Lecture Compiler Construction

Franz Wotawa
wotawa@ist.tugraz.at

Institute for Software Technology
Technische Universität Graz
Inffeldgasse 16b/2, A-8010 Graz, Austria

Summer term 2017
Compilers are everywhere

- Programming languages like Java, C#, C, C++, Pascal, Modula, SML, Lisp, VHDL, Basic,..
- Graphical languages also need compilers (HTML, \LaTeX,..)
- Means for communication between computers like XML
- Natural languages

*Compilers translate a sentence written in one language to another language.*
Example – HTML

```html
<!DOCTYPE HTML PUBLIC "...- > <html> <head> <meta content=text/html; ..." http-equiv=Content-Type- > <title>Hello World</title> </head> <body> <h1>Hello World!</h1> </body> </html>
```

Hello World!
Another example

```java
outer:
for (int i = 2; i < 1000; i++)
{
    for (int j = 2; j < i; j++)
    {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

(Source: en.wikipedia.org/wiki/Java-bytecode)
Compiler vs. Interpreter

**Compiler:** Translates from one into another language where programs are directly executed (C, C++, Pascal, ...).

**Interpreter:** Executes a program directly without converting it (BASIC, batch languages, ...).

*Exact boundary difficult to define nowadays (Byte code interpreter vs. CPU, which executes machine code statements.*

*Front end (analysis phase) of compilers is always needed!*
Organizational issues

- Lecture (Vorlesung) 2h
- Practical exercises (Übung) 1h
  - Examples from the Compiler Construction theory
  - Hands on part: Development of a compiler
  - More information soon (via email and/or the webpage)
- Office hours:
  - Franz Wotawa: Tuesday, 13:00–14:00
  - Roxane Koitz: Monday, 13:00–14:00
Lecture dates

- Monday, 6.3.2017, 16:00 - 18:00 (preliminary discussion)
- Tuesday, 7.3.2017, 16:00 - 17:30 (lexical analysis)
- Monday, 20.3.2017, 16:00 - 19:00 (parsing)
- Tuesday, 21.3.2017, 16:00 - 17:30 (attributed grammars)
- Monday, 27.3.2017, 16:00 - 19:00 (type checking)
- Tuesday, 28.3.2017, 16:00 - 17:00 (runtime environment)
- Monday, 3.4.2017, 16:00 - 19:00 (code generation)
- Tuesday, 4.4.2017, 16:00-17:30 (code optimization, Q&A)
Exam

- Written form only; no papers, books etc. allowed!
- Content: DFA, NFA, LL(1), SLR(1), Attributed Grammars, Type Checking, Code Generation, Code Optimization, etc.)
- 0. date: Monday, 15.5.2017, 18:00-19:00, i13, (1 groups, 1 hour)
- 1. date: Monday, 19.6.2017, 18:00-20:00, i13, (2 groups, each 1 hour)
- 2. date: in October 2017, will be announced soon


Copy of slides but no lecture notes

Compiler construction webpage:
http://www.ist.tugraz.at/cb16.html
What is a compiler?

- Program
- Source language $\Rightarrow$ Target language
- Error reports if source code contains errors

Source program $\rightarrow$ Compiler $\rightarrow$ Target program

$\downarrow$

Error messages
Implications of definition

There are (very) many compilers

Thousands of source languages (e.g. Pascal, Modula, C, C++, Java, \ldots)

Thousands of target languages (other high-level programming languages, machine code)

Classification: single-pass, multi-pass, debugging, or optimizing

But: basics of creating a compiler are mostly consistent
Concept of compilation

- **Analysis phase**
  - Split source program into tokens
  - Generate intermediate code (intermediate representation of source program)
  
- **Synthesis phase**
  - Generate target program based on intermediate code

- Enables reuse of program parts for different source and target languages
Concept of compilation – Analysis

- Operations used within program are stored in hierarchical structure
- Syntax Tree
- Other tools which perform an analysis:
  - Structure editors
  - Pretty printers
  - Static checkers
  - Interpreters
  - Web browser
Language-processing systems

skeletal source program
  ↓
  preprocessor
  ↓
  source program
  ↓
  compiler
  ↓
  target assembly language
  ↓
  assembler
  ↓
  relocatable machine code

↓
relocatable machine code
↓
loader/link-editor
↓
absolute machine code

←−
library, relocatable object files
Components of a compiler

- lexical analyser
- semantic analyser
- syntax analyser
- intermediate code generator
- code optimizer
- code generator
- error handler
- symbol-table manager
- source code

Target program
Lexical analysis, scanning

Example:

\[ \text{pos} := \text{init} + \text{rate} \times 60 \]

Tokens:

1. Identifier \( \text{pos} \)
2. Assignment symbol \( := \)
3. Identifier \( \text{init} \)
4. Plus operator
5. Identifier \( \text{rate} \)
6. Multiplication operator
7. Constant (number) \( 60 \)
Syntax analysis, parsing

- Grammatical analysis
- Token $\leftrightarrow$ rules of grammar
- Rules of grammar (example)
  1. Each *identifier* is an *expression*
  2. Each *number* is an *expression*
  3. Assuming that $ex_1$ and $ex_2$ are *expressions*, then $ex_1 + ex_2$ and $ex_1 * ex_2$ are *expressions* as well
  4. A term: *identifier* $:= Expression$ is a *statement*
- Parse tree
Example - Parse tree

```
assignment
statement
:=

pos

expression

identifier

init

expression

rate

*

60

expression

number
```

F. Wotawa (IST @ TU Graz)
Semantic analysis

- Check for semantic errors
- Type checking
- Identification of operators and operands
- Necessary for code generation
- Example: conversion from int to real

```
statement
pos := init + rate * 60
becomes
pos := init + rate * int2real(60)
```
Intermediate code generation

E.g.: **three-address code**

Example:

1. \( \text{tmp1} := \text{int2real}(60) \)
2. \( \text{tmp2} := \text{id3} \times \text{tmp1} \)
3. \( \text{tmp3} := \text{id2} + \text{tmp2} \)
4. \( \text{id1} := \text{tmp3} \)

using the *symbol table*

<table>
<thead>
<tr>
<th>pos (id1)</th>
<th>real</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>init (id2)</td>
<td>real</td>
<td>...</td>
</tr>
<tr>
<td>rate (id3)</td>
<td>real</td>
<td>...</td>
</tr>
</tbody>
</table>
Code optimizer

- Goal: faster machine code
- Example:

1. \( \text{tmp1} := \text{int2real}(60) \)
2. \( \text{tmp2} := \text{id3} \times \text{tmp1} \)
3. \( \text{tmp3} := \text{id2} + \text{tmp2} \)
4. \( \text{id1} := \text{tmp3} \)

\[ \downarrow \]

1. \( \text{tmp1} := \text{id3} \times 60.0 \)
2. \( \text{id1} := \text{id2} + \text{tmp1} \)
Code generator

- Generation of actual target code (Machine code, Assembler)
- Example:
  1. MOVF id3, R2
  2. MULF #60.0, R2
  3. MOVF id2, R1
  4. ADDF R2, R1
  5. MOVF R1, id1
PART 2 - LEXICAL ANALYSIS
Goals of this section

- Specification and implementation of lexical analysers
- Easiest method:
  1. Create diagram which describes structure of tokens
  2. Manually translate diagram into a program
- Regular expressions in finite state machines
Tasks

- Filter comments, white spaces, ..
- Establish relations between error messages of compiler and source code (e.g. via line number)
- Preprocessor functions (e.g. in C)
- Create copy of source program containing error messages
Why separate lexical analysis and parsing?

1. **Simpler design**: e.g. removal of comments and white spaces enables use of simpler parser design
2. **Improve efficiency**: large portion of time is invested into reading of programs and conversion into tokens
3. **Enhance portability of compilers**
4. *(There are tools for both phases)*
Tokens, patterns, lexemes

**Pattern:** Rule describing a set of strings (a token)

**Token:** Set of strings described by a pattern

**Lexem:** Sequence of characters which are matched by a pattern of a token

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexem</th>
<th>Patterns (verbal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>if</td>
<td>if</td>
</tr>
<tr>
<td>relation</td>
<td>&gt;, &lt;, ...</td>
<td>&lt; or &gt; or ...</td>
</tr>
<tr>
<td>id</td>
<td>pi, test_cases</td>
<td>letter followed by letters, digits, or ‘ ’</td>
</tr>
<tr>
<td>num</td>
<td>2.7, 0, 12e-4</td>
<td>any numeric constant</td>
</tr>
<tr>
<td>literal</td>
<td>“segmentation fault”</td>
<td>any characters between “ and “</td>
</tr>
</tbody>
</table>
**Tokens**

- **Terminals** in the grammar (checked by the parser)
- In programming languages:
  - keywords, operators, identifiers, constants, strings, brackets
- Language conventions:
  1. Tokens have to occur in specific places within the code
  2. Tokens are separated by white spaces
     - *Example (Fortran):* `DO 5 I = 1.25` is interpreted as `DO5I = 1.25`
  3. Keywords are reserved and must not be used as identifiers
     1. is (currently) not used
     2. is accepted
     3. is reasonable (and simplifies compiler construction)
        - `IF THEN THEN THEN = ELSE; ELSE ELSE ELSE = THEN;`
Token attributes

- Store lexeme for further processing
- Store pointers to entries in symbol table
- Line number or position of token within source code

\[ U = 2 \times R \times \pi \]

\[ \langle \text{id}, \text{pointer to } U \rangle \]
\[ \langle \text{assign\_op} \rangle \]
\[ \langle \text{num}, \text{integer value 2} \rangle \]
\[ \langle \text{mult\_op} \rangle \]
\[ \langle \text{id}, \text{pointer to } R \rangle \]
\[ \langle \text{mult\_op} \rangle \]
\[ \langle \text{id}, \text{pointer to } \pi \rangle \]
Lexical errors

- \( \text{fi} ( a == 1) \ldots \)
  - \( \text{fi} \) is interpreted as undefined identifier

- String sequences which cannot be matched to a token
  - e.g.: \( 2.3e*4 \) instead of \( 2.3e+4 \).

- Error-correcting:
  1. **Panic Mode**: Removal of faulty characters until a token can be matched
  2. Remove a faulty character
  3. Include a new character
  4. Replace a character
  5. Change order of neighboring characters

- Some errors (like the first example) may be only recognizable by the parser
Specification of tokens

- **Strings over an alphabet:**
  Alphabet ... finite set of symbols *example*: \{0, 1\} Binary alphabet
  String (word, sentence) ... finite sequence of symbols of the alphabet
  \(\epsilon\) ... empty string

- **Language:** Set of all strings over an alphabet

- **Operators:**
  - **Concatenation** \(xy\) is a string whereby \(y\) is attached to the end of \(x\)
  - \(x^i\) is defined as: \(x^0 = \epsilon, \ i > 0 \rightarrow x^i = x^{i-1}x\)
  - **Prefix:** `computer` \(\rightarrow\) `comp`
  - **Suffix:** `computer` \(\rightarrow\) `uter`
  - **Substring:** `computer` \(\rightarrow\) `put`
  - **Subsequence:** `computer` \(\rightarrow\) `opt`
Operations on languages

- **Union** \( L \cup M = \{x | x \in L \lor x \in M\} \)
- **Concatenation** \( LM = \{xy | x \in L \land y \in M\} \)
- **Kleen Closure** \( L^* = \bigcup_{i=0}^{\infty} L^i \)
- **Positive Closure** \( L^+ = \bigcup_{i=1}^{\infty} L^i \)

**Example:**
- \( L = \{A, \ldots, Z\} \), \( M = \{0, 1, \ldots, 9\} \)
- \( L \cup M \) Set of letters and numbers
- \( LM \) Strings containing a letter followed by a number
- \( L^4 \) Strings which are made up of four letters
- \( L^* \) Strings made up of letters including \( \epsilon \)
- \( M^+ \) Number strings containing at least one number
Regular expressions

- **Example**: Pascal identifier `letter ( letter | digit )*`

Regular expression $r$ defines a language over an alphabet $\Sigma$ using the following rules:

1. The regular expression $\epsilon$ describes $\{\epsilon\}$
2. $a \in \Sigma$: The regular Expression $a$ describes $\{a\}$
3. $r, s$ are regular expressions (languages $L(r), L(s)$):
   1. $(r)|(s)$ describes $L(r) \cup L(s)$
   2. $(r)(s)$ describes $L(r)L(s)$
   3. $(r)^*$ describes $(L(r))^*$
   4. $(r)^+$ describes $(L(r))^+$

- Regular sets
Examples

- Assuming $\Sigma = \{a, b\}$
  1. $a | b$ describes the language $\{a, b\}$
  2. $(a | b)(a | b)$ describes $\{aa, ab, ba, bb\}$
  3. $a^*$ describes $\{\epsilon, a, aa, aaa, aaaa, \ldots\}$
  4. $(a | b)^*$ describes all strings which contain no or multiple $a$’s and $b$’s

- Not all languages can be described by regular expressions, e.g. $\{wcw \mid w \text{ is a string made up of } a \text{’s and } b \text{’s}\}$

- Regular expressions may only describe words containing either a fixed amount of repetitions or an infinite amount of repetitions
A regular definition is a sequence of the form:

\[
d_1 \rightarrow r_1 \\
\ldots \\
d_n \rightarrow r_n
\]

whereby \(d_i\) is a distinct name and \(r_i\) is a regular expression over \(\Sigma \cup \{d_1, \ldots, d_n\}\)

Example:

\[
\begin{align*}
\text{letter} & \rightarrow A|\ldots|Z|a|\ldots|z \\
\text{digit} & \rightarrow 0|1|\ldots|9 \\
\text{id} & \rightarrow \text{letter(letter|digit)}^*
\end{align*}
\]
### Token detection

- **Goal:** Identify lexeme of token in input buffer
- **Output:** Token-attribute-pair

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Token</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>white_space</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>-</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
<td>-</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>pointer to table entry</td>
</tr>
<tr>
<td>num</td>
<td>num</td>
<td>pointer to table entry</td>
</tr>
<tr>
<td>&lt;</td>
<td>relop</td>
<td>LT</td>
</tr>
<tr>
<td>&gt;</td>
<td>relop</td>
<td>GT</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Actions performed by lexical analyser (after `get_next_token` of the parser)

States connected by directed edges

Edges are labeled:
- Labels contain symbols expected in input buffer in order to progress to next state
- Label `other` denotes all symbols not used by any other edge originating from the same node

One start state

End states after matching a token

Deterministic (not necessarily)
Regular expression `letter (letter | digit)∗` can be described by the following transition diagram:

Assume the input buffer contains:

<table>
<thead>
<tr>
<th>Pos</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and the current-symbol pointer is pointing to position 1.
Tokenmatching - Example

1. Current symbol = $P$, position = 1
   ⇒ Transition from state 1 to state 2 and read new symbol

2. Current symbol = $I$, position = 2
   ⇒ Remain in state 2 and read new symbol

3. Current symbol = SPACE, position = 3
   ⇒ Transition from state 2 to state 3

4. State 3 is an end state
Example – Implementing a Lexer directly

Pseudo code

