT {
   final int SIZE = 256;
   Bucket table[] = new Bucket[SIZE];

   int hash(String s) {
        int h=0;
        for(int i=0; i<s.length(); i++)
           h=h*65599+s.charAt(i);
        return h;
   }

   void insert(String s, Binding b) {
        int index=hash(s)%SIZE
        table[index]=new Bucket(s,b,table[index]);
   }

   Object lookup(String s) {
        int index=hash(s)%SIZE
        for (Binding b = table[index]; b!=null; b=b.next)
          if (s.equals(b.key)) return b.binding;
        return null;
   }

   void pop(String s) {
        int index=hash(s)%SIZE
        table[index]=table[index].next;
   }
}
```

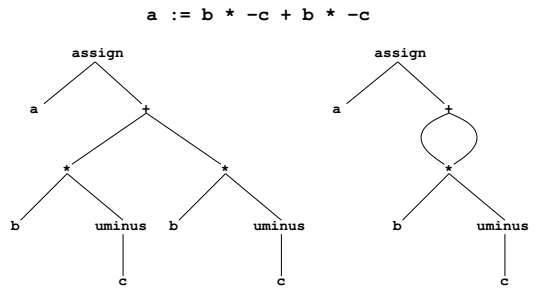
PROGRAM 5.2. Hash table with external chaining.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

Overview

- Why generate intermediate code?
  - re-targeting code is facilitated
  - easier to optimize code (can be done independently of target machine instruction set)
- Will look at two kinds:
  - Graphical representation (syntax trees)
  - Three-address code (TAC) (quads, triples)

Syntax Trees

- Natural way to represent the hierarchical structure of a program
- Can have more than one representation:



Syntax tree DAG

Building Syntax Trees

- Can be generated by appropriate semantic actions:

Production	Semantic Rules
$S \rightarrow$ **id** := E	S.nptr := mknode('assign', mkleaf(id,id.place),E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr := mknode('+', E_1.nptr, E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr := mknode('*', E_1.nptr, E_2.nptr)
$E \rightarrow -E_1$	E.nptr := mkunode('uminus', E_1.nptr)
$E \rightarrow (E_1)$	E.nptr := E_1.nptr
$E \rightarrow$ **id**	E.nptr := mkleaf(**id**, **id**.place)

- Can create DAGS if node creation routines try to return pointers to existing nodes rather than always creating new ones

Building Syntax Trees with a Compiler-Compiler

- CUP specification:

