richardwu.ca

CS 444/644 Course Notes

Compiler Construction

Ondřej Lhoták • Winter 2019 • University of Waterloo

Last Revision: April 16, 2019

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 January 7, 2019

1.1 Basic overview of a compiler

A **compiler** takes a source language and translates it to a target language. The source language could be, for example, C, Java, or JVM bytecode and the target language could be, for example, machine language or JVM bytecode.
A compiler could be divided into two parts:

Front-end analysis Front-end could be further divided into two parts: the first being scanning and parsing (assignment 1) and the second being context-sensitive analysis (assignment 2,3,4).

Some refer to context-sensitive analysis as "middle-end".

Back-end synthesis The backend could also be divided into two parts: the first being optimization (CS 744) and the second being code generation (assignment 5).

1.2 Overview of front-end analysis

Goal: is the input a valid program? An auxiliary step is to also generate information about the program for use in synthesis later on.
There are several steps in the front-end:

Scanning Split sequence of characters into sequence of tokens. Each token consists of its lexeme (actual characters) and its kind.

There are tools for generating the DFA expressions from a regular language e.g. lex.

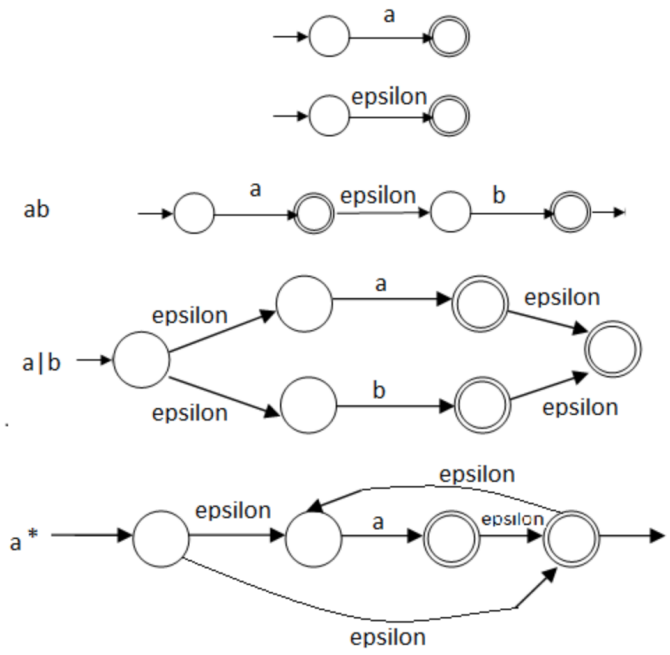
2 January 9, 2019

2.1 Scanning tools (lex)

Our goal is to specify our grammar in terms of regular expressions (regex) which lex can convert into a scanning DFA.
Review of regex to language (set of words):

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

The corresponding NFAs we can construct per each regex rule are



Let Σ be the set of characters, Q the set of states, q_0 the initial state, A the accepting states, and δ the transition function, an NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$ where

$$\text{NFA} : \delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q \text{ (subset of Q)}$$
$$\text{DFA} : \delta : Q \times \Sigma \to Q$$

i.e. the transition function of an NFA returns a subset of states in Q.
We note in our above NFAs some accepting states are equivalent (connected by an ϵ transition).
We define

Definition 2.1 (ϵ-closure I)**.** The ϵ-closure(S) of a set of states S is the set of states reachable from S by (0 or more) ϵ-transitions.

Another equivalent recursive definition

Definition 2.2 (ϵ-closure II)**.** Smallest set S' such that

$$S' \supseteq S$$
$$S' \supseteq \{q \mid q' \in S', q \in \delta(q', \epsilon)\}$$

We note that any NFA can be converted in a corresponding DFA. The input is an NFA $(\Sigma, Q, q_0, A, \delta)$ and we'd like to get a DFA $(\Sigma, Q', q_0', A', \delta')$ where

$$q_0' = \epsilon\text{-closure}(\{q_0\})$$
$$\delta'(q', a) = \epsilon\text{-closure}\Big(\bigcup_{q \in q'} \delta(q, a)\Big)$$

we note that each state of our DFA is a set of states in the original NFA e.g. $\{1, 2, 4\}$ may be a state in the DFA.

We generate more states in our DFA by iterating through every $a \in \Sigma$ and applying rule two to our initial state q'_0. We do this until no further states can be generated from existing DFA states and all $a \in \Sigma$ have been exhausted. We note that the entire set of states Q' in the DFA can be recursively defined as the smallest set of subsets of Q such that

$$Q' \supseteq \{q'_0\}$$
$$Q' \supseteq \{\delta'(q', a) \mid q' \in Q'\} \qquad \forall a \in \Sigma$$

We note that if any accepting state is included in a state of the DFA, we can accept it (since we can reach the corresponding accepting state in the NFA). Thus

$$A' = \{q' \in Q' \mid q' \cap A \neq \varnothing\}$$

3 January 14, 2019

3.1 DFA recognition scanning

The algorithm for using a DFA to recognize if a word is valid in the grammar

Algorithm 1 DFA recognition

input word w, DFA $M = (\Sigma, Q, q_0, \delta A)$
output boolean $w \in L(M)$?

```
q ← q_0
for i from 1 to |w| do
    q ← δ(q, w[i])
return q ∈ A
```

Scanning is similar where we take a *sequence of symbols* and convert it into a *sequence of tokens* using a DFA. In **maximal munch** we have

Algorithm 2 Maximal munch (abstract)

input DFA M specifying language L of valid tokens, string of symbols w
output sequence of tokens, each token $\in L$ that concantenates to w

```
while until end of output do
    Find a maximal prefix of remaining input that is in L
    ERROR if no non-empty prefix in L
```

note that **maximal munch** takes the **maximal prefix**: if we do not take the maximal prefix we may run into ambiguity. A more concrete implementation

Algorithm 3 Maximal munch (concrete)

```
while until end of output do
    Run DFA and record last seen accepting state until it gets stuck (or it transitions to ERROR state)
    Backtrack DFA and the input to last seen accepting state
    ERROR if there is no accepting state
    Output prefix as the next token
    Set DFA back to start state
```

Note that in Java which uses maximal munch, $a - -b$ would be parsed as $a(- - b)$ which would return a parsing error (as opposed to $a - (-b)$ since $--$ is a token in Java).

3.2 Constructing the scanning DFA

The overall algorithm to convert regular expressions denoting tokens to a scanning DFA is

Algorithm 4 Regex to scanning DFA

input REs $R_1, \ldots, R_n$ for token kinds in priority order
output DFA with accepting states labelled with token kinds
1: Construct an NFA M for $R_1 \mid R_2 \mid \ldots \mid R_n$
2: Convert NFA to DFA M' (each state of M' is set of states in M)
3: For each accepting state of M', output highest priority token kind of the set of NFA accepting states

4 January 16, 2019

4.1 Context-free grammar

Regular expressions are great for scanning but would not work for an arbitrary depth of tokens when it comes to parsing grammar.
To address this we define **context-free grammars** which uses recursion to specify arbitrary structures in the grammar at an arbitrary depth in the parse tree.

Definition 4.1 (Context free grammar)**.** A **context-free grammar** is a 4-tuple $G = (N, T, R, S)$ where we have (NB: notation for each class)

Terminals T (e.g. a, b, c; denoted with lowercase alphabet)

Non-terminals N (e.g. A, B, C, S; denoted with uppercase alphabet)

Symbols The set of symbols are $V = N \cup T$ (e.g. W, X, Y, Z)

String of terminals T^* (e.g. w, x, y, z)

String of symbols V^* (e.g. α, β, γ)

Production rules $R \subseteq N \times V^*$ (e.g. $A \to \alpha$)

Start non-terminal S

Definition 4.2 (Directly derives)**.** $\beta A \gamma \Rightarrow \beta \alpha \gamma$ if $A \to \alpha \in R$: that is $\beta A \gamma$ **directly derives** $\beta \alpha \gamma$.

Definition 4.3 (Derives)**.** $\alpha \Rightarrow^* \beta$ if $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$: that is α **derives** β.

Definition 4.4 (Sentential form)**.** α is a **sentential form** if $S \Rightarrow^* \alpha$.

Definition 4.5 (Sentence)**.** x is a **sentence** if $x \in T^*$ and x is a sentential form.

Definition 4.6 (Language)**.** $L(G) = \{x \in T^* \mid S \Rightarrow^* x\}$ is the **language generated by** G (set of sentences).

4.2 Recognizer and parsing

The task of a **recognizer** is to see if $x \in L(G)$ for some grammar G.
The task of **parsing** is to find a derivation from S to x.

Example 4.1. Suppose we have a grammar G with $R = \{A \to BgC, B \to ab, C = ef\}, S = A, N = \{A, B, C\}, T = \{a, b, e, f, g\}$.
We can derive the following sentence

$$A \Rightarrow BgC \Rightarrow abgC \Rightarrow abgef$$

notice this is the only sentence in $L(G)$.
We could represent this as a tree where A is at the root, B, g, C are each children of A (3 children), a, b and e, f are children of B and C, respectively.
Note that for this particular grammar, we may have multiple derivations for $abgef$ (we could have expanded C first) but we have one unique parse tree.

Definition 4.7 (Ambiguous grammar)**.** A grammar is **ambiguous** if $\exists > 1$ parse tree for the same sentence.

Definition 4.8 (Left(Right) derivation)**.** In a **left(right) derivation** we always expand the left(right)-most non-terminal.

There is a *one-to-one correspondence* between parse trees, left derivations, and right derivations: that is given a sentence with a unique parse tree, it has a unique left derivation and a unique right derivation.

4.3 Top-down parser

The simplest approach for parsing a sentence x from S (start symbol) is using a **top-down parser**:

Algorithm 5 Top-down parser

1: $\alpha \leftarrow S$
2: **while** $\alpha \neq x$ **do**
3: Replace **first** non-terminal A in α with β, assuming $A \to \beta \in R$

Replacing the *first* non-terminal results in a left derivation.

5 January 21, 2019

5.1 LL(1) parser

Example 5.1. Given a grammar with the following production rules

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

Suppose we wanted to derive $a + a$. Intuitively we have $E \Rightarrow aE' \Rightarrow a + a$. Our intuition told us to use $E' \to +a$ instead of $E' \to' \epsilon$.

To improve on our top-down parsing, we introduce the LL(1) parser:

Algorithm 6 LL(1) parser

input x is the input string to parse
1: $\alpha \leftarrow S$
2: **while** $\alpha \neq x$ **do**
3: Let A be the first non-terminal in α ($\alpha = yA\gamma$)
4: Let a be the first terminal after y in x ($x = ya\zeta$)
5: $A \to \beta \leftarrow \text{predict}(A, a)$
6: Replace A with β in α

where predict(A, a) follows our intuition of picking the rule with A on the LHS that works best when a is the next terminal: our **lookahead**.

Remark 5.1. The first "L" represents scanning the **input from left to right**; the second "L" denotes **left-canonical derivation** (always expanding the leftmost non-terminal); the "1" denotes a 1 **symbol lookahead**

Definition 5.1 (Augmented grammar)**.** In order to make our 1 lookahead algorithm work with the start rule, we need to augment it:

1. Add a fresh terminal "$"
2. Add production $S' \to S\$$ where S' is our new start and S was the start of our original grammar.

We also need to append "$" to our input. Our augmented grammar becomes

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

If we implement $LL(1)$ naively with string replacements we could see that the algorithm runs in $O(n^2)$.
To do this in $O(n)$ time we could utilize a stack for our α derivation. Our revised algorithm proceeds as

Algorithm 7 LL(1) parser

input x is the input string to parse
1: Push $S\$$ onto stack
2: **for** a in $x\$$ **do**
3: **while** top of stack is a non-terminal A **do**
4: Pop A
5: Find $A \to \beta$ in predict(A, a) or ERROR
6: Push β
7: Pop b, terminal on top of stack
8: **if** $b \neq a$ **then**
9: ERROR

We define

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

A grammar is $LL(1)$ iff $|\text{predict}(A, a)| \leq 1$ for all A, a.

