CS 444: Compiler Construction

Gabriel Wong

Contents

Notes taken from lectures by Ondřej Lhoták in Winter 2016.

Introduction

A *compiler* translates from a source language (eg. Java) to a target machine language (eg. x86).

Scanning, parsing (A1) and context sensitive analysis (A234) are part of the compiler *front end*. The front end is analysis of the source.

Optimization (CS 744) and code generation (A5) are part of the compiler *back end*. The back end does synthesis in the target language.

Front-end Analysis

The *front end* is responsible for

- Checking if input is a valid program
- Gather information

Formal Languages

Alphabet (Σ) Finite set of symbols
Word A finite sequence of symbols from an alphabet
Language A (precisely defined) set of words

Lexical Analysis / Scanning

We typically use *regular languages* for lexical analysis. *Lexical analysis* is the process of transforming a sequence of characters into a sequence of tokens.

Scanning usually uses the maximal munch algorithm, where the longest token possible is created. Also DFAs are usually used.

DFAs for scanning can be designed directly by hand for most purposes. Lex is a tool that can generate a DFA from a regular expression.

Regular Expressions

Regular Expression (e)	Meaning ($L(e)$)
$a \in \Sigma$	$\{a\}$
ϵ	$\{\epsilon\}$
$e_1 e_2$	$\{xy \mid x \in L(e_1), y \in L(e_2)\}$
$e_1 \mid e_2$	$L(e_1) \cup L(e_2)$
e^*	$L(\epsilon \mid e \mid ee \mid eee \mid ...)$
$\varnothing$	$\{\}$

Regex to DFA

We first begin by creating an NFA for the regular expression:

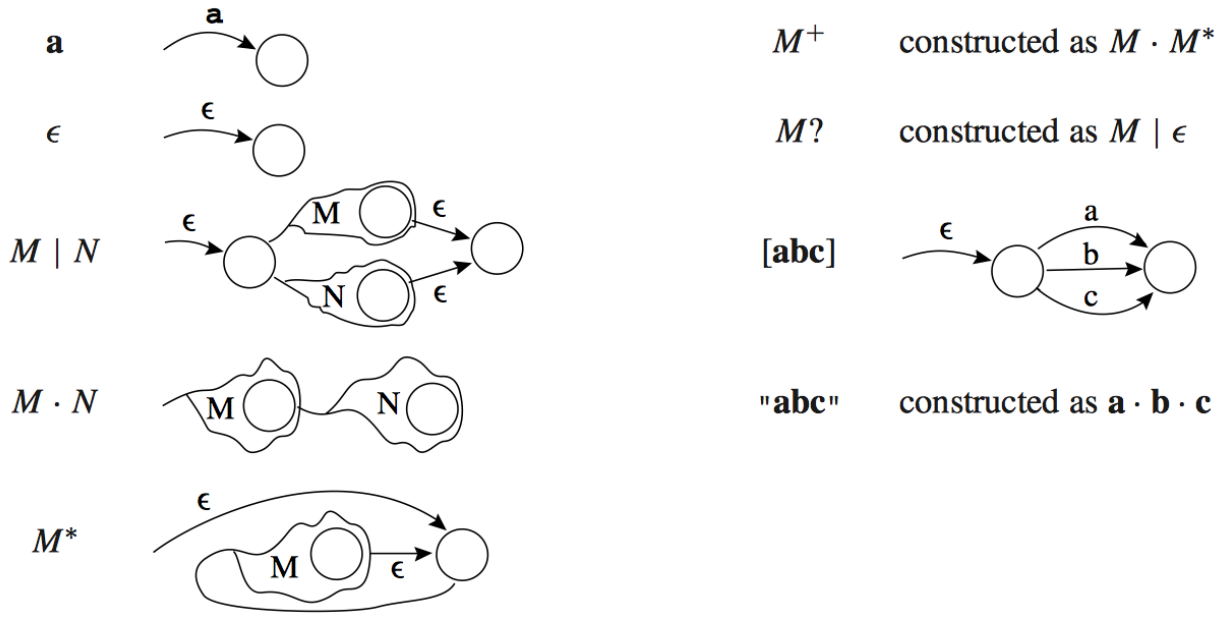


Figure 1: Constructing an NFA from a regular expression

Definition of NFA and DFA

An NFA/DFA is a 5-tuple: $< \Sigma, Q, q_0, A, \delta >$

Σ Alphabet
Q Set of states
q_0 Start state
A Set of accepting states
δ Transition function

The difference between NFAs and DFAs is the δ function.

DFA δ**:** $Q \times \Sigma \to Q$
NFA δ**:** $Q \times \Sigma \to 2^Q$

Computing ϵ-closure

ϵ-closure The ϵ-closure(S) of a set of states S is the smallest set S' such that $S' \supseteq S$ and $S' \supseteq \{q \mid q' \in S', q \in \delta(q', \epsilon)\}$.
Intuitively, the ϵ-closure(S) is the set of states, including S, reachable by only ϵ transitions from some state in S.

The following pseudocode is an algorithm to calculate the ϵ-closure of a set S.

function CALCULATEEPSILONCLOSURE(S)
  worklist $\leftarrow$ elements of S
  while worklist is not empty **do**
    remove some q from worklist

```
        for all q' in δ(q, ε) do
            if q' is not in S' then
                add q' to S' and to worklist
            end if
        end for
    end while
    return S'
end function
```

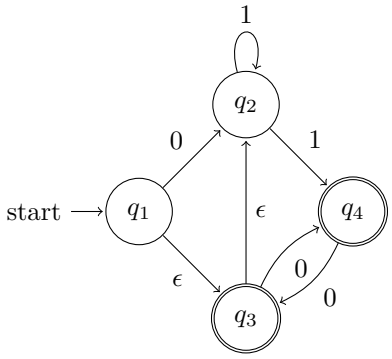
NFA to DFA Conversion

The general idea is to have states of the DFA represent a set of states of the NFA.

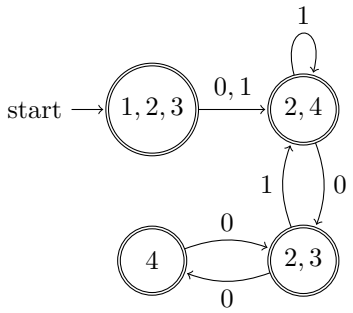
Then we have

$$
\begin{aligned}
q'_0 &= \epsilon\text{-closure}(\{q_0\}) \\
\delta'(q', a) &= \epsilon\text{-closure}\left(\bigcup_{q \in q'} \delta(q, a)\right) \\
Q' &= \text{smallest set } Q' \text{ of subsets of } Q \text{ such that} \\
&\quad q'_0 \in Q' \text{ and } Q' \supseteq \{\delta'(q', a) | q' \in Q', a \in \Sigma\} \\
A' &= \{q' \in Q' | q' \cap A \neq \varnothing\}
\end{aligned}
$$

Given the following NFA,



Our DFA has $q' = \epsilon\text{-closure}(\{1\}) = \{1, 2, 3\}$ and looks like



Scanning

Scanning is the process of converting a sequence of symbols to a sequence of tokens. A *token* has a *lexeme*, the symbols that make up the token and a kind.

Given a language L of valid lexemes and input string w of characters, a scanner outputs a sequence of tokens $\in L$ whose concatenation is w.

Maximal munch scanning outputs tokens greedily by length.

This is pseudocode for maximal munch scanning

while not end of input **do**
    output maximal prefix of remaning input that is in L
    error if no prefix
end while

Maximal munch may fail to scan a "valid" program that can be scanned. For example,

$$L = \{aa, aaa\}, w = aaaa$$

This can be scanned as $aa|aa$. But maximal much will output aaa and then report an error.

In Java (JLS 3.2), `a--b` is not a valid expression in Java due to maximal munch. It is lexically valid as `a|--|b` but fails parsing. However `a - - b` is a valid expression since it scans as `a|-|-|b` and parses as `a -(-b)`. **Note:** in Joos 1W, decrement is not implemented, but we still need to scan it to fail in this case.

Concretely using a DFA,

while not end of input **do**
    run DFA on remaining input until end of input or there is no transition
    if not in accepting state **then**
        Backtrack DFA and input to last seen accepting state
        ERROR if no accepting state was seen
    end if
    output token
    set DFA back to q_0
end while

We should remember the last-seen accepting state to easily backtrack.

Constructing Scanning DFA

Input: Regular expressions $R_1, R_2, ..., R_n$ for token kinds, prioritized

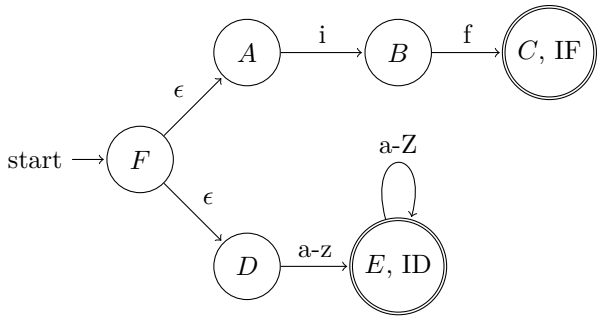
Output: DFA with accepting states labelled with kinds

1. Construct NFA for $R_1|R_2|...|R_n$ with each accepting state labelled with kind
2. Convert NFA to DFA
3. For each accepting state of DFA, set kind to NFA state with highest priority

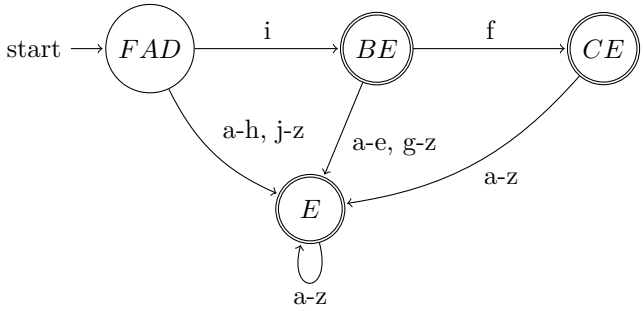
For example if we have the following tokens, prioritized

Kind	Regular Expression
IF	if
ID	[a-z][a-Z]*

Our NFA is



The resulting DFA is



IF has a higher priority than ID so our labelled states are

State	Kind
BE	ID
CE	IF
E	ID

Parsing

After the scanner converts a sequence of characters to a sequence of tokens, the parser converts the sequence of tokens to a parse tree. The scanner's Σ are the characters. The parser's Σ are the kinds output by the scanner (ie. the scanner's words).