```
expr  ::=
   expr:e1 PLUS expr:e2
    {: RESULT = new Abstract.BinaryArithmeticExpr(e1,sym.PLUS,e2); :}
 | expr:e1 MINUS expr:e2
    {: RESULT = new Abstract.BinaryArithmeticExpr(e1,sym.MINUS,e2);:}
 | expr:e1 TIMES expr:e2
    {: RESULT = new Abstract.BinaryArithmeticExpr(e1,sym.TIMES,e2);:}
 | expr:e1 DIVIDE expr:e2
    {: RESULT = new Abstract.BinaryArithmeticExpr(e1,sym.DIVIDE,e2);:}
...
```


Three-Address Code (TAC)

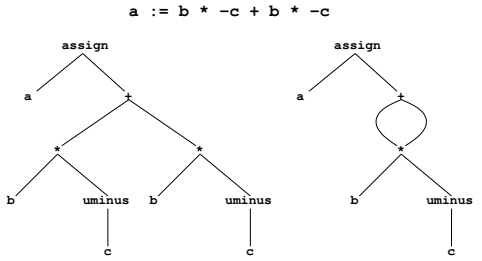
- General form: x := y op z
- Can build complicated expressions using multiple TAC instructions
- For example, x + y * z could be translated to:

  t_1 := y * z
  t_2 := x + t_1

- Using names for intermediate values allows TAC to be easily rearranged

TAC Code & Syntax Trees

- TAC is a linear-representation of a syntax tree, e.g.



Syntax tree DAG

t_1 := −c
t_2 := b * t_1
t_3 := −c
t_4 := b * t_1
t_5 := t2 + t_4
a := t_5

t_1 := −c
t_2 := b * t_1
t_5 := t_2 + t_2
a := t_5

Examples of Three-Address Code

Binary Operators:	`a := b + c`
	`a := b - c`
	`a := b * c`
	`a := b / c`

Unary operators:	`a := -b`
Assignment:	`a := b`
Unconditional jump:	`goto a`
Conditional jump:	`ifz b goto a` `ifnz b goto a`

Function call & return:	`arg a`
	`a := call b`
	`return a`

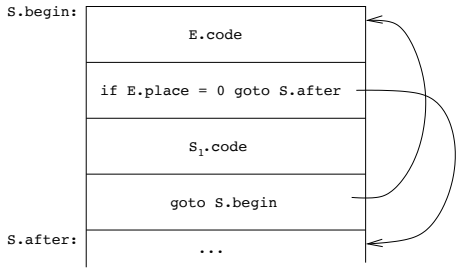
Syntax Directed Translation into Three-Address Code

Production	Semantic Rules
$S \rightarrow$ **id** := E	S.code := E.code ‖ gen(**id**.place ':=' E.place)
$E \rightarrow E_1 + E_2$	E.place := newtemp; E.code := E_1.code ‖ E_2.code ‖ gen(E.place ':=' E_1.place '+' E_2.place)
$E \rightarrow E_1 * E_2$	E.place := newtemp; E.code := E_1.code ‖ E_2.code ‖ gen(E.place ':=' E_1.place '*' E_2.place)
$E \rightarrow -E_1$	E.place := newtemp; E.code := E_1.code ‖ gen(E.place ':=' 'uminus' E_1.place)
$E \rightarrow (E_1)$	E.place := E_1.place; E.code := E_1.code
$E \rightarrow$ **id**	E.place := **id**.place; E.code := ''

Syntax Directed Translation into Three-Address Code *continued*

Production	Semantic Rules
$S \rightarrow$ **while** E **do** S_1	S.begin := newlabel; S.after := newlabel; S.code := gen(S.begin ':') ‖ E.code ‖ gen('if' E.place '=' 0 'goto' S.after) ‖ S_1.code ‖ gen('goto' S.begin) ‖ gen(S.after ':')

TAC for While Loop



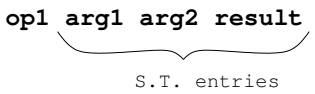