5.2 First, nullable and follow sets

Definition 5.2 (First set)**.** If $\beta \Rightarrow^* a\gamma$ then $a \in \text{first}(\beta)$ where first(β) is the **first set** of β.

Definition 5.3 (Nullable set)**.** If $\beta \Rightarrow^* \epsilon$ then β is **nullable**.

Definition 5.4 (Follow set)**.** If $S' \Rightarrow^* \gamma A a \delta$ then $a \in \text{follow}(A)$ where follow(A) is the **follow set** of A.

Note that in our above example grammar, we have

$$\text{nullable} = \{\epsilon\}$$

and

$$\begin{aligned}\text{first}(\epsilon) &= \{\}\\ \text{first}(+a) &= \{+\}\\ \text{first}(aE') &= \{a\}\\ \text{first}(E\$') &= \{a\}\end{aligned}$$

In general to compute the **follow set**, we define it as

1. If $B \to \alpha A \gamma$ then $\text{first}(\gamma) \subseteq \text{follow}(A)$
2. If $B \to \alpha A \gamma$ and $\text{nullable}(\gamma)$ then $\text{follow}(B) \subseteq \text{follow}(A)$

6 January 23, 2019

6.1 Note on LL(k) parsers

Note that in LL(k) parsers we perform leftmost derivation. This means that the sentence $3-2-1$ would be parsed (with parentheseses denoting derivations/subtrees) $3-(2-1)$ which would incorrectly result in 2.
Ideally we wanted $(3-2)-1$ or **left associative**. In general, LL(k) parser cannot parse arbitrary left associative languages.

Remark 6.1. One could imagine an LL(1) parser as generating the parse tree from top to bottom, left to right where the parse tree is rooted at S and the leaves are the tokens in the input x.

An alternative to LL(k) parsers which is a top-down parser (from start symbol S to input) are bottom-up parsers.

6.2 Bottom-up parsing

Example 6.1. Suppose we have the same grammar as before

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

Suppose we wanted to parse the sentence $a + a\$$. We proceed in a bottom up fashion

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

where we scan from left to right the tokens in the input and reduce with "intuition".

Remark 6.2. Given a parse tree rooted at S with leaves as tokens of input x, a bottom-up parser generates the parse tree from the bottom-left corner and moves upwards and rightwards until the start symbol is reached.

proceed to look at a particular bottom-up parser.

6.3 LR(0) parser

The best way to implement these parsers is to think about invariants and ensuring our algorithm always maintains the invariants:

LL(k) invariant The invariant for an LL(k) parser is that

$$S \Rightarrow^* \alpha$$
$$S \Rightarrow^* \text{ seen input } + \text{ stack}$$

i.e. S always derives the string of processed symbols we've derived so far.

LR(k) invariant Here the invariant is that

$$\alpha \Rightarrow^* x$$
$$\text{stack } + \text{ unseen input } \Rightarrow^* x$$

that is our current string of processed symbols plus the rest of the unseen input should be able to derive our entire input. Equivalently

$$S \Rightarrow^* \alpha$$
$$S \Rightarrow^* \text{ stack } + \text{ unseen input}$$

in other words we require the stack to always be a viable prefix:

Definition 6.1 (Viable prefix)**.** α is a **viable prefix** if it is a prefix of some sentential form i.e.

$$\exists \beta \text{ s.t. } S \Rightarrow^* \alpha\beta$$

In LR(0) parsing as above, we keep a stack of symbols we have processed so far. We have the following algorithm

Algorithm 8 LR(0) parser

```
1: for a in x$ do
2:     while Reduce(stack) = {A → γ} do
3:         Pop γ off stack (i.e. pop |γ| tokens)          ▷ Reduce
4:         Push A onto stack
5:     if Reject(stack + a) then ERROR
6:     Push a onto stack                                  ▷ Shift
```

where we define

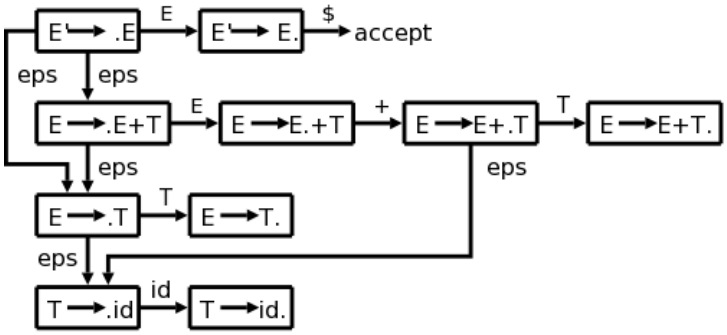
$$\text{Reduce}(\alpha) = \{A \to \gamma \mid \exists \beta \text{ s.t. } \alpha = \beta\gamma \text{ and } \beta A \text{ is a VP}\}$$

where VP stands for **viable prefix** (see above) and Reject(α) is true if α is not a VP.

Question 6.1. How do we check if α is a VP? Note that the set of *all* VPs is itself a context-free language.

Exercise 6.1. Given a CFG G, construct a CFG G' for the set of VPs of G.

Remark 6.3. Only some VPs α's occur during the algorithm, thus we can build a simpler NFA for VPs that actually occur.
We can construct the NFA (such as the following) systematically from a given grammar



We define the construction of the LR(0) NFA as follows:

$\Sigma = T \cup N$
$Q = \{A \to \alpha \cdot \beta \mid A \to \alpha\beta \in R\}$
$q_0 = S' \to S\$$
$A = Q$ all states are accepting i.e. stack is always a VP
$\delta(A \to \alpha \cdot B\beta, \epsilon) = \{B \to \cdot\gamma \mid B \to \gamma \in R\}$
$\delta(A \to \alpha \cdot X\beta, X) = \{A \to \alpha X \cdot \beta\}$

7 January 28, 2019

To parse, we can simply just follow the NFA and reduce if the NFA ends up in state $A \to \gamma\cdot$ on input $\beta\gamma$ (note: Reduce($\beta\gamma$) contains $A \to \gamma$ because the NFA would also accept βA; a VP is always kept on the stack).
However, to make our process deterministic, we need to convert the NFA to a DFA.

Definition 7.1 (Reduce-reduce conflict (LR(0)))**.** If the resulting DFA state contains $A \to \gamma\cdot$ and $B \to \delta\cdot$ for two different productions, a **reduce-reduce conflict** occurs.

Definition 7.2 (Shift-reduce conflict (LR(0)))**.** If the DFA state contains $A \to \gamma\cdot$ and $B \to \alpha \cdot X\beta$, then a **shift-reduce conflict** occurs.

Definition 7.3 (LR(0) grammar)**.** An LR(0) grammar is **unambiguous** if there are no conflicts using the above NFA and DFA construction.

Remark 7.1. Instead of walking the DFA and returning the start state every time a reduction happens, we can **optimize** this procedure to be *linear time* using an additional stack for DFA states. The invariant is that $\delta(q_0,$ processed stack$) =$ top of state stack i.e. the top of our state stack is always our current state in the DFA. We also need to synchronize the number of times we push and pop from both stacks (e.g. a reduction rule that pops 3 symbols would require popping the state stack 3 times).

Example 7.1. Here is an example of LR(0) parsing based on the DFA states:

20

Example LR Parsing

Grammar:
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ **id**

Stack	Input	Action
$ 0	**id*id+id$**	shift 5
$ 0 **id** 5	***id+id$**	reduce 6 goto 3
$ 0 *F* 3	***id+id$**	reduce 4 goto 2
$ 0 *T* 2	***id+id$**	shift 7
$ 0 *T* 2 * 7	**id+id$**	shift 5
$ 0 *T* 2 * 7 **id** 5	**+id$**	reduce 6 goto 10
$ 0 *T* 2 * 7 *F* 10	**+id$**	reduce 3 goto 2
$ 0 *T* 2	**+id$**	reduce 2 goto 1
$ 0 *E* 1	**+id$**	shift 6
$ 0 *E* 1 + 6	**id$**	shift 5
$ 0 *E* 1 + 6 **id** 5	**$**	reduce 6 goto 3
$ 0 *E* 1 + 6 *F* 3	**$**	reduce 4 goto 9
$ 0 *E* 1 + 6 *T* 9	**$**	reduce 1 goto 1
$ 0 *E* 1	**$**	accept

7.1 LR(1) parser

For the grammar

$$S \rightarrow E\$$$
$$E \rightarrow a + E$$
$$E \rightarrow a$$

we end up having $E \rightarrow a \cdot +E$ and $E \rightarrow a\cdot$ in the same DFA state (reduce-shift conflict), therefore the grammar is NOT LR(0).
Is there a way to disambiguate the grammar? Surely if we could *lookahead one token* to see if either the next token of the input is \$ (then we would reduce with $E \rightarrow a\cdot$) or if it is + (then we would shift +), then we could remedy our conflict.
This is the idea behind **LR(1) parsing**: LR parsing with 1 lookahead token. The algorithm is similar to LR(0) parsing:

Algorithm 9 LR(1) parser

```
1: for a in x$ do
2:     while Reduce(stack, a) = {A → γ} do
3:         Pop γ off stack (i.e. pop |γ| tokens)          ▷ Reduce
4:         Push A onto stack
5:     if Reject(stack + a) then ERROR
6:     Push a onto stack                                  ▷ Shift
```

where we define our new Reduce function with a lookahead token a:

$$\text{Reduce}(\alpha, a) = \{A \rightarrow \gamma \mid \exists \beta \text{ s.t. } \alpha = \beta\gamma \text{ and } \beta A a \text{ is a VP}\}$$

7.2 SLR(1) parser

How do we determine if βAa is a VP for a reduction rule $A \to \gamma\cdot$?

Option 1 (SLR(1)) If $a \notin \text{follow}(A)$ then βAa definitely is not a VP

Option 2 (LR(1)) Construct a better DFA (discussed later)

For **Option 1** (called **SLR(1)**) we can combine both the DFA for LR(0) and our follow set:

- Use LR(0) DFA to decide if βA is a VP
- Use follow(A) to decide if βAa is a VP

We redefine what it means for there to be conflicts in our SLR(1) DFA:

Definition 7.4 (Reduce-reduce conflict (SLR(1)))**.** If the resulting DFA state contains $A \to \gamma\cdot$ and $B \to \delta\cdot$ for two different productions, *and* $\text{follow}(A) \cap \text{follow}(B) \neq \varnothing$ then a **reduce-reduce conflict** occurs.

Definition 7.5 (Shift-reduce conflict (SLR(1)))**.** If the DFA state contains $A \to \gamma\cdot$ and $B \to \alpha \cdot X\beta$, *and* $X \in \text{follow}(A)$ then a **shift-reduce conflict** occurs.

Remark 7.2. SLR(1) uses the same parser as LR(1): the only difference is how the two checks if βAa is a VP.