```
lexem = "";
if (next_char ∈ letter) then
    lexem = lexem + next_char;
    get_next_char();
while (next_char ∈ letter ∪ digit) do
    lexem = lexem + next_char;
    get_next_char();
return IDENTIFIER(lexem);
```
Generalization – Finite automata

Nondeterministic finite automaton (NFA) \((S, \Sigma, move, s_0, F)\)

1. Set of states \(S\)
2. Set of input symbols \(\Sigma\)
3. Transition relation \(move : S \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^S\), relating state-symbol-pairs to states
4. Initial state \(s_0 \in S\)
5. Set of end states \(F \subseteq S\)

- NFA can be visualized in form of directed graph (transition graph)
- Distinctions from transition diagram:
  - Nondeterministic
  - \(\epsilon\)-moves allowed
NFA for $aa^* | bb^*$ (language, accepting NFA):

\[(\{0, 1, 2, 3, 4\}, \{a, b\}, \text{move}, 0, \{2, 4\})\] with \text{move}:

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>new state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\epsilon$</td>
<td>{1,3}</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>{2}</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>{2}</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>{4}</td>
</tr>
</tbody>
</table>
Deterministic finite automaton

Deterministic finite automaton (DFA) is a special case of a NFA whereby:

1. No state has an $\epsilon$-move
2. For each state $s$ there is at most one edge which is marked with an input symbol $a$

- DFAs contain exactly one transition for each input
- Each entry in transition table contains exactly one state

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
DFA simulation

*Input:* Input string \( x \), DFA \( D \)

*Output:* ’Yes’ if \( D \) accepts \( x \), ’No’ otherwise

\[
s := s_0 \\
c := \text{nextchar} \\
\textbf{while} \ c \neq \text{eof} \ \textbf{do} \\
\hspace{1em} s := \text{move}(s, c) \\
\hspace{1em} \text{if} \ s \equiv \bot \ \textbf{then} \\
\hspace{2em} \text{return } '\text{No}' \\
\hspace{1em} c := \text{nextchar} \\
\textbf{end} \\
\textbf{if} \ s \in F \ \textbf{then} \\
\hspace{1em} \text{return } '\text{Yes}' \\
\textbf{else} \\
\hspace{1em} \text{return } '\text{No}' \\
\textbf{end} \\
\]
DFA for $aa^* | bb^*$:
$$(\{0, 1, 2\}, \{a, b\}, move, 0, \{1, 2\})$$

whereby

$move$:

<table>
<thead>
<tr>
<th>state</th>
<th>symbol</th>
<th>new state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
Conversion NFA $\rightarrow$ DFA

**Subset construction algorithm**

Idea:

1. Each DFA state corresponds to a set of NFA states.
2. After reading $a_1 a_2 \ldots a_n$ the DFA is in a state which is equivalent to a subset of the NFA. This subset corresponds to the path of the NFA when reading $a_1 a_2 \ldots a_n$.

Number of DFA states is exponential in the number of NFA states (worst case)
Subset construction

\textbf{Input:} NFA $N = (S, \Sigma, move, s_0, F)$
\textbf{Output:} DFA $D$, accepting the same language as $N$

Initially, $\epsilon$-\textit{closure}(s$_0$) is the only state in $Dstates$ and it is unmarked

\textbf{while} there is an unmarked state $T$ in $Dstates$ \textbf{do}

\hspace{1cm} \text{mark } T$

\hspace{1cm} \textbf{for} each input symbol $a$ \textbf{do}

\hspace{2cm} $U := \epsilon$-\textit{closure}(move($T,a$))

\hspace{2cm} \textbf{if} $U \notin Dstates$ \textbf{then}

\hspace{3cm} \text{Add } U \text{ as unmarked state to } Dstates$

\hspace{2cm} \textbf{end}

\hspace{2cm} $Dtran[T,a] := U$

\hspace{1cm} \textbf{end}

\hspace{1cm} \textbf{end}
\( \varepsilon \)-closure, move

- \( \varepsilon \)-closure\((s) \): Set of NFA states which are accessible from a state \( s \) via \( \varepsilon \)-moves
- \( \varepsilon \)-closure\((T) \): Set of NFA states which are accessible from a state \( s \in T \) via \( \varepsilon \)-moves
- move\((T, a) \): Set of NFA states which are accessible from a state \( s \in T \) via a move relation of the input symbol \( a \)
Calculation of $\epsilon$-closure

**Input:** NFA $N$, set of states $T$

**Output:** $\epsilon$-closure

Push all states in $T$ onto stack
Initialize $\epsilon$-closure($T$) to $T$

while stack $\neq \emptyset$ do
  Pop $t$, the top element of stack
  for each state $u$ with an edge from $t$ to $u$ labeled $\epsilon$ do
    if $u \not\in \epsilon$-closure($T$) then
      Add $u$ to $\epsilon$-closure($T$)
      Push $u$ onto stack
  end
end

Example NFA $\rightarrow$ DFA
Thompson’s Construction

Input:

Regular expression $r$ over an alphabet $\Sigma$

Output: NFA $N$, which accepts $L(r)$

$\epsilon$

$\epsilon$

$a \in \Sigma$

$s | t$

$s^*$

$[s]$

whereby

$[s] = (s | \epsilon)$
NFA simulation

*Input:* NFA $N$, input string $x$
*Output:* 'Yes' if $N$ accepts $x$, 'No' otherwise

\[
S := \epsilon\text{-closure}(\{s_0\})
\]
\[
a := \text{nextChar}
\]

**while** $a \neq \text{eof} \ **do**$
\[
S := \epsilon\text{-closure}(\text{move}(S,a))
\]
\[
a := \text{nextChar}
\]
**end**

**if** $S \cap F \neq \emptyset$ **then**
\[
\text{return 'Yes'}$
\]
**else**
\[
\text{return 'No'}$
\]
**end**
Check whether an input string $x$ is contained in a language defined by the regular expression $r$:

1. Construct NFA (Thompson’s construction) and apply simulation algorithm for NFAs
2. Construct NFA (Thompson’s construction), convert NFA to DFA and apply simulation algorithm for DFAs

Time-space tradeoffs

<table>
<thead>
<tr>
<th>Automaton</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>$O(</td>
<td>r</td>
</tr>
<tr>
<td>DFA</td>
<td>$O(2^{</td>
<td>r</td>
</tr>
</tbody>
</table>
Regular expression $\rightarrow$ DFA

- Direct conversion
- Notation:
  - NFA state $s$ is important $\iff s$ has at least one non-$\epsilon$-move to a successor
  - Extended regular expression (Augmented regular expression) $(r)\#$
  - Illustration of extended expressions via syntax trees (cat-node, or-node, star-node)
  - Each leaf node is labeled either with a symbol (of the alphabet) or with $\epsilon$
  - Each leaf node ($\neq \epsilon$) has a designated number (position)
- Remark: NFA-DFA-conversion only via important states = positions
Syntax tree – Example

Syntax tree for 

$$\left( (a|b)^* abb \# \right)$$
Algorithm - Idea

- **Approach:**
  1. Create syntax tree for extended regular expression
  2. Calculate four functions: `nullable`, `firstpos`, `lastpos`, `followpos`
  3. Construct DFA using `followpos`

- **DFA states correspond to sets of positions (important NFA states)**

- **Position** $i$, $\text{followpos}(i)$ are positions $j$ for which there exists an input string $\ldots cd \ldots$ where $i$ corresponds to $c$ and $j$ corresponds to $d$
followpos – Example

- Syntax tree for \((a|b)^* abb#\)
- \(followpos(1) = \{1, 2, 3\}\)
- Explanation: Suppose we see an \(a\). This symbol could belong either to \((a|b)^*\) or to the following \(a\). If we see a \(b\) then this symbol has to belong to \((a|b)^*\). Thus positions 1, 2, 3 are contained in \(followpos(1)\).
- Informal definition of functions (node \(n\), string \(s\))
  - \(firstpos\) ... positions, which match the first symbol of \(s\)
  - \(lastpos\) ... positions, which match the last symbol of \(s\)
  - \(nullable\) ... True, if node \(n\) can create a language containing the empty string, false otherwise
### Rules for `nullable, firstpos, lastpos`

<table>
<thead>
<tr>
<th>Node $n$</th>
<th><code>nullable</code></th>
<th><code>firstpos</code> [lastpos]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf $n$ labeled</td>
<td>$\epsilon$</td>
<td>$\emptyset$ [\emptyset]</td>
</tr>
<tr>
<td>Leaf $n$ labeled</td>
<td>$\text{false}$</td>
<td>${i} \ [{i}]$</td>
</tr>
<tr>
<td>position $i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$c_1</td>
<td>c_2$</td>
<td>$\text{nullable}(c_1) \lor \text{nullable}(c_2)$</td>
</tr>
<tr>
<td>$c_1 \bullet c_2$</td>
<td>$\text{nullable}(c_1) \land \text{nullable}(c_2)$</td>
<td>$\begin{cases} \text{if } \text{nullable}(c_1) \text{ then} \ \quad \text{firstpos}(c_1) \cup \text{firstpos}(c_2) \ \text{else } \text{firstpos}(c_1) \end{cases}$ $\begin{cases} \text{if } \text{nullable}(c_2) \text{ then} \ \quad \text{lastpos}(c_1) \cup \text{lastpos}(c_2) \ \text{else } \text{lastpos}(c_2) \end{cases}$</td>
</tr>
<tr>
<td>$c_1^*$</td>
<td>$\text{true}$</td>
<td>$\text{firstpos}(c_1) \ [\text{lastpos}(c_1)]$</td>
</tr>
</tbody>
</table>
\textit{firstpos, followpos} – Example

\begin{itemize}
\item \{1,2,3\} \rightarrow 6
\item \{1,2,3\} \rightarrow \{5\} \rightarrow 6 \# \{6\}
\item \{1,2,3\} \rightarrow \{4\} \rightarrow 5 \rightarrow \{4\} \rightarrow b \{4\}
\item \{1,2\} \rightarrow \{1,2\} \rightarrow 3 \rightarrow \{3\}
\item \{1,2\} \rightarrow \{1,2\}
\item \{1\} \rightarrow a \{1\} \rightarrow 1
\item \{2\} \rightarrow b \{2\} \rightarrow 2
\end{itemize}
If $n$ is a cat-node ($\bullet$), $c_1$ is the left, $c_2$ is the right child and $i$ is a position in $\text{lastpos}(c_1)$, then all positions of $\text{firstpos}(c_2)$ are contained in $\text{followpos}(i)$

If $n$ is a star-node ($\ast$) and $i$ is contained in $\text{lastpos}(n)$, then all positions of $\text{firstpos}(c)$ are contained in $\text{followpos}(i)$

<table>
<thead>
<tr>
<th>Node</th>
<th>followpos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>3</td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td>{5}</td>
</tr>
<tr>
<td>5</td>
<td>{6}</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
</tr>
</tbody>
</table>
followpos-graph → NFA

A NFA without $\epsilon$-moves can be created based on a followpos-graph:

1. Mark all positions in $firstpos$ of the root node of the syntax tree as initial states
2. Label each directed edge $(i, j)$ with the symbol of position $i$
3. Mark position which belongs to # as end state

→ followpos-graph can be converted to DFA (using subset construction)
DFA-algorithm

Input:

Regular expression $r$

Output: DFA $D$, accepting the language $L(r)$

Initially, the only unmarked state in $Dstates$ is $\text{firstpos}(\text{root})$

while there is an unmarked state $T \in Dstates$ do

Mark $T$

for each input symbol $a$ do

Let $U$ be the set of positions that are in $\text{followpos}(p)$ for some position $p \in T$ where the symbol of $p$ is $a$.

if $U \neq \emptyset$ and $U \notin Dstates$ then

Add $U$ as unmarked state to $Dstates$

end

$DTran(T, a) = U$

end

end
Goals

- Definition of the syntax of a programming language using context-free grammars
- Methods for parsing of programs
  ⇒ determine whether a program is syntactically correct
- Advantages (of grammars):
  - Precise, easily comprehensible language definition
  - Automatic construction of parsers
  - Declaration of the structure of a programming language (important for translation and error detection)
  - Easy language extensions and modifications
Tasks

- Parser types:
  - Universal parsers (inefficient)
  - Top-down-parser
  - Bottom-up-parser

- Only subclasses of grammars (LL, LR)
- Collect token informations
- Type checking
- Immediate code generation
Syntax error handling

Error types:
- Lexical errors (spelling of a keyword)
- Syntactic errors (closing bracket is missing)
- Semantic errors (operand is incompatible to operator)
- Logic Errors (infinite loop)

Tasks:
- Exact error description
- Error recovery → consecutive errors should be detectable
- Error correction should not slow down the processing of correct programs
Problems during error handling

- Spurious Errors: Consecutive errors created by error recovery
  *Example*: Compiler issues error-recovery resulting in the removal of the declaration of $\text{pi}$
  $\rightarrow$ Error during semantic analysis: $\text{pi}$ undefined

- Error is detected late in the process $\rightarrow$ error message does not point to the correct position within the code

- Too many error messages are issued
Error-recovery

- **Panic mode:**
  Skip symbols until input can by synchronized to a token

- **Phrase-level recovery:**
  Local error corrections, e.g. replacement of ‘,’ by a ‘;;’

- **Error productions:**
  Extension of grammar to handle common errors

- **Global correction:**
  Minimal correction of program in order to find a matching derivation (cost intensive)
A grammar is a 4-tupel $G = (V_N, V_T, S, \Phi)$ whereby:

- $V_N$ Set of nonterminal symbols
- $V_T$ Set of terminal symbols
- $S \in V_N$ Start symbol
- $\Phi : (V_N \cup V_T)^* V_N (V_N \cup V_T)^* \rightarrow (V_N \cup V_T)^*$
  Set of production rules (rewriting rules)
  $(\alpha, \beta)$ is represented as $\alpha \rightarrow \beta$

**Example:**

$$\begin{align*}
\{S, A, Z\}, \{a, b, 1, 2\}, S, \{S \rightarrow AZ, A \rightarrow a, A \rightarrow b, Z \rightarrow \epsilon, Z \rightarrow 1, Z \rightarrow 2, Z \rightarrow ZZ\}\}
\end{align*}$$
**Direct derivation** \( \sigma, \psi \in (V_T \cup V_N)^* \).

\( \sigma \) can be directly derived from \( \psi \) (in one step; \( \psi \Rightarrow \sigma \)), if there are two strings \( \phi_1, \phi_2 \), so that \( \sigma = \phi_1 \beta \phi_2 \) and \( \psi = \phi_1 \alpha \phi_2 \) and \( \alpha \rightarrow \beta \in \Phi \).

**Example:**

<table>
<thead>
<tr>
<th>( \psi )</th>
<th>( \sigma )</th>
<th>Rule used</th>
<th>( \phi_1 )</th>
<th>( \phi_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( A \ Z )</td>
<td>( S \rightarrow AZ )</td>
<td>( \epsilon )</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>( aZ )</td>
<td>( a1 )</td>
<td>( Z \rightarrow 1 )</td>
<td>( a )</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>( AZ Z )</td>
<td>( A2Z )</td>
<td>( Z \rightarrow 2 )</td>
<td>( A )</td>
<td>( Z )</td>
</tr>
</tbody>
</table>
Derivation
Production:

A string $\psi$ produces $\sigma$ ($\psi \xrightarrow{\dagger} \sigma$), if there are strings $\phi_0, \ldots, \phi_n$ ($n > 0$), so that $\psi = \phi_0 \Rightarrow \phi_1, \phi_1 \Rightarrow \phi_2, \ldots, \phi_{n-1} \Rightarrow \phi_n = \sigma$.

Example: $S \Rightarrow AZ \Rightarrow AZZ \Rightarrow A2Z \Rightarrow a2Z \Rightarrow a21$

Reflexive, transitive closure: $\psi \xrightarrow{\ast} \sigma \iff \psi \xrightarrow{\dagger} \sigma$ or $\psi = \sigma$

Accepted language: A grammar $G$ accepts the following language

$$L(G) = \{ \sigma | S \xrightarrow{\ast} \sigma, \sigma \in (VT)^* \}$$
Parse trees

Example:

\[ E \rightarrow E + E \mid E \ast E \mid \text{id} \]

⇒ 2 derivations (and parse trees) for \text{id+id*id}

Grammar is ambiguous
Classification of grammars

**Chomsky** (*restriction of production rules* $\alpha \rightarrow \beta$)

- **Unrestricted Grammar**: no restrictions
- **Context-Sensitive Grammar**: $|\alpha| \leq |\beta|$
- **Context-Free Grammar**: $|\alpha| \leq |\beta|$ and $\alpha \in V_N$
- **Regular Grammar**: $|\alpha| \leq |\beta|$, $\alpha \in V_N$ and $\beta$ is in the form of: $aB$ or $a$ whereby $a \in V_T$ and $B \in V_N$
Grammar examples

- **Regular grammar:** \((a|b)^*abb\)
  
  \[
  A_0 \rightarrow aA_0 | bA_0 | aA_1 \\
  A_1 \rightarrow bA_2 \\
  A_2 \rightarrow bA_3 \\
  A_3 \rightarrow \epsilon
  \]

- **Context-sensitive grammars:**
  
  \[L_1 = \{wcw | w \in (a|b)^*\}\]
  
  But
  
  \[L'_1 = \{wcw^R | w \in (a|b)^*\}\] is context-free

  \[L_2 = \{a^nb^mc^nd^m | n \geq 1, m \geq 1\}\]

  \[L_3 = \{a^nb^nc^n | n \geq 1\}\]
Conversions

- Remove ambiguities

\[
stmnt \rightarrow \text{if } expr \ \text{then} \ stmnt | \text{if } expr \ \text{then} \ stmnt \ \text{else} \ stmnt | \text{other}
\]

2 parse trees for \( \text{if } E_1 \ \text{then if } E_2 \ \text{then } S_1 \ \text{else } S_2 \).

Prefer left tree

Associate each \textit{else} with the closest preceding \textit{then}
Removing left recursions

- A grammar is left-recursive if there is a nonterminal $A$ and a production $A \rightarrow A \alpha$
- Top-Down-Parsing can’t handle left recursions
- Example:
  convert $A \rightarrow A \alpha | \beta$ to:
  $A \rightarrow \beta A_1$
  $A_1 \rightarrow \alpha A_1 | \epsilon$
Algorithm to eliminate left recursions

*Input:*

Grammar $G$ without cycles and $\epsilon$-productions

*Output:* Grammar without left recursions

Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$

for $i := 1$ to $n$ do

for $j := 1$ to $i - 1$ do

Replace each production $A_i \rightarrow A_j \gamma$

by the productions $A_i \rightarrow \delta_1 \gamma | \ldots | \delta_k \gamma$,  
where $A_j \rightarrow \delta_1 | \ldots | \delta_k$ are all the current $A_j$-productions

end

Eliminate the immediate left recursion among the $A_i$-productions

end
Left factoring

- Important for predictive parsing
- Elimination of alternative productions

Example:
\[
\text{stmt} \rightarrow \text{if expr then stmt else stmt} \\
\text{stmt} \rightarrow \text{if expr then stmt}
\]

Solution: For each nonterminal \( A \) find the longest prefix \( \alpha \) for two or more alternative productions

If \( \alpha \neq \epsilon \) then replace all \( A \)-productions
\[
A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \ldots | \alpha \beta_n | \gamma 
\]

(\( \gamma \) does not start with \( \alpha \)) with:

\[
A \rightarrow \alpha A_1 | \gamma \\
A_1 \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n
\]

Apply transformation until no prefixes \( \alpha \neq \epsilon \) can be found
Idea: Construct parse tree for a given input, starting at root node

Recursive-descent parsing (with backtracking)

**Example:**

\[ S \rightarrow cAd \]

\[ A \rightarrow ab | a \]

Matching of \( cad \)

Predictive parsing (without backtracking, special case of recursive-descent parsing)

Left-recursive grammars can lead to infinite loops!
Predictive parsers

- Recursive-descent parser without backtracking
- Possible if production which needs to be used is obvious for each input symbol
- Transition diagrams
  1. Remove left recursions
  2. Left factoring
  3. For each nonterminal $A$:
     1. Create a initial state and an end state
     2. For each production $A \rightarrow X_1X_2 \ldots X_n$ create a path leading from the initial state to the end state while labeling the edges $X_1, \ldots, X_n$
Predictive parsers (II)

- **Processing:**
  1. Start at the initial state of the current start symbol.
  2. Suppose we are currently in the state $s$ which has an edge whose label contains a terminal $a$ and leads to the state $t$. If the next input symbol is $a$ then go to state $t$ and read a new input symbol.
  3. Suppose the edge (from $s$) is marked by a nonterminal $A$. In that case go to the initial state of $A$ (without reading a new input symbol). If we reach the end state of $A$ then go to state $t$ which is succeeding $s$.
  4. If the edge is marked by $\epsilon$ then go directly to $t$ without reading the input.

- Easily implemented by recursive procedures.
Example - Predictive parser

\[
E \rightarrow \text{id} E_1 | (E) \\
E_1 \rightarrow \text{op} E | \epsilon
\]

E()

if nextToken=id then
getNextToken
E1()

if nextToken=( then
getNextToken
E()
if nextToken=) then accept

E1()

if nextToken=op then
getNextToken
E()
else return
Non-recursive predictive parser

- **Input buffer**: String to be parsed (terminated by a $)
- **Stack**: Initialized with the start symbol and contains nonterminals which are not derivated yet (terminated by a $)
- **Parsing table**: $M(A, a)$, $A$ is a nonterminal, $a$ a terminal or $\$$
Top-down parsing with stack

Mode of operation:

1. $X$ is a terminal: If $X = a = \$, then the input was matched. If $X = a \neq \$, pop $X$ off the stack and read next input symbol. Otherwise an error occurred.

2. $X$ is a nonterminal: Fetch entry of $M(X, a)$. If this entry is an error skip to error recovery. Otherwise the entry is a production of the form $X \rightarrow UVW$. Replace $X$ on the stack with $WVU$ (afterward $U$ is the top most element on the stack).
Example

Grammar

\[
E \rightarrow \text{id } E_1 | (E) \\
E_1 \rightarrow \text{op } E | \epsilon
\]

Parsing table \( M(X, a) \)

<table>
<thead>
<tr>
<th>NONTERMINAL</th>
<th>id</th>
<th>op</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>( E \rightarrow \text{id } E_1 )</td>
<td>( E_1 \rightarrow \text{op } E )</td>
<td>( E \rightarrow (E) )</td>
<td>( E_1 \rightarrow \epsilon )</td>
<td>( E_1 \rightarrow \epsilon )</td>
</tr>
</tbody>
</table>

Derivation of \( \text{id op id} \).
### Example (II)

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id op id $</td>
<td>$E \rightarrow \text{id }E_1$</td>
</tr>
<tr>
<td>$E_1$</td>
<td>id op id $</td>
<td>$E_1 \rightarrow \text{op }E$</td>
</tr>
<tr>
<td>$E_1$</td>
<td>op id $</td>
<td>$E \rightarrow \text{id }E_1$</td>
</tr>
<tr>
<td>$E$</td>
<td>id $</td>
<td>$E_1 \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E_1$</td>
<td>id $</td>
<td>$E \rightarrow \text{id }E_1$</td>
</tr>
<tr>
<td>$E_1$</td>
<td>$</td>
<td>$E_1 \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$ $</td>
<td>$</td>
<td>$E \rightarrow \text{id }E_1$</td>
</tr>
</tbody>
</table>
FIRST and FOLLOW

- Used when calculating parse table
- $\textit{FIRST}(\alpha)$ Set of terminals, which can be derived from $\alpha$ ($\alpha$ string of grammar symbols)
- $\textit{FOLLOW}(A)$ Set of terminals which occur directly on the right side next to the nonterminal $A$ in a derivation
  If $A$ is the right most element of a derivation, then $\$$ is contained in $\textit{FOLLOW}(A)$
Calculation of FIRST

- $\text{FIRST}(X)$ for a grammar symbol $X$
  1. $X$ is a terminal: $\text{FIRST}(X) = \{X\}$
  2. $X \rightarrow \epsilon$ is a production: Add $\epsilon$ to $\text{FIRST}(X)$
  3. $X$ is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production
     $a$ is in $\text{FIRST}(X)$ if:
     1. An $i$ exists; $a$ is in $\text{FIRST}(Y_i)$ and $\epsilon$ is in every set $\text{FIRST}(Y_1) \ldots \text{FIRST}(Y_{i-1})$
     2. $a = \epsilon$ and $\epsilon$ is in every set $\text{FIRST}(Y_1) \ldots \text{FIRST}(Y_k)$

- $\text{FIRST}(X_1 X_2 \ldots X_n)$:
  Each non-$\epsilon$ symbol of $\text{FIRST}(X_1)$ is in the result
  If $\epsilon \in \text{FIRST}(X_1)$, then each non-$\epsilon$ symbol of $\text{FIRST}(X_2)$ is in the result and so on
  Is $\epsilon$ in every $\text{FIRST}$-set, then it it also is contained in the result
Calculation of FOLLOW

In order to calculate $FOLLOW(A)$ of a nonterminal $A$ use following rules:

1. Add $\$$ to $FOLLOW(S)$, whereby $S$ is the initial symbol
2. For each production $A \rightarrow \alpha B \beta$, add all elements of $FIRST(\beta)$ except $\epsilon$ to $FOLLOW(B)$
3. For each production $A \rightarrow \alpha B$ and $A \rightarrow \alpha B \beta$ with $\epsilon \in FIRST(\beta)$, add each element of $FOLLOW(A)$ to $FOLLOW(B)$
Example

Grammar:

\[ E \rightarrow \text{id} \ E_1 | (E) \]
\[ E_1 \rightarrow \text{op} \ E | \epsilon \]

FIRST sets:

\[ FIRST(E) = \{ \text{id}, () \} \]
\[ FIRST(E_1) = \{ \text{op}, \epsilon \} \]

FOLLOW sets:

\[ FOLLOW(E) = FOLLOW(E_1) = \{ $, ) \} \]
Construction of parsing tables

Input:

Grammar \( G \)

Output: Parsing table \( M \)

1. For each production \( A \rightarrow \alpha \) do Steps 2 and 3.
2. For each terminal \( a \) in \( FIRST(\alpha) \), add \( A \rightarrow \alpha \) to \( M(A, a) \).
3. If \( \epsilon \) is in \( FIRST(\alpha) \), add \( A \rightarrow \alpha \) to \( M(A, b) \) for each terminal \( b \) in \( FOLLOW(A) \). If \( \epsilon \) is in \( FIRST(\alpha) \) and \( \$ \) is in \( FOLLOW(A) \), add \( A \rightarrow \alpha \) to \( M(A, \$) \).
4. Make each undefined entry of \( M \) be \textit{error}.

Example: See table of last example grammar
LL(1) Grammars

- Parsing table construction can be used with arbitrary grammars
- Multiple elements per entry may occur
- **LL(1) Grammar**: Grammar whose parsing table contains no multiple entries
- L . . . Scanning the Input from LEFT to right
- L . . . Producing the LEFTMOST derivation
- 1 . . . Using 1 input symbol lookahead
Properties of LL(1)

- No ambiguous or left-recursive grammar is LL(1)
- \( G \) ist LL(1) \( \iff \) For each two different productions \( A \to \alpha | \beta \) it is necessary that:
  1. No strings may be derived from both \( \alpha \) and \( \beta \) which start with the same terminal \( a \)
  2. At most one of the productions \( \alpha \) or \( \beta \) may be derivable to \( \epsilon \)
  3. If \( \beta \to^* \epsilon \), then \( \alpha \) may not derive any string which starts with an element in \( FOLLOW(A) \)
- Multiple entries in the parsing table can occasionally be removed by hand (without changing the language recognized by the automaton)
Heuristics in panic-mode error recovery:

1. Initially, all symbols in $FOLLOW(A)$ can be used for synchronization: Skip all tokens until an element in $FOLLOW(A)$ is read and remove $A$ from the stack.

2. If $FOLLOW$ sets don’t suffice: Use hierarchical structure of program constructs. E.g. use keywords occurring at the beginning of a statement as addition to the synchronization set.

3. $FIRST(A)$ can be used as well: If an element in $FIRST(A)$ is read, continue parsing at $A$.

4. If a terminal which can’t be matched is at the top of the stack, remove it.
Bottom-up parsing

- Shift-reduce parsing
- Reduction of an input towards the start symbol of the grammar
- Reduction step:
  Replace a substring, which matches the right side of a production with the left side of that same production
- Example:

  \[
  S \rightarrow aABe \\
  A \rightarrow Abc|b \\
  B \rightarrow d \\
  \]

  \[
  abbcde \\
  aAbcde \\
  aAde \\
  aABe \\
  S 
  \]
**Handles**

- Substring, which matches the right side of a production and leads to a valid derivation (rightmost derivation)

**Example (ambiguous grammar):**

Rightmost derivation of `id + id * id`:

<table>
<thead>
<tr>
<th>Right-Sentential Form</th>
<th>Handle</th>
<th>Reducing Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id + id * id</code></td>
<td><code>id</code></td>
<td><code>E → id</code></td>
</tr>
<tr>
<td><code>id + id * E</code></td>
<td><code>id</code></td>
<td><code>E → id</code></td>
</tr>
<tr>
<td><code>id + E * E</code></td>
<td><code>E * E</code></td>
<td><code>E → E * E</code></td>
</tr>
<tr>
<td><code>id + E</code></td>
<td><code>id</code></td>
<td><code>E → id</code></td>
</tr>
<tr>
<td><code>E + E</code></td>
<td><code>E + E</code></td>
<td><code>E → E + E</code></td>
</tr>
</tbody>
</table>