```
S.begin := newlabel;
S.after := newlabel;
S.code := gen(S.begin ':') ||
    E.code ||
    gen('if' E.place '=' 0 'goto' S.after) ||
    S1.code ||
    gen('goto' S.begin) ||
    gen(S.after ':')
```

Implementations of Three-Address Code

- Three-address code is an abstract form of intermediate code which can be implemented in many different ways
- We will look at **quads**, **triples**

Quads

- **quads:** record structure with four fields



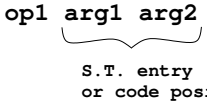
- temporary variables are created as needed and placed in the S.T.

a := b * −c + b * −c

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Triples

- **triples:** record structure with three fields



- to avoid entering temps into S.T., can refer to temp value by position

a := b * −c + b * −c

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	(4)	a

Local Declarations

P → D	{ offset := 0 }
D → D ; D	
D → **id** : T	{ enter(**id**.name, T.type, offset); offset := offset + T.width; }
T → **integer**	{ T.type := integer; T.width := 4; }
T → **real**	{ T.type := real; T.width := 8; }
T → array [**num**] of T_1	{ T.type := array(**num**.val, T_1.type); T.width := **num**.val × T_1.width }
T → ↑T_1	{ T.type := pointer(T_1.type); T.width := 4 }

Declarations in Nested Procedures

P → M D { addwidth(top(tblptr), top(offset)); }
pop(tblptr); pop(offset); }

M → ϵ { t := mktable(nil);
push(t,tblptr); push(0,offset); }

D → D_1 ; D_2

D → **proc id ;** N D_1 ; S { t := top(tblptr);
addwidth(t,top(offset));
pop(tblptr); pop(offset);
enterproc(top(tblptr), **id**.name, t) }

D → **id** : T { enter(top(tblptr),**id**.name, T.type, top(offset));
top(offset) := top(offset) + T.width }

N → ϵ { t := mktable(top(tblptr));
push(t,tblptr); push(0,offset); }

mktable(previous)	creates a new symbol table
enter(table,name,type,offset)	creates a new entry for *name* in S.T. pointed to by *table*
addwidth(table,width)	records cumulative width of all entries in *table*
enterproc(table,name,newtable)	creates new entry for procedure *name* in S.T. pointed to by *table*

Records

T → **record** L D **end** { T.type := record(top(tblptr));
T.width := top(offset);
pop(tblptr); pop(offset) }

L → ϵ { t := mktable(nil);
push(t,tblptr); push(0,offset) }

Assignments & Arithmetic Expressions

S → **id** := E { p := lookup(**id**.name);
if p $\neq$ nil **then** emit(p ':=' E.place)
else error }

E → E_1 + E_2 { E.place := newtemp;
emit(E.place ':=' E_1.place '+' E_2.place) }

E → E_1 * E_2 { E.place := newtemp;
emit(E.place ':=' E_1.place '*' E_2.place) }

E → −E_1 { E.place := newtemp;
emit(E.place ':=' 'uminus' E_1.place) }

E → (E_1) { E.place := E_1.place }

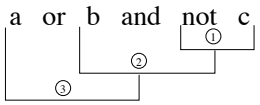
E → **id** { p := lookup(**id**.name);
if p $\neq$ nil **then** E.place := p
else error }

Boolean Expressions

- Grammar:

  E → E **or** E | E **and** E | **not** E | (E) | **id relop id** | **true** | **false**
- Two methods of translating:
  - encode true and false numerically, then evaluate boolean expressions as arithmetic expressions
  - flow of control: represent value of boolean expression by a position reached in a program

Numerical Representation



- Three-address code sequence for and/or/not:

  t_1 := not c
  t_2 := b and t_1
  t_3 := a or t_2

- Three-address code sequence for relational operators:

  a **relop** b $\equiv$ if a **relop** b then 1 else 0

```
100:  if a relop b goto 103
101:  t := 0
102:  goto 104
103:  t := 1
104:
```

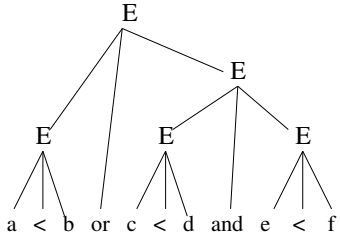
Numerical Representation Translation Scheme

$E \rightarrow E_1$ **or** E_2	{ E.place := newtemp; emit(E.place ':=' E_1.place 'or' E_2.place) }
$E \rightarrow E_1$ **and** E_2	{ E.place := newtemp; emit(E.place ':=' E_1.place 'and' E_2.place) }
$E \rightarrow$ **not** E_1	{ E.place := newtemp; emit(E.place ':=' 'not' E_1.place) }
$E \rightarrow$ (E_1)	{ E.place := E_1.place }

$E \rightarrow$ **id**$_1$ **relop** **id**$_2$	{ E.place := newtemp;
	emit('if' **id**$_1$.place **relop**.op **id**$_2$.place 'goto' nextstat + 3);
	emit(E.place ':=' '0');
	emit(**'goto'** nextstat + 2);
	emit(E.place ':=' '1'); }

$E \rightarrow$ **true**	{ E.place := newtemp; emit(E.place ':=' '1') }
$E \rightarrow$ **false**	{ E.place := newtemp; emit(E.place ':=' '0') }

Example Expression