7.3 Distinction between LR(0), SLR(1), LALR(1), LR(1)

The distinction between different types of parser depends on how it determines VPs (ranked from most general to most specific; prior types imply later types):

General Determines if $\beta A\gamma$ is a sentential form (where γ is the rest of input)

LR(1) Determines if βAa is a VP

LALR(1) Determines if βA is a VP and $a \in \ldots$??? (discussed later)

SLR(1) Determines if βA is a VP and $a \in \text{follow}(A)$

LR(0) Determines if βA is a VP

8 January 30, 2019

8.1 Generating the LR(1) parse table

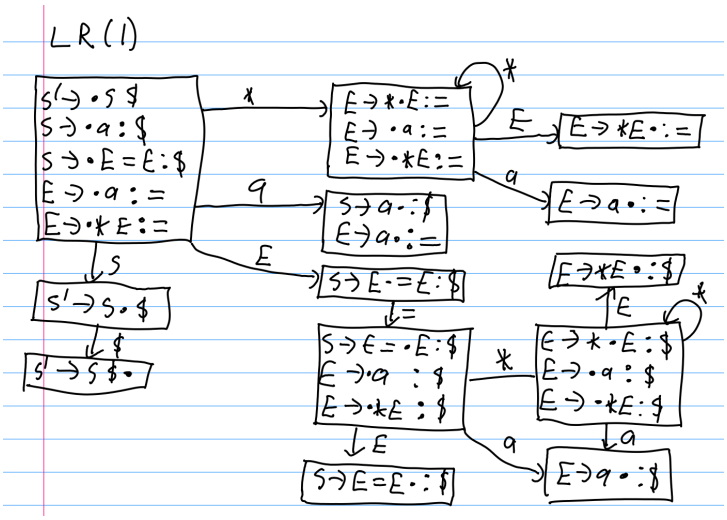
As noted previously, the difference between LR(1) and SLR(1) is how they determine whether a proposed prefix is a **viable prefix**.

Example 8.1. The following grammar is LR(1) but not SLR(1):

$$S' \to S\$$$
$$S \to a$$
$$S \to E = E$$
$$E \to a$$

Note that given input $a\$$, SLR(1) will try to determine whether to reduce a to S or E. It will check if E is a VP and if the lookahead token, \$, is in follow(E). Similarly for S.
Note that in SLR(1) both E and S are viable candidates but only S is the correct reduction since if we reduce to E we would require an $=$ afterwards.

The **idea** behind LR(1) (vs SLR(1)) is to keep *specific follow sets* in each NFA state. In the example above, we would keep track of the specific follow set of the first E_1 and for the second E_2 in their corresponding NFA states. When we are constructing the NFA, if we are in an NFA state where we have a bullet point $\cdot$ in front of a non-terminal (e.g. $S \to \cdot E = E\$$), we generate an ϵ transition out of that state (e.g. to $E \to \cdot a\$$).
For the new generated state we annotate it with the follow set of the specific instance of the non-terminal (in this case it is the first E so we annotate the new state with $=$).
Finally once we convert the NFA to a DFA we end up with something like this:



where each state is annotated with the lookahead symbol from the follow set.

Remark 8.1. The lookahead generated from the follow sets only matter for states with *reductions*; however, to carry through the follow set lookaheads we need to include them in intermediary state during generation.

That is: if NFA ends up in a state $A \to \gamma \cdot : a$ on a stack $\beta\gamma$, then reduce$(\beta\gamma, a)$ contains $A \to \gamma$ because the NFA would also accept $\beta A a$.
This answers the question: is $\beta A a$ a VP?

Definition 8.1 (Reduce-reduce conflict (LR(1)))**.** If DFA state contains $A \to \gamma \cdot : a$ and $B \to \delta \cdot : a$ then we have a **reduce-reduce conflict**.

Definition 8.2 (Shift-reduce conflict (LR(1)))**.** If DFA state contains $A \to \gamma \cdot : a$ and $B \to \alpha \cdot a\beta : b$ then we have a **shift-reduce conflict**.

To build the LR(1) NFA we define it declaratively as:

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

9 February 4, 2019

9.1 LALR(1) grammar

Idea: we use the LR(0) DFA with follow sets local to each LR(1) DFA state.

Definition 9.1 (Core of a state)**.** The **core** of a LR(1) DFA state is defined as

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

i.e. the set of items belonging to the state q.

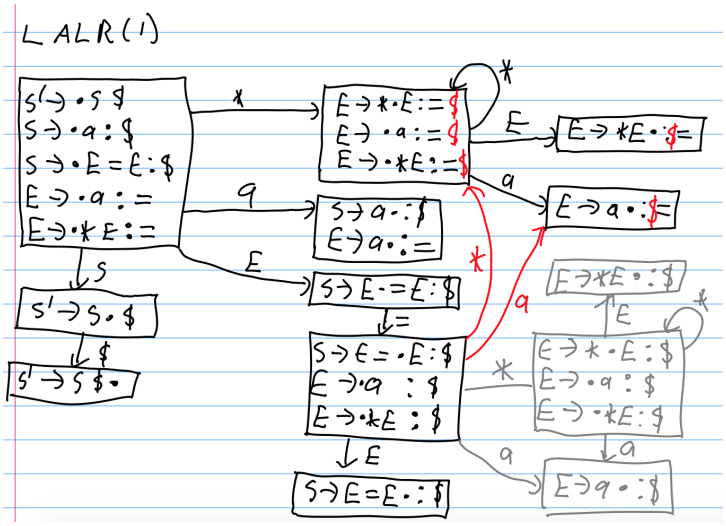
Fact 9.1. We can replace each LR(1) DFA state by its **core** i.e. its LR(0) DFA state.

The algorithm to convert an LR(1) DFA's states to LALR(1) DFA states:

Algorithm 10 LR(1) to LALR(1)

input M is the LR(0) DFA, M' is the LR(1) DFA
1: **for** each state q of M **do**
2: **for** each item $A \to \alpha \cdot \beta$ of q **do**
3: $l' \leftarrow \{a \mid q' \text{ is a state of } M', q \in core(q'), (A \to \alpha \cdot \beta : a) \in q'\}$
4: q with lookaheads l' is the new LALR(1) state/core

That is: we simply coalesce the item sets and merge (take the union) of the lookahead tokens per each item/production rule. An example of a transformation to LALR(1):



9.2 Abstract syntax tree

The grammar for the abstract syntax tree can be ambiguous (i.e. we can have multiple parse trees for a given sentence in the AST). This allows us to keep the AST grammar simpler.
The AST is built recursively from the parse tree. We define a new type for each kind of AST node (e.g. `Expression`, `Statement`, etc.).
The later phases of the compiler will compute things about the AST nodes in order to do semantic analysis.

Remark 9.1. We could theoretically proceed with just the parse tree from e.g. LR(1) parsing instead of building an AST, but building an AST helps us simplify concepts and reduces the complexity of pattern matching.
E.g. instead of having to match `Term (Factor * Factor)` for an arithemetic expression we can simply match `Expression`.

9.3 Weeding

Weeding permits us to verify invariants that may be easier by traversal of the AST. For example in Java (Joos) a class cannot be both abstract and final. While we can specify this in the grammar and catch it during parsing, it may be easier to simply traverse the AST and verify the two modifiers do not exist underneath a given class declaration.

Definition 9.2 (Weeding)**. Weeding** is the process of checking the language requirements that could have been enforced by grammars but not easily in code.

9.4 Context-sensitive/semantic analysis (middle end)

The next phase of the compiler is to associate identifiers and nodes in the AST with their semantic meaning.
We will first look at **declarations** and and **usages**. Take the following example:

```
1  class A {
2    public int x;
3    public boolean y;
4    public void m(int y) {
5      x = y+1;
6    }
7  }
```

We note that there are declarations A, x, y, m, y (paramaeter). We can map the usages of every variable to their corresponding declarations intuitively.
Things can get even more complicated with subclasses and nested declarations:

```
1  class B extends A {
2    String z;
3    public void m(boolean b) {
4      y = b;
5      {
6        int y = 42;
7        x = y;
8        (new B()) m(y);
9      }
10     {
11       String y = "foo";
12       y = y + z;
13     }
14   }
15 }
```

We note that the declaration of m is separate from the declaration of m in `class A`. We also note the reference of y in the first bracketed scope is to `int y` and not `public boolean y` declared in A.
Finally, note that the usage of m in the first bracketed scope references `public void m` declared in `class A`: we infer this by the fact that its reference to y is a reference to the y declared as `int y`.

10 Februrary 11, 2019

10.1 Name resolution

The goal of name resolution is to **link usages of names with their corresponding declarations**.

Definition 10.1 (Scope)**.** A **scope** is an area of a program where a declaration **has effect** and **is visible**.

Definition 10.2 (Environment)**.** An **environment** is a map from names (strings) to declarations/meanings. During implementation, one can either map the names to the AST declaration node or to some other data structure used for semantic anaylsis.

Note that a *new scope* should correspond to a *new environment*. We form a stack of environments as we traverse through scopes.
How would operations on our environment look as we traverse our AST?

Declaring a name If we find a declaration for a name, we:

1. Search for name in current environment
2. If name already exists, ERROR
3. Else insert name into environment

Name lookup/resolution To resolve a usage:

1. Search innermost environment
2. If not found, search recursively in enclosing environments
3. If not found in any enclosing environments, ERROR

We look at an example where **namespaces** come into play:

Example 10.1. Suppose we have the following code snippet

```
1 int x = 0;
2 int x (int x) { ... }
3 x = x(x);
```

We note that in the third line, the three `x`'s refer to `int x`, `int x (int x)`, and `int x`, respectively.
For the declaration `int x` and `int x (int x)`, we note that they may both be declared within the same scope, howeer their literal identifier collide. We therefore must introduce **namespaces** for variables and methods in *separate environment*. We **syntactically determine** which namespace a declaration belongs to.

In Java (JLS 6.5.1) we have 6 namespaces:

1. package
2. type (class, interface)
3. method
4. expression (variable, parameter, field)
5. package $\cup$ type

   For example, given `import P.C` and assuming we infer that `P.C` is a class, `P` could either be a package or an (outer) class.

   Outer classes and thus this namespace is not required in Joos.
6. ambiguous

10.2 Name resolution in Java

While the general idea of name resolution can be applied to all languages, in Java specifically we can perform name resolution as follows:

1. Build global environment (set of classes)
2. Resolve type names (type namespace)
3. Check class hierarchy (inheritance)
4. Disambiguate namespaces of ambiguous names (since we have a hierarchy now)
5. Resolve "expressions" (variables, static fields)
6. Type checking
7. Resolve methods and instance fields

Note that in Java (and Joos), we are able to topologically perform the above steps in order. Some other languages have mutual dependency between the above stepsand thus name resolution will need to be performed recursively. Assignment 2 consists of the first 3 steps and assignment 3 consists of the last 4 steps.

10.3 Building the global environment

The global environment should record all class names along with their corresponding package names from the files passed to the compiler for linking.

Remark 10.1. In a normal Java compiler, it is only necessary to specify the single Java file to be compiled and the compiler will automatically resolve imported packages and classes.

10.4 Resolving type names

There are two ways to reference a name:

Qualified names always have `.` in their names (e.g. `a.b.c.d`).

We simply traverse the sequence of names listed starting from the top-level name.

Remark 10.2. If there is a usage `c.d` and a single-type import `a.b.c`, then `a.b.c.d` will never be resolved to `c.d`.

Simple names have no `.` in their names. We traverse the namespaces in the following priority order:

1. Enclosing class/interface
2. Single-type imports (e.g. `import a.b.c`)
3. Type in same package
4. Import-on-demand package (e.g. `import a.b.*`)

Remark 10.3. If there is any ambiguity within any of those namespaces (e.g. two type `c` imported by two single-type imports) then we raise a compile error.

Remark 10.4. The default package (packages without a name e.g. package containing `main`) is **NOT** the root package.

Therefore one would need to prefix the default package with some unique package name to avoid usages referencing the default package in other packages.

10.5 Simple class hierarchy checks (JLS 8,9)

We first verify class declarations. Some simple constraints in JLS 8 and 9:

1. For `class A extends B`, `B` must be a **class** (8.1.3)
2. For `class C implements D`, `D` must be an **inferface** (8.1.4)
3. No duplicate interfaces e.g. `class E implements F, F` (8.1.4)
4. For `class A extends B`, `B` cannot be `final` (8.1.3)
5. Constructors for the same class must have distinct signatures (i.e. parameter types) (8.8.2)

11 February 13, 2019

11.1 Formal constructs for class hierarchy

We attempt to define a formal model for inheritance and the class hierarchy. Consider the following examples:

Example 11.1. Given `class A extends B implements C, D, E` then we define

$$\text{super}(A) = \{B, C, D, E\}$$

which are the **direct superclasses**.