```latex
E → E + E
E → E * E
E → (E)
E → id
```

F. Wotawa (IST @ TU Graz)
Stack implementation

- Initially: Stack $\cdot$ $\cdot$ $\cdot$ $\cdot$ Input $\cdot$ $\cdot$ $\cdot$ $\cdot$ $w$
- Shift $n \geq 0$ symbols from input onto stack until a handle can be found
- Reduce handle (replace handle with left side of production)
### Example shift-reduce parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $</td>
<td>$id + id * id $</td>
<td>shift</td>
</tr>
<tr>
<td>(2) $ id</td>
<td>$ + id * id $</td>
<td>reduce by $E \rightarrow id$</td>
</tr>
<tr>
<td>(3) $ E</td>
<td>$ + id * id $</td>
<td>shift</td>
</tr>
<tr>
<td>(4) $ E +</td>
<td>$ id * id $</td>
<td>shift</td>
</tr>
<tr>
<td>(5) $ E + id</td>
<td>$ * id $</td>
<td>reduce by $E \rightarrow id$</td>
</tr>
<tr>
<td>(6) $ E + E</td>
<td>$ * id $</td>
<td>shift</td>
</tr>
<tr>
<td>(7) $ E + E *</td>
<td>$ id $</td>
<td>shift</td>
</tr>
<tr>
<td>(8) $ E + E * id</td>
<td>$</td>
<td>reduce by $E \rightarrow id$</td>
</tr>
<tr>
<td>(9) $ E + E * E</td>
<td>$</td>
<td>reduce by $E \rightarrow E * E$</td>
</tr>
<tr>
<td>(10) $ E + E</td>
<td>$</td>
<td>reduce by $E \rightarrow E + E$</td>
</tr>
<tr>
<td>(11) $ E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Viable prefixes, conflicts

- **Viable prefix**: Right sentential forms which can occur within the stack of a shift-reduce parser
- **Conflicts**: (Ambiguous grammars)

\[
stmt \rightarrow \text{if } expr \text{ then } stmt \\
| \quad \text{if } expr \text{ then } stmt \text{ else } stmt \\
| \quad \text{other}
\]

**Configuration:**
- Stack
  
  \[
  \ldots \text{if } expr \text{ then } stmt \quad \text{else} \ldots
  \]
- Input

No unambiguous handle (shift-reduce conflict)
LR parser

- LR($k$) parsing
- L . . . Left-to-right scanning
- R . . . Rightmost derivation in reverse

Advantages:
- Can be used for (nearly) every programming language construct
- Most generic backtrack-free shift-reduce parsing method
- Class of LR-grammars is greater than those of LL-grammars
- LR-parsers identify errors as early as possible

Disadvantage: LR-parser is hard to build manually
LR-parsing algorithm

Stack stores $s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$ ($X_i$ grammar, $s_i$ state)

Parsing table = action- and goto-table

$s_m$ current state, $a_i$ current input symbol

\[
\text{action}[s_m, a_i] \in \{\text{shift, reduce, accept, error}\}
\]

\[
\text{goto}[s_m, a_i] \text{ transition function of a DFA}
\]
LR-parsing mode of operation

- **Configuration** \((s_0 X_1 s_1 \ldots X_m s_m, a_i a_{i+1} \ldots a_n)\)

- Next step (move) is determined by reading of \(a_i\)
  
  Dependent on \(\text{action}[s_m, a_i]\):
  
  1. \(\text{action}[s_m, a_i] = \text{shift } s\)
     New configuration: \((s_0 X_1 s_1 \ldots X_m s_m a_i s, a_{i+1} \ldots a_n)\)
  2. \(\text{action}[s_m, a_i] = \text{reduce } A \rightarrow \beta\)
     New configuration: \((s_0 X_1 s_1 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n)\)
     whereby \(s = \text{goto}[s_{m-r}, A], r\ \text{length of } \beta\)
  3. \(\text{action}[s_m, a_i] = \text{accept}\)
  4. \(\text{action}[s_m, a_i] = \text{error}\)
Example

1) \( E \rightarrow E + T \)
2) \( E \rightarrow T \)
3) \( T \rightarrow T \ast F \)
4) \( T \rightarrow F \)
5) \( F \rightarrow (E) \)
6) \( F \rightarrow \text{id} \)

<table>
<thead>
<tr>
<th>State</th>
<th>(\text{id} )</th>
<th>(\ast)</th>
<th>()</th>
<th>$</th>
<th>\text{action}</th>
<th>\text{goto}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td></td>
<td></td>
<td>$</td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>s6</td>
<td></td>
<td></td>
<td>acc</td>
<td>1 2 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>r2</td>
<td>s7</td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>8 2 3</td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9 3</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>s11</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>r1</td>
<td>s7</td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
</tr>
</tbody>
</table>
Construction of SLR parsing tables

- **LR(0)-items**: Production with dot at one position of the right side
  
  *Example*: Production $A \rightarrow XYZ$ has 4 items: $A \rightarrow .XYZ$, $A \rightarrow X.YZ$, $A \rightarrow XY.Z$ and $A \rightarrow XYZ$.
  
  Exception: Produktion $A \rightarrow \epsilon$ only has the item: $A \rightarrow$.

- Augmented grammar: Grammar with new start symbol $S'$ and production $S' \rightarrow S$.

- Functions: *closure* and *goto*

  - $\text{closure}(I)$ ($I \ldots$ set of items)
    
    1. All $I$ are within *closure*
    2. If $A \rightarrow \alpha.B\beta$ is part of *closure* and $B \rightarrow \gamma$ is a production, then add $B \rightarrow .\gamma$ to *closure*
Construction, *goto*

- *goto*(I, X) with I as set of items and X a symbol of the grammar
- goto = closure of set of all items A → αX.β for all A → α.Xβ in I

**Example:** I = \{E' → E., E → E. + T\}

\[ \text{goto}(I, +) = \{ E \rightarrow E + .T, T \rightarrow .T * F, T \rightarrow .F, F \rightarrow .(E), F \rightarrow .id \} \]

- Sets-of-items construction (Construction of all LR(0)-items)

**items(G')**

\[ I_0 = \text{closure}(\{S' \rightarrow .S\}) \]

\[ C = \{ I_0 \} \]

repeat
  for each set of items I ∈ C and each grammar symbol X such that goto(I, X) is not empty and not in C do
    Add goto(I, X) to C
  until no more sets of items can be added to C
SLR parsing table

**Input:** Augmented grammar $G'$

**Output:** SLR parsing table

1. Calculate $C = \{I_0, I_1, \ldots, I_n\}$, the set of LR(0)-items of $G'$

2. State $i$ is created by $I_i$ as follows:
   1. If $A \rightarrow \alpha.a\beta$ is in $I_i$ and $\text{goto}(I_i, a) = I_j$, then $\text{action}(i, a) = \text{shift} \ j$ ($a$ is a terminal symbol)
   2. If $A \rightarrow \alpha.$ is in $I_i$, then $\text{action}[i, a] = \text{reduce} \ A \rightarrow \alpha$ for all $a \in \text{FOLLOW}(A) \land A \neq S'$
   3. If $S' \rightarrow S.$ is in $I_i$, then $\text{action}[i, \$] = \text{accept}$

3. For all nonterminal symbols $A$: $\text{goto}[i, A] = j$ if $\text{goto}(I_i, A) = I_j$

4. Every other table entry is set to $\text{error}$

5. Initial state is determined by the item set with $S' \rightarrow .S$
If we receive a table without multiple entries using the SLR-parsing-table-algorithm then the grammar is SLR(1)
Otherwise the algorithm fails and an algorithm for extended languages (like LR) needs to be utilized
⇒ generally results in increased processing requirements
Shift/reduce-conflicts can be partially resolved
The process usually involves the determination of operator binding strength and associativity
Error handling can be directly incorporated into the parsing table
Translation of context-free languages

Information ↔ attributes of grammar symbols

Values of attributes are defined by “semantic rules”

2 possibilities:
- Syntax directed definitions (high-level spec)
- Translation schemes (implementation details)

Evaluation: (1) Parse input, (2) Generate parse tree, (3) Evaluate parse tree
Syntax directed definitions

- Generalization of context-free grammars
- Each grammar symbol has a set of attributes
- Synthesized vs. inherited attributes
- Attribute: string, number, type, memory location, ...
- Value of attribute is defined by semantic rules
  - **Synthesized**: Value of child node in parse tree
  - **Inherited**: Value of parent node in parse tree
- Semantic rules define dependencies between attributes
- Dependency graph defines calculation order of semantic rules
- Semantic rules can have side effects
Form of a syntax directed definition

- Grammar production: $A \rightarrow \alpha$
- Associated semantic rule: $b := f(c_1, \ldots, c_k)$
- $f$ is a function
- **Synthesized**: $b$ is a synthesized attribute of $A$ and $c_1, \ldots, c_k$ are grammar symbols of the production
- **Inherited**: $b$ is an inherited attribute of a grammar symbol on the right side of the production and $c_1, \ldots, c_k$ are grammar symbols of the production
- $b$ depends on $c_1, \ldots, c_k$
Example

“Calculator”-program: *\textit{val*} is a synthesized attribute for nonterminals $E, T$ and $F$

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow En$</td>
<td>$\text{print}(E.val)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1+T$</td>
<td>$E.val := E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val := T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T_1*F$</td>
<td>$T.val := T_1.val * F.val$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.val := F.val$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$F.val := E.val$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F.val := \text{digit.lexval}$</td>
</tr>
</tbody>
</table>
S-attributed grammar

- Attributed grammar exclusively using synthesized attributes
- **Example-evaluation**: $3*5+4n$ (annotated parse tree)
### Inherited attributes

- Definition of dependencies of program language constructs and their context
- **Example**: (type checking)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow TL$</td>
<td>$L.in := T.type$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$T.type := \text{integer}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{real}$</td>
<td>$T.type := \text{real}$</td>
</tr>
<tr>
<td>$L \rightarrow L_1, \text{id}$</td>
<td>$L_1.in := L.in$ addtype($\text{id}.entry, L.in$)</td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>addtype($\text{id}.entry, L.in$)</td>
</tr>
</tbody>
</table>
**Inherited attributes – Annotated parse tree**

- real $id_1$, $id_2$, $id_3$
Dependency graphs

- Show dependencies between attributes
- Each rule is represented in the form \( b := f(c_1, \ldots, c_k) \)
- Nodes correspond to attributes; edges to dependencies

**Definition:**

```plaintext
for each node \( n \) in the parse tree do
    for each attribute \( a \) of the grammar symbol at \( n \) do
        construct a node in the dependency graph for \( a \)
    for each node \( n \) in the parse tree do
        for each semantic rule \( b := f(c_1, \ldots, c_k) \) associated
            with the production used at \( n \) do
            for \( i := 1 \) to \( k \) do
                construct an edge from the node for \( c_i \) to the node for \( b \)
```
Dependency graph – Example
Topological sort

- Arrangement of $m_1, \ldots, m_k$ nodes in a directed, acyclic graph where edges point from smaller nodes to bigger nodes. If $m_i \rightarrow m_j$ is an edge, then the node $m_i$ is smaller than the node $m_j$.

- Important for order in which the attributes are calculated.

**Example (cont.):**

1. $a_4 := \text{real}$
2. $a_5 := a_4$
3. \text{addtype}(id_3\text{.entry}, a_5)$
4. $a_7 := a_5$
5. \text{addtype}(id_2\text{.entry}, a_7)$
6. $a_9 := a_7$
7. \text{addtype}(id_1\text{.entry}, a_9)$
Example - syntax trees

- Abstract syntax tree = simplified form of a parse tree
- Operators and keywords are supplied to intermediate nodes by leaf nodes
- Productions with only one element can collapse
- Examples:

```
if-then-else
B   S_1   S_2

+   *
3   5
4
3    4

F. Wotawa (IST @ TU Graz)
Syntax trees – Expressions

- **Functions (return value: pointer to new node):**
  - `mknode(op, left, right)`: node label `op`, 2 child nodes `left`, `right`
  - `mkleaf(id, entry)`: leaf `id`, entry in symbol table `entry`
  - `mkleaf(num, val)`: leaf `num`, value `val`

- **Syntax directed definition:**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E → E_1 + T</code></td>
<td><code>E.nptr := mknode('+', E_1.nptr, T.nptr)</code></td>
</tr>
<tr>
<td><code>E → E_1 - T</code></td>
<td><code>E.nptr := mknode('-', E_1.nptr, T.nptr)</code></td>
</tr>
<tr>
<td><code>E → T</code></td>
<td><code>E.nptr := T.nptr</code></td>
</tr>
<tr>
<td><code>T → (E)</code></td>
<td><code>T.nptr := E.nptr</code></td>
</tr>
<tr>
<td><code>T → id</code></td>
<td><code>T.nptr := mkleaf(id, id.entry)</code></td>
</tr>
<tr>
<td><code>T → num</code></td>
<td><code>T.nptr := mkleaf(num, num.val)</code></td>
</tr>
</tbody>
</table>
Syntax tree for $a-4+c$
Evaluation of S-attributed definitions

- Attributed definition exclusively using synthesized attributes
- Evaluation using bottom-up parser (LR-parser)
- Idea: store attribute information on stack

<table>
<thead>
<tr>
<th>State</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>X</td>
<td>X.x</td>
</tr>
<tr>
<td>Y</td>
<td>Y.y</td>
</tr>
<tr>
<td>top →</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Semantic rule:

\[ A.a := f(X.x, Y.y, Z.z) \]

Production: \( A \rightarrow XYZ \)

Before \( XYZ \) is reduced to \( A \), value of \( Z.z \) stored in \( \text{val}[top] \), \( Y.y \) stored in \( \text{val}[top - 1] \), \( X.x \) in \( \text{val}[top - 2] \)
Example - S-attributed evaluation

“Calculator”-example:

<table>
<thead>
<tr>
<th>Production</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E^n$</td>
<td>$\text{print}(\text{val}[\text{top} - 1])$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$\text{val}[\text{ntop}] := \text{val}[\text{top} - 2] + \text{val}[\text{top}]$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$\text{val}[\text{ntop}] := \text{val}[\text{top} - 2] + \text{val}[\text{top}]$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$\text{val}[\text{ntop}] := \text{val}[\text{top} - 2] * \text{val}[\text{top}]$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$\text{val}[\text{ntop}] := \text{val}[\text{top} - 1]$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$\text{val}[\text{ntop}] := \text{val}[\text{top} - 1]$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$\text{val}[\text{ntop}] := \text{val}[\text{top} - 1]$</td>
</tr>
</tbody>
</table>

- Code executed before reduction
- $\text{ntop} = \text{top} - r + 1$, after reduction: $\text{top} := \text{ntop}$
### Result for $3*5+4n$

<table>
<thead>
<tr>
<th>Input</th>
<th>state</th>
<th>val</th>
<th>Production used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3*5+4n$</td>
<td>_</td>
<td>_</td>
<td></td>
</tr>
<tr>
<td>*5+4n</td>
<td>3</td>
<td>3</td>
<td>$F \rightarrow \text{digit}$</td>
</tr>
<tr>
<td>*5+4n</td>
<td>$F$</td>
<td>3</td>
<td>$T \rightarrow F$</td>
</tr>
<tr>
<td>5+4n</td>
<td>$T*$</td>
<td>3 _</td>
<td></td>
</tr>
<tr>
<td>+4n</td>
<td>$T*5$</td>
<td>3 _ 5</td>
<td>$F \rightarrow \text{digit}$</td>
</tr>
<tr>
<td>+4n</td>
<td>$T*F$</td>
<td>3 _ 5</td>
<td>$T \rightarrow T*F$</td>
</tr>
<tr>
<td>+4n</td>
<td>$T$</td>
<td>15</td>
<td>$E \rightarrow T$</td>
</tr>
<tr>
<td>+4n</td>
<td>$E$</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4n</td>
<td>$E+$</td>
<td>15 _</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>$E+4$</td>
<td>15 _ 4</td>
<td>$F \rightarrow \text{digit}$</td>
</tr>
<tr>
<td>n</td>
<td>$E+F$</td>
<td>15 _ 4</td>
<td>$T \rightarrow F$</td>
</tr>
<tr>
<td>n</td>
<td>$E+T$</td>
<td>15 _ 4</td>
<td>$E \rightarrow E+T$</td>
</tr>
<tr>
<td>n</td>
<td>$E$</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>$En$</td>
<td>19 _</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>19</td>
<td></td>
<td>$L \rightarrow En$</td>
</tr>
</tbody>
</table>
**L-attributed definitions**

- **Definition:** A syntax directed definition is L-attributed if each inherited attribute of $X_j, 1 \leq j \leq n$, on the right side of $A \rightarrow X_1, \ldots, X_n$ is only dependent on:
  1. the attributes $X_1, \ldots, X_{j-1}$ to the left of $X_j$ and
  2. the inherited attributes of $A$

- Each S-attributed grammar is a L-attributed grammar.

- Evaluation using depth-first order

```
procedure dfvisit(n : node)
    for each child $m$ of $n$, from left to right do
        evaluate inherited attributes of $m$
        dfvisit($m$)
    end
    evaluate synthesized attributes of $n$
end
```
Translation schemes

- Translation scheme = context-free language with attributes for grammar symbols and semantic actions which are placed on the right side of a production between grammar symbols and are confined within {}.

- **Example:**

  \[ T \rightarrow T_1 \ast F\{T.val := T_1.val \ast F.val\} \]

- If only synthesized attributes are used, the action is always placed at the end of the right side of a production.

- Note: Actions may not access attributes which are not calculated yet (limits positions of semantic actions).
If both inherited and synthesized attributes are used the following needs to be taken into consideration:

1. An inherited attribute of a symbol on the right side of a production has to be calculated in an action which is positioned to the left of the symbol

2. An action may not reference a synthesized attribute belonging to a symbol which is positioned to the right of the action

3. A synthesized attribute of a nonterminal on the left side can only be calculated if all referenced attributes have already been calculated $\Rightarrow$ actions like these are usually placed at the end of the right side
Example translation scheme

\[ S \rightarrow A_1 A_2 \quad \{ A_1.in := 1; A_2.in := 2 \} \]
\[ A \rightarrow a \quad \{ \text{print}(A.in) \} \]

- Above grammar does not fulfill the three conditions for translation schemes.
- The inherited attribute \( A.in \) is not yet defined at the point in time when it should be printed.
- But: For each L-attributed grammar a translation scheme can be found which fulfills the three conditions, e.g.:

\[ S \rightarrow \{ A_1.in := 1 \} A_1 \quad \{ A_2.in := 2 \} A_2 \]
\[ A \rightarrow a \quad \{ \text{print}(A.in) \} \]
Top-down translation

- Removal of left recursions in translation scheme is necessary
  
  \[
  E \rightarrow E_1 + T \quad \{E.val := E_1.val + T.val\}
  \]
  
  \[
  E \rightarrow E_1 - T \quad \{E.val := E_1.val - T.val\}
  \]
  
  \[
  E \rightarrow T \quad \{E.val := T.val\}
  \]
  
  \[
  T \rightarrow (E) \quad \{T.val := E.val\}
  \]
  
  \[
  T \rightarrow \text{num} \quad \{T.val := \text{num}.val\}
  \]
  
- Example:
Example top-down translation

\[
E \rightarrow T \quad \{ R.i := T.val \}
\]

\[
R \rightarrow + \quad T \quad \{ R_1.i := R.i + T.val \}
\]

\[
R_1 \quad \{ R.s := R_1.s \}
\]

\[
R \rightarrow - \quad T \quad \{ R_1.i := R.i - T.val \}
\]

\[
R_1 \quad \{ R.s := R_1.s \}
\]

\[
R \rightarrow \epsilon \quad \{ R.s := R.i \}
\]

\[
T \rightarrow ( \quad E \quad ) \quad \{ T.val := E.val \}
\]

\[
T \rightarrow \text{num} \quad \{ T.val := \text{num}.val \}
\]
Evaluation of $9 - 5 + 2$

```
E
   /\   \
  /   \  \
 T.val = 9  R.i = 9
   \    /
    \  /
   num.val = 9

-   T.val = 5  R.i = 4
   \    /
    \  /
   num.val = 5

+   T.val = 2  R.i = 6
   \    /
    \  /
   num.val = 2
```
Summary transformation

Given translation scheme:

\begin{align*}
A & \rightarrow A_1 Y & \{ A.a := g(A_1.a, Y.y) \} \\
A & \rightarrow X & \{ A.a := f(X.x) \}
\end{align*}

After removal of left recursions:

\begin{align*}
A & \rightarrow X R \\
R & \rightarrow Y R | \epsilon
\end{align*}

Transformed scheme:

\begin{align*}
A & \rightarrow X & \{ R.i := f(X.x) \} \\
R & \rightarrow Y & \{ R.s := R.s \} \\
R & \rightarrow Y & \{ R.s := R.s \} \\
R & \rightarrow \epsilon & \{ R.s := R.i \}
\end{align*}
Predictive parsing with schemes

**Input:** syntax-directed translation scheme; **Outp.:** Syntax-directed translator

1. For each nonterminal $A$, construct a function that has a formal parameter for each inherited attribute of $A$ and that returns the values of the synthesized attributes of $A$. This function has a local variable for each attribute of each grammar symbol that appears in a production for $A$.

2. As previously described (see predictive parsing), the code for nonterminal $A$ decides what production to use based on the current input symbol.

3. The code for each production does the following (evaluation from left to right):
   1. Token $X$ with synthesized attribute $x$: Save the value of $x$ in a variable $X.x$. Generate a call to match token $X$.
   2. Nonterminal $B$: Generate $c := B(b_1, \ldots, b_k)$; $b_1, \ldots, b_k$ variables for inherited attributes of $B$; $c$ variable for synthesized attribute of $B$.
   3. For an action, copy the code into the parser, replacing each reference to an attribute by the variable for that attribute.
Example - predictive parsing

Grammar:

- **Grammar:**

  - **E** → **T** \{R.i := \(T.val\)}
  - **R** \{E.val := R.s\}
  - **R** → **op**
    - **T** \{R1.i := mknnode(\(op.lexeme, R.i, T.nptr\))\}
    - **R1** \{R.s := R1.s\}
  - **R** → **\(\epsilon\)** \{R.s := R.i\}
  - **T** → (**E**
    - **E**
    - **)**) \{T.val := E.val\}
  - **T** → **num** \{T.val := num.val\}

- **Functions:**

  - **function E : node**
  - **function R(i : node) : node**
  - **function T : node**
Procedure without translation scheme

```pseudocode
procedure R()
begin
    if lookahead = op then begin
        match(op);
        T();
        return R();
    end else begin
        return;
    end
end
```
Parsing function $R$

function $R (i: \text{node}) : \text{node}$

\begin{verbatim}
var $nptr$, $i_1$, $s_1$, $s$: node; $oplexeme$: char;

begin
    if $\text{lookahead} = \text{op}$ then begin
        $oplexeme$ := $\text{lexval}$;
        $\text{match}(\text{op})$;
        $nptr$ := $T()$;
        $i_1$ := $\text{mknode}(\text{oplexeme}, i, nptr)$;
        $s_1$ := $R(i_1)$;
        $s$ := $s_1$
    end else
    $s$ := $i$;

    return $s$
end
\end{verbatim}
Bottom-up with inherited attribute

- Implementation of L-attributed grammars in bottom-up parsers
- For LL(1)-grammars and many LR(1)-grammars
- Removal of embedding actions from translation schemes:
  - Actions have to be placed at end of right side of a production
  - Ensured by new *marker* nonterminals

**Example:**

\[
\begin{align*}
E & \rightarrow TR \\
R & \rightarrow +T\{print('\plus\}')R|-T\{print('\minus\}')R|\epsilon \\
T & \rightarrow \text{num}\{print(\text{num.val})\} \\
M & \rightarrow \epsilon\{print('\plus\}') \\
N & \rightarrow \epsilon\{print('\minus\}')
\end{align*}
\]
Inherited attributes on the stack

- **Idea:** Production $A \rightarrow XY$, synthesized attribute $X.x$ and inherited attribute $Y.y$
  - Before a reduction (of $X Y$), $X.x$ is on the stack
  - In the case of $Y.y = X.x$ (copy action), the value of $X.x$ can be used whenever the value of $Y.y$ is required

- **Example:** Parser for variable declarations `real p, q, r`
Variable declaration - example

\[
D \rightarrow T \quad \{ L.in := T.type \}
\]

\[
T \rightarrow \text{int} \quad \{ T.type := \text{integer} \}
\]

\[
T \rightarrow \text{real} \quad \{ T.type := \text{real} \}
\]

\[
L \rightarrow L_1 \quad \{ L_1.in := L.in \}
\]

\[
L \rightarrow \text{id} \quad \{ \text{addtype}(\text{id}.entry, L.in) \}
\]

\[
L \rightarrow \text{id} \quad \{ \text{addtype}(\text{id}.entry, L.in) \}
\]
Calculation using the stack

<table>
<thead>
<tr>
<th>Input</th>
<th>state</th>
<th>Production Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>real p,q,r</td>
<td>-</td>
<td>T → real</td>
</tr>
<tr>
<td>p,q,r</td>
<td>real</td>
<td>T → real</td>
</tr>
<tr>
<td>p,q,r</td>
<td>T p</td>
<td>T → real</td>
</tr>
<tr>
<td>,q,r</td>
<td>TL</td>
<td>T → real</td>
</tr>
<tr>
<td>,q,r</td>
<td>TL , q</td>
<td>T → real</td>
</tr>
<tr>
<td>q,r</td>
<td>TL , r</td>
<td>T → real</td>
</tr>
<tr>
<td>,r</td>
<td>TL</td>
<td>T → real</td>
</tr>
<tr>
<td>,r</td>
<td>L , id</td>
<td>T → L , id</td>
</tr>
<tr>
<td>r</td>
<td>TL</td>
<td>T → L , id</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>D → TL</td>
</tr>
</tbody>
</table>

Implementation:

<table>
<thead>
<tr>
<th>Production</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → TL</td>
<td>val[top] := integer</td>
</tr>
<tr>
<td>T → int</td>
<td>val[top] := real</td>
</tr>
<tr>
<td>T → real</td>
<td>addtype(val[top], val[top − 3])</td>
</tr>
<tr>
<td>L → L , id</td>
<td>addtype(val[top], val[top − 1])</td>
</tr>
</tbody>
</table>
Positions of attributes on the stack need to be known

When the reduction $C \rightarrow c$ is conducted, it is unknown whether the value of $C.i$ is located in $\text{val}[\text{top} - 1]$ or in $\text{val}[\text{top} - 2]$! It depends on whether a $B$ is located on the stack.

Solution: Introduction of a marker $M$:

- $S \rightarrow aAC$
  - $C.i := A.s$
  - $C.s := g(C.i)$

- $S \rightarrow aABMC$
  - $M.i := A.s; C.i := M.s$

- $C \rightarrow c$
  - $C.s := g(C.i)$

- $M \rightarrow \epsilon$
  - $M.s := M.i$
Problems (cont.)

- Simulation of semantic rules which are no copy actions
- Usage of marker!

\[ S \rightarrow aAC \quad C. i := f(A. s) \]
\[ S \rightarrow aANC \quad N. i := A. s; C. i := N. s \]
\[ N \rightarrow \epsilon \quad N. s := f(N. i) \]
Bottom-up parsing . . .

. . . with calculation of inherited attributes

**Input:** L-attributed definition (and LL(1)-grammar)

**Output:** Parser, which calculates attribute values on stack

1. **Assumptions:** Each nonterminal $A$ has an inherited attribute $A.i$, each grammar symbol $X$ has a synthesized attribute $X.s$. If $X$ is a terminal, then $X.s$ is the lexical value of $X$ (supplied by the lexical analyser). The values are stored on the stack in form of an array $val$.

2. For each production $A \rightarrow X_1 \ldots X_n$ create $n$ new markers (nonterminals) $M_1, \ldots, M_n$ and replace the production with $A \rightarrow M_1 X_1 \ldots M_n X_n$.
   
   **Note:** synthesized values for $X_i$ are stored in the $val$ array entry, which belongs to $X_i$. Inherited values $X_i.i$ are stored in entries which are associated to $M_i$.

3. **Invariant:** The new inherited attribute $A.i$ (if existing) is always directly beneath the position of $M_1$ within the $val$ array.
Simplifications

- Reduction of markers:
  1. If $X_j$ has no inherited attribute, then no marker $M_j$ is required $\Rightarrow$ positions of attributes on the stack are shifting!
  2. If $X_1.i$ exists and is calculated by $X_1.i = A.i$, then $M_1$ is not required
Removal of inherited attributes

• Replacement of inherited attributes by synthesized ones
• Not always possible
• Requires modification of grammar!

**Example:** Declarations in Pascal

\[
\begin{align*}
D & \rightarrow L : T \\
T & \rightarrow \text{integer} | \text{char} \\
L & \rightarrow L, \text{id} | \text{id}
\end{align*}
\]

\[
\begin{align*}
D & \rightarrow \text{id} L \\
T & \rightarrow \text{integer} | \text{char}
\end{align*}
\]
Difficult syntax directed definition

The following definition cannot be processed by bottom-up parsers using current approaches

\[ S \rightarrow L \quad L\.count := 0 \]
\[ L \rightarrow L_1 \_1 \quad L_1\.count := L\.count + 1 \]
\[ L \rightarrow \epsilon \quad print(L\.count) \]

**Reason:** \( L \rightarrow \epsilon \) receives the number of 1s by means of inheritance. However, as \( L \rightarrow \epsilon \) is used in the reduction first, no value is specified yet!
Recursive evaluators

- Evaluation of attributes
- Based on parse tree
- Not possible in conjunction with parsing
- Order of nodes which are visited during evaluation is arbitrary
- For each nonterminal a translation function exists
- Extensions may visit nodes more than once
- Order of node visits needs to regard the following:
  1. Each inherited attribute of a node has to be calculated before the node is visited
  2. Synthesized attributes are calculated before the node is left (for the last time)
- Order is determined by dependencies
Example – Recursive evaluators

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow LM$</td>
<td>$L.i := l(A.i)$</td>
</tr>
<tr>
<td></td>
<td>$M.i := m(L.s)$</td>
</tr>
<tr>
<td></td>
<td>$A.s := f(M.s)$</td>
</tr>
<tr>
<td>$A \rightarrow QR$</td>
<td>$R.i := r(A.i)$</td>
</tr>
<tr>
<td></td>
<td>$Q.i := q(R.s)$</td>
</tr>
<tr>
<td></td>
<td>$A.s := f(Q.s)$</td>
</tr>
</tbody>
</table>

function $A(n, ai)$

if $production(n) = 'A \rightarrow LM'$ then

li := l(ai)
l := L(child(n, 1), li)
mi := m(ls)
ms := M(child(n, 2), mi)
return $f(ms)$

if $production(n) = 'A \rightarrow QR'$ then

ri := r(ai)
rs := R(child(n, 2), ri)
qi := q(rs)
qs := Q(child(n, 1), qi)
return $f(qs)$
Static Program Checking

- **Type Checks** Check of the used type. Error if operands are incompatible with the used operator. Example: $1.2 + 2$ ($real + int$).