Context-Free Grammar

Regular languages are not enough for parsing. This is because regular languages cannot express unbounded levels of nesting. For example, no regular expression can match `(((((((a)))))))` for unbounded number of parenthesis.

A *context-free grammar* is a 4-tuple $G =< N, T, R, S >$ with sets

T terminals
N non-terminals
$S \in N$ start symbol
$R \subseteq N \times V^*$ production rules
$V = T \cup N$ symbols
V^* sequence of symbols
T^* sequence of terminals

We have the following conventions to simplify notation:

$$
\begin{aligned}
a, b, c &\in T \\
A, B, C, S &\in N \\
A \to \alpha &\in R \\
X, Y, Z &\in V \\
\alpha, \beta, \gamma &\in V^* \\
x, y, z &\in T^*
\end{aligned}
$$

Directly derives ($\Rightarrow$): $\beta A \gamma \Rightarrow \beta \alpha \gamma$ if $A \to \alpha \in R$
Derives ($\Rightarrow^*$): $\alpha \Rightarrow^* \beta$ if $\alpha = \beta$ or ($\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$)
Context-Free Language L(G): $L(G) = \{x \in T^* \mid S \Rightarrow^* x\}$
Parsing Is $x \in L(G)$?
Find a derivation of x (parse tree).
Right derivation A derivation where each step is of the form $\beta A x \Rightarrow \beta \alpha x$.
Every step expands the right-most non-terminal.
Left derivation A derivation where each step is of the form $xA\gamma \Rightarrow x\alpha\gamma$
Every step expands the left-most non-terminal.

There is a one-to-one correspondence (bijection) between

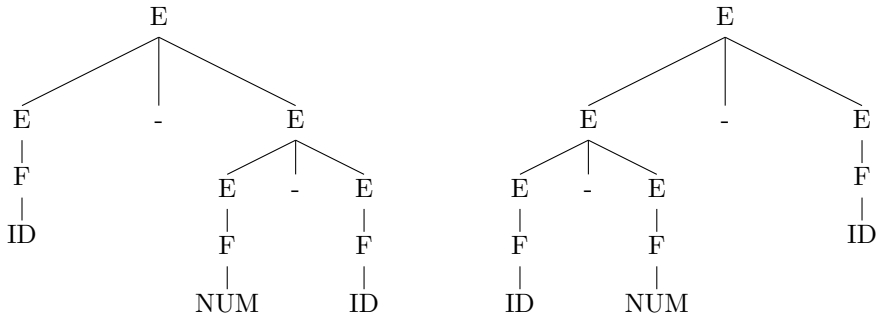
- Parse trees
- Right-derivations
- Left-derivations

A grammar is *ambiguous* if some word has more than 1 parse tree, more than 1 right-derivation or more than 1 left derivation.

Example: Consider the following context free grammar:

$$
\begin{aligned}
E &\to F \mid E - E \\
F &\to ID \mid NUM
\end{aligned}
$$

The grammar is ambiguous since there are multiple parse trees for `ID - NUM - ID`:



Top-down / Bottom-up Parsing

In *top-down* parsing, we do $S \Rightarrow ... \Rightarrow \underbrace{x}_{\text{input}}$. That is, we make derivations starting from the start symbol, until we obtain the input.

$\alpha \leftarrow S$
while $\alpha \neq x$ **do**

select $A \to \beta \in R$
replace some A in α with β
end while

In *bottom-up* parsing, we start from our input x. We apply production rules in reverse until we obtain the start symbol.

$\alpha \leftarrow x$
while $\alpha \neq S$ **do**
replace some β in α with A where $A \to \beta \in R$
end while

In both top-down and bottom-up parsing, we would like to process input from left to right.

LL parsing is weaker than LR parsing. LL cannot parse left-associative grammars, which is better for programming languages. LR can parse both left-associative and right-associative grammars.

LL (top-down) parsers are weaker since they need to know which rule should be used to expand a non-terminal using only a constant lookahead. It needs to apply a production rule without knowing whether the production body is correct. On the other hand, LR (bottom-up) parsers work on the roots of the parse tree. A LR parser can defer applying a production rule until it has produced the complete production rule body.

Example: For the grammar

$$E \to a$$
$$E \to E + a$$

When we try to parse $a + a$, a $LL(1)$ parser will get stuck. A $LL(1)$ parser cannot distinguish between $(E \Rightarrow a)$ or $(E \Rightarrow E + a)$. In either case, the next token can possibly be a. For a $LL(k)$ parser, we can see that it'll fail for a sufficiently long $a + a + ... + a$.

LL(1) Parsing

$$\underbrace{L}_{\text{left-to-right}} \overbrace{L}^{\text{left derivation}} \; (\; \underbrace{1}_{\text{1 symbol of lookahead}} \;)$$

LL stands for left-to-right, left derivation. The 1 in $LL(1)$ means 1 symbol of lookahead. Recall that a left derivation is when we always expand the leftmost non-terminal. Then our algorithm becomes

$\alpha \leftarrow S$
while $\alpha \neq x$ **do**
$A \leftarrow$ the first nonterminal in α $(\alpha = yA\gamma)$
$a \leftarrow$ the first terminal in x after y
$(A \to \beta) \leftarrow \text{Predict}(A, a)$ ▷ Production such that some derivation of β starts with a
replace A in α with β
end while

Example: Consider the following grammar

$$E \to a \; E'$$
$$E' \to + \; a$$
$$E' \to \epsilon$$

Let's perform the derivation for input $a + a$

$$\begin{aligned} & E \\ \Rightarrow\ & a\ E' \\ \Rightarrow\ & a + a \end{aligned}$$

Augmented Grammar

Let's see what happens if we try to parse a. We get $E \Rightarrow a\ E'$ but when we try to expand E', there is no terminal after a in our input.

To solve this, we *augment* the grammar. We add a new terminal \$ and add a rule $S' \to S\$$. S' becomes the new start symbol.

LL(1) Algorithm

We want our algorithm to run in $O(n)$ time. The loop in the previous code will run $O(n)$ times. So we need the body to run in constant time. Naively if we represent α using an array, this algorithm will take $O(n^2)$ time.

We can represent unprocessed input with a stack.

$$\alpha = y \underbrace{w\ A^{\text{bottom}}_{\text{top}}\ \gamma}_{\text{stack}}$$

The idea is to do one of the following

- Pop w and check that it matches the actual input (ie. check w becomes y).
- Pop A, choose $A \to \beta = \text{predict}(A, a)$ and push β.

```
function PARSE(x)
    push S$
    for a in x do
        while top of stack is A ∈ N do
            pop A
            find A → β in Predict(A,a) or ERROR
            push β
        end while
        pop b
        if b ≠ a then
            ERROR
        end if
    end for
    accept
end function
```

Predict$(A, a) = \{A \to \beta \in R \mid \exists\gamma : \beta \Rightarrow^* a\gamma$ or $(\beta \Rightarrow^* \epsilon$ and $\exists\gamma\delta : S\$ \Rightarrow^* \gamma A a\delta)\}$

first$(\beta) = \{a \mid \exists\gamma : \beta \Rightarrow^* a\gamma\}$ (the set of terminal symbols that can be first in a derivation of β)

nullable$(\beta) = \beta \Rightarrow^* \epsilon$ (β can be derived to ϵ)

follow$(A) = \{a \mid \exists\gamma\delta : S\$ \Rightarrow^* \gamma A a\delta\}$ (the set of terminals that can follow A)

Computing follow:

- If $B \to \alpha A \gamma \in R$, $\text{first}(\gamma) \subseteq \text{follow}(A)$
- If $B \to \alpha A \gamma \in R$ and $\text{nullable}(\gamma)$, $\text{follow}(B) \subseteq \text{follow}(A)$

Example: For the following grammar

$$\begin{aligned} S &\to E\$ \\ E &\to aE' \\ E' &\to +a \\ E' &\to \epsilon \end{aligned}$$

we have the following

$$\begin{aligned} \text{nullable}(\epsilon) &= \text{true} \\ \text{nullable}(+a) &= \text{false} \\ \text{nullable}(aE') &= \text{false} \\ \text{nullable}(E\$) &= \text{false} \\ \text{follow}(S) &= \{\} \\ \text{follow}(E) &= \{\$\} \\ \text{follow}(E') &= \{\$\} \\ \text{first}(\epsilon) &= \{\} \\ \text{first}(+a) &= \{+\} \\ \text{first}(aE') &= \{a\} \\ \text{first}(E\$) &= \{a\} \end{aligned}$$

From this, we can generate our Predict table

Predict	S	E	E'
a	$S \to E\$$	$E \to aE'$	
$+$			$E' \to +a$
$\$$			$E' \to \epsilon$

It is a coincidence that each rule only appears once in the table. However, it is not a coincidence that each cell has a maximum of one rule. That means that there is no ambiguity when parsing.

LL(1) Grammar A grammar is $LL(1)$ if $|\text{Predict}(A, a)| \leq 1$ for all A, a

LR(0) Parsing

Our algorithm will have the invariant that Stack + remaining input $\Rightarrow^*$ input.

If the top elements of our stack form the right side of a production (handle), we can perform a *reduction*. Otherwise, we *shift* the next symbol from the input onto the stack.