```
100: if a < b goto 103      107: t2 := 1
101: t1 := 0                108: if e < f goto 111
102: goto 104               109: t3 := 0
103: t1 := 1                110: goto 112
104: if c < d goto 107      111: t3 := 1
105: t2 := 0                112: t4 := t2 and t3
106: goto 108               113: t5 := t1 or t4
```

Flow of Control Method

- Translate expression into series of TAC instructions involving a sequence of unconditional and conditional jumps to one of two labels: E.true and E.false
- Example #1:

  a < b $\equiv$ if a < b goto E.true
  goto E.false

- Example #2: $E \rightarrow E_1$ or E_2
  - if E_1 is true: E_1.true $\equiv$ E.true
  - if E_1 is false: E_2 must be evaluated, therefore E_1.false is the label for the first statement in the code for E_2
- Example #3: $E \rightarrow$ not E_1

  No code is generated: E_1.true $\equiv$ E.false
  E_1.false $\equiv$ E.true

Flow of Control Translation Scheme

$E \rightarrow E_1$ **or** E_2	{ E_1.true := E.true E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code ‖ gen(E_1.false ':') ‖ E_2.code }
$E \rightarrow E_1$ **and** E_2	{ E_1.true := newlabel; E_1.false := E.false E_2.true := E.true; E_2.false := E.false; E.code := E_1.code ‖ gen(E_1.true ':') ‖ E_2.code }
$E \rightarrow$ **not** E_1	{ E_1.true := E.false; E_1.false := E.true; E.code := E_1.code }
$E \rightarrow (E_1)$	{ E_1.true := E.true; E_1.false := E.false; E.code := E_1.code }
$E \rightarrow$ **id**$_1$ **relop** **id**$_2$	{ E.code := gen('if' **id**$_1$.place **relop**.op **id**$_2$.place 'goto' E.true) ‖ gen('goto' E.false) }
$E \rightarrow$ **true**	{ E.code := gen('goto' E.true) }
$E \rightarrow$ **false**	{ E.code := gen('goto' E.false) }

Branching Statements

if-then:

```
     ...
     <code for test>
     if false, goto END:
     <statements>
END: ...
```

if-then-else:

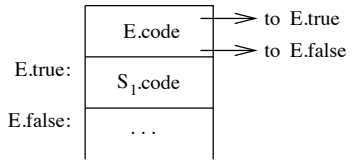
```
          ...
          <code for test>
          if false, goto FALSE_PART:
          <statements>
          goto END:
FALSE_PART:
          <statements>
END:      ...
```

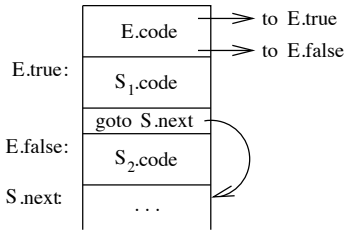
Loop Statements

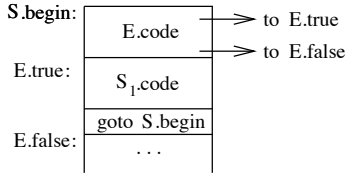
while-do:

```
BEGIN_LOOP:
          <code for test>
          if false, goto END_LOOP:
          <statements>
          goto BEGIN_LOOP:
END_LOOP:  ...
```

Flow of Control Formats







Branching and Loop Statements Translation Scheme

$S \rightarrow$ **if** E **then** S_1 { E.true := newlabel;
E.false := S.next;
S_1.next := S.next;
S.code := E.code || gen(E.true ':') || S_1.code }

$S \rightarrow$ **if** E **then** S_1 **else** S_2 { E.true := newlabel;
E.false := newlabel;
S_1.next := S.next;
S_2.next := S.next;
S.code := E.code ||
gen(E.true ':') || S_1.code ||
gen('goto' S.next) ||
gen(E.false ':') || S_2.code }

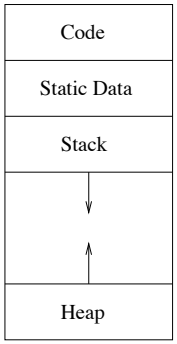
$S \rightarrow$ **while** E **do** S_1 { S.begin := newlabel;
E.true := newlabel;
E.false := S.next;
S_1.next := S.begin
S.code := gen(S.begin ':') || E.code ||
gen(E.true ':') || S_1.code ||
gen('goto' S.begin) }

Storage Allocation Strategies

- Three primary strategies:
  - Static allocation
  - Stack allocation
  - Heap allocation
- Adopted strategies often combination of all three
- Will concentrate on stack allocation

Run-Time Storage Organization

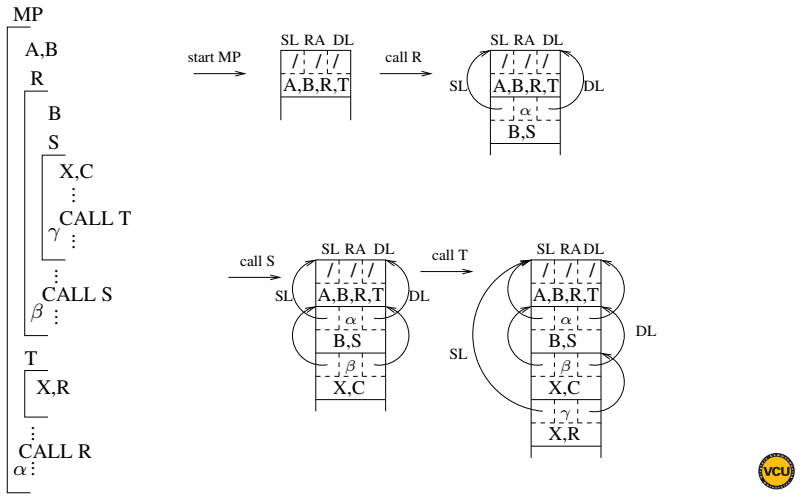
- Memory subdivided to hold:
  1. generated target code
  2. data objects
  3. stack of subroutine activations
- Typical structure:



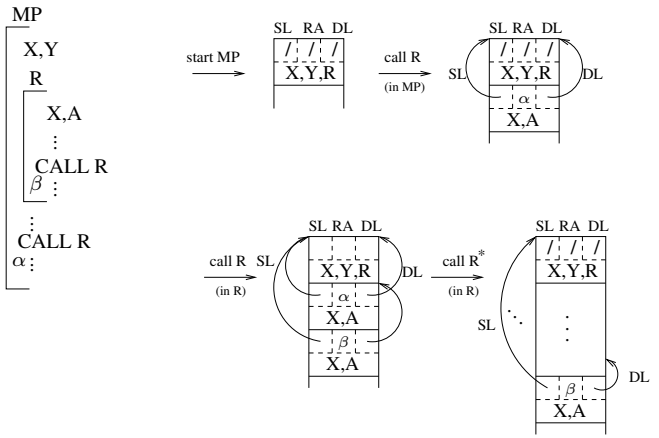
Activation Records

- Each subprogram call generates an *activation record* (AR) or *frame* on the run-time stack
- Activation records typically contain:
  - Temporary values
  - Local data
  - Saved machine status (registers)
  - Stack control links (dynamic links)
  - Variable access links (static links)
  - Subprogram parameters
  - Subprogram return values
  - Subprogram return address

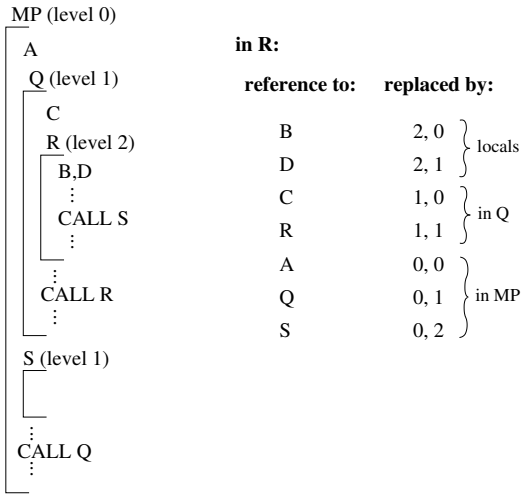
The Run-Time Stack



Recursion



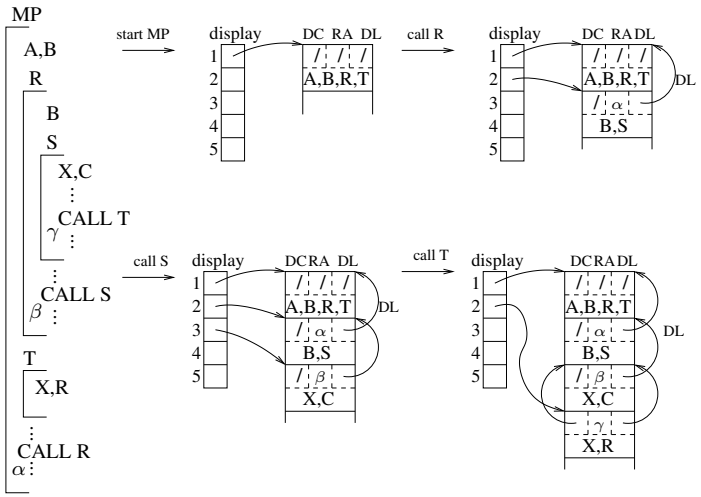
Addressing for Non-local Scopes



in R:

reference to:	replaced by:	
B	2, 0	locals
D	2, 1	
C	1, 0	in Q
R	1, 1	
A	0, 0	in MP
Q	0, 1	
S	0, 2	

Using a Display



Parameter Passing

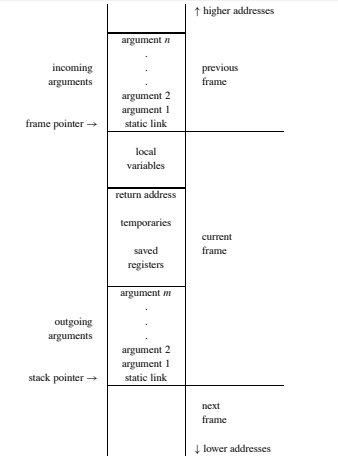


FIGURE 6.2. A stack frame.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Optimizations

- Use registers to minimize memory traffic:
  - for parameters: studies show few routines have more than 6 parameters
  - for locals & temporaries
  - return address
  - displays
- Dynamic link can be eliminated if one has a set frame size

Overview

- Intermediate representations usually use an unbounded number of temporaries
- Target machine, however, has a bounded number of registers
- Two temporaries $t1$ and $t2$ can be stored in the same register if they are never “in use” at the same time
- Therefore to properly map an unbounded number of temporaries to a bounded set of registers, a compiler must be able to determine which variables are in use at any given time; excess temporaries can be stored in memory
- A variable is said to be **live** if it holds a value that may be needed in the future
- Liveness analysis begins with a **control-flow graph**

Liveness of Variables