- **Flow-of-Control Checks** Check if the transfer of the program execution is possible. Example: `break` needs an enclosing loop. `goto label` needs a defined label.

- **Uniqueness Checks** Check if an object has been defined exactly once. Example: In Pascal each identifier must be unique.

- **Name-related Checks** In some languages, names (e.g. for procedures) are used which need to occur at a different location (e.g. at the end of a procedure).
Tasks

- Check if the type system of the language is satisfied.
- Separate type checker is not always necessary.

Typesystems (Examples):

- “If both operands of the arithmetic operators of addition, subtraction and multiplication are of type integer, then the result is of type integer”
- “The result of the unary & operator is a pointer to the object referred to by the operand. If the type of the operand is ‘...’, the type of the result is ’pointer of ...’.”
A type expression is:

1. **a Basic Type** integer, boolean, char, and real as well as a special Basic Type type_error or void.

2. **the Type Name**

3. **a composite type** in the form of:
   - **Arrays.** array\((I, T)\); set of indexes \(I\), type \(T\)
   - **Products.** \(T_1 \times T_2\)
   - **Records.** record\(((N_1 \times T_1) \times \ldots \times (N_k \times T_k))\); name \(N_i\), types \(T_i\)
   - **Pointers.** pointer\((T)\)
   - **Functions.** \(T_1 \rightarrow T_2\)

4. **and type variables.**
type row = record
    address: integer;
    lexeme: array[1..15] of char
  end;

var table: array[1..101] of row;

row can be represented as
  record((address × integer), (lexeme × array(1..15, char))).

function f(a,b: char): ↑ integer;
is represented as: char × char → pointer(integer).
Graphical Representation of Types

- as DAG (Directed Acyclic Graph)

- or as a tree
Typesystems

- Set of Rules
  - specified using attributed grammars (or verbally)
- Static vs. Dynamic Checking of Types
- Sound Typesystem = static type checking is sufficient
- Language is strongly typed = the compiler guarantees that an accepted program runs without type errors.

But some checks can only occur dynamically

```plaintext
table: array[0..255] of char;
i: integer;
```

The correctness of the call `table[i]` in the program can not be checked by the compiler.

- Error Recovery is important (even for type errors)
Type Checker Spec

Language Definition:

\[ P \rightarrow D ; E \]
\[ D \rightarrow D ; D | \text{id : } T \]
\[ T \rightarrow \text{char} | \text{integer} | \text{array [ num ] of } T | \uparrow T \]
\[ E \rightarrow \text{literal} | \text{num} | \text{id} \downarrow E \text{ mod } E | E[E] | E \uparrow \]

Example:

- key: integer ;
  key mod 1999

- array [256] of char
  array(1...256, char)
### 1. Secure Type Info

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \rightarrow D ; E$</td>
<td>${ \text{addtype( id.entry, } T.\text{type}) }$</td>
</tr>
<tr>
<td>$D \rightarrow D ; D$</td>
<td>${ T.\text{type} := \text{char} }$</td>
</tr>
<tr>
<td>$D \rightarrow \text{id} : T$</td>
<td>${ T.\text{type} := \text{integer} }$</td>
</tr>
<tr>
<td>$T \rightarrow \text{char}$</td>
<td>${ T.\text{type} := \text{array}(1 \ldots \text{num}.\text{val}, T_1.\text{type}) }$</td>
</tr>
<tr>
<td>$T \rightarrow \text{integer}$</td>
<td>${ T.\text{type} := \text{pointer}(T_1.\text{type}) }$</td>
</tr>
<tr>
<td>$T \rightarrow \text{array [ num ] of } T_1$</td>
<td></td>
</tr>
<tr>
<td>$T \rightarrow \uparrow T_1$</td>
<td></td>
</tr>
</tbody>
</table>
2. Type Checking – Expressions

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow \text{literal}$</td>
<td>$E.type := \text{char}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{num}$</td>
<td>$E.type := \text{integer}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$E.type := \text{lookup}(\text{id}.entry)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \mod E_2$</td>
<td>$E.type := \begin{cases} \text{integer} &amp; \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{integer} \ \text{type_error} &amp; \text{else} \end{cases}$</td>
</tr>
<tr>
<td>$E \rightarrow E_1[E_2]$</td>
<td>$E.type := \begin{cases} \text{t} &amp; \text{if } E_1.type = \text{array}(s, t) \text{ and } E_2.type = \text{integer} \ \text{type_error} &amp; \text{else} \end{cases}$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \uparrow$</td>
<td>$E.type := \begin{cases} \text{t} &amp; \text{if } E_1.type = \text{pointer}(t) \ \text{type_error} &amp; \text{else} \end{cases}$</td>
</tr>
</tbody>
</table>
### 3. Type Checking – Statements

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $S \rightarrow \text{id} := E$ | $S\.type := \begin{cases} 
\text{if } \text{id\.type} = E\.type \text{ then void} \\
\text{else type\_error}
\end{cases}$ |
| $S \rightarrow \text{if } E \text{ then } S_1$ | $S\.type := \begin{cases} 
\text{if } E\.type = \text{boolean} \text{ then } S_1\.type \\
\text{else type\_error}
\end{cases}$ |
| $S \rightarrow \text{while } E \text{ do } S_1$ | $S\.type := \begin{cases} 
\text{if } E\.type = \text{boolean} \text{ then } S_1\.type \\
\text{else type\_error}
\end{cases}$ |
| $S \rightarrow S_1 ; S_2$ | $S\.type := \begin{cases} 
\text{if } S_1\.type = \text{void and} \\
\quad S_2\.type = \text{void} \text{ then void} \\
\text{else type\_error}
\end{cases}$ |
4. Type Checking – Functions

- **Syntax Extension:**
  \[ T \rightarrow T' \rightarrow' T \]
  \[ E \rightarrow E (E) \]
  Definition
  FunctionCall

- **Type Extraction + Type Checking:**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T \rightarrow T_1' \rightarrow' T_2 )</td>
<td>( T.type := T_1.type \rightarrow T_2.type )</td>
</tr>
</tbody>
</table>
| \( E \rightarrow E_1 (E_2) \)                           | \( E.type := \left\{ \begin{array}{ll}
  \text{if } E_1.type = s \rightarrow t \text{ and } \\
  E_2.type = s \text{ then } t \\
  \text{else type_error}
\end{array} \right. \) |

- **Example:**
  \[ \text{root} : ((\text{real} \rightarrow \text{real}) \times \text{real}) \rightarrow \text{real} \]
Type Equivalence

When are types equivalent???

- structural equivalence
- name equivalence
function sequiv(s, t) : boolean; begin
  if s and t are the same basic type then
    return true
  else if $s = \text{array}(s_1, s_2)$ and $t = \text{array}(t_1, t_2)$ then
    return $s_1 = t_1$ and sequiv($s_2, t_2$)
  else if $s = s_1 \times s_2$ and $t = t_1 \times t_2$ then
    return sequiv($s_1, t_1$) and sequiv($s_2, t_2$)
  else if $s = \text{pointer}(s_1)$ and $t = \text{pointer}(t_1)$ then
    return sequiv($s_1, t_1$)
  else if $s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$ then
    return sequiv($s_1, t_1$) and sequiv($s_2, t_2$)
  else return false end
Encoding of Type Expressions

**Expression as Bit vector** (efficient storage and comparison)

Example:

<table>
<thead>
<tr>
<th>Type Constructor</th>
<th>Encoding</th>
<th>Basic Type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>01</td>
<td>boolean</td>
<td>0000</td>
</tr>
<tr>
<td>array</td>
<td>10</td>
<td>char</td>
<td>0001</td>
</tr>
<tr>
<td>freturns</td>
<td>11</td>
<td>integer</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real</td>
<td>0011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type expression</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>00 00 00 0001</td>
</tr>
<tr>
<td>freturns(char)</td>
<td>00 00 11 0001</td>
</tr>
<tr>
<td>pointer(freturns(char))</td>
<td>00 01 11 0001</td>
</tr>
<tr>
<td>array(pointer(freturns(char)))</td>
<td>10 01 11 0001</td>
</tr>
</tbody>
</table>
Name vs. structural Equivalence

- Example (Pascal Programm)

  ```pascal
  type link = ↑ cell;
  var next: link;
      last : link;
      p : ↑ cell;
      q,r : ↑ cell;
  
  Do all variables have the same type?
  Depends on the typesystem (and the compiler in pascal!)
  Implementation of the above example creates implicit types (e.g.
  type np : ↑ cell for variable p).
  ```
Cyclic Typedefinition (Example)

```haskell
type link = ↑ cell;
  cell = record
    info: integer;
    next: link;
  end;
```

F. Wotawa (IST @ TU Graz)
Type conversion / Coercions

- Statement of the problem: $x + i$ with $x$ as a real- and $i$ as an integer variable.
- There exist only operators for (real + real) or (int + int)
- Type conversion necessary! $x = \text{int2real}(i)$
- Implicit (by the compiler) or explicit (by the programmer) possible
- Implicit = Coercion
- Loss of information should be prevented (int → real but not real → int).
- Performance!!!

```
for I := 1 to N do X[I] := \text{int2real}(1) \quad \text{(PASCAL; X is an array of reals) needs 48.4 \mu s}
```
```
for I := 1 to N do X[I] := 1.0 \quad \text{needs only 5.4 \mu s.}
```
## Type conversion - Semantic Rules (1)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$E.type := \text{lookup}(\text{id}.entry)$ &lt;br&gt; $E.txt := \text{id}.entry$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \ \text{op} \ E_2$</td>
<td>$E.type := \begin{cases} \text{integer} &amp; \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{integer} \ \text{real} &amp; \text{if } E_1.type = \text{integer} \text{ and } E_2.type = \text{real} \ \text{real} &amp; \text{if } E_1.type = \text{real} \text{ and } E_2.type = \text{integer} \ \text{real} &amp; \text{if } E_1.type = \text{real} \text{ and } E_2.type = \text{real} \ \text{type_error} &amp; \text{else} \end{cases}$</td>
</tr>
</tbody>
</table>
Type conversion - Semantic Rules (2)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $E \rightarrow E_1 \ op \ E_2$ | $E.txt := \begin{cases} 
  \text{if } E_1.type = \text{integer and } E_2.type = \text{integer} & \text{then } E_1.txt \circ E_2.txt \\
  \text{else if } E_1.type = \text{integer and } E_2.type = \text{real} & \text{then } \text{int2real}(E_1.txt) \circ E_2.txt \\
  \text{else if } E_1.type = \text{real and } E_2.type = \text{integer} & \text{then } E_1.txt \circ \text{int2real}(E_2.txt) \\
  \text{else if } E_1.type = \text{real and } E_2.type = \text{real} & \text{then } E_1.txt \circ E_2.txt \\
  \text{else type\_error} & 
\end{cases}$ |
| $E \rightarrow \text{num}$      | $E.type := \text{integer}$                         |
|                                     | $E.txt := \text{val}$                             |
| $E \rightarrow \text{num.\text{num}}$ | $E.type := \text{real}$                           |
|                                     | $E.txt := \text{val}$                             |
Overloading

- Symbols with different meaning (dependent on application context)
  - *mathematics*: + operator (integer, reals, complex numbers)
  - *ADA*: ()-Expression for array access AND function calls
- Overloading is resolved, when the meaning is clear (operator identification)
- Overloading can often be resolved by the types of operands.
Overloading - Possible Types

Example (ADA):

function "*"(i,j: integer) return complex;
function "*"(i,j: complex) return complex;

Possible types for * are:

- integer \times integer \rightarrow integer
- integer \times integer \rightarrow complex
- complex \times complex \rightarrow complex

Assumption: 2,3,5 are integer

- 3 \times 5 is either integer or complex.
- So 2 \times (3 \times 5) must be of type integer.
- (3 \times 5) \times z is of type complex, if z is of type complex.
Handling Overloading

- Instead of a type the set of all possible types must be stored in an attribute.
- **Attribute** `types`
  
  \[
  \begin{align*}
  E' \rightarrow E & \quad E'.types = E.types \\
  E \rightarrow \text{id} & \quad E.types = \{\text{lookup( id.entry)}\} \\
  E \rightarrow E_1 ( E_2 ) & \quad E.types = \{t | \exists s \in E_2.types \land s \rightarrow t \in E_1.types\}
  \end{align*}
  \]
- **Example:** \(3 \times 5\)

```
E: {i,c}
  \|--
  E: {i}
    \|--
    3: {i}
      \|--
      \{ i \times i \rightarrow i, i \times i \rightarrow c, c \times c \rightarrow c \}
      \|--
      \{ i \times i \rightarrow i \}
    \|--
    \{ i \times i \rightarrow c \}
  \|--
  \{ i \times i \rightarrow i \}
  \|--
  \{ i \times i \rightarrow c \}
  \|--
  E: {i}
    \|--
    5: {i}
```
### Uniqueness of Types

Expressions may only have one type (otherwise type_error)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| $E' \rightarrow E$ | $E'.\text{types} := E.\text{types}$  
$E.\text{unique} := \text{if } E'.\text{types} = \{t\} \text{ then } t \text{ else } \text{type_error}$ |
| $E \rightarrow \text{id}$ | $E.\text{types} := \{\text{lookup(id.entry)}\}$ |
| $E \rightarrow E_1(\ E_2\ )$ | $E.\text{types} := \{s' | \exists s \in E_2.\text{types} \land (s \rightarrow s') \in E_1.\text{types}\}$  
$t := E.\text{unique}$  
$S := \{s | s \in E_2.\text{types} \land s \rightarrow t \in E_1.\text{types}\}$  
$E_2.\text{unique} := \text{if } S = \{s\} \text{ then } s \text{ else } \text{type_error}$  
$E_1.\text{unique} := \text{if } S = \{s\} \text{ then } s \rightarrow t \text{ else } \text{type_error}$ |
Polymorphic Functions

- Polymorphic Function = Function, whose argument may have an arbitrary type
- Polymorphic refers to functions and operators
- Examples: Built-in operators for array-access, pointer manipulation
- Reason for polymorphism:
  - Code can be used for various data structures
  - Example: finding the length of lists (e.g. ML)
    ```
    fun length(lptr) =
        if null(lptr) then 0
        else length(tl(lptr))
    length([sun,mon,tue]), length([1,2,3,4])
    not possible in PASCAL!
    ```
Type Variables

- Variables, that allow us to talk about unknown types
- Note: Type Variables as greek letters $\alpha, \beta, \ldots$
- *Type Inference* = problem of deciding the type of an expression taking into account the application (of the expression).

```plaintext
type link ↑ cell;
procedure mlist ( lptr : link; procedure p)
begin
  while lptr <> nil do begin
    p(lptr);
    lptr := lptr↑.next
  end
end;

mlist: link $\times$ procedure $\rightarrow$ void
p: link $\rightarrow$ void
```
Example - Type Inference

Program:

```plaintext
def function deref(p);
begin
  return p↑
end;
```

Derivation:

1. Type of \( p \) is \( \beta \) (Assumption)
2. From \( p↑ \) follows that \( p \) must be a pointer. Therefore it holds:
   \[ \beta = \text{pointer}(\alpha). \]
3. Furthermore, we know that the type of \( p↑ \) must be \( \alpha \).
4. Therefore, it follows: \( \forall \alpha : \text{pointer}(\alpha) \rightarrow \alpha \) is the type of the function deref.
Language for Polymorphism

- Type expression of the form $\forall \alpha. E(\alpha)$ denotes a ‘polymorph type’.
- Language definition:
  
  $P \rightarrow D ; E$
  
  $D \rightarrow D ; D|\text{id} : Q$
  
  $Q \rightarrow \forall \text{type_variable} . Q|T$
  
  $T \rightarrow T' \rightarrow' T|T \times T|(T)$
  
  $|\text{unary.constructor}(T)$
  
  $|\text{basic.type} | \text{type.variable}$
  
  $E \rightarrow E(E)|E,E|\text{id}$
Example

deref : \( \forall \alpha. \text{pointer}(\alpha) \rightarrow \alpha \); 
q : \text{pointer}(\text{pointer}(\text{integer})) ;
deref(deref(q))

\textbf{apply} : \alpha_0

deref_0 : \text{pointer}(\alpha_0) \rightarrow \alpha_0

\textbf{apply} : \alpha_i

deref_i : \text{pointer}(\alpha_i) \rightarrow \alpha_i \quad q : \text{pointer}(\text{pointer}(\text{integer}))
Differences in Type Handling

In distinction from former type handling (without polymorphism):

1. Arguments of polymorph functions in an expression may have different types.

2. The concept of type equivalence is different.

3. Calculated Types must be used in further consequence. The effect of the unification of two expressions must be preserved.

\[ \alpha \] is assigned the type \( t \). If \( \alpha \) is referenced elsewhere, \( t \) must be used!

Terms: Substitution, Instances, Unification
Substitution, Instances

- **Substitution** = function that maps type variables to type expressions. \( S : \text{type}\_\text{variables} \mapsto \text{type}\_\text{expressions} \)

- **Example**: \( \alpha \mapsto \text{pointer} (\text{integer}) \)

- **Application of a substitution**: 
  
  ```
  function subst(t : type_expression) : type_expression
  begin
      if t is a basic type then return t
      else if t is a variable then return S(t)
      else if t is \( t_1 \rightarrow t_2 \) then return subst(\( t_1 \)) \( \rightarrow \) subst(\( t_2 \))
  end
  ```

- \( S(t) \) ... Instance. We write \( s < t \iff s \) is instance of \( t \).
Examples

- **Instances:**
  
  \[
  \begin{align*}
  \text{pointer}(&\text{integer}) < \text{pointer}(\alpha) \\
  \text{pointer}(&\text{real}) < \text{pointer}(\alpha) \\
  \text{integer} \rightarrow \text{integer} < \alpha \rightarrow \alpha \\
  \text{pointer}(\alpha) < \beta \\
  \alpha < \beta
  \end{align*}
  \]

- **No Instances:**
  
  \[
  \begin{align*}
  \text{integer} & \quad \text{real} & \quad \text{substitution on Basic Types not possible} \\
  \text{integer} \rightarrow \text{real} & \quad \alpha \rightarrow \alpha & \quad \text{inconsistent replacement of } \alpha \\
  \text{integer} \rightarrow \alpha & \quad \alpha \rightarrow \alpha & \quad \text{all occurrences must be replaced}
  \end{align*}
  \]
Unification

- 2 types \( t_1, t_2 \) are unifiable if there exists a substitution \( S \), so that \( S(t_1) = S(t_2) \) holds.