Example: For the following grammar

$$S \to E\$$$
$$E \to E + a$$
$$E \to a$$

we do the following reductions (shifting operations omitted, but the pipe denotes boundary between stack and input):

$$\begin{aligned} & a \mid +a\$ \\ \Leftarrow\ & E + a \mid \$ \\ \Leftarrow\ & E\$ \mid \\ \Leftarrow\ & S \end{aligned}$$

The following is pseudocode for the LR(0) algorithm:

```
for all terminal a in the input x$ do
    while (A → γ) ← Reduce(stack) do
        pop |γ| symbols                                  ▷ Reduce
        push A
    end while
    if Reject(stack + a) then
        ERROR
    end if
    push a                                               ▷ Shift
end for
```

Sentential form Given a grammar G, α is a *sentential form* if $S \Rightarrow^* \alpha$.
Viable prefix α is a *viable prefix* if it is the prefix of a sentential form.

$$\exists \beta : S \Rightarrow^* \underbrace{\alpha}_{\text{stack}} \overbrace{\beta}^{\beta \Rightarrow^* \text{remaining input}}$$

Then we'll keep a second invariant that the stack is a viable prefix. This helps guide our parser toward the start symbol. To summarize, our invariants are

1. Stack + unseen input $\Rightarrow^*$ input
2. The stack is a *viable prefix*

For z = seen input, y = unseen input, $x = zy$ = input, and α = stack,

$$\alpha \Rightarrow^* z$$
$$\alpha y \Rightarrow^* zy \qquad \text{(Invariant 1)}$$
$$\exists \beta : S \Rightarrow^* \alpha\beta \qquad \text{(Invariant 2)}$$

Then we have $\beta \Rightarrow^* y$ and

$$S \Rightarrow^* \alpha\beta \Rightarrow^* \alpha y \Rightarrow^* zy = x$$

To keep our invariant when reducing we want

$$\text{Reduce}(\alpha) = \{A \to \gamma \in R \mid \exists \beta : \alpha = \beta\gamma \text{ and } \beta A \text{ is a viable prefix}\}$$

To keep our invariant when shifting we want

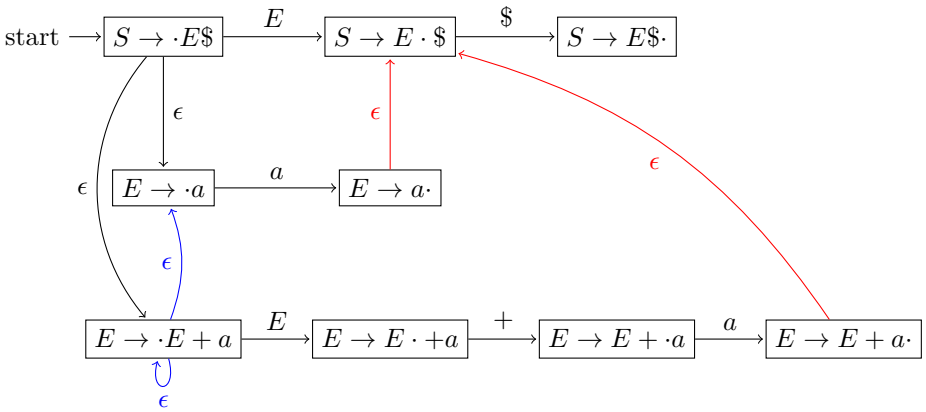
$$\text{Reject}(\alpha) = \alpha \text{ is not a viable prefix}$$

How do we check if α is a viable prefix? Turns out that for a context-free grammar G, the language of viable prefixes of G is also a context-free language. The key is to build a parser to only answer for a limited subset of α's (stacks).

Consider the grammar

$$\begin{aligned} S &\to E\$ \\ E &\to E + a \\ E &\to a \end{aligned}$$

We have the following NFA to recognize prefixes for the grammar:



Note that the red edges can be removed from the NFA. This is because when we reach the end of a production, we have to reduce. To see why this is true, consider if we don't reduce at the end of a production. If we don't reduce, then we may miss a possible derivation of the input. By failing to reduce, the parser would fail to consider derivations with that non-terminal expanded.

The blue edges are present because E can be derived by either production rule with E as the *head*. Generally for a non-terminal symbol, we need to consider all its production rules.

This is how to formally construct a $LR(0)$ NFA:

$$\begin{aligned} \Sigma &= N \cup T \\ A = Q &= \{A \to \alpha \cdot \beta \mid A \to \alpha\beta \in R\} \\ q_0 &= \{S' \to \cdot S\$\} \\ \delta(A \to \alpha \cdot X\beta, X) &= \{A \to \alpha X \cdot \beta\} && \text{(Shift)} \\ \delta(A \to \alpha \cdot B\beta, \epsilon) &= \{B \to \cdot\gamma \mid B \to \gamma \in R\} && \text{(Possible reduction)} \end{aligned}$$

Notice that all states are accepting states since we are looking for viable prefixes.

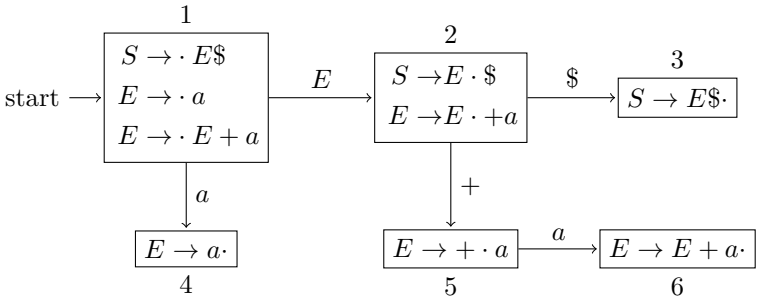
To use the NFA, run it on the stack. If the state ends in a dot ($A \to \gamma\cdot$), reduce $A \to \gamma$). If the state does not end in a dot, then shift.

What happens if we encounter two states ending in a dot? Or a state ending in a dot and another state not ending in a dot? Then we have a conflict. To determine if we have a conflict, we can convert the NFA to a DFA.

- If a DFA state contains $A \to \gamma\cdot$ and $B \to \alpha \cdot X\beta$, there's a shift-reduce conflict.
- If a DFA state contains $A \to \gamma\cdot$ for two different rules, there's a reduce-reduce conflict.

In essence, there is no conflict only if all DFA states each contain only one *item* or contain multiple shift items. A grammar is **LR(0)** if the **LR(0)** DFA has no conflicts.

This is the DFA from the NFA



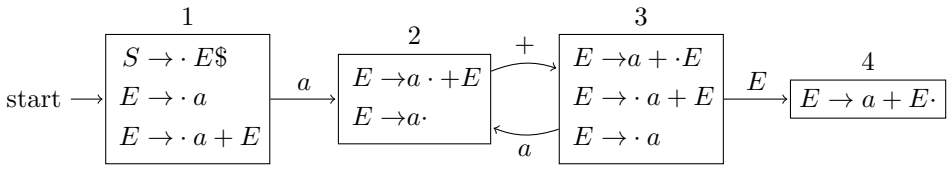
Let's use the DFA to parse $a + a\$$.

$\|a + a\$$	(State 1)
$a\| + a\$$	(Shift a: [2])
$E\| + a\$$	(Reduce $E \to a$: [1] then goto [2])
$E+\|a\$$	(Shift +: [5])
$E + a\|\$$	(Shift a: [6])
$E\|\$$	(Reduce $E \to E + a$: [1] then goto [2])
$E\$\|$	(Accept when \$ is shifted)

A $LR(0)$ grammar is not powerful enough for most purposes. Consider the grammar

$$S \to E\$$$
$$E \to a + E$$
$$E \to a$$

We have a shift-reduce conflict in state 2:



Notice that the next symbol can tell us whether to shift or reduce.

$a \mid \boxed{\$}$	(Reduce $E \to a$)
$a \mid \boxed{+}\ a\$$	(Shift and eventually do $E \to a + E$)

This brings us to LR(1) parsing.

LR(1) Parsing

We'll use the stronger invariant that our stack and a symbol is a viable prefix. We slightly modify our parsing algorithm,

for all terminal a in the input $x\$$ **do**
  while $(A \to \gamma) \leftarrow$ Reduce(stack, a) **do**
    pop $|\gamma|$ symbols ▷ Reduce
    push A
  end while
  if Reject(stack + a) **then**
    ERROR
  end if
  push a ▷ Shift
end for

with a new Reduce() function

$$\text{Reduce}(\alpha, a) = \{A \to \gamma \mid \exists \beta : \alpha = \beta\gamma \text{ and } \beta A a \text{ is a viable prefix}\}$$

Using a LR(0) NFA to keep our invariant does not work since we would need the red ϵ-transitions that we removed. There are some possible solutions:

1. If $a \notin \text{follow}(A)$, then $\beta A a$ is never a viable prefix. Resolving conflicts this way gives a SLR(1) parser. The 'S' stands for simple.
2. Build a better NFA/DFA (LR(1) DFA)

SLR(1) Parser

The SLR(1) Parser has a weaker Reduce() function than LR(1) but is stronger than LR(0)

$$\text{Reduce}(\alpha, a) = \{A \to \gamma \mid \exists \beta : a = \beta\gamma \text{ and } \beta A \text{ is a viable prefix and } a \in \text{follow}(A)\}$$

Going back to our example, $\text{follow}(E) = \{\$\} \not\ni +$. Then a SLR(1) parser can resolve this conflict.

We can enforce this constraint using our LR(0) DFA, except being more lax for conflicts:

- If a state contains $A \to \gamma\cdot$ and $B \to \alpha \cdot X\beta$ **and** $X \in \text{follow}(A)$, then it is a shift-reduce conflict.
- If a state contains $A \to \gamma\cdot$ and $B \to \delta\cdot$ **and** $\text{follow}(A) \cap \text{follow}(B)$, then it is a reduce-reduce conflict.

An SLR(1) parser is still not strong enough though. Consider the following grammar

$$\begin{aligned} S' &\to S\$ \\ S &\to a \\ S &\to E = E \\ E &\to a \end{aligned}$$

If we build the DFA and shift a, we'll have a state with items

$$\begin{aligned} S &\to a\cdot \\ E &\to a\cdot \end{aligned}$$

And we have follow sets

$$\begin{aligned} \text{follow}(S) &= \{\$\} \\ \text{follow}(E) &= \{=, \$\} \end{aligned}$$

There is still a reduce-reduce conflict since $\text{follow}(S) \cap \text{follow}(E) = \{\$\}$ and is not empty.

The problem is that we are using a global follow set. We can resolve more conflicts by keeping a "local" follow set for each state. We can only enter that state if the lookahead symbol matches the lookahead symbol in the state.

Comparison Between LR Parsers

$$\begin{aligned} \alpha &= \beta\gamma && \text{(stack)} \\ A \to \gamma &\in R \\ y &= \text{remaining input} \\ a &= \text{first symbol of } y \end{aligned}$$

We have the following implications:

$$\begin{aligned} &\beta Ay \text{ is a sentential form} && \text{(must reduce } A \to \gamma \text{ otherwise we'd miss a derivation)} \\ \Rightarrow &\beta Aa \text{ is a viable prefix} && \text{(LR(1), more reductions than necessary)} \\ \Rightarrow &\beta A \text{ is a viable prefix } a \in ? && \text{(LALR(1))} \\ \Rightarrow &\beta A \text{ is a viable prefix and } a \in \text{follow}(A) && \text{(SLR(1))} \\ \Rightarrow &\beta A \text{ is a viable prefix} && \text{(LR(0))} \end{aligned}$$

LR(1) NFA

It is similar to the LR(0) DFA except that each item has a lookahead symbol. The idea is an item $(A \to \alpha \cdot \beta, x)$ indicates that α is at the top of the stack and a prefix of the input is derivable from βx.