Remark 11.1. If inheritance unspecified (e.g. `class A`) then super(A) = {java.lang.Object}.
Note that super(java.lang.Object) = {}.

Example 11.2. Given `interface F extends G, H, I` we have

$$\text{super}(F) = \{G, H, I\}$$

We now give the formal inference rules. First let's look at types (classes and interfaces):

Definition 11.1 (Subtypes)**.** Let $S < T$ denote that S is a **strict subtype** of T where

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

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

We also define $S \leq T$ as S is a **subtype** of T where

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

$$\frac{}{S \leq S}$$

Definition 11.2 (Super set)**.** We define the **super set** super(T) as all super classes of T i.e. super(T) = $\{S \mid T < S \text{ or } T \leq S\}$.

Definition 11.3 (Declare set)**.** We define the **declare set** declare(T) to be the set of *methods and fields* declared in T.

Remark 11.2.
- interface methods are implicitly `public abstract`
- interface fields are implicitly `static`

Definition 11.4 (Inherit set)**.** We define the **inherit set** $\text{inherit}(T)$ as the *methods and fields* that T inherits.

Definition 11.5 (Contain set)**.** We define the **contain set** $\text{contain}(T) = \text{declare}(T) \cup \text{inherit}(T)$.

Now we look at type methods:

Definition 11.6 (Replace (methods))**.** We say $\text{replace}(m, m')$ if method m overrides method m'.

Definition 11.7 (Replace (fields))**.** We say $\text{replace}(f, f')$ if field f hides field f'.

Definition 11.8 (Signature)**.** We define $\text{sig}(m)$ as method m's signature which consists of its **name** and its **parameter type**.
It does **not** include *return type* or *modifiers*.

We have the following inference rule for **overriding methods in superclasses** (8.4.6):

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

Definition 11.9 (No declaration)**.** We define $\text{nodecl}(T, m)$ as T does not declare method with $\text{sig}(m)$.
That is $\forall m' \in \text{declare}(T), \text{sig}(m') \neq \text{sig}(m)$.

Definition 11.10 (Modifications)**.** We define $\text{mods}(m)$ as the set of method modifiers on m (e.g. `abstract`).

We also have the inference rule for **inheriting non-abstract methods** (8.4.6.4):

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

Definition 11.11 (All abstract)**.** We define $\text{allabs}(T, m)$ as:

$$\forall S \in \text{super}(T) \forall m' \in \text{contain}(S), \text{sig}(m') = \text{sig}(m) \Rightarrow \texttt{abstract} \in \text{mods}(m')$$

That is all inherited methods with the same $\text{sig}(m)$ are abstract.

So we have the inference rule for **inheriting abstract methods** (8.4.6.4):

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

that is: we can only inherit `abstract` methods if there are no concrete methods with the same signature. We thus need to define **replacing abstract with concrete methods** (8.4.6.4):

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

Finally we have the inference rule for **inheriting fields**:

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

A note on interfaces and `java.lang.Object`:

Remark 11.3. An interface with no superinterfaces implicitly declares an `abstract` version of every `public` method in `java.lang.Object` (JLS 9.2).
For example an empty interface `interface I` implicitly has `i.equals(i)`.
So `class C implements I` would implicitly contain abstract methods of `java.lang.Object`.

11.2 Hierarchy checks (JLS, Joos)

We enumerate the class hierarchy checks we need to perform using the notation and definitions we introduced above:

1. No cycles in hierarchy i.e. $\nexists T, T < T$
2. No duplicate methods i.e.
$$\forall m, m' \in \text{declare}(T), m \neq m' \Rightarrow \text{sig}(m) \neq \text{sig}(m')$$
3. One return type per unique signature i.e.
$$\forall m' \in \text{contain}(T), \text{sig}(m) = \text{sig}(m') \Rightarrow \text{type}(m) = \text{type}(m')$$
4. Classes with abstract methods must be abstract i.e.
$$\forall m \in \text{contain}(T), \texttt{abstract} \in \text{mods}(m) \Rightarrow \texttt{abstract} \in \text{mods}(T)$$
5. Static methods can only override static methods i.e.
$$\forall (m, m') \in \text{replace}, \texttt{static} \in \text{mods}(m) \iff \texttt{static} \in \text{mods}(m')$$
Remark 11.4. Note this is an $\iff$ relationship: we cannot override with a non-static method if a static method is inherited *nor* can we override with a static method the inherited method is non-static.
6. Methods can only override methods with the same return type i.e.
$$\forall (m, m') \in \text{replace}, \text{type}(m) = \text{type}(m')$$
7. Only public methods can override public methods i.e.
$$\forall (m, m') \in \text{replace}, \texttt{public} \in \text{mods}(m') \Rightarrow \texttt{public} \in \text{mods}(m)$$
Remark 11.5. We only have a $\Rightarrow$ relationship: this means we may override with a `public` method m with the same signature of m' if m' is **not** `public`.
8. Omitted.
9. We cannot override `final` methods i.e.
$$\forall (m, m') \in \text{replace}, \texttt{final} \notin \text{mods}(m')$$
10. We cannot declare two fields with the same name i.e.
$$\forall f, f' \in \text{declare}(T), f \neq f' \Rightarrow \text{name}(f) \neq \text{name}(f')$$

11.3 Disambiguating namespaces (JLS 6.5.2)

Consider the expression $a_1.a_2.a_3.a_4$. We need to ascertain whether a_i are packages, types, or fields.
Consider also the expression $a_1.a_2.a_3.a_4()$. Then we know a_4 is a method, but we still need to disambiguou the rest of a_is.
Consider the following code snippet:

```
1  class X {
2    X X (X X) {
3      return (X)X.X(X);
4    }
5  }
```

how would the `X`'s be disambiguated? We follow the below procedure.
Given $a_1.....a_n$ proceed left to right and start with a_1:

1. If local variable a_1 is in scope, we use it.

   $a_2, \ldots, a_n$ must be *instance fields*.

2. If field $a_1 \in$ contain(current class) we use it.

   $a_2, \ldots, a_n$ must be *instance fields*.

3. For each k from 1 to n, if $a_1.....a_k$ is a **type** in the *global environment*, then a_{k+1} is a `static` field and $a_{k+2}, \ldots, a_n$ are *instance fields*.

Remark 11.6. If at any step e.g. step 1 we discover a_2 is not an instance field of a_1, we *do not* proceed to later steps and instead throw a compile error.

12 February 25, 2019

12.1 Resolving variables/static fields

In general this is straightforawrd: we simply lookup the variable in the innermost environment and if not found, we search outwards in enclosing scopes.
An issue with Java specifically is **shadowing**. A solution is to create a new environment for each block, but check outer environments when adding a name.
For example considder the following code snippet:

```
1  { int x;
2      {
3          int y;  // OK
4          int z;  // OK?
5      }
6      {
7          int x;  // not allowed
8          int y;  // OK
9      }
10     int z;  // OK?
11 }
```

Note that declaring `int x` again inside a nested scope when `int x` is already declared in an enclosing scope is not allowed. However, re-declaring `int y` in disjoint non-nested scopes is fine.
A solution is to create a *new scope* for every variable declaration (which would end at its semantic scope). For example given:

```
{
  int x;
  y = 3;  // error
  int y;
  y = 5;  // OK
  int z;
}
```

The corresponding program with scopes created for each new variable declaration would look something like:

```
{
  int x;
  y = 3;  // error
  {
    int y;
    y = 5;  // OK
    {
        int z;
    }
  }
}
```

12.2 Type checking

We provide two different perspectives/definitions of a type:

Definition 12.1 (Type (definition 1))**.** A **type** is a set of values (with operations on them).

Definition 12.2 (Type (definition 2))**.** A **type** is a way to interpret bit sequences.

We now differentiate between two categories of types:

Definition 12.3 (Static type)**.** The **static type** of an expression E is a set containing all possible values of E.

Definition 12.4 (Dynamic type)**.** The **dynamic type** is a runtime value that indicates how to interpret some of the other bits.

Definition 12.5 (Declared type)**.** The **declared type** of a variable is an assertion that the variable will only contain values of that type.

We now distinguish between two different ways we go about type checking:

Definition 12.6 (Static type checking)**.** Prove (using mathematical inference/deduction) that every expression will evaluate to a value in its type.

Definition 12.7 (Dynamic type checking)**.** Runtime check that the dynamic type (tag) is declared in the variable to which it is assigned.

A static type checker does two things:

1. Infers a type for each sub-expression
2. Check that expressions are used correctly with operations (e.g. `1 + true` is an error).

Definition 12.8 (Type correct)**.** A program is **type correct** if type assertions hold in all executions.

Remark 12.1. Note that a naive definition of type correctness is undecidable. Consider the following program:

```
1 int x;
2 if(program halts) {
3   x = true;
4 }
```

Note that if the program halts then `x = true` makes the entire program type incorrect, otherwise the program is indeed type correct since `x = true` is not evaluated.
Therefore we could reduce the halting problem to this definition of type correctness.

Instead we have a more refined definition of type correctness:

Definition 12.9 (Statically type correct)**.** A program is **statically type correct** if it obeys a system of type inference rules (**type system**).

Definition 12.10 (Soundness)**.** A type system is **sound** if statically type correct $\Rightarrow$ type correct.

12.3 Introduction to type system for Joos

We introduce some notation:

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

In class C, local environment L, if the current method has return type σ, then expression E has tpe τ.
Also:

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

means statement S is statically type correct.
We now introduce the inference rules:

Literals Literals are straightforward. For example integer literals have the rule:

$$\frac{}{C, L, \sigma \vdash 42 : \text{ int}}$$

The other literals:

$$\frac{}{\texttt{true} : \text{ boolean}} \qquad \frac{}{\text{“abc”} : \text{ string}}$$

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

Note “null” is a special type with only the `null` literal.

Operations The *not* operator:

$$\frac{E : \text{ boolean}}{!E : \text{ boolean}}$$

We first define the *numeric type class* as:

$$num(\sigma) = \sigma \in \{\text{int, short, char, bytes}\}$$

Then the addition operation is:

$$\frac{E_1 : \tau_1 \quad E_2 : \tau_2 \quad num(\tau_1) \quad num(\tau_2)}{E_1 + E_2 : \text{ int}}$$

$$\frac{E_1 : \text{ string} \quad E_2 : \tau_2 \quad \tau_2 \neq \text{ void}}{E_1 + E_2 : \text{ string} \quad E_2 + E_1 : \text{ string}}$$

12.4 Instance fields/methods

After type-checking, every sub-expression has a type. Consider the following two usages:

```
1 a.b       // field access
2 a.b(c)    // method access
```

We know the type of `a` from type-checking, thus for the first usage we can simply lookup the field `b` in type `a`, and for the second usage we can look up for a method `b` in type `a` (we check the contains set).
If `b` is an *overloaded* method, we disambiguate based on the arguments `c`.

13 February 27, 2019

13.1 Pseudocode for type checking

We have the following pseudocode for type checking an AST node E:

1. Find an inference rule of the form:
$$\frac{\text{premises}}{C, L\sigma \vdash E : \tau}$$
2. Check premises
3. Return τ if premises are satisfied

We repeat the above for all inference rules for E until either we find one whose premises are satisfied or we raise a type/compile error.