- In praxis, we are interested in the **Most General Unifier (MGU)**.
  1. \( S(t_1) = S(t_2) \)
  2. Every substitution \( S' \) with \( S'(t_1) = S'(t_2) \) must be an instance of \( S \).
Checking Polymorphic Functions

- **2 Functions:**
  1. \( \text{fresh}(t) \) replaces all Variables in the type expression \( t \) with new variables. A pointer to the node representing the new expression is returned.
  2. \( \text{unify}(m, n) \) unifies the two expressions \( m \) and \( n \). As a side effect the substitution is performed.

- **Translation Schema:**

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
</table>
| \( E \rightarrow E_1 \ ( E_2 ) \) | \( p := mkleaf(\text{newtypevar}); \) 
  \( \text{unify}(E_1.\text{type}, mknod(e' \rightarrow', E_2.\text{type}, p)); \) 
  \( E.\text{type} := p \) |
| \( E \rightarrow E_1 \ , E_2 \) | \( E.\text{type} := mknod(e' \times', E_1.\text{type}, E_2.\text{type}) \) |
| \( E \rightarrow \text{id} \) | \( E.\text{type} := \text{fresh}(\text{id}.\text{type}) \) |
Example Type Checking

\[ \text{apply} : \alpha_0 \]

\[ \text{deref}_0 : \text{pointer}(\alpha_0) \to \alpha_0 \]

\[ \text{apply} : \alpha_i \]

\[ \text{deref}_i : \text{pointer}(\alpha_i) \to \alpha_i \]

\[ q : \text{pointer}(\text{pointer}(\text{integer})) \to \beta \]

Summary (Bottom-up type detection):

<table>
<thead>
<tr>
<th>Expression : Type</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ q : \text{pointer}(\text{pointer}(\text{integer})) ]</td>
<td>[ \alpha_i = \text{pointer}(\text{integer}) ]</td>
</tr>
<tr>
<td>[ \text{deref}_i : \text{pointer}(\alpha_i) \to \alpha_i ]</td>
<td></td>
</tr>
<tr>
<td>[ \text{deref}_i(q) : \text{pointer}(\text{integer}) ]</td>
<td></td>
</tr>
<tr>
<td>[ \text{deref}_0 : \text{pointer}(\alpha_0) \to \alpha_0 ]</td>
<td>[ \alpha_0 = \text{integer} ]</td>
</tr>
<tr>
<td>[ \text{deref}_0(\text{deref}_i(q)) : \text{integer} ]</td>
<td></td>
</tr>
</tbody>
</table>
Unification Algorithm

*Input.* A graph and a pair of nodes $m$ and $n$, which should be unified.

*Output.* True, if the nodes can be unified, False otherwise.

*Method.* A node is represented by the record $[\text{constructor, left, right, set}]$, where $\text{set}$ is the Set of equivalent nodes. A node of $\text{set}$ is chosen as representative of this set. In the beginning, each set contains only the node itself.

- $\text{find(n)}$ returns the representative node
- $\text{union(m,n)}$ merges the equivalence sets. The new representative node is a node which does not correspond with a variable. If there exists no such node, a former representative node is chosen as the new one.
function unify(m, n : node) : boolean begin
    s := find(m);
    t := find(n);
    if s = t then return true
    else if s and t are nodes that represent the same basic type then return true
    else if s is an op-node with children \( s_1, s_2 \) and
    t is an op-node with children \( t_1, t_2 \) then begin
        union(s, t);
        return unify(s_1, t_1) and unify(s_2, t_2) end
    else if s or t represents a variable then begin
        union(s, t)
        return true end
    else return false end
Example – Unification

Type expression:

\[(\alpha_1 \rightarrow \alpha_2) \times \text{list}(\alpha_3)) \rightarrow \text{list}(\alpha_2)\]

\[(\alpha_3 \rightarrow \alpha_4) \times \text{list}(\alpha_3)) \rightarrow \alpha_5\]

Question: \textit{unify}(1, 9) = ?
Objectives/Tasks

- Relate static source code to actions at program runtime.
- Names in the source code relate to (not necessarily the same) data objects on the target machine.
- Allocation and deallocation of data objects need to be managed (run-time support packages).
- procedure activation
- Store data objects accordingly to their data type.
Definitions

- Procedure definition = name + body
- Procedures with return value = functions
- Program = procedure (e.g. main)
- Procedure name in one location in the code = procedure call
- Variables (identifier) in procedure definition = formal parameters
- Arguments of a procedure call = actual parameters (substitute formal parameters after call)
program sort(input,output)
    var a: array[0..10] of integer;
procedure readarray;
    var i: integer;
    begin .... end;
procedure partition(y,z: integer) : integer;
    var i,j,x,v : integer;
    begin .... end;
procedure quicksort(m,n : integer);
    var i: integer;
    begin
        if (n>m) then begin
            i := partition(m,n); quicksort(m,i-1); quicksort(i+1,n)
        end
    end;
begin
    a[0]:=-9999; a[10]:=9999;
    readarray; quicksort(1,9);
end.
Activation Trees

Assumptions:
1. Sequential control flow
2. Procedure activation starts at the beginning of the body. After finishing the procedure, the statement located after the procedure call is executed.

Activation = execution of the body

Life time of a procedure = step sequence (time) from the first to the last step during the procedure execution.

Recursive procedure calls are possible (do not have to occur directly) \( P \rightarrow Q \rightarrow \ldots \rightarrow P \)
Activation Trees / Definition

1. Each node represents a procedure activation.
2. The root node represents the activation of the main program.
3. Node $a$ is the parent node of $b$ $\iff$ control flow: In $a$, $b$ is called (activated).
4. Node $a$ is left of node $b$ $\iff$. The lifetime of $a$ is ahead of the lifetime of $b$. 
Example Activation Tree
Control Stack

- Control flow = Depth processing (left to right) of the activation tree
- Control stack = stack to save all procedures at their lifetime
- Beispiel:

```
q(2,3) q(1,0) p(1,3)
q(1,3) p(1,9)
q(1,9) r
```

[Diagram showing the control stack and activation tree structure]
Explicit or implicit

```pascal
var i : integer;
```

scope of the variable given through *Scope Rules*

variables can be used within the scope

global vs. local variables

variables with the same name may denote different objects (due to their scope)

sequence of variable access (first local, then global variable if the name is identical,...)
Name Binding

- **Data object**: storage location, which can store a value

- **A variable (variable name)** can reference different data objects during runtime.

- **Program language semantics**:
  - **Environment**: function that maps names to storage locations
  - **State**: function that maps the storage locations to values

- Environment and State are different!!!!

**Example**: \( \pi \) is associated with the address 100, which stores the value 0. After \( \pi := 3.14 \) the storage location 100 has the value 3.14; \( \pi \), however, still points to 100.
A variable $x$ is bound to storage location $s$, if the storage location is associated with $x$.

Location does not always have to be a (real) storage location (in the main memory of the computer); e.g. complex datatypes

Correlation between STATIC and DYNAMIC notations:

<table>
<thead>
<tr>
<th>STATIC NOTATION</th>
<th>DYNAMIC COUNTERPART</th>
</tr>
</thead>
<tbody>
<tr>
<td>def. of a procedure</td>
<td>activation of a procedure</td>
</tr>
<tr>
<td>declaration of a name</td>
<td>bindings of the name</td>
</tr>
<tr>
<td>scope of a declaration</td>
<td>lifetime of a binding</td>
</tr>
</tbody>
</table>
Important Questions

... regarding organization of memory management and name binding.

1. Are there recursive procedures?
2. What happens with the values of local variables after finishing the procedure execution.
3. Can a procedure reference non-local variables?
4. How are parameters passed to a procedure?
5. Can procedures be passed as parameters?
6. Can procedures be returned as a return value?
7. Is there dynamic memory allocation?
8. Does memory have to be deallocated explicitly? (or does this happen implicitly (garbage collection)?)
Memory management

- Run time memory for:
  1. generated target code
  2. data objects
  3. control stack for procedure activation

- typical layout:

```
<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Data</td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Heap</td>
</tr>
</tbody>
</table>
```
Activation Record

...for information storage at a procedure call:

1. temporary values (evaluation of expressions,..)
2. local data (local variables,..)
3. machine data to save (program counter, registers,..)
4. access links (link to non-local data)
5. control link (link to activation record of the called procedure)
6. current parameters (usually stored in register)
7. return value of the called procedure
Compile-Time Layout

- Memory = blocks of Bytes, Byte = smallest addressable unit
- usually: 1 Byte = 8 Bit; n Bytes = Word
- Memory for a variable (or parameter) is dependant on the type. 
  Example: Basic Types (int, real, boolean,..) = n Bytes
- Storage Layout dependant on addressing: 
  Example:

  - **Aligned** Integers may only reside at certain addresses (addresses that are divisible by 4)
  - **Padding** 10 characters are necessary to save a string, but there must be 12 Bytes allocated.

- arrays, records are written to a memory range that is of sufficient size.
Memory Allocation Strategies

1. static allocation (at compile time)
2. stack allocation
3. heap allocation
Static Allocation

- Memory mapping is determined at compile time.
- Local values remain stored even after procedure termination.
- No run time support package necessary.
- Limitation:
  1. Size of the data structures must be known at compile time.
  2. Recursive procedures are only possible with restrictions (all recursive calls share the same memory!).
  3. Data structures can not be created dynamically.
Stack Allocation

**Idea:**
- Control Stack
  - procedure is activated $\rightarrow$ activation record is pushed to the stack
  - procedure activation terminates $\rightarrow$ activation record is popped from the stack
- local values are deleted after termination of the activation
Example

<table>
<thead>
<tr>
<th>Activation Tree</th>
<th>activation record</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>s</td>
<td>Frame for s</td>
</tr>
<tr>
<td></td>
<td>a : array</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>s</td>
<td>r is activated</td>
</tr>
<tr>
<td></td>
<td>a : array</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i : integer</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>s</td>
<td>Frame for r has been popped and q(1, 9) pushed</td>
</tr>
<tr>
<td></td>
<td>a : array</td>
<td></td>
</tr>
<tr>
<td></td>
<td>q(1, 9)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i : integer</td>
<td></td>
</tr>
</tbody>
</table>

s

\[ q(1,9) \]
<table>
<thead>
<tr>
<th>Activation Tree</th>
<th>activation record</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Activation Tree Diagram]</td>
<td>![Activation Record]</td>
<td>Control has just returned to $q(1,3)$</td>
</tr>
</tbody>
</table>

- **Activation Tree Diagram**
  - $s$ at the root
  - Branches from $s$ to $q(1,9)$
  - Branches from $q(1,9)$ to $p(1,9)$
  - Branches from $q(1,9)$ to $p(1,3)$
  - Branches from $p(1,9)$ to $q(1,3)$
  - Branches from $p(1,3)$ to $q(1,0)$

- **Activation Record**
  - $s$
  - $a : array$
  - $q(1,9)$
  - $i : integer$
  - $q(1,3)$
  - $i : integer$
Calling/Return Sequences

- **Calling Sequence**: allocate the activation records and fill in the fields
- **Return Sequence**: recover machine state
- Calling sequences do not necessarily equal activation record
  1. caller evaluates the current parameters
  2. return address, stack top are stored in the activation record of the calling procedure (callee).
  3. The callee stores the register values and other status information
  4. The callee initializes the local data and starts the execution.
possible return sequence:

1. the callee stores the return value
2. the stack, register information,.. are restored
3. the caller copies the return value into his activation record

task division:
Data with variable length

- storage not directly in the activation record
- a pointer to the data is stored
Dangling References

- Reference to memory is used but memory has already been deallocated.
- Logical programming error
- Cause of mysterious bugs
- Example:

```c
main() {
    int *p;
    p = dangle();
}

int *dangle() {
    int i = 23;
    return &i;
}
```
Heap Allocation

Necessary if:
1. value of local variable needs to be retained (after activation)
2. the called procedure survives the calling procedure

Memory is allocated and deallocated at request

Memory management is necessary:
1. linked list for storing free blocks
2. free sections should be filled in an optimal way
Access to nonlocal names

- lexical or static scope rules (declaration of names decided at compile time)
- static scope with most closely nested scopes
- dynamic scope rules (declaration of names decided at run time; activations are considered)
Blocks

- most closely nested:
  1. the scope of a declaration in block $B$ contains $B$.
  2. If the name $x$ in $B$ is not declared, an occurrence of $x$ in $B$ is in the scope of a declaration of $x$ in the surrounding block $B'$:
     1. $x$ is declared in $B'$.
     2. $B'$ is the closest immediate surrounding block of $B$ which declares $x$. 
Example – Scopes

```c
main() {
    int a = 0; \(B_0\)
    int b = 0;
    {
        int b = 1; \(B_1\)
        {
            int a = 2; \(B_2\)
        }
        {
            int b = 3; \(B_3\)
        }
    }
}
```
Memory Handling

1. Memory is provided via a stack. If a block is executed, memory for the local names is allocated. This memory will be deallocated after termination of the block.

2. Alternatively, it is possible to provide the memory for all blocks of a procedure at the same time. Memory can be decided at compile time (except if there is variable memory)
Global Data

- memory space can be allocated statically
- procedures can be passed as parameter (C: pointer)
- Example:

```pascal
program pass(input, output);

var m : integer;

function f(n: integer) : integer;
begin f := m + n end { f };

function g(n: integer) : integer;
begin g := m * n end { g };

procedure b(function h(n: integer) : integer);
begin write(h(2)) end { b };

begin
  m := 0;
  b(f); b(g); writeln
end.
```
Nested Procedures

- procedure definitions in procedures
- Example:

  ```pascal
  program sort(input, output)
  ...
  procedure exchange(i, j: integer);
  ...
  procedure quicksort(m, n: integer);
  var k, v: integer;
  begin
    ...
    exchange(i, j);
    ...
  ```
Nesting Depth, Access Lists

- **Nesting Depth**: depth of the nesting of procedures/blocks... (programs: 1, procedure in program: 2, ...)
- **Access List**: implementation of the access to nested procedures
  - *access link*: field in the activation record of a procedure
  - If \( P \) is declared in \( Q \), the access link of \( P \) point to the access link of \( Q \) (in the last activation of \( Q \))
Example – Access Lists

\[
\begin{align*}
s & \quad \text{access link} \\
& \quad a, x \\
& \quad q(1, 9) \\
& \quad \text{access link} \\
& \quad k, v \\
\end{align*}
\]

F. Wotawa (IST @ TU Graz)
Search for non-local names

- Procedure $P$ is in nesting depth $n_p$ and accesses $a$ with $n_a \leq n_p$.
  1. $P$ is being executed. The activation record of $P$ is located at the top of the stack. Walk along $n_p - n_a$ access links.
  2. Subsequently, we reach the activation record, which contains $a$. An offset value returns the actual position of $a$.

- $(n_p - n_a, offset)$ defines the address of $a$. Computation can be done at compile time.
Procedure Access (nested)

- $P$ calls procedure $X$.
  1. $n_P < n_X$: $X$ lies lower than $P$. Therefore, $X$ must be defined in $P$ (or $X$ cannot be accessed from $P$). The access link of $X$ points to $P$.
  2. $n_P \geq n_X$: Walk along $n_P - n_X + 1$ access links. This activation record contains $P$ as well as $X$. 
Procedure Parameter

- passing a procedure as parameter
- not allowed in all languages
- handling of links (similar to already described method):
  - assuming \(c\) calls \(b\) and passes \(f\) as parameter
  - a link from \(f\) to \(c\) is computed
  - this link is used as access link when \(f\) is actually called.
Dynamic Scope

- New activation uses existing binding of non-local names in their memory. A in the called activation references the same memory like the calling activation. New bindings are provided for local names of the called procedure.

- Semantics of static scope and dynamic scope are different!
program dynamic (input, output);
  var r: real;
procedure show;
  begin write(r : 5:3) end;
procedure small;
  var r : real;
  begin r := 0.125; show end;
begin
  r := 0.25;
  show; small; writeln;
  show; small; writeln;
end
Example – Dynamic Scope (2)
Example – Dynamic Scope (3)

- **Example:**
  - Static Scope: Output = 0.250 0.250 nl 0.250 0.250 nl
  - Dynamic Scope: Output = 0.250 0.125 nl 0.250 0.125 nl

- **Deep access:** Control links are used as access links. Search in the stack (from top to bottom) for the first entry of a non-local name.

- **Shallow access:** Current value of a name is deposited in a (statically) allocated location. At an activation of \( P \), local name \( n \) uses the location. The old value of the location can be cached in the activation record and is therefore restorable.
Parameter Passing

How are parameters passed at a call?

```pascal
procedure exchange(i, j: integer);
    var x: integer;
    begin
        x := a[i]; a[i] := a[j]; a[j] := x
    end
```

- Call-by-Value
- Call-by-Reference
- Copy-Restore
- Call-by-Name
Call-by-Value

- Formal parameters are considered as local names.
- The caller evaluates the actual parameter and passes them the associated formal parameters.
- Pointers can also be passed as values.
Call-by-Reference

- Instead of the value (as in Call-by-Value) a pointer to the memory location of the actual parameter is passed.
- `var` parameter in PASCAL are references.
- Arrays are usually passed as reference.
Copy-Restore

- Hybrid method between Call-by-Value and Call-by-Reference.
- Method:
  1. Before executing the procedure, the actual parameters are evaluated. The R-Values (values) are passed to the respective formal parameters (as Call-By-Value). Additionally, the L-Values (locations) are computed.
  2. After procedure termination, the actual R-Values are copied back to the L-Values of the actual parameters (if available).
Call-by-Name

- Procedures are treated as macros, i.e. instead of the procedure call, the body of the procedure is substituted; all formal parameters in the body are replaced by the actual parameters. (Macro-Expansion)

- The local names of the called procedures must be different to the name of the calling procedure. (Variable renaming is partially necessary)

- The actual parameters are put in braces to avoid problems.