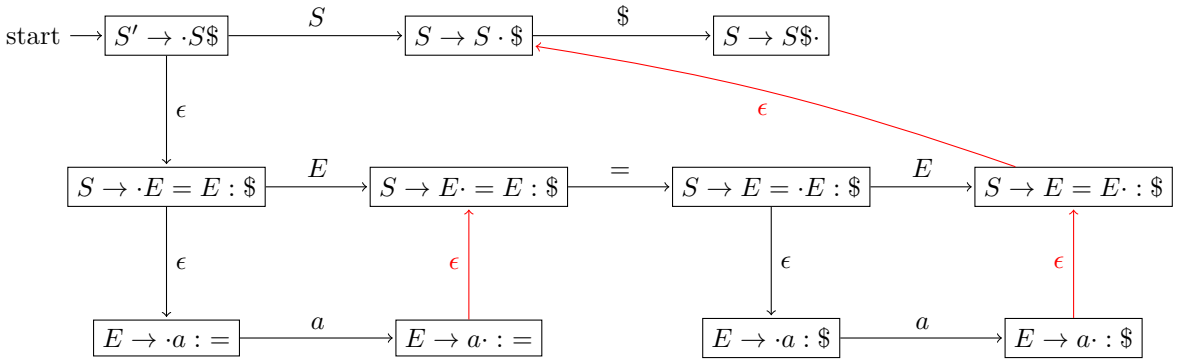
$$\begin{aligned} \Sigma &= N \cup T \\ A = Q &= \{A \to \alpha \cdot \beta : a \mid A \to \alpha\beta \in R, a \in T\} \\ q_0 &= S' \to \cdot S\$: \$ \\ \delta(A \to \alpha \cdot X\beta : a, X) &= \{A \to \alpha X \cdot \beta : a\} \\ \delta(A \to \alpha \cdot B\beta : a, \epsilon) &= \{B \to \cdot\gamma : b \mid B \to \gamma \in R, b = \text{first}(\beta a)\} \end{aligned}$$

When shifting, we use the same lookahead symbol as in the previous state. When considering expanding a non-terminal, a lookahead symbol that any terminal that could follow the non-terminal.

Example: For the grammar

$$\begin{aligned} S' &\to S\$ \\ S &\to E = E \\ E &\to a \end{aligned}$$

we have the following LR(1) NFA:



Again the red edges aren't necessary since we would have to reduce before taking the transition.

To use the DFA:

- $(A \to \gamma\cdot : a)$ reduce only if a is the next input symbol.
- $(A \to \alpha \cdot a\beta : b)$ shift if next symbol is a.

Then we have a conflict when the following are in the same state:

- $(A \to \gamma\cdot : \underline{a})$ and $(B \to \alpha \cdot \underline{a}\beta : b)$ is a shift-reduce conflict
- $(A \to \gamma\cdot : \underline{a})$ and $(B \to \delta\cdot : \underline{a})$ is a reduce-reduce conflict

LALR(1) Parser

The LA in LALR stands for look ahead. The motivation for using LALR(1) is that LR(1) DFAs can have very many states due to many states differing only in lookahead symbol. We would like the LR(1) DFA to be small like the LR(0) DFA.

Let S be an LR(1) DFA state. We define

$$\text{core}(S) = \{A \to \alpha \cdot \beta \mid A \to \alpha \cdot \beta : a \in S\}$$

We can see that if every state of the LR(1) DFA is replaced with its core, the result is the LR(0) DFA.

For each LR(0) DFA state, use a local follow set from lookahead symbols of corresponding LR(1) DFA states:

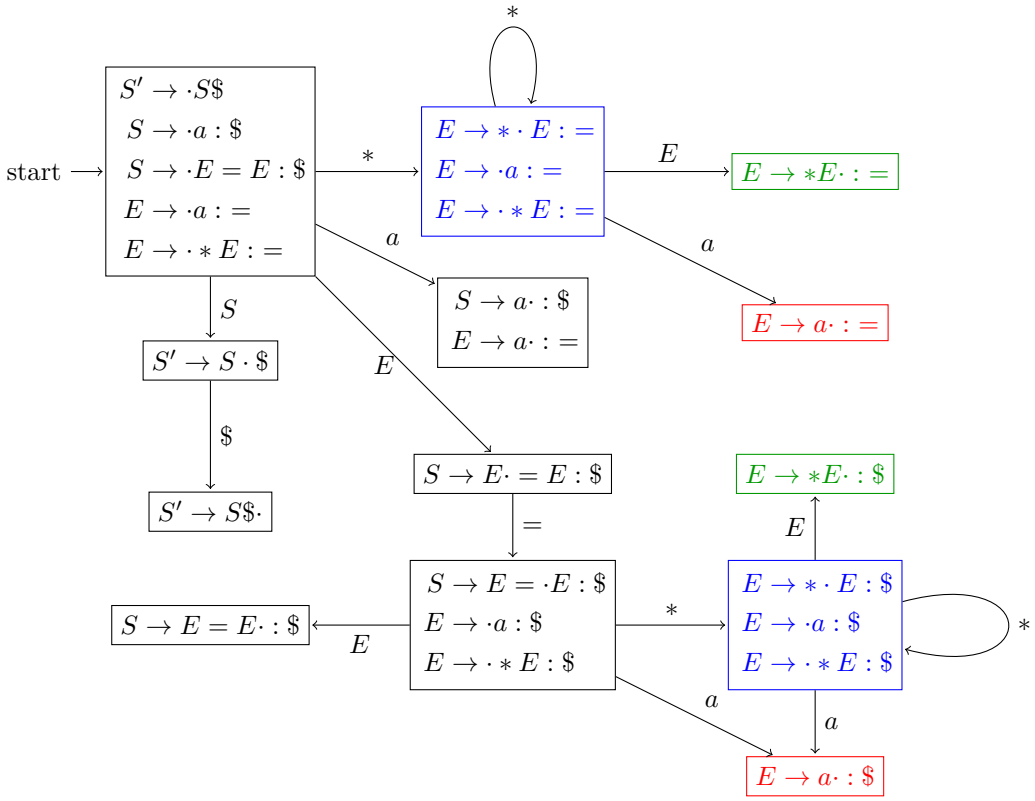
$M \leftarrow$ the LR(0) DFA
$M' \leftarrow$ the LR(1) DFA
for all state q of M **do**
  for all $A \to \alpha \cdot \beta \in q$ **do**
    lookahead $\leftarrow \{a \mid q' \text{ is a state of } M',\ q = \text{core}(q'),\ A \to \alpha \cdot \beta : a \in q'\}$
  end for
end for

In essence, we are merging states in the LR(1) DFA which differ only in lookahead symbol. And in our LR(0) DFA, we keep a set of lookahead symbols in each state.

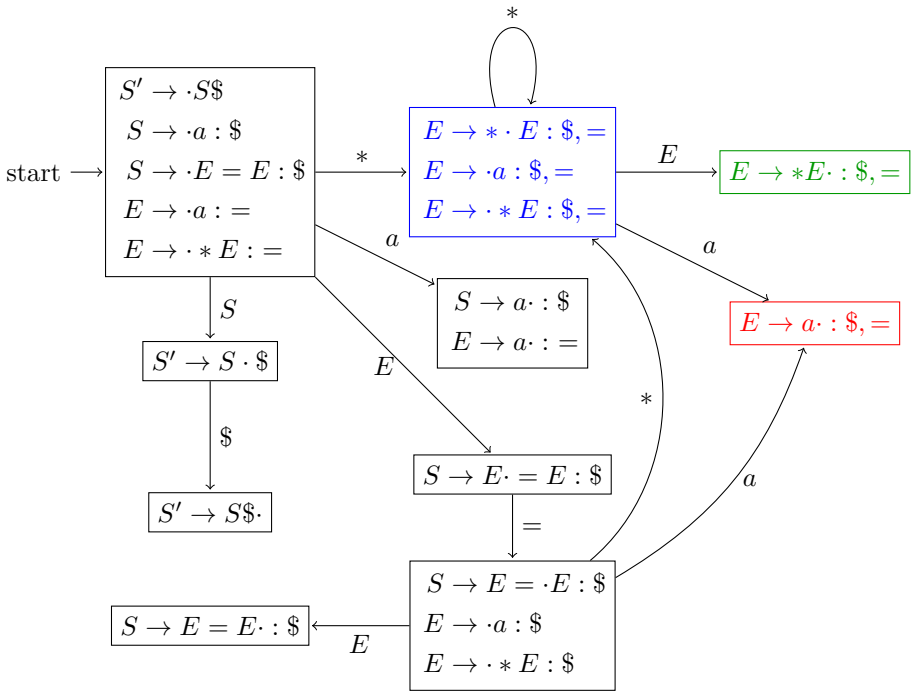
Example: For the grammar

$$\begin{aligned} S' &\to \cdot S\$ \\ S &\to \cdot a : \$ \\ S &\to \cdot E = E : \$ \\ S &\to \cdot a : = \\ E &\to \cdot * E : = \end{aligned}$$

This is the LR(1) DFA. Notice how many states there are for such a simple grammar.



The colored states differ only in lookahead symbols so they are combined for a LALR(1) DFA:

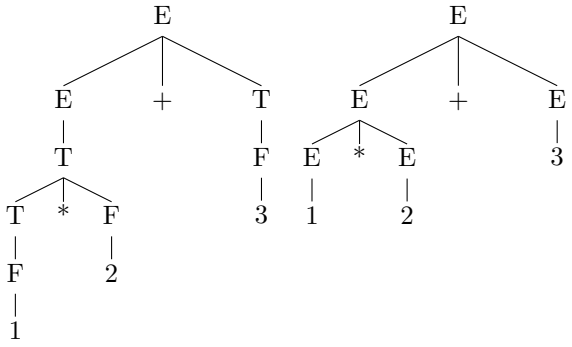


The DFA is now smaller by 3 states. For more complicated grammars with more types of terminal symbols, the LALR(1) DFA will have significantly less states than the LR(1) DFA.