13.2 More inference rules for type system of Joos

For **local variable usages** we have

$$\frac{L(n) = \tau}{C, L, \sigma \vdash n : \tau}$$

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

where $L(n)$ is the type (if it exists) of a variable.
For **statements**, note that we do not assign it a type in Java:

$$\frac{C, L, \sigma \vdash E : \text{boolean} \quad C, L, \sigma \vdash S}{C, L, \sigma \vdash \texttt{if(E) S}}$$

For a **block of statements** we have:

$$\frac{\forall i \vdash S_i}{\vdash \{S_1; S_2; \ldots S_n;\}}$$

To handle **variable declarations**, we have the rule:

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

For **assignments**, we have:

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

where the := is a special relation known as **assignability (JLS 5)**, which has the inference rules (for Joos specifically):

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

$$\frac{}{\text{int} := \text{short}}$$

$$\frac{}{\text{short} := \text{byte}}$$

$$\frac{}{\text{int} := \text{char}}$$

$$\frac{D \leq C}{C := D} \quad \text{assignment to superclass}$$

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

$$\frac{}{C := \text{null}}$$

For **fields** we have the rules:

$$\frac{\vdash \texttt{static}\ \ \tau f \in \text{contains}(D)}{\vdash D.f : \tau}$$

$$\frac{\vdash \texttt{static}\ \ \notin \text{mods}(f) \quad E : D \quad \tau f \in \text{contains}(D)}{\vdash E.f : \tau}$$

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

$$\frac{\vdash E_1 : D \quad E_2 : \tau_1 \quad \texttt{static}\ \ \notin \text{mods}(f) \quad \tau_2 f \in \text{contains}(D) \quad \tau_2 := \tau_1}{\vdash E_1.f = E_2 : \tau_2}$$

For **comparison** we have (note these rules hold for both == and !=)

$$\frac{\vdash E_1 : \tau_1 \quad E_2 : \tau_2 \quad \text{num}(\tau_1) \quad \text{num}(\tau_2)}{\vdash E_1 == E_2 : \text{boolean}}$$

$$\frac{\vdash E_1 : \tau_1 \quad E_2 : \tau_2 \quad \tau_1 := \tau_2 \vee \tau_2 := \tau_1}{\vdash E_1 == E_2 : \text{boolean}}$$

For **casting**, we have two ways to cast:

$$\frac{\vdash E_1 : \tau_1 \quad \tau_1 := \tau_2 \vee \tau_2 := \tau_1}{\vdash (\tau_2)E : \tau_2}$$

Note $\tau_2 := \tau_1$ corresponds to **upcasting** (*always succeeds*) whereas $\tau_1 := \tau_2$ corresponds to **downcasting** (*may fail, needs runtime check*).
For numeric types:

$$\frac{\vdash E_1 : \tau_1 \quad \text{num}(\tau_1) \quad \text{num}(\tau_2)}{\vdash (\tau_2)E_1 : \tau_2}$$

Finally we have **runtime checks** for the expression `instance of`:

$$\frac{\vdash E : \tau \quad \tau := D \vee D := \tau}{E\ \texttt{instance of}\ D : \text{boolean}}$$

Remark 13.1. Note that if there is no assignability between τ_1, τ_2 then it *always fails* **except for interfaces**.

For **methods invocations**:

$$\frac{\vdash E : D \quad \forall i \vdash E_i : \tau_i \quad \tau m(\tau_1, \ldots, \tau_n) \in \text{contains}(D)}{\vdash E.m(E_1, \ldots, E_n) : \tau}$$

Remark 13.2. We do not have implicit casting of arguments with assignability semantics in Joos.
In Java there are specific rules for determining the "maximally specific" method declaration since the method may be overloaded (see 15.12.2).
One can simply use upcasts in Joos to work around this.

For **method returns**:

$$\frac{C, L, \sigma \vdash E : \tau \quad \sigma := \tau}{C, L, \sigma \vdash \texttt{return } E}$$

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

For **arrays**:

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

$$\frac{\vdash E_1 : \tau_1[] \quad E_2 : \tau_2 \quad \text{num}(\tau_2) \quad E_3 : \tau_3 \quad \tau_1 := \tau_3}{\vdash E_1[E_2] = E_3 : \tau_1}$$

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

Note that **array assignability (JLS 5)** has its own rules for the standard library types:

$$\frac{}{\texttt{Object} := \sigma[]}$$

$$\frac{}{\texttt{Cloneable} := \sigma[]}$$

$$\frac{}{\texttt{java.io.Serializable} := \sigma[]}$$

We also have the following rule for **multidimensional arrays**:

$$\frac{\sigma[] := \tau[]}{\sigma[][] := \tau[][]}$$

14 March 4, 2019

14.1 Potential issues with arrays

Recall that we have the inference rules

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

$$\frac{D \leq C}{C := D}$$

Now consider the following code fragment:

```
1 Apple[] as = new Apple[1];
2 Fruit[] fs = as;
3 fs[0] = new Orange();
4 Apple a = as[0];
```

Note that `as[0]` now points to an `Orange`, but we've explicitly typed `a` as an `Apple`. This is an **unsound** program.

Remark 14.1. In Java, the above code statically type checks and compiles but the assignment `fs[0] = new Orange()` will throw a runtime error.
It does this by marking the initial `as` allocated array as type `Apple[]` (dynamic type tag) and checking it during assignment.

Remark 14.2. Unsound programs are possible even if they statically type check.
Formally, we have statically type correct $\subseteq$ type correct $\subseteq$ all programs where unsound programs can occur at all levels.

How do we deal with this? To preserve type safety, we must check the dynamic type tag of an array at every array write (JLS 10.10) and raise an `ArrayStoreException` if a violation occurs.

14.2 Static program analysis

Consider another "unsound" behaviour with casts:

```
1 Orange o = (Orange) new Apple();
```

As a programmer there may be instances where we want to intentionally introduce unsoundness e.g. with the above cast.

Remark 14.3. The above cast raises a `ClassCastException` runtime error in Java.

In **static program analysis**, we want to prove *properties* of run-time behaivour without actually running the program.
Some applications:

- Prevent bugs
- Generate efficient code
- Inform programmer

Some properties we can derive:

- Constant expression
- Method execution always ends with return statement
- Will current value of variable ever be read?
- Will a statement ever be executed?
- Read/write dependencies
- Will program terminate or loop infinitely? (in general this is undecidable but we can still do this analysis for some programs)

- Is an array access in bounds?

Example 14.1. Suppose we wanted to see if the following outputs "Hello world":

```
1 main() {
2   if(...) { ... }
3   printf("Hello world");
4 }
```

Suppose we wanted to write an assembler that given a program f, decides whether the output is "Hello world". Clearly this is undecidable in general.

Theorem 14.1 (Rice's theorem)**.** Let R be any **non-trivial** property of the output of the program.
Given program P, it is **undecidable** whether P has property R.
We define *non-trivial* as $\exists P \in R$ and $\exists P \notin R$.

We can however give a **conservative approximation**.

Definition 14.1 (Conservative analysis)**.** An analysis is **conservative** if its result is never untrue.
For example an analysis outputting **maybe** and **no** is conservative (assuming it's sound).

Definition 14.2 (More precise analysis)**.** A more **precise** analysis gives definitive answers for more programs.
For example an analysis outputting **maybe**, **no** and **yes** is more precise than a conservative one (assuming it's sound).

Java requires **reachability analysis** and **definite assignment** (every local variable must be written to before it is read).

14.3 Java reachability analysis (JLS 14.20)

Java specifies specific reachability analysis that must be performed. A conservative analysis looks like:

Algorithm 11 Java reachability (conservative) analysis

1: Define: in[s] - can statement s start executing? (no/maybe)
2: Define: out[s] - can statement s finish executing? (no/maybe)
3: Error if in[s] = no for any s
4: Error if out[method body] = maybe (for non-void methods)

We define $in[s]$ and $out[s]$ for every AST node recursively.
For `return` statements we have for both `L : return` and `L : return E`:

$$out[L] = no$$

For `if` statements we have `L : if (E) S` where:

$$in[S] = in[L]$$
$$out[L] = out[S] \vee in[L] = in[L] \qquad out[S] \Rightarrow in[S] \Rightarrow in[L]$$

where $in[S] = in[L]$ since to execute S, regardless of whether E evaluates to `true` or `false` we require at least $in[L]$ and we define

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

(this holds only for this specific analysis: one will need to define this merge $\vee$ specifically for a given analysis).
For `if-else` statements we have `L : if (E) S1 else S2` where:

$$in[S1] = in[L]$$
$$in[S2] = in[L]$$
$$out[L] = out[S1] \vee out[S2]$$

We have a special case for an **infinite loop** where `L : while(true) S` and `breaks` do not exist (Joos):

$$in[S] = in[L]$$
$$out[L] = no$$

similarly for `L: while(false) S`:

$$in[S] = no$$
$$out[L] = in[L]$$

and in general for `L: while(E) S`:

$$in[S] = in[L]$$
$$out[L] = in[L] \qquad\qquad in[S] \vee in[L] = in[L]$$

Note the above also applies for `if`s but the spec does not include this, presumably because the initial motivation was to detect infinite loops.

15 March 6, 2019

15.1 Constant expressions

We note that comparison between literals can be converted into a constant. For example `while (1 == 1)` becomes `while (true)`; however, for an expression x in general `while (x == x)` cannot have any constant reduction (consider x is a method call e.g. RNG).
For more constant expressions see JLS 15.28.

15.2 More reachability analysis

For block statements `L : {S1; S2}`:

$$in[S1] = in[L]$$
$$in[S2] = out[S1]$$
$$out[L] = out[S2]$$

For any other statement `L : other stmt`:

$$out[L] = in[L]$$

For method bodies `L : method body`, we do not know if someone will invoke the method at any point, so:

$$in[L] = maybe$$

and if the method is non-void, then `out[L]` must be no.

Remark 15.1. Performing reachability analysis of ins/outs is naive quadratic time: **memoization** reduces this to linear time.

Remark 15.2. We can do this analysis in topological order from the top of tree and work downwards. Handling cycles will be discussed later.

15.3 Java definite assignment

In Java we require a variable to be initialized/defined before usage. For example:

```
1 int x;
2 x = 42;
3 return x;
```

Note it is fine for a never referenced variable to not be initialized.
In Joos, we relax definite assignment e.g. requiring `int x = 42;`. That is:

1. Local variable must be initialized when declared
2. Local variable cannot appear in own initializer e.g. `int x = x`

However in general we define (note in/out is now a set rather than a boolean) the sets of **definitely assigned variables**:

in[s] set of variables that definitely have been initialized before s starts execution

out[s] set of variables definitely initialized after execution of s

Then for any expression E that reads x, **error** if $x \notin in[E]$.

Remark 15.3. Java permits assignments within assignments e.g. `int x = (y = 5)`.

So we define for `L { τ x = E; S }` :

$$in[E] = in[L]$$
$$in[S] = out[E] \cup \{x\}$$
$$out[L] = out[S]$$

for a similar expression without initialization `L : { τ x; S }`:

$$in[S] = in[L]$$
$$out[L] = out[S]$$

For a simple assignment expression `L : x = E`:

$$in[E] = in[L]$$
$$out[L] = out[E] \cup \{x\}$$

Remark 15.4. Consider the following:

```
1 boolean x;
2 if (b && (x = true)) {
3   println(x);
4 } // fine
5
6 boolean x;
7 if (b || (x = true)) {
8   println(x);
9 } // error
```

Note in the first snippet we are guaranteed `x` will be initialized if `println(x)` is executed. This does not hold in the latter example.