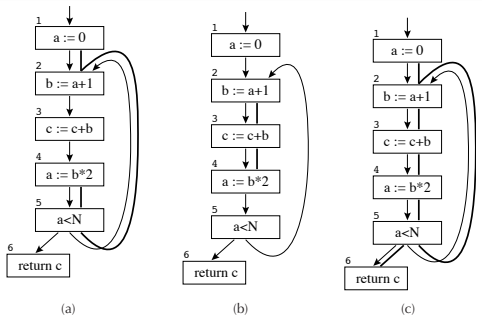


FIGURE 10.2. Liveness of variables a, b, c.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

- Determining the live range of a variable is an example of a **dataflow** problem

Dataflow Terminology & Definitions

- A flow graph has:
  - out edges that lead to **successor** nodes; the set **succ[n]** is the set of successors of node *n*
  - in edges that come from **predecessor** nodes; the set **pred[n]** is the set of predecessors of node *n*
- An assignment to a variable or temporary **defines** that variable; an occurrence of a variable in an expression **uses** the variable
- The sets **def[n]** and **use[n]** are the set of variables defined and used in node *n*
- A variable is **live** on an edge if there is a directed graph from that edge to a **use** of the variable that does not go through a **def** of that variable
- A variable is **live-in** at a node if it is live on any of the in-edges of the node; it is **live-out** at a node if it is live on any out-edges of the node

Dataflow Equations

1. If a variable is in **use[n]**, then it is **live-in** at node *n*
2. If a variable is **live-in** at a node *n*, then it is **live-out** at all nodes in **pred[n]**
3. If a variable is **live-out** at node *n* and not in **def[n]**, then the variable is also **live-in** at *n*

$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

EQUATIONS 10.3. Dataflow equations for liveness analysis.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

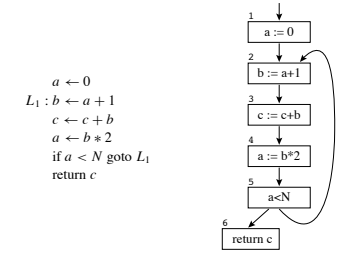
(**in[n]** = live-in at *n*, **out[n]** = live-out at *n*)

Computation of Liveness

for each n
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$
repeat
  for each n
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

ALGORITHM 10.4. Computation of liveness by iteration.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

Computation of Liveness Example



GRAPH 10.1. Control-flow graph of a program.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

for each n
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$
repeat
  for each n
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

ALGORITHM 10.4. Computation of liveness by iteration.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

Computation of Liveness Example *continued*

Calculations using example graph:

			1st		2nd		3rd		4th		5th		6th		7th	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

TABLE 10.5. Liveness calculation following forward control-flow edges. From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

TABLE 10.6. Liveness calculation following reverse control-flow edges. From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

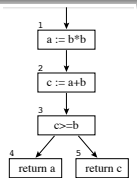
Time Complexity of Liveness Analysis

- Worst case: $O(N^4)$!
- However, ordering the nodes using depth-first search usually brings the number of repeat-loop iterations to two or three, plus the live-sets are usually sparse, so the algorithm runs between $O(N)$ and $O(N^2)$