Abstract Syntax Tree

A parse tree can sometimes be painful to deal with. The LR(1) grammar for programming languages is usually quite complex to avoid with ambiguities and to enforce order of operations.

To simplify later stages, it's usually a good idea to construct an abstract syntax tree from the parse tree. The abstract syntax tree is allowed to have an ambiguous grammar since parsing is already complete. The AST grammar is then much simpler. We construct the abstract syntax tree by recursively walking the parse tree.



Parse Tree Abstract Syntax Tree

For this simple grammar, creating an abstract syntax tree only gives a small benefit. However for a grammar such as Java's, there is a huge reduction in complexity of the tree.

Semantic Analysis

Semantic analysis is the phase of the compiler concerned with the meaning of identifiers.

Scope An area of a program where a declaration has effect and is visible.
Environment A map from names (strings) to declarations (AST node).

We keep an *environment* for each *scope* since scopes have different sets of active variables.

Example: For the following Java program,

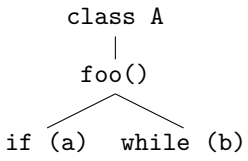
```
public class A {

  private int a;
  private boolean b;

  public boolean foo(boolean a) {
```

```
    a = true;
    if (a) {
      b = true;
      int b = 5;
      b = b + 7;
    while (b) {
      a = false;
      break;
    }
    return b;
  }
}
```

we have this hierachy of scopes:



Implementing Environments

The following are typically used to implement an environment:

- A hash map to map names to AST nodes.
- Reference to the scope (AST node) which the environment is for.
- Reference to the outer enclosing environment.

To lookup a name:

1. Search for the name in the local environment
2. Recursively search in the enclosing environments

Namespaces

Many languages have different namespaces for different types of identifiers. A **namespace** is a set of identifiers that we search in.

For example, in `methodCall(x + 5)`, `methodCall` is found in the method namespace and `x` is in the field namespace.

Namespaces are determined syntactically (based on position in grammar). That is clear from the example above.

These are the namespaces in Java/Joos:

- Expression (variable, field)
- Method
- Type (class, interface, enum)
- Package

In Java, we also have a "type or package" namespace due to nested classes and an ambiguous namespace, but those aren't in Joos.

Java Name Resolution

These are the steps for Java name resolution:

1. Create class environment

The class environment is a set of all classes. We resolve the fully qualified names (name including package) of all classes.

2. Resolve type names

In Java there are qualified names. If a name has a dot, then it must be fully qualified (eg. `java.util.List`). There are also simple names without a dot. This is how we resolve simple names:

a) Is it the enclosing class/interface?
b) Is it a single type import? (`import foo.bar.baz`)
c) Is it a type in the same package?
d) Is it an import on demand? (`import foo.bar.baz.*`)

Each step must be unambiguous. Otherwise it is a compile-time error.

3. Check class hierachy

We can perform some simple checks straight away:

a) If class `A` extends `B`, `B` must be a class.
b) If class `C` implements `D`, `D` must be an interface.
c) We cannot have duplicate interfaces (eg. `class E implements F, F`).
d) Cannot extend a final class.
e) Cannot have two constructors with the same signature (easier to do than methods since constructors aren't inherited).

Let's define a model for the remaining checks:

super(A) Set of superclasses of `A`

For `class A extends B implements C, D, E`, we have super(A) = { B, C, D, E }. If a class does not extend an object, the default superclass is `java.lang.Object`. Also super(`java.lang.Object`) = {}.

For `interface F extends G, H, I`, we have super(F) = { G, H, I }. Also methods in interfaces are implicitly public abstract. Fields are implicitly static.

Define $S < T$ to mean that S is a strict subtype of T.

S is a strict subtype of its direct superclasses:

$$\frac{T \in \text{super}(S)}{S < T}$$

The strict subtype relation is transitive:

$$\frac{\exists T' : S < T' \quad T' < T}{S < T}$$

$S \leq T$ is the subtype relation:

$$\frac{S < T}{S \leq T} \qquad \frac{}{S \leq S} \qquad \frac{}{S = S}$$

declare(T) Methods and fields declared in T
inherit(T) Methods and fields that T inherits from supertypes
contain(T) Methods and fields contained in T. $\text{contain}(T) = \text{declare}(T) \cup \text{inherit}(T)$
replace(m, m') Method m overrides m'. Field or static method m hides m'.
nodecl(T, m) Method m is not declared in T
$\text{nodecl}(T, m) = \forall m' \in \text{declare}(T) : \text{sig}(m) = \text{sig}(m')$
allabs(T, m) If m is in a supertype of T, then m is abstract.
$\text{allabs}(T, m) = \forall S \in (T)\ \forall m' \in \text{contain}(S) : \text{sig}(m) = \text{sig}(m') \Rightarrow \text{abstract} \in \text{mods}(m')$

Declared methods override their corresponding methods in supertypes:

$$\frac{m \in \text{declare}(T) \quad S \in \text{super}(T) \quad m' \in \text{contain}(S) \quad \text{sig}(m) = \text{sig}(m')}{(m, m') \in \text{replace}}$$

Inherit concrete methods that are not declared:

$$\frac{S \in \text{super}(T) \quad m \in \text{contain}(S) \quad \text{nodecl}(T,\ m) \quad \text{abstract} \notin \text{mods}(m)}{m \in \text{inherit}(T)}$$

Inherit abstract methods that do not have a corresponding concrete method:

$$\frac{s \in \text{super}(T) \quad m \in \text{contain}(S) \quad \text{nodecl}(T, m) \quad \text{abstract} \in \text{mods}(m) \quad \text{allabs}(T, m)}{m \in \text{inherit}(T)}$$

Concrete methods from supertypes override abstract methods from supertypes:

$$\frac{\begin{array}{r} S \in \text{super}(T) \quad m \in \text{contain}(S) \quad S' \in \text{super}(T) \quad m' \in \text{contain}(S') \quad \text{nodecl}(T, m) \\ \text{sig}(m) = \text{sig}(m') \quad \text{abstract} \notin \text{mods}(m) \quad \text{abstract} \in \text{mods}(m') \end{array}}{(m, m') \in \text{replace}}$$

Fields in supertypes that aren't declared are inherited:

$$\frac{S \in \text{super}(T) \quad f \in \text{contain}(S) \quad \forall f' \in \text{declare}(T) : \text{name}(f') \neq \text{name}(f)}{f \in \text{inherit}(T)}$$

JLS 9.2: an interface without any super interfaces implicitly declares an abstract version of every public method in `java.lang.Object`.

Now having defined a model, we can define constraints on our class hierachy (numbering corresponds to dOvs well-formedness constraints):

1) $\nexists T : T < T$ (no cycles)
2) $\forall m, m' \in \text{declare}(T) : m \neq m' \Rightarrow \text{sig}(m) \neq \text{sig}(m')$ (declared methods all have different signatures)
3) $\forall m, m' \in \text{contain}(T) : \text{sig}(m) = \text{sig}(m') \Rightarrow \text{type}(m) = \text{type}(m')$ (methods with the same signature must have the same return type)
4) $\forall m \in \text{contain}(T) : \text{abstract} \in \text{mods}(m) \Rightarrow \text{abstract} \in \text{mods}(T)$ (types containing abstract methods must be abstract)

5) $\forall(m, m') \in \text{replace} : \text{static} \in \text{mods}(m) \Leftrightarrow \text{static} \in \text{mods}(m')$ (method must be static if and only if overriding a static method)
6) $\forall(m, m') \in \text{replace} : \text{type}(m) = \text{type}(m')$ (method overriding another method must have the same the return type)
7) $\forall(m, m') \in \text{replace} : \text{public} \in \text{mods}(m') \Rightarrow \text{public} \in \text{mods}(m)$ (method overriding another method cannot have a stricter access modifier)
8) No 8
9) $\forall(m, m') \in \text{replace} : \text{final} \notin \text{mods}(m')$ (final method cannot be overridden)
10) $\forall f, f' \in \text{declare}(T) : f \neq f' \Rightarrow \text{name}(f) \neq \text{name}(f')$ (inherited field with same name must have same type)

4. Disambiguate ambiguous names

We expect an expression for names in the form of $a_1.a_2.a_3$. For $a_1.a_2.a_3.a_4()$ we expect a method.

These are the steps to disambiguate names:

1) If a local variable a_1 exists in the environment then $a_2.a_3.a_4$ are instance fields of a_1.
2) If field $a_1 \in$ contain(current class) then $a_2.a_3.a_4$ are instance fields of the current class.
3) Find the type with shortest length:

for k from 1 to $n-1$ **do**
  if $a_1.a_2.(...).a_k$ is a type **then**
    a_{k+1} is a static field and $a_{k+2}, ..., a_n$ are instance fields
  end if
end for

5. Resolve expressions (variables, fields)

Generally, we just search in environments moving outwards through nested scopes.

However Java differs from C in that we cannot have local variables with overlapping scopes:

```
void method() {
    int x;
    {
        int y;
    }
    {
        int x; // not allowed
        int y; // allowed
    }
}
```

The solution is to check if a variable is declared in enclosing environments in the same method when inserting a variable.

Also, we can only use a variable after it has been declared:

```
void method() {
    y = 42; // not allowed
    int y;
    y = 42; // allowed
}
```

One possible solution is to introduce a new block for each variable declaration:

```
void method() {
    {
        int x;
        x = 1;
        {
            int y;
            y = 2;
        }
    }
}
```

6. Type checking

Type A collection of values or an interpretation of bits
Static Type For an expression E, the *static type* is a set containing all possible values of E
Dynamic Type A runtime value that indicates how to interpret some bits
Declared Type The *declared type* of a variable is an assertion that the variables will only contain values of that type
Type Checking Enforcing declared type assertions
Static Type Checking Prove that every expression evaluates to a value in its type
Dynamic Type Checking Runtime check that the tag of a value is in the declared type of the variable to which it is assigned

Java mostly uses static type checking but can also perform dynamic type checking (eg. casts, `instanceof`).

A static type checker:

- Infers a static type for every (sub)expression.
- Proves that every expression evalues to a value in its type.
- Checks that expressions are used correctly. For example, `1 + true` or `needsBoolean(1)` is not allowed.