For `if` statements e.g. `L : if(E) S`:

$$in[E] = in[L]$$
$$in[S] = out_{true}[E]$$
$$out[L] = out[S] \cap out_{false}[E]$$

where $out_{true}[E]$ is the set of initialized variables if E indeed evaluates to true (and similarly $out_{false}[E]$).
The intersection allows us to account for both cases when E is true and false simulatenously: if we don't know E is true or false then the intersection gives us a conservative estimate (since we want in/out to know **definite assignments**).
For the `&&` operator where `L : E1 && E2`:

$$in[E1] = in[L]$$
$$in[E2] = out_{true}[E1]$$
$$out_{true}[L] = out_{true}[E2]$$
$$out_{false}[L] = out_{false}[E1] \cap out_{false}[E2]$$

16 March 11, 2019

16.1 `x86` assembly language family

The name `x86` comes from the 8086 processors produced by *Intel* in 1978 and was one of the first architectures to support eight 16-bit registers (whereas most other processors only supported 8-bit registers). It used the `x86-16` instruction set.
The 80386 released in 1986 was the first 32-bit Intel architecture with eight 32-bit registers. It used the `x86-32` (also known as `IA-32` or `i386`) instruction set.
Opteron was released in 2003 had sixteen 64-bit registers and uses the `x86-64` instruction set.
We will be using the `i386` instruction set for Joos.

16.2 i386 registers

In `i386` we have eight **general-purpose registers**: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, and `ebp`.
These registers have special names attached to them: for example when copying between two arrays there are special instructions where `esi` is implicitly treated as the "source index" and `edi` is treated as the "destination index".
We treat the first 6 registers as just ordinary registers for this course.
We also have special instructions (e.g. push/pop) where it treats `esp` as containing the **stack pointer** and `ebp` usually as the **frame pointer** (beginning of function), although we may use it as any ordinary register.

Remark 16.1. One should never explicitly uses `esp` since `esp` is used for a lot of special instructions implicitly.

There are also six **segment registers**: `cs`, `ds`, `es`, `fs`, `gs`, `ss`. We will ignore these registers.

16.3 i386 instructions

To copy data to/from registers, we have the following instructions:

```
mov eax, ebx          ; eax = ebx (copy)
mov [eax], ebx        ; *eax = ebx (de-reference and copy)
mov eax, 42           ; eax = 42 (assign constant)
mov eax, label        ; eax = label (assigns location of label)
mov eax, [label]      ; eax = *label (assigns value (of instr.) at label)
mov eax, [esp + ebx*2 + 5]    ; more complex arithmetic
```

We also have instructions for constants/words:

```
dd 1234               ; .word 1234 (32-bits)
db "hello"
```

where `dd` stands for "data double-word" (32 bits) and `db` stands for "data bytes".
We have arithmetic instructions:

```
add eax, ebx          ; eax += ebx
sub eax, ebx          ; eax -= ebx
imul eax, ebx         ; eax *= ebx (i for signed)
idiv ebx              ; eax = edx:eax / ebx (edx:eax is 64 bits)
                      ; edx = edx:eax % ebx
```

Remark 16.2. Warning: since `idiv` prepends `eax` with `edx`, one needs to set `edx` to the appropraite sign of `eax`: that is `edx = 0` if `eax` ≥ 0 and `edx = -1` (mask of 1's) if `eax` < 0.
The `cdq` instruction will set `edx` to the sign of `eax`.

Some instructions for control flow:

```
jmp label             ; eip = label
cmp eax, ebx          ; sets appropraite flags (see next)
jg label              ; jump if eax > ebx (from cmp, signed)
                      ; also jge, jl, jle for >=, <. <=
ja label              ; jump if eax > ebx (from cmp, unsigned)
                      ; also jae, jb, jbe for >=, <, <=
push eax              ; pushes eax onto stack
pop eax               ; pops off stack into eax
call label            ; pushes eip, jumps to label
ret                   ; pops to eip
```

For system calls and interrupts:

```
int 0x80              ; performs system call 0x80
```

We will need to set appropriate registers with arguments we want to pass through to system calls. For example, the `exit(code)` system call in Linux, we will set `eax = 1` and `ebx = code`.

16.4 i386 directives

We have a few directives available to us:

```
dd, db              ; see above
global label        ; export (make label available in other files)
extern label        ; import label from another file
align 4             ; pad with 0x0 bytes to align to memory address divisible by 4
```

17 March 13, 2019

17.1 Code generation

The main tasks of code generation is to:

- Plan data layout (express as code)
- Generate code for each AST node (botom-up)
- Write supporting run-time code

17.2 Java data

We need to allocate memory and organize data for the following in Java:

- Literals
- Local variables
- Objects, where instances require:
  - Field values
  - Dynamic class tag (vpointer)
- Classes, which require:
  - Pointers to method implementations (vtable)
  - Subtype testing
  - Static field values
- Arrays, which require:
  - Elements
  - Length
  - Dynamic type tag (vpointer)

Primative data types in Joos include:

- `int` (signed and unsigned), 32 bits
- `short` (signed) and `char`, 16 bits
- `byte`, 8 bits
- `boolean`, 1 bit
- `reference`, 32 bits

Remark 17.1. 1. One may choose to use 32 bits for everything, OR

2. One may choose to pack bits/bytes together

Remark 17.2. For arithmetic involving `short` and `char`, note that the result is a 32 bit `int`, but semantically the inputs are 16 bits.
One should cast the `short` and `char` to `int` first (since each are signed, one should sign extend them) then perform the arithmetic as per usual.

17.3 Data storage options

- **Constants**: one can either specify it directly in the instruction (e.g. `move eax, 42`) or use labelled memory locations (e.g. `label: db "string"`)
- **Registers**: limited number so will require backup
- Fixed (labeled) memory locations
- **Stack**: LIFO, efficient (referenced by fixed offset from stack pointer)
- **Heap**: allocate/free in any order, any size. Requires runtime data structures and algorithms to manage memory

Remark 17.3. Constants, registers, and fixed memory locations are statically decoded/accessed by the compiler (hardcoded in assembly) whereas the stack and heap can by dynamically referenced.

A good mapping between types of data and storage:

- Literals: constants
- Local variables: stack
- Objects: heap
- Classes: fixed locations
- Arrays: heap

17.4 Stack offsets

Given the following code snippet:

```
1 void m() {
2   int a = 0;
3   int b = 0;
4   int c = 0;
5 }
```

We can create a frame to store the local variables ($3 \times 4 = 12$ bytes) i.e. we first do `sub esp, 12` to move the head of the stack downwards (stack grows from top to bottom) then our local variables will simply be `a = [esp + 8], b = [esp + 4], c = [esp + 0]`.
Some problems:

- Offsets change when `esp` changes
- Require fixed frame size (does not work if we try `alloc a` or allocate variable-length array (VLA), but this is fine in Java, maybe not in C)
- Difficult to interpret stack trace (for debugging)

One solution for the first and third problem is to use the `ebp` register as a **frame pointer**. Before we allocate the frame on the stack, we set `mov ebp, esp` (i.e. keep a reference to the start of the frame) and all local variables will reference `ebp` throughout e.g. `a = [ebp - 4], b = [ebp - 8], c = [ebp - 12]`.

17.5 Procedure calls

During procedure calls, one needs to pass the parameters and receive the return value. One needs to decide on how and **which registers are to be saved and restored**.
The **callee-save** protocol specifies that the **callee must preserve and restore certain register values**. In `x86` the registers `ebx, esi, edi, ebp, esp` are typically required to be preserved.
The **caller-save** protocol permits register values in `eax, ecx, edx` (for `x86`) to change in a procedure call.
For example, the following is a typical prologue/epilogue during a procedure call:

```
1 push ebp
2 mov ebp, esp
3 sub esp, 12     ; 12 is frame stack
4 ; push callee-save regs on stack
5 ...
6 ; pop callee-save regs
7 mov esp, ebp
8 pop ebp
9 ret
```

We also need a convention for storing arguments before a procedure (**Application Binary Interface (ABI)**) so the procedure knows how to access those parameters. A typical convention is to push arguments onto the stack from **left to right**:

```
1 int f(int a, int b, int c){
2   int i = 0;
3   return a;
4 }
5
6 f(x,y,z);
```

To invoke `f(x,y,z)` we would have:

```
1 push x
2 push y
3 push z
4 call f
```

and when `f` returns we typical return the value in register `eax`. This works well for Joos since everything is at most 32 bits.
Thus inside `f` we'll have:

```
1 push ebp
2 mov ebp, esp
3 sub esp, 4                ; equivalent to pushing i = 0
4 mov eax, [ebp + 16]       ; return a
5 mov esp, ebp
6 pop ebp
7 ret
```

Note that we have `i = ebp - 4`, `c = [ebp + 8]`, `b = [ebp + 12]`, `c = [ebp + 16]` (**NB:** our first parameter begins at +8 instead of +4 because we pushed the previous `ebp` (at `ebp + 0`) and `eip` (from `call/ret` at `ebp+4`) onto the stack!)

Remark 17.4. Pascal ABI is also **left to right** whereas cdecl ABI is **right to left**.
The cdecl ABI allows variable-length argument lists such as for `printf`: `printf` can simply keep walking down the stack (which is ordered left to right since we inserted right to left) until it reaches the old `ebp`.

17.6 Code generation for objects and classes

The per-object data includes the **class tag** and **instance field values**.
The per-class data includes **static field values** and **method implementations**.

Definition 17.1 (Type-compatible)**.** We say $T := S$: the memory layout of an S object must be **compatible** with T.
The **idea** is to make the T layout a *prefix* of the S layout (both *object and class* layouts).

Example 17.1. Consider the following:

```
1  class A {
2    int fa;
3    int ma() { ... }
4    int ma2() { ... }
5  }
6
7  class B extends A {
8    int fb;
9    int ma() { ... }
10   int mb() { ... }
11 }
```

For objects of type A and B, we can have their memory layout as such:

object A
class tag (points to class A)
`fa`

object B
class tag (points to class B)
`fa`
`fb`

where each class tag points to the address of their respective classes.
If we wanted to `read fa`, we can simply do:

```
1  mov eax, [eax + 4]
```

for either object A or B.
As for the class memory layouts:

class A
`ma` (points to A's implementation of `ma`)
`ma2` (points to A's implementation of `ma2`)

class B
`ma` (points to B's implementation of `ma`)
`ma2` (points to A's implementation of `ma2`)
`mb` (points to B's implementation of `mb`)

Then for `call ma`, we can simply do:

```
1  mov eax, [eax];          ; store vtable/class tag
2  mov eax, [eax + 0];      ; + 0: offset of ma
3  call eax
```

Question 17.1. When does this offset approach fail? With interfaces!
Consider the following:

```
1  interface A {
2    int ma();          // offset 0
3  }
```

```
4
5  interface B {
6    int mb();          // offset 0
7  }
8
9  class C implements A, B {
10   int ma() { ... }  // offset 0
11   int mb() { ... }  // offset 4
12 }
```

Note that an object of class C will have `ma` and `mb` at offsets 0 and 4, respectively.
If we had a declared object of type B, then we would incorrectly access `ma` at offset 0.
We thus need a function at **runtime** $f : C \times S \to I$ where:

- C: concrete (runtime) class of object
- S: "selector" i.e. name/signature of method
- I: method implementation

Solution. Option 1: Selector Indexed Table We can implement this directly in assembly using a large **2D table** where we have **columns for every class** and **rows for every selector** i.e. method name/signature for every *interfaces*.

For every class, we keep a reference to its respective column in the table at the 0th offtset.

Lookup time is $O(1)$.

For example, `call mb` for an object is:

```
1 mov eax, [eax]      ; class vpointer
2 mov eax, [eax]      ; select table column
3 mov eax, [eax + 4]  ; (global) offset of mb
4 call eax
```

This table solution is called a **Selector Indexed Table**.

This table takes $O(SC)$ space (for S selectors/unique method signatures in interfaces and C classes).

We can save space by:

- reusing columns/rows (e.g. if classes implement the exact same interfaces)
- return unused selectors to heap memory manager

Option 2: Hash Table We may also implement a hash table separately (e.g. in C) as part of the runtime library, take its assembly and interface it with the compiler so the compiler would do lookups via this hash table.