- Other, better methods exist

Conservative Approximation

- If a value *a* is needed when execution reaches node *n*, then we can be assured that *a* is live-out in any solution of the equations
- However, the converse is *not* true: any variable *d* that is live-out at node *n* does not mean that it will really be used
- A conservative approximation of liveness is one that may erroneously believe a variable is alive, but it will never erroneously believe it is dead (why is this important?)
- A dataflow equation used for compiler optimization should be set up so that any solution to it provides conservative information to the optimizer; imprecise information may lead to sub-optimal but never incorrect programs

Static vs. Dynamic Liveness



GRAPH 10.8. Standard static dataflow analysis will not take advantage of the fact that node 4 can never be reached. From *Modern Compiler Implementation in Java*, Cambridge University Press, ©1998 Andrew W. Appel

Dynamic liveness A variable *a* is dynamically live at node *n* if some execution of the program goes from *n* to a use of *a* without going through any definition of *a*

Static liveness A variable *a* is statically live at node *n* if there is some path of control-flow edges from *n* to some use of *a* that does not go through a definition of *a*

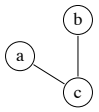
- Clearly if *a* is dynamically live it is also statically live
- An optimizing compiler must perform optimizations on the basis of static liveness

Interference Graphs

- Liveness analysis can be used for register allocation and code optimizations
- If we have a set of temporaries $a, b, c, \ldots$ that must be allocated to registers $r_1, \ldots, r_k$, a condition that prevents a and b from being allocated to the same register is called **interference**
- Interference information can be represented as a matrix or graph:

	a	b	c
a			x
b			x
c	x	x	

(a) Matrix



(b) Graph

FIGURE 10.9. Representations of interference.
From *Modern Compiler Implementation in Java*,
Cambridge University Press, ©1998 Andrew W. Appel

Register Allocation using Graph Coloring

- register allocation assigns intermediate-code temporary variables to actual machine registers
- Method:
  1. construct an *interference graph*
  2. *color* the graph *spilling* registers where needed
- The graph coloring problem derives from the old mapmakers' rule that no adjacent countries on a map should have the same color; in our case we "color" the interference graph with colors corresponding to registers
- The goal: if our target machine has K registers we wish to color the graph with K (or less) colors
- If a K-coloring is not possible, some variables will have to be *spilled* to memory

Spilling Example

- Given the following code for expression **(b * c) + a - (c * a)**:

$t_1 \leftarrow b$
$t_2 \leftarrow c$
$t_3 \leftarrow t_1 * t_2$
$t_4 \leftarrow a$
$t_5 \leftarrow t_3 + t_4$
$t_6 \leftarrow t_2 * t_4$
$t_7 \leftarrow t_5 - t_6$

- If $K = 2$ (with registers r1, r2), where operands must be registers:

```
r2 = b
r1 = c
r2 = r2 * r1
spill1 = r1
r1 = a
r2 = r2 + r1
spill2 = r2
r2 = spill1
r1 = r2 * r1
r2 = spill2
r1 = r2 - r1
```

Coloring by Simplification

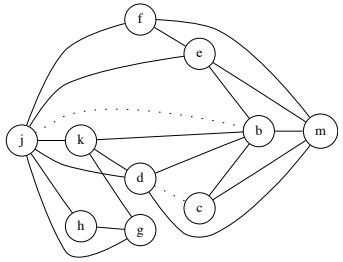
- General register allocation and graph coloring are *NP*-complete problems
- Linear-time approximation algorithm:
  1. **Build** – construct the interference graph
  2. **Simplify** – color the graph using a simple heuristic:
     - ★ suppose graph G contains a node m with fewer than K neighbors (where K = # of registers)
     - ★ let G' be the graph $G - \{m\}$ obtained by removing node m; we therefore know that if G' can be colored, so then can G
     - ★ repeatedly remove (and push onto a stack) nodes of degree less than K until all nodes have been removed
  3. **Spill** – if at some point the heuristic fails (i.e. a node n is encountered with degree $\geq K$):
     - ★ mark node n for spilling, remove it from the graph and push it onto the stack
     - ★ continue with the simplification
  4. **Select** – assign colors (registers) to nodes of the graph:
     - ★ rebuild the graph from nodes from the stack, assigning a non-interfering register to each node when popped
     - ★ add code for spills when spill nodes are popped from stack

Graph Coloring Example

```
live-in: k j

g := mem[j+12]

h := k - 1

f := g * h

e := mem[j+8]

m := mem[j+16]

b := mem[f]

c := e + 8

d := c

k := m + 4

j := b

live-out: d k j
```

Graph Coloring Example *continued*

live-in: k j
```
g := mem[j+12]
h := k - 1
f := g * h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
```
live-out: d k j



GRAPH 11.1. Interference graph for a program. Dotted lines are not interference edges but indicate move instructions.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel

Coalescing

- Redundant move instructions can be removed from an interference graph; if there is no edge in the graph between the source and destination of a move, then the move can be eliminated
- The source and destination nodes are *coalesced* (combined) into a new node whose edges are the union of the edges being replaced
- Algorithm can be found in text

stack	coloring
e	1
m	2
f	3
j&b	4
c&d	1
k	2
h	2
g	1

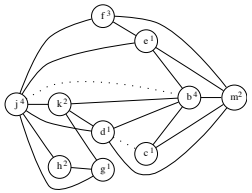


FIGURE 11.6. A coloring, with coalescing, for Graph 11.1.
From *Modern Compiler Implementation in Java,*
Cambridge University Press, ©1998 Andrew W. Appel