Type Correct A program is *type correct* if type assertions hold in all executions

Checking if a program is type correct is actually undecidable since we can reduce the halting problem it type correctness. For example it is undecidable whether the following program is type correct due to undecidability of the halting problem:

```
int x;
if (halts(someProgram)) {
    x = true;
}
```

Statically Type Correct A program is *statically type correct* if it satisfies type rules (a **type system**)
Soundness A type system is *sound* if static type correctness implies type correctness
A type system that is not sound is pretty useless since it doesn't really check anything

Here's a type system for Joos:

We'll define notation. The following means in class C, local environment L, where current method returns type σ, E has type τ.

$$C, L, \sigma \vdash E : \tau$$

In class C, local environment L, where current method returns type σ, S is syntatically type correct. In essence, statements which don't have a type are always correct.

$$C, L, \sigma \vdash S$$

These rules define types for literals:

$$\frac{}{42 : \texttt{int}} \qquad \frac{}{\texttt{true} : \texttt{boolean}} \qquad \frac{}{\texttt{"abc"} : \texttt{java.lang.String}}$$

$$\frac{}{\texttt{'a'} : \texttt{char}} \qquad \frac{}{\texttt{null} : \texttt{null}}$$

These are the rules for variables:

$$\frac{L(n) : \tau}{C, L, \sigma \vdash n : \tau} \qquad \frac{}{C, L, \sigma \vdash \texttt{this} : C}$$

$L(n) = \tau$ means that n has type τ in local environment L.

These are rules for operators (JLS 15). In Java, we can concatenate anything to a `String`.

$$\frac{E : \texttt{boolean}}{!E : \texttt{boolean}} \qquad \frac{E_1 : \texttt{String} \quad E_2 : \tau_2 \quad \tau_2 \neq \texttt{void}}{E_1 + E_2 : \texttt{String}} \qquad \frac{E_1 : \tau_1 \quad E_2 : \texttt{String} \quad \tau_1 \neq \texttt{void}}{E_1 + E_2 : \texttt{String}}$$

$$\frac{E_1 : \tau_1 \quad E_2 : \tau_2 \quad \text{num}(\tau_1) \quad \text{num}(\tau_2)}{E_1 + E_2 : \texttt{int}} \qquad \text{num}(\sigma) = \sigma \in \{\texttt{byte}, \texttt{short}, \texttt{char}, \texttt{int}\}$$

The rules for other arithmetic operators (`-`, `*`, `/`, `%`) are the same as addition, minus the `String` rules.

These are rules for statements:

$$\frac{\vdash S_1 \quad \vdash S_2}{\vdash S_1; S_2}$$

$$\frac{C, L[n \to \tau] \quad \sigma \vdash S}{C, L, \sigma \vdash \{\tau\ n; S\}}$$

$L[n \to \tau]$ means to use L except lookup for n is replaced with τ.

$$\frac{\vdash E : \texttt{boolean} \quad \vdash S}{\vdash \texttt{if}(E)\ S}$$

The same applies other statements such as while, if-else, etc.

$$\frac{L(n) : \tau_1 \quad C, L, \sigma \vdash E : \tau_2 \quad \tau_1 := \tau_2}{C, L, \sigma \vdash n = E}$$

The $\tau_1 := \tau_2$ means τ_2 is assignable to τ_1. We need to define type assignability.

$$\frac{}{\tau := \tau}$$

$$\frac{}{\texttt{short} := \texttt{byte}} \qquad \frac{}{\texttt{int} := \texttt{char}} \qquad \text{(widening)}$$

$$\frac{\sigma := \tau \quad \tau := \rho}{\sigma := \rho} \qquad \text{(transitivity)}$$

$$\frac{D \leq C}{C := D} \qquad \frac{}{C := \texttt{null}}$$

$$\frac{}{C, L, \sigma \vdash \texttt{this} : C} \qquad \text{(this expression)}$$

$$\frac{\texttt{static}\ \tau\ f \in D}{C, L, \sigma \vdash D.f := \tau} \quad \text{(static field)}$$

$$\frac{E : D \quad \tau_2\ f \in D}{C, L, \sigma \vdash E.f : \tau_2} \quad \text{(instance field)}$$

$$\frac{E : \tau_1 \quad \texttt{static}\ \tau_2\ f \in D \quad \tau_2 := \tau_1}{D.f = E : \tau_2} \quad \text{(static field assignment)}$$

$$\frac{E_1 : D \quad \tau_2\ f \in D \quad E_2 : \tau_3 \quad \tau_2 := \tau_3}{E_1.f = E_2} \quad \text{(instance field assignment)}$$

Upcast is safe but addition of downcast makes the type system unsound:

$$\frac{E : \tau_1 \quad \overbrace{\tau_2 := \tau_1}^{\text{upcast}} \vee \overbrace{\tau_1 := \tau_2}^{\text{downcast}}}{(\tau_2)E : \tau_2} \quad \text{(cast)}$$

$$\frac{E : \tau_1 \quad \text{num}(\tau_1) \quad \text{num}(\tau_2)}{(\tau_2)E : \tau_2} \quad \text{(numerical cast)}$$

$$\frac{E_1 : \tau_1 \quad E_2 : \tau_2 \quad \tau_1 := \tau_2 \vee \tau_2 := \tau_1}{E_1 == E_2 : \texttt{bool}} \quad \text{(comparison)}$$

$$\frac{E : \tau_1 \quad \sigma := \tau_1}{C, L, \sigma \vdash \texttt{return}\ E} \quad \text{(return type)}$$

$$\frac{\sigma : \texttt{void}}{C, L, \sigma \vdash \texttt{return}}$$

In Joos, the argument type must match the method signature exactly:

$$\frac{E : D \quad E_i : \tau_i \quad \tau\ m(T_1, ..., T_k) \in D}{E.m(E_1, ..., E_k) : \tau} \quad \text{(method invocation in Joos)}$$

This is the method invocation rule for Java which we don't need for the assignment:

$$\frac{E : D \quad E_i : \sigma_i \quad \tau\ m(\tau_1, ..., \tau_k) \in D \quad \tau_i := \sigma_i \quad \forall \gamma : (\gamma\ m(\gamma_1, ..., \gamma_R) \in D \wedge \gamma_i := \sigma_i) \Rightarrow (\forall i : \gamma_i := \tau_i)}{E.m(E_1, ..., E_k) : \tau}$$

These are rules for new class instances and arrays:

$$\frac{E_i : \tau_i \quad D(\tau_1, ..., \tau_k) \in D}{\texttt{new}\ \ D(E_1, ..., E_k) : D}$$

$$\frac{E_1 : \tau_1[] \quad E_2 : \tau_2 \quad \text{num}(\tau_2)}{E_1[E_2] : \tau_1}$$

$$\frac{E_1[E_2] : \tau_1 \quad E_3 : \tau_2 \quad \tau_1 := \tau_2 \quad \text{num}(E_2)}{E_1[E_2] = E_3}$$

$$\frac{E : \tau[]}{E.\texttt{length} : \texttt{int}}$$

$$\frac{E : \tau_2 \quad \text{num}(\tau_2)}{\texttt{new}\ \ \tau_1[E] : \tau_1[]}$$

$$\frac{}{\texttt{java.lang.Object} := \sigma[]} \qquad \frac{}{\texttt{java.lang.Cloneable} := \sigma[]} \qquad \frac{}{\texttt{java.io.Serializable} := \sigma[]}$$

$$\frac{D \leq C}{C[] := D[]} \qquad \frac{D \leq C}{C := D}$$

Java's array type system is unsound. For example the following is allowed under Java's type system:

```
B[] bs = new B[1];
Object[] os = bs;
os[0] = new C();
B b = bs[0];
```

To preserve type safety, we must check the dynamic type tag at every array write (JLS 10.10). If the tag does not match, throw an `ArrayStoreException`.

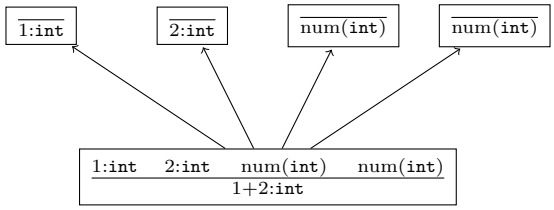
How do we practically use the rules?

function TYPECHECK(E)
Find type rule of the form

$$\frac{\text{premises}}{C, L, \sigma \vdash E : \tau}$$

Check premises by calling typecheck recursively
return τ
end function

Type derivation A tree whose ndoes are type rules. The root is the judgement to be proven and the leaves are axioms (rules with no premises).



The type derivation is useful to keep since the type rules can be used for code generation. For example the following rule

$$\frac{E_1 : \texttt{String} \quad E_2 : \tau \quad \tau \neq \texttt{void}}{E_1 + E_2}$$

can be made to generate `String.concat(`E_1`, String.valueOf(`E_2`))`.

Things get tricky if we can have ambiguous derivations (not Joos). In Java we can have $\frac{E:\texttt{int}}{E:\texttt{float}}$ which generates bytecode to convert an int to a float. Then we have the following ambiguous derivations:

$$\frac{\dfrac{i:\texttt{int} \quad j:\texttt{int}}{i+j:\texttt{int}}}{i + j : \texttt{float}} \qquad \frac{\dfrac{i:\texttt{int}}{i:\texttt{float}} \quad \dfrac{j:\texttt{int}}{j:\texttt{float}}}{i + j : \texttt{float}}$$

Java avoids this by specifying that operands should only be coerced if necessary (JLS 5.6).

7. Resolve methods and instance fields

For an expression $a.b$, look for a field named b in type of a.

For a method call $a.b(c)$, look for method b in type of a. We need to choose from overloaded methods using the type of c.

Static Analysis

Static analysis Proving properties about run-time behaviour (without running the program)

Some applications of static analysis include:

- Generate efficient code
- Prevent bugs
- Inform programmers

Rice's Theorem: Let R be a non-trivial property of the output of a program. Given program P, it is undecidable whether the output of P has the property R.

We can show Rice's Theorem by reducing proving whether the output of a program is n to the halting problem:

```
function HALTS(a, i)
    function T(n)
        a(i)
        return n
    end function
    return ISIDENTITYFUNCTION(t)
end function
```

t returns n if and only if $a(i)$ halts.

R is non-trivial means $\exists P \in R$ and $\exists P \notin R$. That is, R is a property that is not always true or always false.

For static analysis, we make conservative approximations. We return "yes," "no," or "maybe" for whether a program has a given property. It is always valid to return "maybe." A more **precise** analysis gives a definitive answer for more programs rather than maybe.