Problems: Call $\text{swap}(i, a[i])$ is expanded to:
\[
temp := i; \ i := a[i]; \ a[i] := \text{temp}
\]
Instead of $a[i]=i$ we write $a[a[i]]=i$. $\text{temp} := x; \ x := y; \ y := \text{temp}$
Symbol Table

- entries correspond to the declaration of names
- store binding and scope information
- storage allocation information (that is needed at run time)
- storage of the symbol table in a list
- storage of the symbol table in a hash table
Objectives/Tasks

- provide a target machine independent format
- advantages:
  - easy adaptation on different target machines
  - machine independent code optimization can be realized
- attributed grammars can be used
Languages

- graphical representation (syntax tree)
- **Three-Address Code**
  \[
  x := y \ op \ z
  \]
  - \(x, y, z\) are arbitrary numbers, constants, names (variables), or temporary variables
  - \(\text{op}\) is an operator
Three-Address Code Statements

1. assignments $x := y \ op \ t$
2. assignments with unary operator $x := \ op \ y$
3. copy statements $x := y$
4. unconditional jumps $\text{goto } L$
5. conditional jumps $\text{if } x \ relop \ y \ \text{goto } L$
6. param $x$, call $p$, $n$ calls procedure $p$ with $n$ parameters. return $y$ where $y$ is optional.
8. addresses and pointer assignments: $x := & y$, $x := *y$ and $*x := y$
Generating the Three-Address Code (simplified)

- **S-attributed grammar**
- \( S.code \) represents the three address code
- \( E.place \) the name which contains the value of the non-terminal \( E \).
- \( E.code \) the sequence of three address code statements, that evaluate \( E \).
Assignments

\[
\begin{align*}
S & \rightarrow \text{id} := E & S.code := E.code \mid gen(\text{id}.place \ ' := \ E.place) \\
E & \rightarrow E_1 + E_2 & E.place := \text{newtemp}; \\
& & E.code := E_1.code \mid E_2.code \mid gen(E.place \ ' := \ E_1.place \ ' + \ ' E_2.place) \\
E & \rightarrow E_1 \times E_2 & E.place := \text{newtemp}; \\
& & E.code := E_1.code \mid E_2.code \mid gen(E.place \ ' := \ E_1.place \ ' * \ ' E_2.place) \\
E & \rightarrow - E_1 & E.place := \text{newtemp}; \\
& & E.code := E_1.code \mid gen(E.place \ ' := \ ' \text{uminus} \ ' E_1.place) \\
E & \rightarrow ( E_1 ) & E.place := E_1.place; E.code := E_1.code \\
E & \rightarrow \text{id} & E.place := \text{id}.place; E.code := \text{"} \\
\end{align*}
\]
If-then-else

Statement: \( S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \)

Generated Code:

\[
S\text{.else} := \text{newlabel}; \\
S\text{.after} := \text{newlabel}; \\
S\text{.code} := \begin{cases}
E\text{.code} &|\|
\text{gen( 'id' } E\text{.place } '=' '0' \text{ 'goto' } S\text{.else}) &|\|
S_1\text{.code} &|\|
\text{gen( 'goto' } S\text{.after}) &|\|
\text{gen(S\text{.else '':')}} &|\|
S_2\text{.code} &|\|
\text{gen(S\text{.after '':')}}
\end{cases}
\]
While-loops

- **Statement:** $S \rightarrow \textbf{while } E \textbf{ do } S_1$

- **Generated Code:**

  \[
  S.\text{begin} := \text{newlabel}; \\
  S.\text{after} := \text{newlabel}; \\
  S.\text{code} := \begin{cases} \\
    \text{gen}(S.\text{begin } ':') || \\
    E.\text{code} || \\
    \text{gen( 'if' } E.\text{place } '=' '0' 'goto' S.\text{after} ) || \\
    S_1.\text{code} || \\
    \text{gen( 'goto' } S.\text{begin} ) || \\
    \text{gen}(S.\text{after } ':') \\
  \end{cases}
  \]
Implementation of the Three-Address Code

- **Quadruple** = record with 4 fields:
  - $op$  
  - $arg_1$  
  - $arg_2$  
  - $result$

  arguments and temporary variables are usually pointer to symbol table entries

- **Triple** = “Quadruple” without result field. Instead, the position of the triple that calculates a value is stored in the argument.
**Examples**

\[ a := b \times -c + b \times -c \]

<table>
<thead>
<tr>
<th>Quadruple</th>
<th>Triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \begin{array}{</td>
<td>c</td>
</tr>
</tbody>
</table>
provision of the memory space for local names of a procedure (relative addresses of the activation record or memory of the static data area)

**procedure declarations:**

1. offset . . . next free relative address
2. initialization offset = 0
3. offset is used for the current data object
4. then, the offset is increased by the size of the current data object

**enter(name, type, offset)** creates an entry in the symbol table for name, assigns it the type type and offset as relative address.
Attributed Grammar for Declarations

\[
P \to \{ \text{offset} := 0 \}\ D
\]
\[
D \to D \ ; \ D
\]
\[
D \to \text{id : } T \{ \text{enter( id.name, T.type, offset) }
\quad \text{offset := offset + T.width} \}
\]
\[
T \to \text{integer}\{T.type := \text{integer}; T.width := 4\}
\]
\[
T \to \text{real}\{T.type := \text{real}; T.width := 8\}
\]
\[
T \to \text{array \ [ \text{num} \ ] of } T_1\{T.type := \text{array(num.val, T}_1\text{.type); }
\quad \text{T.width := num.val \times T}_1\text{.width}\}
\]
\[
T \to^{↑} T_1\{T.type := \text{pointer(T}_1\text{.type); T.width := 4}\}
\]
**Scope Information**

- each procedure has its own symbol table
- for every procedure declaration a symbol table is created
- + a link to the symbol table of the enclosing procedure
- offset is now local!
- example grammar:
  \[
  P \rightarrow D \\
  D \rightarrow D ; D | \text{id : } T | \text{proc id ; } D ; S
  \]
- grammar definition... (Exercise)
create symbol table for fields (marker \( L \)).

names are stored in the new symbol table

grammar:

\[
T \rightarrow \mathbf{record \ LD \ end} \quad \{ \begin{align*}
T\.type & := \text{record}(\text{top}(\text{tblptr})); \\
T\.width & := \text{top}(\text{offset}) \\
\text{pop}(\text{tblptr}); \text{pop}(\text{offset})
\end{align*}\}
\]

\[
L \rightarrow \epsilon \quad \{ \begin{align*}
t & := \text{maketable}(\text{nil}); \\
\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset})
\end{align*}\}
Assignments

- assumption so far: names are represented by oneself
- correct if name for pointer is in their symbol table
- generalization by attribute: \textit{name} for identifier
- \textit{lookup(\textbf{id}.name)} results in the entry
- advantage: usable even if the entry is declared in an enclosing procedure
- By defining \textit{lookup}, the scope of a language regarding an identifiers is defined.
Grammar

\[
S \rightarrow \text{id} := E \\
E \rightarrow E_1 + E_2 \\
E \rightarrow E_1 \ast E_2 \\
E \rightarrow -E_1 \\
E \rightarrow (E_1) \\
E \rightarrow \text{id}
\]

\[
p := \text{lookup}(\text{id}.name);
\]

\[
\text{if } p \neq \text{nil} \text{ then}
\]

\[
\text{emit}(p' :=' E\text{.place})
\]

\[
\text{else error}
\]

\[
E \rightarrow E_1 + E_2
\]

\[
E\text{.place} := \text{newtemp};
\]

\[
\text{emit}(E\text{.place'} :=' E_1\text{.place' +'} E_2\text{.place})
\]

\[
E \rightarrow E_1 \ast E_2
\]

\[
E\text{.place} := \text{newtemp};
\]

\[
\text{emit}(E\text{.place'} :=' E_1\text{.place' \ast'} E_2\text{.place})
\]

\[
E \rightarrow -E_1
\]

\[
E\text{.place} := \text{newtemp};
\]

\[
\text{emit}(E\text{.place'} :=' \text{'uminus'} E_1\text{.place})
\]

\[
E \rightarrow (E_1)
\]

\[
E\text{.place} := E_1\text{.place}
\]

\[
E\text{.place} := \text{newtemp}
\]

\[
E \rightarrow \text{id}
\]

\[
p := \text{lookup}(\text{id}.name);
\]

\[
\text{if } p \neq \text{nil} \text{ then}
\]

\[
E\text{.place} := p
\]

\[
\text{else error}
\]
Addressing Array Elements

- array access fast, if the elements are stored in one block.
- access to the element at position \( i \) (\( w \) ... element size):

\[
base + (i - low) \times w
\]

can be rewritten to:

\[
i \times w + (base - low \times w)
\]

advantage: \( base - low \times w = c \) can be calculated at compile time!

- two dimensional arrays \( A(i_1, i_2) \):
  - row major (row-by-row) \( base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w \)
    where \( n_2 = high_2 - low_2 + 1 \).
  - column major (column-by-column)
Translation Schema for Array Access (1)

1. \[ S \rightarrow L := E \]

   \[ \text{if } L.offset = \text{null then} \]
   \[ \text{emit}(L.place' := E.place); \]

   \[ \text{else} \]
   \[ \text{emit}(L.place'[L.offset'] := E.place) \]

2. \[ E \rightarrow E_1 + E_2 \]

   \[ E.place := \text{newtemp} \]
   \[ \text{emit}(E.place' := E_1.place' + E_2.place) \]

3. \[ E \rightarrow (E_1) \]

   \[ E.place := E_1.place \]
Translation Schema (2)

1. \( E \rightarrow L \)
   
   \( \text{if } L.\text{offset} = \text{null} \text{ then} \)
   
   \( E.\text{place} := L.\text{place} \)
   
   \( \text{else} \)
   
   \( E.\text{place} := \text{newtemp} \)
   
   \( \text{emit}(E.\text{place } := L.\text{place } [L.\text{offset}]) \)

2. \( L \rightarrow \text{Elist} \)
   
   \( L.\text{place} := \text{newtemp} \)
   
   \( L.\text{offset} := \text{newtemp} \)
   
   \( \text{emit}(L.\text{place } := c(\text{Elist.array})) \)
   
   \( \text{emit}(L.\text{offset } := \text{Elist.place } \ast \text{width(\text{Elist.array})}) \)

3. \( L \rightarrow \text{id} \)
   
   \( L.\text{place} := \text{id.place}; L.\text{offset} := \text{null} \)
1. $Elist \rightarrow Elist_1, E$

   \begin{align*}
   t & := \text{newtemp} \\
   m & := Elist_1.\text{ndim} + 1 \\
   \text{emit}(t '\, :=' Elist_1.\text{place} ',*' \text{limit}(Elist_1.\text{array}, m)) \\
   \text{emit}(t '\, :=' t '+' E.\text{place}) \\
   Elist.\text{array} & := Elist_1.\text{array} \\
   Elist.\text{ndim} & := m
   \end{align*}

2. $Elist \rightarrow \text{id [ E}$

   \begin{align*}
   Elist.\text{array} & := \text{id.\text{place}} \\
   Elist.\text{place} & := E.\text{place} \\
   Elist.\text{ndim} & := 1
   \end{align*}
Boolean Expressions

2 main tasks:
1. calculating logical values
2. changing program procedure

grammar:

\[ E \to E \text{ or } E | E \text{ and } E | \text{ not } E | ( E ) | \text{id relop id} | \text{true} | \text{false} \]

2 methods to represent boolean values:
1. true and false are coded as numbers (e.g. true = 1, false = 0).
2. Flow-of-Control: values are represented as positions in the code
Example 1: \( a \lor (b \land (\neg c)) \)

\[
\begin{align*}
t_1 &:= \neg c \\
t_2 &:= b \land t_1 \\
t_3 &:= a \lor t_2
\end{align*}
\]

Example 2: \( a < b \)

100: if \( a < b \) then goto 103
101: \( t := 0 \)
102: goto 104
103: \( t := 1 \)
104:
Translation Schema (Bool. Expr.) I

\[ E \rightarrow E_1 \text{ or } E_2 \]
\[ E \rightarrow E_1 \text{ and } E_2 \]
\[ E \rightarrow \text{not } E_1 \]
\[ E \rightarrow \text{id}_1 \text{ relop id}_2 \]

\[ E \rightarrow \text{true} \]
\[ E \rightarrow \text{false} \]

\[ E \rightarrow E_1 \text{ or } E_2 \quad E\.place := \text{newtemp}; \text{emit}(E\.place := \text{'or'} E_1\.place \text{ or } E_2\.place) \]
\[ E \rightarrow E_1 \text{ and } E_2 \quad E\.place := \text{newtemp}; \text{emit}(E\.place := \text{'and'} E_1\.place \text{ and } E_2\.place) \]
\[ E \rightarrow \text{not } E_1 \quad \]
\[ E \rightarrow \text{id}_1 \text{ relop id}_2 \quad E\.place := \text{newtemp} \]
\[ \text{emit('if' id}_1\.place \text{ relop.opid}_2\.place \text{'goto' nextstat + 3}) \]
\[ \text{emit(E\.place := '0')} \]
\[ \text{emit('goto' nextstat + 2)} \]
\[ \text{emit(E\.place := '1')} \]
\[ E \rightarrow \text{true} \quad E\.place := \text{newtemp}; \text{emit(E\.place := '1')} \]
\[ E \rightarrow \text{false} \quad E\.place := \text{newtemp}; \text{emit(E\.place := '0')} \]
Short-Circuit Code

- Representation of the boolean expressions without generating code for operators \texttt{and, or, not}
- values are represented by positions in the code

\textit{Jumping Code}

Example: \(a < b \lor c < d \land e < f\)

\begin{align*}
100: & \text{ if } a < b \text{ goto } 103 & 107*: & t2 := 1 \\
101*: & t1 := 0 & 108: & \text{ if } e < f \text{ goto } 111 \\
102: & \text{ goto } 104 & 109*: & t3 := 0 \\
103*: & t1 := 1 & 110: & \text{ goto } 112 \\
104: & \text{ if } c < d \text{ goto } 107 & 111*: & t3 := 1 \\
105*: & t2 := 0 & 112*: & t4 := t2 \text{ and } t3 \\
106: & \text{ goto } 108 & 113*: & t5 := t1 \text{ or } t4
\end{align*}
Flow-of-Control Statements

- Statements:
  
  - if \( E \) then \( S_1 \)
  
  \[ S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \]
  
  - while \( E \) do \( S_1 \)

- Use labels to represent true and false
- Dependent on the evaluation of \( E \), branch out.
- Attributed grammar (see above)
Use \( E.\text{true} \) (\( E.\text{false} \)) if \( E \) evaluates to true.

Example \( E_1 \text{ or } E_2 \) is true, if \( E_1 \) is true.

not all expressions are evaluated (like e.g. in C)

Example: \( a < b \text{ or } (c < d \text{ and } e < f) \)

```
if a < b goto Ltrue
  goto L1
L1:  if c < d goto L2
      goto Lfalse
L2:  if e < f goto Ltrue
      goto Lfalse
```
Translation Schema (Bool. Expr.) II

\[ E \rightarrow E_1 \text{ or } E_2 \]
\[ E_1.true := E.true; E_1.false := \text{newlabel}; E_2.true := E.true \]
\[ E_2.false := E.false; E.code := E_1.code || \text{gen}(E_1.false):' || E_2.code \]

\[ E \rightarrow E_1 \text{ and } E_2 \]
\[ E_1.true := \text{newlabel}; E_1.false := E.false; E_2.true := E.true \]
\[ E_2.false := E.false; E.code := E_1.code || \text{gen}(E_1.true):' || E_2.code \]

\[ E \rightarrow \text{not } E_1 \]
\[ E_1.true := E.false; E_1.false := E.true; E.code := E_1.code \]

\[ E \rightarrow \text{id}_1 \text{ relop } \text{id}_2 \]
\[ E.code := \begin{cases} 
\text{gen} \left( \text{if' } \text{id}_1.place \text{ relop } \text{op' } \text{id}_2.place \text{'goto' } E.true \right) || \\
\text{gen}'\text{goto' } E.false
\end{cases} \]

\[ E \rightarrow \text{true} \]
\[ E.code = \text{'goto' } E.true \]

\[ E \rightarrow \text{false} \]
\[ E.code = \text{'goto' } E.false \]
Mixed Mode

- consideration (so far) simplified
- in practice, mixed expressions are possible
- Example 1: \((a + b) < c\)
- Example 2: \((a < b) + (b < a)\)
- introduce synthetic attribute \(E.type\)
  \[
  E.type = \begin{cases} 
  \text{arith} & \text{Arithmetic expression} \\
  \text{bool} & \text{Boolean expression} 
  \end{cases}
  \]
- Code Generation for \(E + E, E \times E, \ldots\) needs to be changed.
Back patching

- easiest implementation of attributed grammars:
  1. generate a syntax tree
  2. generate the translation depth-first
- problem with Single Pass:
  labels for control flow are unknown
- trouble-shooting:
  1. jump statements are generated with empty labels
  2. these statements are saved in a list
  3. the target labels are registered once they are known
Procedure Calls

- usage of run time routines for handling the parameters, the call itself and the return of values.

- grammar:

  \[
  S \rightarrow \text{call id} \ (Elist) \\
  Elist \rightarrow Elist, E | E
  \]

- Calling Sequence must be reproduced.
Attributed Grammar (simplified)

- Call-by-Reference
- Memory is statically allocated
- Grammar:

1. \[ S \rightarrow \text{call id } (Elist) \]
   
   \[
   \{ \begin{align*}
   &\text{for each item } p \text{ on queue do} \\
   &\quad \text{emit('param' } p) \\
   &\quad \text{emit('call' id.place)}
   \end{align*}
   \]

2. \[ Elist \rightarrow Elist, E \]
   
   Append \( E\).place to the end of queue

3. \[ Elist \rightarrow E \]
   
   Initialize queue to contain only \( E\).place
Objectives/Tasks

- Generate machine code (from the intermediate code)
  - Inputs:
    - intermediate code
    - symbol table
  - possible output:
    - absolute machine code
    - relocatable machine code
    - assembler code
**Problem – Instruction Selection**

- **easiest method:** each three-address code statement is assigned instructions (code skeleton) $x := y + z$
  is represented by:

  ```plaintext
  MOV y, R0
  ADD z, R0
  MOV R0, x
  ```

- **Issue: not efficient**

  ```plaintext
  MOV b, R0
  ADD c, R0
  a := b + c
  MOV R0, a
  d := a + e
  MOV a, R0  not necessary
  ADD e, R0
  MOV R0, d
  ```
quality of the generated code depends on (1) speed and (2) size

depends on the target machine

There are often several possible instructions to execute a command

Example: \( a := a + 1 \)

via increment command:

- MOV a, R0
- INC R0
- MOV R0, a

via addition:

- MOV a, R0
- ADD # 1, R0
- MOV R0, a
Register Allocation

- register operations are usually fast
- limited number of registers available
- during register allocation, variables are selected to be stored in the register
- during register assignment, variables are assigned to registers
- optimal assignment of registers to variables is NP-complete! (even under realistic assumptions)
- register assignment partially depends on the target machine
The order of the target code evaluation has an effect on efficiency. It is possible that fewer registers are needed. Finding the optimal order is also NP-complete.