The space is $O\left(\sum_c \in C m(c)\right)$ where $m(c)$ is the methods in class c i.e. linear in the number of methods.

The time is however not $O(1)$ since we may have to do a hash table lookup upon every call. We can however cache the last-seen class/answer for each call site.

17.7 Array types

We note that arrays of reference types e.g. `Foo[]` technically inherit from `java.lang.Object` and will require their own vtable/class data table for method implementations.
They do not however contain any static fields so only pointers to their appropriate implementations are necessary.

17.8 Subtype testing

We may need to test for subtype assignability during runtime, e.g.

```
1 if( o instanceof T) ...    // instanceof
2 a = (A)o                   // casting
3 fruits[i] = orange         // assigning arrays
```

That is given object o and type T, is $T :=$ classtag(o)?
Ignoring interfaces and only considering *single inheritance*, we can perform DFS, assigning each class an interval that spans the smallest and largest number assigned to its descendant classes.
For example if `B extends A` and `C extends A`, then after DFS traversal we have $[2, 3]$ for `B`, $[4, 5]$ for `C`, and $[1, 6]$ for `A`.
Then $B \leq A$ iff interval of B is contained in interval of A.
What about multiple inheritance via interfaces? We can construct a similar lookup table of size $O(T^2)$ where T is the number of types that tells us whether a given type T is a subtype of another type T'. This table can work exactly as the Selector Indexed Table we used for method dispatching.
Again, similar optimizations can be performed on this table.

18 March 20, 2019

18.1 Code generation for control-flow statements

`if else` statements Given `L: if(C) T else E` we have:

```
1 ; C.code
2 cmp eax, 0
3 je else42           ; unique else label
4 ; T.code
5 jmp end42           ; unique end label
6 else42:
7 ; E.code
8 end42:
```

An *alternative* is to define `iffalse` and `iftrue` functions in our compiler where

`iffalse(C, label)` evaluates `C` and branch to `label` if false

`iftrue(C, label)` evaluates `C` and branch to `label` if true

and we can replace e.g. the first 3 lines with `iffalse(C, else42)`.

Remark 18.1. We only really need one of `iffalse` or `iftrue`.

`&& ||` Note these operators can be treated as *control flow expressions* since they short-circuit.

They are similar to `if else`, e.g. for `L: E1 && E2`:

```
1 ; E1.code
2 cmp eax, 0
3 je endX
4 ; E2.code
5 endX:
```

Remark 18.2. The register `eax` still contains the entire expression's result whether `E1` evaluates to true or false.

If we implemented `iffalse`, then:

```
iffalse(E1 && E2, l) =
  iffalse(E1, l)
  iffalse(E2, l)
```

18.2 Code generation for expressions

Arithmetic Suppose we are given `E: E1 - E2`, then:

```
; E1.code
push eax
; E2.code
pop ebx
sub ebx, eax
mov eax, ebx
```

Remark 18.3. Using the stack is the simplest but wastes more instructions than necessary.

We could have specified a separate register where `E1.code` would place its results instead of pushing and popping off the stack.

We will re-visit this later on how to optimize this.

Assignment Given `E1 = E2`, we need to evaluate the **value of `E2`** and the **address of `E1`**, where either expressions could be complicated.

Note that we need to ensure that `E1` is an **l-value** to prevent invalid assignments like `5 = 42`.

Definition 18.1 (l-value)**.** An **l-value** is an expression that can be assigned to.

In Java, **variables**, **object fields**, **class fields**, and **array elements** are all l-values.

The idea is to **generate the address of l-values separately** i.e. some code to get the address for `E1` when we want `E1.addr`.

Thus we have:

```
; E1.addr
push eax
; E2.code
pop ebx
; if E1 is array access, check assignability
mov [ebx], eax
```

To get the address of an l-value `E1.addr`, we have:

Local variable `V` For `V.addr`:

```
mov eax, ebp
add eax, (offset of V)
```

Class field `C.f` For `C.f.addr`:

```
mov eax, (label of C.f)
```

Object field `O.f` For `O.f.addr`:

```
; O.code
; null check
add eax, (offset of f)
```

Remark 18.4. Note we have `O.code` instead of `O.addr`: this is because the value of the object is the address of its instance, whereas `O.addr` is where the object is declared i.e. local variable.

Array access `a[i]` For `a[i].addr`:

```
1 ; a.code
2 push eax
3 ; i.code
4 pop ebx
5 ; null check of a
6 ; bounds check
7 add eax, (header offset)  ; # of words at the header of array instance
8 shl eax, 2                ; multiply offset by 4 to get byte offset
9 add eax, ebx
```

Remark 18.5. The null check happens after evaluating `i` as per the Java spec.

Thus if we wanted to read the value of `E1` e.g. for `E1.code` then we have:

```
1 ; E1.addr
2 mov eax, [eax]      ; dereference address of E1
```

Method calls (static type of o is a class) For **non-static** and **non-constructor** method calls on an object i.e. `o.m(a)`, we follow the left-to-right Pascal argument semantics:

```
1  ; o.code
2  ; null check
3  push eax                          ; stack: [o]
4  ; a.code
5  push eax                          ; stack: [o, a]
6  mov eax, [esp + 4]                ; get address of o
7  mov eax, [eax]                    ; get vtable
8  mov eax, [eax + (offset of m)]    ; address of m body
9  call eax
10 add esp, 8                        ; pop args o and a
```

Remark 18.6. Recall the first item on the stack is at `esp + 0`.

Method calls (static type of `o` is an interface) If the static type of `o` is an interface, rather than simply getting the vtable and using the offset of `m` to get `m`'s body, we need to do a lookup in the Selector Indexed Table for `m`'s body.

19 March 25, 2019

19.1 Instance creation

For `new A(a)`, we have:

```
1  ; create object:
2
3  mov eax, (size of A in bytes)
4  call __malloc                    ; zeros memory
5  mov [eax], (vtable of class A)
6
7  ; call constructor:
8
9  push eax                         ; push this as first argument
```

```
; a.code                          ; evaluate parameters
push eax                          ; push the argument a
; call constructor of A
add esp, 4                        ; pop parameters
pop eax                           ; pointer to object
```

Generating the code for the **constructor**:

1. Call superclass constructor
2. Execute field initializers
3. Explicit constructor body

19.2 Compiler optimization

There are a couple of ways to optimize our compiler:

- Better code generation
- Static analysis and optimization
- Heap management and garbage collection (GC)

19.3 Code generation optimization

So far, for each AST node we've generated multiple assembly instructions in a RISC fashion.
In CISC, we have more powerful and complex instructions that will allow us to implement multiple AST nodes in *one instruction*.
We categorize the ways we can optimize code generation:

- instruction selection: which instructions
- instruction scheduling: in which order to enable pipelining
- register allocation

Example 19.1. Consider the following array assignment `a[b] = c`. We have

```
; a.code:
mov eax, ebp
add eax, -4       ; offset of a in stack
mov eax, [eax]    ; dereference to a object
push eax

; b.code:
move eax, ebp
add eax, -8       ; offset of b in stack
mov eax, [eax]

; compute a[b] addr
pop ebx
add eax, 2        ; header of a object
shl eax, 2
add eax, ebx
push eax

; c.code
```

```
20 mov eax, ebp
21 add eax, -12
22 mov eax, [eax]
23
24 ; assign c to a[b]
25 pop ebx
26 mov [ebx], eax
```

Note that for `a.code`, we could've simply done `mov eax, [ebp - 4]`.

19.4 Peephole optimization

In general, we can use **peephole optimization** that does *search and replace* on generated code. We express patterns for instructions to look for and replace. For the above pattern, we can replace:

```
1 mov Ra, Rb
2 add Ra, C
3 mov Ra, [Ra]
```

with

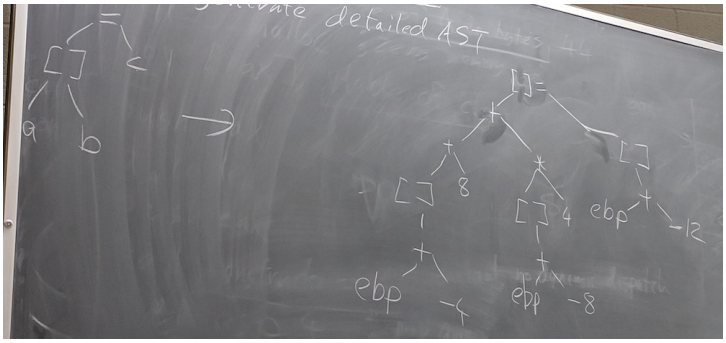
```
1 mov Ra, [Rb + C]
```

One would repeatedly apply search and replaces until no changes.

20 March 27, 2019

20.1 Tree-tiling instruction selection

We first generate a **detailed AST** whereby the leafs are individual tokens in assembly. For example for `a[b] = c` we have:



What if we had instructions that performed the operations of sub-trees of the right tree? We could transform subtrees on the right with single instructions.
We can map each subtree to instructions (these are our **tiles**):

- `ebp` → `mov eax, ebp`
- `c` (some constant `c`) → `mov eax, c`
- `x + y`:

```
1 ; x
2 push eax
3 ; y
4 pop ebx
5 add eax, ebx
```

- `[x]` → `mov eax, [eax]`
- `[x] = y`:

```
1 ; x
2 push eax
3 ; y
4 pop ebx
5 move [ebx], eax
```

We observe that we can combine the instructions for some of the subtrees:

- `[r + c]` → `mov eax, [r + c]`
- `[(x + c2) + (c1 * y)]`:

```
1 ; x
2 push eax
3 ; y
4 pop ebx
5 move eax, [ebx + eax * c1 + c2]
```

Thus `[(x + c2) + (c1 * y)] = z` is:

```
1 ; x
2 push eax
3 ; y
4 push eax
5 ; z
6 pop ebx
7 pop ecx
8 move [ecx + ebx * c1 + c2], eax
```

If we assigned a cost to each node and transformation (e.g. +1 for every instruction) then we can find the tiling that minimizes the overall cost. We can come up with a **tiling algorithm**:

Algorithm 12 Tree-tiling algorithm

input AST T
output $mincost[n]$ - minimum cost of tiling subtree under node n
output $tile[n]$ - tile to use at node n to achieve min cost

1: **for** each node n of T in bottom-up order (DFS) **do**
2: **for** each tile t compatible with subtree under node n **do**
3: $tilecost[t] \leftarrow cost[t] + \sum_{c \in \text{ children of } t \in T} mincost[c]$
4: $tile[n] \leftarrow t$ such that $tilecost[t]$ is minimal
5: $mincost[n] \leftarrow tilecost[tile[n]]$

20.2 Register allocation

We note that we can transform our tree above into a DAG by merging nodes that are equivalent. However, register allocation optimization on DAGs is in general **NP-hard**.
We will thus focus on register allocation optimization on **trees**.
We define $r[n]$ as the *minimum number* of registers needed to evaluate node n. If n has no children then $r[n]$ is constant (usually 1).