Java requires

- Reachability analysis (JLS 14.20)
  - All statements must *potentially* execute
  - No execution of a non-void method ends without a return statement
- Definite assignment (not in Joos)
  - Every local variable must be written before it is read

Java Reachability Analysis (JLS 14.20)

We define in[S] as whether S can start executing and out[S] as whether S can finish executing.

It is an error if in[S] = no for any S. That means a statement is not reachable. We define out[`return` E] = no. It is an error if out[non-void method body] = maybe. That means it is possible that a non-void method body does not return an expression. We define

$$\text{no} \vee \text{maybe} = \text{maybe}$$

For `if` statements, we assume the expression can evaluate to either true or false for the purpose of reachability.

For statement L : `if(`E`)` S,

$$\text{in}[S] = \text{in}[L]$$
$$\text{out}[L] = \text{in}[L] \vee \cancel{\text{out}[S]}$$

For statement L : `if(`E`)` S_1 `else` S_2,

$$\begin{aligned}\text{in}[S_1] &= \text{in}[L] \\ \text{in}[S_2] &= \text{in}[L] \\ \text{out}[L] &= \text{out}[S_1] \vee \text{out}[S_2]\end{aligned}$$

Loops have a few special cases. In the general case for statement L : `while(`E`)` S,

$$\begin{aligned}\text{in}[S] &= \text{in}[L] \\ \text{out}[L] &= \text{in}[L] \vee \cancel{\text{out}[S]}\end{aligned}$$

Java requires special treatment for `while` loops when E is a constant expresson. Note that in Joos, we don't have `break` statements. Then when E evaluates to a constant expression `true`,

$$\begin{aligned}\text{in}[S] &= \text{in}[L] \\ \text{out}[L] &= \text{no}\end{aligned}$$

When E evaluates to a constant expression `false`,

$$\begin{aligned}\text{in}[S] &= \text{no} \\ \text{out}[L] &= \text{in}[L]\end{aligned}$$

JLS 15.28 defines the meaning of *constant expressions*. Informally, a constant expression is an expression composed of literals, operators and final variables whose identifiers are constant expressions.

For `return` statements L : `return` and L : `return` E,

$$\text{out}[L] = \text{no}$$

For sequential statements $L : \{S_1; S_2\}$,

$$\begin{aligned}\text{in}[S_1] &= \text{in}[L] \\ \text{in}[S_2] &= \text{out}[S_1] \\ \text{out}[L] &= \text{out}[S_2]\end{aligned}$$

All other statements L can finish if they are reachable. That is,

$$\text{out}[L] = \text{in}[L]$$

We assume that all method bodies may potentially execute. For L : method body,

$$\text{in}[L] = \text{maybe}$$

Notice that there are no cycles in the in and out equations. in is inherited top-down from parent or earlier sibling. out is computed bottom-up from children. Then we can do reachability checking through a tree traversal.

Java Definite Assignment Analysis (JLS 16)

Definite assignment analysis is static analysis to ensure that all variables are assigned before they are accessed. For example the following is a compile time error in Java:

```
int foo() {
    int x;
    return x;
}
```

The definite assignment analysis in Joos barely qualifies as static analysis. In Joos, local variables must be initialized when they are declared. Also in Joos, variables cannot occur in its own initializer. The following is allowed in Java but not Joos: `int x = (x = 1);`

We define in$[S]$ as the set of variables that have definitely been initialized before S. We also define out$[S]$ as the set initialized after S.

In any expression E that reads l, if $l \notin \text{in}[E]$, give an error.

These are the rules for a variable declaration with initializer $L : \{\tau\ x = E; S\}$

$$\begin{aligned} \text{in}[E] &= \text{in}[L] \\ \text{in}[S] &= \text{out}[E] \cup \{x\} \\ \text{out}[L] &= \text{out}[S] \end{aligned}$$

For a variable declaration $L : \{\tau\ x; S\}$,

$$\begin{aligned} \text{in}[S] &= \text{in}[L] \\ \text{out}[L] &= \text{out}[S] \end{aligned}$$

For variable assignment $L : x = E$,

$$\begin{aligned} \text{in}[E] &= \text{in}[L] \\ \text{out}[L] &= \text{out}[E] \cup \{x\} \end{aligned}$$

These are the key rules in Java for definite assignment analysis. The full specification is more involved due to boolean short circuiting, ternary expression, and other reachability analysis.

Notice that here also, the equations for in and out do not have cycles. But in general, equations in static analysis can have cycles. One such example is live variable analysis.

Live Variable Analysis

Live variable analysis is the analysis of the set of variables whose current value might be read in the future. It is useful for register allocation and garbage collection. We only care if variables may be read before they are overwritten. That is because we can discard the current value of a variable if it is not read before being overwritten.

We define in$[S]$ as the set of live variables before S and out$[S]$ as the set of live variables after S.

For while loop L : `while(`E`)` S,

$$\begin{aligned} \text{out}[S] &= \text{out}[L] \cup \text{in}[S] \\ \text{in}[L] &= \text{out}[L] \cup \text{in}[S] \end{aligned}$$

If S is empty in the while loop then $\text{in}[S] = \text{out}[S]$ trivially. Then we have a loop since $\text{out}[S] = \text{out}[L] \cup \text{in}[S]$ and $\text{in}[S] = \text{out}[S]$ depend on each other.

Having these cycles do not prevent a solution, but makes it more difficult since we can't just recurse or do a tree traversal.

A partial order is a relation $\sqsubseteq$ that is

1. Reflexive $x \sqsubseteq x$
2. Transitive $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
3. Anti-symmetric $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

If $x \sqsubseteq y$ then if x is a sound approximation so is y.

For example, given

$$X = \{x, y\} \qquad Y = \{x, y, z\}$$

$X \sqsubseteq Y$, meaning X is at least as precise as Y.

Upper Bound z is an upper bound for x and y if $x \sqsubseteq z \wedge y \sqsubseteq z$
Lower Bound z is a lower bound for x and y if $z \sqsubseteq x \wedge z \sqsubseteq y$
Least Upper Bound z is a *least upper bound* of it is an upper bound and for all other upper bounds v of x and y, $z \sqsubseteq v$
Greatest Lower Bound z is a GLB if z is a lower bound ($z \sqsubseteq x, z \sqsubseteq y$) and for all other lower bounds v of x, y, $v \sqsubseteq z$

For example, given

$$X = \{x, y\} \qquad Y = \{y, z\} \qquad Z = \{a, b, c\}$$

The lower upper bound for live variables is union: $V = \{x, y, z, a, b, c\}$

Lattice A lattice is a partially ordered set such that for every pair of elements, there exists

- A least upper bound (join) $\sqcup$
- A greatest lower bound (meet) $\sqcap$

Complete Lattice (L) A set closed under LUB and GUB. That is, for every subset S from L, LUB(S) $\in L$ and GLB(S) $\in L$

Bottom ($\bot$) is an element such that $\forall x : \bot \sqsubseteq x$

Top ($\top$) is an element such that $\forall x : x \sqsubseteq \top$

Fixed Point x is a fixed point of function F such that $F(x) = x$
Monotone Function A function $F : L \to L$ is monotone if $x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)$

Knaster-Tarski Theorem (Fixed Point Theorem)

If L is a complete lattice, and $F : L \to L$ is monotonic, then the set of fixed points of F is a complete sub-lattice and

$$\bigsqcup_{n>0} F^{(n)}(\bot)$$

is the least fixed point of L.

This theorem is useful for static analysis since it gives us an algorithm to find the least fixed point. This gives us the most precise analysis for our rules.

Code Generation

We will generate IA-32 or i386 assembly instead of Java bytecode. IA-32 assembly is part of the x86 architecture family.

The general idea is that the compiler generates assembly files. An assembler generates object files from assembly files. The object files are then linked to form an executable.

IA-32 Assembly

There are 8 general purpose 32-bit registers (eax, ebx, ecx, edx, esi, esp, ebp).

There are 6 segment register (cs, ds, es, fs, gs, ss). These hold the segment addresses of various items, including code segment, data segment, video memory, stack segment, etc.

There are two main categories of x86 assembly syntaxes. There's the AT&T style which looks like

```
addl $4, %esp
```

We use the Intel style syntax which looks like,

```
add esp, 4
```

The assembler we use for this course is the Netwide Assembler (NASM).

Here are some examples of IA-32 instructions. Copy the contents of `ebx` into `eax`:

```
mov eax, ebx
```

Store the contents of `ebx` into the memory location specified by `eax` (ie. `*eax = ebx`):

```
mov [eax], ebx
```

Load the contents of `eax + ebx + 2` into `ebx`:

```
mov ebx, [eax + ebx + 2]
```

Move the address at the label into `eax`:

```
mov eax, label
```

Store the contents at the address given by the label into `eax`:

```
mov eax, [label]
```

When we do addition and subtraction operations, the result is stored in the register of the first operand. The following instructions store `eax + ebx` and `eax - ebx` into `eax` respectively:

```
add eax, ebx
sub eax, ebx
```

Multiplication and division make special use of registers `eax` and `edx`.

For branching we compare registers using: `cmp eax, ebx` `cmp` stores the result of the comparision into the status register. We can branch on results in the status register using `je`, `jne`, `jg`, etc.

We operate on the stack using `push` and `pop`. `push` decrements `esp` then stores into `esp`. `pop` loads a value from `esp` then increments `esp`. For example,

```
push eax
pop eax
```

`call` and `ret` are used for function calls. `call` pushes `eip` onto the stack and then branches:

```
call label
```

`ret` pops from the stack into `eip`. The `int` instruction generates an interrupt. Generally `int 0x80` is used to make a system call.

Assembler Directives

`global label` will cause `label` to have an external symbol definition in the object file. `extern label` declares a symbol that is defined in another module. The object file will contain an external symbol reference. The linker resolves these symbols.

`db`, `dw`, `dd` can be used to declare constants in the output file:

```
db    0x55                ; just the byte 0x55
db    0x55,0x56,0x57      ; three bytes in succession
db    'a',0x55            ; character constants are OK
db    'hello',13,10,'$'   ; so are string constants
dw    0x1234              ; 0x34 0x12
dw    'a'                 ; 0x61 0x00 (it's just a number)
dw    'ab'                ; 0x61 0x62 (character constant)
dw    'abc'               ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678          ; 0x78 0x56 0x34 0x12
```

Reads from memory are aligned on the size of the data bus. We can use `align c` to align instructions to a multiple of `c`.