Kinds of Code Generation
- direct code generation + optimization
- use flow-of-control to create better code
- tree-directed code-selection techniques (Tree-rewriting)
Assumptions: byte-addressable machine with 4 Bytes for a Word and \( n \) registers \( R_0, \ldots R_{n-1} \)

Instructions are of the form: \( \text{opsource, destination} \)

Examples: MOV, ADD, SUB, \( \ldots \)

Address Modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Form</th>
<th>Address</th>
<th>Added Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>( M )</td>
<td>( M )</td>
<td>1</td>
</tr>
<tr>
<td>register</td>
<td>( R )</td>
<td>( R )</td>
<td>0</td>
</tr>
<tr>
<td>indexed</td>
<td>( c(R) )</td>
<td>( c + \text{contents}(R) )</td>
<td>1</td>
</tr>
<tr>
<td>indirect register</td>
<td>( *R )</td>
<td>( \text{contents}(R) )</td>
<td>0</td>
</tr>
<tr>
<td>indirect indexed</td>
<td>( *c(R) )</td>
<td>( \text{contents}(c + \text{contents}(R)) )</td>
<td>1</td>
</tr>
</tbody>
</table>
Examples

- MOV R0, M stores the content of R0 in location M.
- MOV 4(R0), M stores the content of 4 + contents(R0) in location M.
- MOV *4(R0), M stores the value of contents(4 + contents(R0)) in M.
- Further Address-Mode: MOV # 1, R0 stores the constant 1 in register R0.
Instruction Costs

- costs of an instruction = 1 + costs associated with source- and destination addresses.
- costs correspond to the length of the instructions
- minimizing the costs minimizes the size of the instructions and the execution time (register operations are usually faster)

Example

\[ a := b + c \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV b, R0</td>
<td>6</td>
</tr>
<tr>
<td>ADD c, R0</td>
<td>6</td>
</tr>
<tr>
<td>MOV R0, a</td>
<td>6</td>
</tr>
<tr>
<td>MOV b, a</td>
<td>6</td>
</tr>
<tr>
<td>ADD c, a</td>
<td>6</td>
</tr>
<tr>
<td>MOV *R1, *R0</td>
<td>2</td>
</tr>
<tr>
<td>ADD *R2, *R0</td>
<td>3</td>
</tr>
<tr>
<td>ADD R2, R1</td>
<td>3</td>
</tr>
<tr>
<td>MOV R1, a</td>
<td>3</td>
</tr>
</tbody>
</table>
Run-time Storage Management

- handling of the memory management
- which code needs to be created for the management of the activation records
- 2 strategies:
  - Static Allocation
  - Stack Allocation
- Three-Address-Statements: call, return, halt, action
Static Allocation

- Activation Record is created at compile time
- Caller-Code:
  - MOV # here + 20, callee.static_area
  - GOTO callee.code_area
- Callee-Code (after finishing)
  - GOTO *callee.static_area
Stack Allocation

code for the first procedure

MOV # stackstart, SP

date for the 1. procedure

HALT

Procedure call

ADD # caller.recordsize, SP

MOV # here + 16, *SP

GOTO callee.code_area

Return Sequence

at the Callee:

GOTO *0(SP)

at the Caller:

SUB # caller.recordsize, SP
Flow Graphs

Basic Block

sequence of statements that don’t contain a halt or branching statement (except at the end).

- \( x := y + z \)
- \( x \) is defined
- \( y, z \) are used (referenced)

Algorithm:

1. Determine the set of leaders, i.e., the first statement of a basic block.
   - The first statement is the leader
   - Any statement that is the target of a conditional or unconditional goto is a leader.
   - Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader an all statements up to but not including the next leader or the end of the program.
Example – Basic Blocks

begin
prod := 0;
i := 1;
do begin
    prod := prod + a[i]*b[i];
i := i+1
end while i<=20
end

(1) prod := 0
(2) i := 1
(3) t1 := 4*i
(4) t2 := a[t1]
(5) t3 := 4*i
(6) t4 := b[t3]
(7) t5 := t2*t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
Transformations of Basic Blocks

- Basic Blocks calculate a set of expressions
- Two blocks are equal if they calculate the same set of expressions
- Transformations may not change this set of expressions
- 2 kinds of transformations:
  - Structure-Preserving Transformations
  - Algebraic Transformations
Structure-Preserving Transformations

- **Common subexpression elimination**
  \[a := b + c\]
  \[b := a - d\]
  \[c := b + c\]
  \[d := a - d\]

- **dead-code elimination**
  Assuming \(x\) is defined by \(x := y + z\) but not used afterwards, the statement can be removed.

- **renaming temporary variables**

- **interchange of statements**
Algebraic Transformations

- there are different transformations
- goal is usually optimization of the code

**Example 1**: The following statements can be eliminated

\[
\begin{align*}
x & := x + 0 \\
x & := x \times 1
\end{align*}
\]

**Example 2**

\[
\begin{align*}
x & := y \times 2 
\rightarrow x & := y \times y
\end{align*}
\]
Flow Graphs

- directed graph
- nodes are Basic Blocks
- two nodes $n_1$ and $n_2$ are connected if the block of $n_2$ follows directly after $n_1$ in an execution. This means
  1. There is a conditional and unconditional jump from the last statement of the block of $n_1$ to the first statement of the block of $n_2$.
  2. The block of $n_2$ follows the block of $n_1$ directly and the block of $n_1$ has no unconditional jump in the last statement.

Finding all loops is not always easy. A loop is a set of nodes in the flow graph with:
  1. all nodes in the set are strongly connected
  2. the set of nodes has a distinct start node

A loop that does not contain any other loops is called Inner Loop.
Next-Use Information

- If a name in a register is no longer needed, the register can be used for another name again.
- Used at register allocation and assignment of memory location to temporary names.
- Definition **Use**: Assuming \( x \) is assigned a value in statement \( i \). The statement \( j \) uses \( x \) as an operand and there is a control flow of \( i \) to \( j \), where no other statement redefines \( x \) in this path. We then say that \( j \) USES the value of \( x \), which was defined at location \( i \).
Calculation of Next-Use

- in the Basic Block (procedure calls also start Basic Blocks)
- Algorithm:
  1. From the end of a basic block to its first statement for each statement $i: x := y \text{ op } z$:
     1. Attach to $i$ the information currently found in the symbol table regarding the next use and liveness of $x, y$, and $z$.
     2. In the symbol table, set $x$ to “not live” and “no next use”.
     3. In the symbol table, set $y$ and $z$ to “live” and their next uses to $i$. The order of steps may not be interchanged!
A simple Code Generator

- **Register-Descriptor**: shows which names are currently stored in which registers. In the initial phase, all registers are empty.
- **Address-Descriptor**: points to the location of the current value of a name at runtime. This location is either a register, a stack location, a memory location or a combination.
- **Assumptions**: for each operator there is a target machine operator, information about registers and their assignment are used.
Algorithm Code-Generator

For each statement \( x := y \ op z \) perform the following actions:

1. Invoke a function \( \text{getreg} \) to determine the location \( L \) where the result of \( y \ op z \) should be stored.

2. Consult the address descriptor for \( y \) to determine \( y' \) (one of) the current location(s) of \( y \). Registers are preferred. If the value is not already in \( L \), then generate a statement \( \text{MOV} \) \( y' \), \( L \).

3. Generate the instruction \( \text{OP} \ z' \), \( L \), where \( z' \) is the current location of \( z \). Update the address descriptor of \( x \) to indicate that \( x \) is currently in location \( L \). If \( L \) is a register, update its descriptor and remove \( x \) from all register descriptors.

4. If the current values of \( y \) and/or \( z \) have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of \( x := y \ op z \), those registers no longer will contain \( y \) and/or \( z \), respectively.
Special Cases

- Statements of the form $x := \text{op} \ y$ are converted the same way.

- Statements of the form $x := y$ are handled as follows:
  - If $y$ is in a register, simply change the register and address descriptors to record that the value of $x$ is now found only in the register holding the value of $y$.
  - If $y$ has no next use and is not live on exit from the block, the register no longer holds the value of $y$.
  - If $y$ is in memory, we use $\text{getreg}$ to determine a new register in which to load $y$ and make that register the location of $x$.
  - Alternatively, the statement $\text{MOV} \ y, x$ can be used.

- After the handling of all statements, all names that are live on exist and not yet in their memory location are saved via $\text{MOV}$. 
**Function \textit{getreg}**

Statement $x := y \; \text{op} \; z$.

1. If $y$ is in a register that holds the value of no other name, and $y$ is not live and has no next use after executing the statement, then return the register of $y$ to be $L$. Update the address descriptor of $y$.

2. Failing (1), return an empty register for $L$ if there is one.

3. Failing (2), if $x$ has a next use in the block or $\text{op}$ is an operator that requires a register, find an occupied register $R$. Store the value of $R$ into a memory location, update the address descriptors, and return $R$. If $R$ holds the value of several variables, a move operation for all variables has to be generated.

4. If $x$ is not used in the block, or no suitable occupied register can be found, select the memory location of $x$ as $L$. 
Handling Conditionals

2 Kinds (depend on the target machine)

1. Branches depend of the value of a special register (negative, zero, positive, nonnegative, nonzero, and nonpositive). For such machines, if \( x < y \) goto \( z \) can be implemented by subtracting \( y \) of \( x \) and jumping if the value is negative.

2. Condition Codes: this code shows if a value that has been calculated or loaded into a register is negative, null or positive. Usually there is a comparison operation \( \text{CMP} \ x, y \), which sets the condition code to positive if \( x > y \). Additional command \( \text{CJ} r \ z \) with \( r \in \{>, <, =, \leq, \geq, \neq\} \) jumps to the label \( z \) if the condition code \( >, <, \ldots \) is null. Example:

\[
\text{CMP} \ x, y \\
\text{CJ}< \ z
\]
Peephole Optimization

- uses only a small sequence of statements
- moving window for target code (=peephole)
- this should be replaced by a smaller and faster sequence.
- several executions possible (and preferable).

**Redundant Loads and Stores**

1. MOV R0, a
2. MOV a, R0

Statement (2) can be removed if there is no label to (2)!
Unreachable Code

```c
#define debug 0
...
if ( debug ) {
    ...
}
```

Flow-of-Control Optimizations

```c
goto L1
...
L1: goto L2
→
goto L2
L1: goto L2
```

Algebraic Simplifications

```
x := x + 0
```

Reduction in Strength

\[ x^2 \] is implemented faster than \( x \times x \). Shift-Operation can also be used for division or multiplication with \( 2^n \).

Use of Machine Idioms

Instead of \( i := i+1 \) the increment command can be used.
Objectives/Tasks

- “Optimization” of the generated (intermediate) code (faster, smaller)
- Tradeoff between effort and usage of the optimization
  Most payoff for the least effort
- Focus: target machine independent code optimization
- Optimization independent of the input values!
- Objective: better average results
- Techniques: data and control flow analysis.
Criteria for Code Transformation

- The Behavior of the program must be preserved.
- The transformed program must be faster (smaller) on average.
- The analysis must be worth the effort. Meaning a program analysis, which leads to a better transformation but needs a lot of time is possibly not an option.
Dramatic increase of the runtime (from hours to a few seconds) is only possible, if the program is corrected on all levels:

- better algorithms and data structures (e.g. replace Insert Sort with Quicksort \((N \log N \text{ instead of } N^2)\))
- code transformations
- Utilization of registers and faster (machine language) instructions, or peephole transformations
Organization Compiler optimization

- **Organization:**
  
  ![Diagram showing the organization of compiler optimization]

- **Advantages:**
  
  - Operations for high-level constructs are explicitly available in the intermediate code and can be therefore be optimized.
  - The intermediate code is (relatively) independent of the target machine. Optimizations are possible for many machines (without any changes).
Example–Quicksort

```c
void quicksort(int m, int n) {
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    } /* fragments ends here */
    quicksort(m, j); qicksort(i+1, n);
}
```
Three Address Code – Quicksort

(1)  \( i := m-1 \)
(2)  \( j := n \)
(3)  \( t1 := 4*n \)
(4)  \( v := a[t1] \)
(5)  \( i := i+1 \)
(6)  \( t2 := 4*i \)
(7)  \( t3 := a[t2] \)
(8)  \( \text{if } t3 < v \text{ goto (5)} \)
(9)  \( j := j-1 \)
(10) \( t4 := 4*j \)
(11) \( t5 := a[t4] \)
(12) \( \text{if } t5 > v \text{ goto (9)} \)
(13) \( \text{if } i >= j \text{ goto (23)} \)
(14) \( t6 := 4*i \)
(15) \( x := a[t6] \)
(16) \( t7 := 4*i \)
(17) \( t8 := 4*j \)
(18) \( t9 := a[t8] \)
(19) \( a[t7] := t9 \)
(20) \( t10 := 4*j \)
(21) \( a[10] := x \)
(22) \( \text{goto (5)} \)
(23) \( t11 := 4 * i \)
(24) \( x := a[t11] \)
(25) \( t12 := 4 * j \)
(26) \( t13 := 4*n \)
(27) \( t14 := a[t13] \)
(28) \( a[t12] := t14 \)
(29) \( t15 := 4*n \)
(30) \( a[t15] := x \)
Possibilities of Optimization

- **Common Subexpression**: An Expression $E$, which has been calculated before and which value has not been changed in the meantime. Multiple calculation of expressions can be avoided.

  **Example**: Block B5.

  $t_8 := 4 * j; \ t_9 := a[t_8]; \ a[t_8] := x;$

  is changed to:

  $t_9 := a[t_4]; \ a[t_4] := x;$

- **Copy Propagation**: Introduction of copy statements to simplify equal expressions.

  $$t := d + e$$

  $a := d + e \quad a := t$

  $b := d + e \quad \text{is changed to: } b := t$

  $c := d + e \quad c := t$
Dead Code Elimination

- A variable *lives* at a certain location in the code, if its value can be used or is used afterwards.
- Code is considered *DEAD*, when the calculated values are not used. This code can be ignored.
- Example:
  ```
  debug := false
  ...
  if (debug) print ...
  ```
  
  *Keyword: Constant Folding*

- Copy statements can often be deleted, since the variables are no longer used.
Loop Optimization

- **Code Motion**: Moving the code in an area outside of the loop. While \( i \leq \text{limit}-2 \) ... is changed to \( t = \text{limit}-2; \) while \( i \leq t \) ....

- **Induction Variable Elimination**: Removing some induction variables by using other variables.

- **Reduction in Strength**: Replacement of expensive operators through cheap ones (e.g. multiplication is replaced by addition).
Transformed Quicksort Program

i := m-1
j := n
t1 := 4*n
v := a[t1]
t2 := 4*i
t4 := 4*j

t2 := t2+4
t3 := a[t2]
if t3 < v goto B2

b4

t4 := t4-4
t5 := a[t4]
if t5 > v goto B3

b3

if i >= j goto B6

a[t2] := t5
a[t4] := t3

goto B2

B1

B2

B3

B4

B5

B6

a[t2] := t5
a[t4] := t3

goto B2

B1

B2

B3

B4

B5

B6

a[t2] := t5
a[t4] := t3

goto B2

F. Wotawa (IST @ TU Graz)
Optimization of Basic Blocks

- Represent Basic Blocks as a Directed Acyclic Graph (DAG)
- Utilize the graph for optimization
  - Common Subexpression Elimination:
    
    
    
    
    \[
    \begin{align*}
    a & := b + c \\
    b & := a - d \\
    c & := b + c \\
    d & := a - d
    \end{align*}
    \]

    - Dead Code Elimination: Remove each root node from the DAG which has variables that are no longer used (DEAD variables).
Basic Block Optimization (2)

- **Algebraic Equivalences:**
  
  \[
  \begin{align*}
  x + 0 &= 0 + x = x \\
  x - 0 &= x \\
  x \times 1 &= 1 \times x = x \\
  x/1 &= x
  \end{align*}
  \]

- **Reduction in Strength**
  
  \[
  \begin{align*}
  x \times 2 &= x \times x \\
  2.0 \times x &= x + x \\
  x/2 &= x \times 0.5
  \end{align*}
  \]

- **Constant Folding:** Evaluation of expressions that remain constant. The calculated values are used instead of the constants.

- **Further algebraic transformation:** Associative and communicative laws are used, relational operators (\(<,\),\(>,\)\(\)) are replaced with subtraction and tests, ..
Looking for Loops

- **Dominator**: The node $d$ of a flow graph dominates a node $n$ if every path from the initial node to $n$ leads through $d$.

- **Recognizing loops**:
  - Each node must have a single input node **Header**. This entry dominates all other nodes of the loop.
  - There is at least one way to iterate through the loop.
  
  Search edges in the flow graph $a \rightarrow b$ where $a$ dominates the node $b$ (Tail). Such an edge is a **Back Edge**.
Calculating the Dominator

**Input.** A flow graph $G$ with set of nodes $N$, set of edges $E$ and initial node $n_0$.

**Output.** The relation $dom$.

1. $D(n_0) := \{n_0\}$
2. for $n \in N - \{n_0\}$ do $D(n) := N$
3. while changes to any $D(n)$ occur do
   1. for $n \in N - \{n_0\}$ do
      1. $D(n) := \{n\} \cup \cap_p$ a predecessor $n D(p)$

$d$ is in $D(n)$ iff $d \in dom n$. 

F. Wotawa (IST @ TU Graz)
Natural Loop

- Given: a Back Edge \( n \rightarrow d \).
- The Natural Loop consists of \( d \) and all nodes that reach \( n \) without going through \( d \). \( d \) is the Header.
- Algorithm:
  
  ```
  procedure insert(m)
  if m is not in loop then begin
    loop := loop \cup \{m\}
    push m onto stack
  end
  ```

  ```
  procedure main(n \rightarrow d)
  stack := empty
  loop := \{d\}
  insert(n)
  while stack is not empty do begin
    pop m, the first element of the stack, off stack
    for each predecessor \( p \) of \( m \) do insert(p)
  end
  ```

- Inner Loops: a loop that doesn’t contain any other loops.
Reducible Flow Graphs

Definition: A flow graph is reducible if and only if its edges can be partitioned into two non-intersecting parts, and the parts have the following properties:

1. The forward nodes of the graph form an acyclic graph where each node can be reached from the initial node.
2. All back edges are edges where their heads dominate their tails.

Example: The following graph is not reducible.