Given a tree rooted at A with three children B, C, D, to evaluate A we must evaluate B, C, D in some order (assume side effects), e.g. if order is B, C, D then we have

$r[B]$	registers to evaluate B
$r[C]+1$	registers to evaluate C (and remember B)
$r[D]+2$	registers to evaluate D (and remember B, C)

In general

$$r[A] = \max_i \{i + r[c_i]\}$$

where c_i are children of A sorted such that $r[c_i] \geq r[c_{i+1}]$.
We can then use DP or memoization to evaluate $r[c_i]$.
When generating code for a node, pass in integer r indicating that first r registers are busy. The result should then go into register $r+1$.
Thus from our example tree above, where the root is `A` and we have child `A: [(B + 8) + 4 * C] = D`:

```
1 B: mov eax, [ebp - 4]
2 C: mov eax, [ebp - 8]
3 D: mov eax, [ebp - 12]
4 A: mov [eax + ebx*4 + 8], ecx
```

21 April 1, 2019

21.1 Live variables

Live variables are a set of variables whose current values might be read before it is overwritten.
Similar to reachability analysis, we define $in[S]$ and $out[S]$ sets for an code section S:

$in[S]$ Before S, the set of variables that might be read before being overwritten

$out[S]$ After S, the set of variables that might be read before being overwritten

To illustrate how we construct these sets, we annotate the following code with in/out sets between each line:

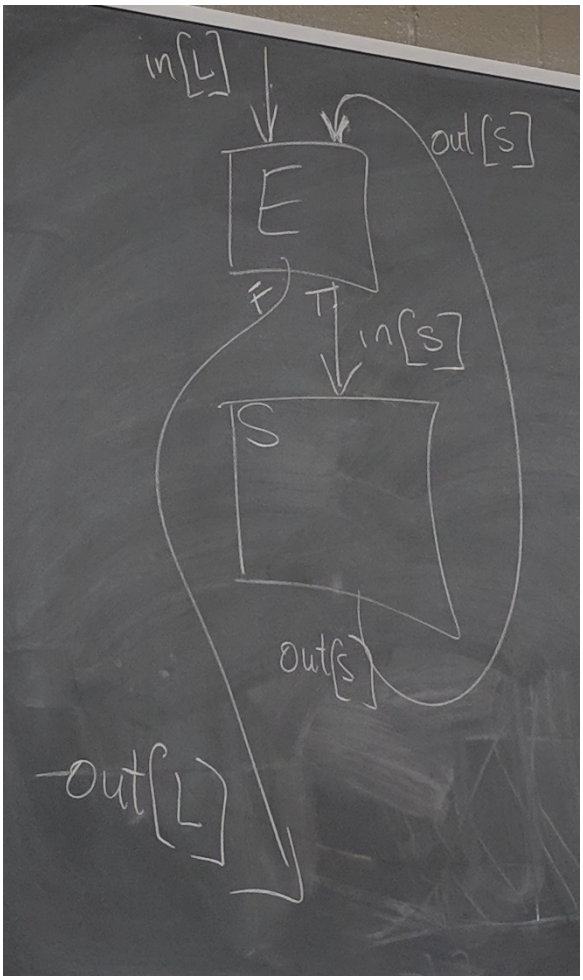
```
1  // {x}
2  z = 5;
3  // {x,z}
4  ... = z;
5  // {x}
6  z = 4;
7  // {x}
8  y = 2;
9  // {x,y}
10 ... = foo(y);
11 // {x}
12 ... = x;
13 // {}
```

Where we worked backwards: there are no live variables at the end of the segment. Moving up to `... = x`, we note that `x` is read so $in($`... = x`$) = \{x\}$.
When we get to `y = 2`, we overwrite `y` so we remove `y` from $in($`y = 2`$)$ from $\{x, y\}$ since we do not care about the value of `y` after the assignment.
We note that `x` is live throughout so it may be wise to store `x` in a register throughout.

We also note that after `z = 4`, `z` is never a live variable so we can simply remove the assignment (assuming no side effects).
In general for **assignments** i.e. `S: lhs = rhs` we have:

$$in[S] = \big(out[S] \setminus KILL[S]\big) \cup GEN[S]$$

where $KILL[S]$ contains variables that are *overwritten* and $GEN[S]$ contains variables that are *read* in the RHS. For example `y = y + 1` would correspond to $in[S] = (out[S] \setminus \{y\}) \cup \{y\} = out[S] \cup \{y\}$.
For `L: while(E) S`, note we have the following flow:



so we have:

$$out[S] = out[L] \cup in[S]$$
$$in[L] = out[L] \cup in[S]$$
$$in[S] = out[S] \cup READ[S]$$

Note that analysis has a cyclic dependency between $in[S]$ and $out[S]$: we cannot use tree traversal (and attribute grammars) to compute the *in* sets.
We want:

- termination guarantees

- some "precision" guanratee

For example, suppose $out[L] = \{x\}$ and `y` is read inside of `S`. We start the process by assuming $in[S] = \{\}$, then:

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

where $in[S]$ and $out[S]$ reaches their **fixed points**.
Let us prove the termination guarantees mathematically:

Definition 21.1 (Partial order)**.** A partial order is a relation $\preceq$ that is

1. reflexive: $x \preceq x$
2. transitive: if $x \preceq y$ and $y \preceq z$ then $x \preceq z$
3. anti-symmetric: if $x \preceq y$ and $y \preceq x$ then $x = y$

For **liveness**, let x, y be each sets of variables over the partial order $\subseteq$ (non-strict set inclusion).
We define two important notions:

1. If $x \preceq y$ and x is a **sound approximation** then so is y
2. If $x \preceq y$ then x is at least as **precise** as y

Definition 21.2 (Upper bound)**.** z is an **upper bound** of x and y if $x \preceq z$ and $y \preceq z$.

Definition 21.3 (Least upper bound (LUB))**.** z is a **least upper bound (LUB)** if it is an upper bound and for all other upper bounds v, $z \preceq v$.

Definition 21.4 (Complete lattice)**.** A **complete lattice** L is a set closed under LUBs.

Definition 21.5 (Bottom)**.** The **bottom** element $\bot$ is an element such that $\forall x, \bot \preceq x$.

Definition 21.6 (Monotone function)**.** $f : L \to L$ is monotone if $x \preceq y$ then $f(x) \preceq f(y)$.

Theorem 21.1. If L is a **finite set** with a least upper bound (**semi-lattice**) and $f : L \to L$ is **monotone**, then:

1. **Fixed point**: $\exists k$ such that if $x = f^k(\bot)$, then $x = f(x)$
2. $\forall y$ such that $y = f(y)$, then $x \preceq y$ (where x is any fixed point from above)

Remark 21.1. Conclusion 1 basically tells us we will terminate eventually. Conclusion 2 tells us that any other fixed point is a superset of any fixed points.
We want the *smallest* set of live variables (fixed points).

The pseudocode for the backwards analysis:

Algorithm 13 Live variable in/out sets

1: Initialize $in[S], out[S] = \perp$ for all S
2: **while** until $in[S], out[S]$ do not change **do**
3: Update all $in[S], out[S]$ according to equations

22 April 3, 2019

22.1 Memory management

The program and static parts of the code is stored somewhere around address $0x0$ (although a certain chunk of memory is reserved). The stack grows from the maximum address downwards, and the heap grows from the end of the static part upwards towards the stack.
The static section is fixed whereas the stack grows linearly. The heap on the other hand requires management by the **runtime**.
Virtual memory abstracts away allocating and freeing memory in the heap area. It offers protection via paging and segments.

Manual memory Instead of garbage collection, we can manually manage memory which involves `malloc(size)` and `free(ptr)`. Note `free` automatically frees the among of bytes allocated by `malloc` at a given pointer: the manager allocates a header right before the pointer to the object (indexed at -1).

`mmap` and `malloc` Note that initially the memory to the memory manager is a block of dead space. The memory manager must invoke `mmap` to the kernel in order to allocate **pages** in memory. The runtime/user code can then invoke `malloc` to allocate **bytes** out of the `mmap`'ed memory. Note that `mmap` involves a context switch.
So the user code can `malloc` and `free` bytes from/to the manager: however, the manager cannot return memory to the kernel that are not full pages. The manager therefore must keep track itself of allocated and free memory by the user code.

22.2 Memory management: free lists

One way the manager can manage freed memory is via link lists that points to continguous sections of free memory. Note generally we have several linked lists: several for titfree objects and one for *free space*. Initially *free space* is the entire `mmap`'ed region. Objects that are `free`'ed later on are appended to their corresponding linked list.
When an allocation is performed, the manager tries to find a free object for the specific object. If none can be found, memory is allocated from the *free space*.
In general to do this efficiently is a hard topic: one could imagine optimizations such as coalescing adjacent free objects, etc.

22.3 Automatic memory management

The defining principle is that `free` should be automatic (no explicit memory management).
The common solution is having a **garbage collector (GC)**: part of the runtime does `free` for us. There are numerous other approaches such as **type-based** memory management (e.g. `Rust`).
Ideally the GC should free an object when we are done with it, but what does "done" mean? This can be reduced to the halting problem, so instead we approximate this idea by freeing once an object becomes **unreachable**.

Reachability Note that reachability starts from outside the heap: we have a pointer to some object in the heap on e.g. the stack or static space. The GC must be able to DFS search through references nested in objects in order to determine what other objects are reachable/referenced still. Therefore, the GC must have enough knowledge that largely depends on the information the compiler surfaces, thus the GC and compiler are usually written together in tandem. For example, the compiler must tell the GC how to look through the **roots** (e.g. the stack and static space) for object references.

22.4 Mark and sweep GC

So we know unreachable objects need to be freed, but how do we reach these unreachable objects? The heap must be **parseable** i.e. the GC must be able to scan through it and find objects to free them. This can be accomplished by having a linked list between the allocated object headers. When perform reachabiity check, we "mark" some field in the header to denote that object is reachable. Then the GC parses through the object and frees the unmarked, unreachable objects. This type of GC is called **mark and sweep**; however, it is a very expensive and uncompetitive version.

22.5 Semispace copying GC

Instead of marking reachable objects and then freeing unreachable objects, instead during reachability parsing we **copy** the reachable objects to a new pool of memory (**tospace**). Finally, we free the entire previous pool of memory (**fromspace**) since all that is remaining are unreachable objects. We then reverse the roles of **tospace** and **fromspace** for the next iteration. This is called **semispace copying**.
Implications:

1. Let L represent our allocated size and H the size of the heap. Note that in semispace copying, our runtime only depends on L (reachability parsing). If $L <<< H$, then mark and sweep is much more expensive such it depends on the size of H when it parses through the *entire* heap.

   Note increasing H requires us to perform GC less often in semispace copying.

2. We need to update the references/pointers to point to the new space. This is easy since the compiler already tells us the references during reachability.

   We must also take care not to copy an object twice (e.g. if it's referenced twice).

   There is a way to perform these updates in one DFS traversal.

   Note that we can no longer use references/addresses as a unique ID for objects.

3. We can only use at most half of the heap

Note that **allocation is super easy**: we no longer need free lists for objects but rather just a **bump-pointer** for free space (where we bump it every time we allocate more space).

22.6 Mark and compact GC

Instead of copying to two regions of the heap, we can instead first mark the objects then copy the object to the beginning of the same pool. However in contrast to copying allocation, we must first traverse and mark in order to find the **forward/new addresses** to which to copy.
To solve this problem, we require 3 sweeps:

- Imagine compacting and store the forward address with the object
- Update the references to the objects with the new forward address

- Actually compact/copy the object

It is unclear how to do this in fewer sweeps. Note that the runtime is $O(H)$, and the constant factor is quite high (since we have a mark phase and 3 sweep phases). This however may work well for infrequent GCs.

22.7 Generational GCs

Generational GCs notice that most objects die young, so we can partition the heap by age. The "nursery" partition is GC'ed frequently and the "old" partition is only collected during a full-heap GC.
We can then combine the various methods of GC and work towards their strength.
When the "nursery" is full, we perform copying allocation where we treat the "old" partition as the **tospace**.
Afterwards the "nursery" is empty and we can perform mark and sweep GC on the "nursery" as per usual.

22.8 Allocation spectrum

On one end of the spectrum `C` and `malloc` simply returns a set of bytes: manager knows no type information and object is returned full of garbage.
On the other end `Haskell` allocation and the manager knows the static type and so object is returned fully initialized and immutable.
In the middle, `Java` and `new` returns objects zeroed.