Strategy for Code Generation

It's a good idea to plan out data layout before starting. For example, write methods to read/write a variable and call a method:

```
readVar(VariableDeclaration v, Register r);
callMethod(MethodDeclaraton m, Expression e...);
```

After that, we can do a tree traversal to generate the code. Typically, code is generated bottom up.

Data Layout

We need strategies on where to store,

- Constants
- Local variables
- Class data
  - Static fields
  - Method bodies
- Objects
  - Instance fields
  - Pointers to the class code
- Arrays
  - Array elements
  - Length of array

One way to make method calls is define a label for methods:

```
global MTH$Classname$method$
MTH $Classname$method$:
    add eax, ebx
    ...
```

To call the method:

```
extern MTH$Classname$method$
call MTH$Classname$method$
```

This is possible, however we run into problems due to inheritance. We don't want to generate duplicate code for inherited methods. Also, we need to perform dynamic dispatch of method calls for methods that are overrided.

Java primitive types have different sizes. For example `int` and pointers to objects are 32 bits while `char` is 16 bits. It may simplify the compiler design if we allocated 32 bits even for primitives that require less space. This avoids the need to worry about aligment.

Constants

We can deal with constants by assigning a label to constants:

```
mov eax, [CONST$uniqueLabel]
CONST$uniqueLabel:
    dd 42
```

Another alternative is to put the constant in the generated instruction:

```
mov eax, 42
```

Commercial compilers generally insert constants into the assembly instructions for efficiency.

Local Variables

We can put variables in dedicated registers. This creates fast generated code. However, a fallback is required since we may have more local variables than registers. Code complexity is increased since local variables can be either in a register or memory address.

Local variables cannot be put at fixed memory locations. Otherwise, variables would behave as local static variables in C. This method fails for recursion or in any other cases when multiple instances of a method is on the call stack.

The preferred method for local variables is to use the stack. This works for primitive types. Objects cannot live on the stack since they would be destroyed when the stack frame is popped. We allocate objects on the heap (using malloc) and use the stack to hold primitives a pointers to objects.

Let's consider how to generate code for the following:

```
void foo() {
    int x = 0;
    int y = 1;
    int z = 2;
    ...
}
```

We could simply generate the following code:

```
push dword 2   ; z = 2
push dword 1   ; y = 1
push dword 0   ; x = 0
```

However we run into problems if the variable initializer is an expression rather than a constant. Evaluating the expression may require use of the stack.

We may then consider allocating space on the stack ahead of time before evaluating the initializers. We store the value of the initializer using fixed offsets relative to esp.

```
sub esp, 12
mov [esp + 8], 0   ; x = 0
mov [esp + 4], 1   ; y = 1
mov [esp + 0], 2   ; z = 2
```

This still doesn't work since the stack pointer can be changed while evaluating an expression to store temporaries. The solution is to keep a frame pointer. The frame pointer is usually stored in ebp. We then reference variables on the stack relative to the frame pointer rather than the stack pointer.

```
push ebp           ; save the old frame pointer
mov ebp, esp       ; set the frame pointer
sub esp, 12        ; allocate the frame on the stack
mov [ebp - 4], 0   ; x = 0
mov [ebp - 8], 1   ; y = 1
mov [ebp - 12], 2  ; z = 2
...
add esp, 12        ; pop the stack frame
pop ebp            ; restore the old frame pointer
```

Method Calls

An application binary interface (ABI) needs to be decided for method calls. An ABI includes

- Data type sizes
- Alignment
- Calling convention
- System call numbers
- Registers which are saved

There are two approaches to saving registers: callee saves or caller saves. The advantage of callee saved is that only registers which are overwrriten need to be saved. The disadvantage is that the caller might not care if a certain set of registers are overwritten.

For caller saved registers, the reverse is true. Only registers that the caller needs preserved are saved. However, the caller might save registers that the callee never modifies. There is an extra benefit of caller saved registers. The caller ensures that its registers are not modified. This is useful to calling into untrusted code.

We'll look at two popular calling conventions.

In CDECL, registers are caller saved. The caller does the following:

- Push registers to stack.
- Arguments pushed to stack from right to left. This is done to support varargs. In C/C++, the order of evaluation in the argument list is undefined, but in Java expressions must be evaluated left to right.
- `call foo` which pushes `eip`.
- When the call returns, the return value is expected to be in `eax`.
- The caller pops the parameters and restores registers.

The callee does the following:

- Push the old base pointer (`push ebp`).
- Set the base pointer: `mov ebp, esp`. Arguments start at `ebp + 8`. Note that `push` decrements the stack pointer before storing. Also after pushing the arguments, `call` pushes `eip` and we just pushed `ebp`.
- Allocate space for local variables (`sub esp, 24`).
- Execute code for function.
- Ensure return value is in `eax` and then deallocate local variables (`add esp, 24`).
- Pop the old base pointer (`pop ebp`)
- `ret`

For example `foo(1, 2, 3)` is translated to the following:

```
push ebx        ; Assume we need to save ebx and ecx
push ecx
```

```
push dword 3    ; Push arguments
push dword 2
push dword 1
call foo        ; Return value should now be in eax
add esp, 12     ; Pop arguments from the stack
pop ecx         ; Restore the registers we pushed
pop ebx
add eax, ebx    ; We can use the return value
```

For the function

```
int foo(int a, int b, int c) {
  int i = 0;
  return a;
}
```

the compiler could generate the following:

```
push ebp           ; Save the base pointer
mov ebp, esp       ; Set the new base pointer
sub esp, 4         ; Allocate 4 bytes for the locals (int i)
mov [ebp + 0], 0   ; Initialize i to 0
mov eax, [ebp + 8] ; Set return value to a (b would be in ebp + 12)
mov esp, ebp       ; Deallocate locals (alternatively `sub esp, 4`)
pop ebp            ; Restore the base pointer
ret                ; Pops into eip
```

In the Pascal calling convention, the caller arguments are pushed left to right. However, the caller is expected to pop the arguments. This leads to uneven pushes and pops.

Object Layout

When we do `new Object()`, we need to allocate and initialize memory for our new object. An object needs to hold instance data and a reference to its concrete class.

A class holds static fields and method implementations. This is complicated by inheritance. Methods and fields can be inherited or overridden.

Consider the following classes:

```
public class Vehicle {
  public int numTires() {
    return 0;
  }
  public void foo() { }
}

public class Car extends Vehicle {
  public void bar() { }
  public int numTires() {
    return 4;
  }
}
```

Generating the code to call methods of `Car` knowing that it doesn't have subclasses is not too difficult. We can resolve methods statically:

```
Car car = new Car();
car.foo();       // call MTH$Vehicle$foo
car.numTires();  // call MTH$Car$numTires
```

However consider generating method calls for `Vehicle`:

```
Vehicle vehicle = new Car();
vehicle.numTires();  // call MTH$Vehicle$numTires? nope
```

We cannot statically dispatch `vehicle.numTires()` since `numTires()` may be overriden in a subclass.

Vtables

The solution is to use a vtable to dynamically dispatch methods. The idea is that each class has an array of pointers to method implementations called a *vtable*. Each object contains a pointer to the start of the vtable. The address of a method is dynamically resolved at runtime using the vtable of the object.

```
; assume that eax has the address of the object
; and a pointer to the vtable is at the beginning of the object
mov eax, [eax]          ; Load address of vtable
mov eax, [eax + offset] ; Load a method based on its compile time offset
call eax
```

We must ensure that inherited methods are at the same offset subclasses. Otherwise, the wrong method will be dispatched when calling a method on the subclass.

For example if we have the vtable for `Vehicle`,

```
VTABLE$Vehicle:
    dd METHOD$Vehicle$numTires
    dd METHOD$Vehicle$foo
```

We need to have the following vtable for `Car`,

```
VTABLE$Car:
    ; These inherited methods must be in the same order as the superclass
    dd METHOD$Car$numTires
    dd METHOD$Vehicle$foo
    ; Declared methods can be in any order
    dd METHOD$Vehicle$bar
```

The key idea for inheritance is that a subclass must appear like a superclass. We must have consistency of layout. The same principle applies to fields.

Dispatching Interface Methods

Now let's consider resolving interface methods. Consider the following classes:

```
interface A {
  ma();
}

interface B {
  mb();
}

class C implements A, B {
   ma() {...}
```

```
    mb() {...}
}
```

Where do we put `ma()` and `mb()` the the vtable for class `C`? Both interfaces `A` and `B` want their function at offset 0 in the vtable.

A solution is to fix offsets for all interface methods globally. We can then generate a table for each class containing all interface methods. If the method is defined, then the address of the implementation is set in the table. Otherwise, the value in the table is garbage.

The table is of quadratic size (number of methods times number of classes). We can perform sparse array optimizations or combine classes which have the same implementations for all interface methods.

More abstractly, this approach defines a function:

$$f : \underbrace{C}_{\text{concrete class}} \times \underbrace{S}_{\text{method}} \to \underbrace{I}_{\text{pointer to implementation}}$$

So instead of using a simple 2D array, we could use a hashtable.

We use the interface table for method calls on an interface and the vtable for method calls on classes.

Subtype Testing

It is sometimes necessary to check subtype at runtime:

- `obj instanceof A`
- `(A) obj`
- `array[0] = new A()` may throw `ArrayStoreException`

Subtype testing is the problem: "Given an object O of some type A, is $A :=$ typeof(O)?"

If we only had single inheritance, we could generate intervals for a given type through tree traversal of the class hierarchy at compile (see CS 341). Runtime typechecking could be performed by comparing the intervals. A class C is a subclass of class P if and only if the interval of C is contained in the interval of P.

However this approach doesn't work for Java due to interfaces. We could perform subtype checking in a similar manner to interface method dispatching. We generate a huge table for whether each class is a subclass of another. Similar optimations could be performed as the interface method table.

Arrays

Arrays in Java are instances of `Object` and implement `Cloneable` and `Serializable`. Then arrays still need to hold a vtable and interface method table. Arrays also need to hold information for type checking and to check for an `ArrayStoreException` when assigning.

An `ArrayStoreException` is thrown since Java's array type system is not fully sound:

```
B[] bs = new B[1];
Object[] os = bs;
os[0] = new C();  // throws ArrayStoreException
```