Tips

- ϵ is not a terminal symbol

Context-free Grammars

Left-associative - $(1 \oplus 2) \oplus 3$

$$E \to E \oplus T \mid T$$
$$T \to N$$

Right-associative - $1 \oplus (2 \oplus 3)$

$$E \to T \oplus E \mid T$$
$$T \to N$$

$$E \to TE'$$
$$E' \to \epsilon \mid \oplus TE'$$
$$T \to N$$

Left-factoring and left-recursion removal

Removing left recursion

$$E \to E \oplus T \mid T$$

is transformed into

$$E \to TE'$$
$$E' \to \epsilon \mid \oplus TE'$$

This transformers a left-associative grammar into a right-associative grammar

Recogniser Code

T

```
tokens.match(Token.T);
```

N

```
parseN();
```

$S_1 \dots S_n$

```
recog(S_1);  ...;  recog(S_n);
```

$S_1|S_2|\dots S_n$

```
if (token.isIn(First(S_1))) {
    recog(S_1);
} ...
} else if (
    tokens.isIn(First(S_n))) {
        recog(S_n);
} else {
    errors.error("Syntax error");
}
```

$[S]$

```
if (token.isIn(First(S))) {
    recog(S);
}
```

$\{S\}$

```
while (token.isIn(First(S))) {
    recog(S);
}
```

(S)

```
recog(S);
```

Methods like `token.getValue()` do not consume the token. `match(NUMBER)` will need to be called immediately afterwards.

First, Follow, and LL(1)

Calculating First sets

- If production of form $N \to \epsilon$, add ϵ to first set for N to indicate nullability
- If production of form $N \to S_1 S_2 \dots S_n$, then if $\forall i \in 1..n, \forall j \in 1..i-1 \cdot S_j$ is nullable, we add current first set for S_i to first set for N
- If every construct $S_1, \dots, S_n$ is nullable, add ϵ to first set for N

Perform for all productions, repeating the process until no sets are modified

LL(1) Grammars

Definition

A BNF grammar is LL(1) if for each nonterminal, N, where $N \to a_1 \mid a_2 \mid \dots \mid a_n$,

the first sets of each pair of alternatives for N are disjoint:

$$\forall i, j \in 1..n \cdot i \neq j \implies First(a_i) \cap First(a_j) = \{\}$$

if N is nullable, $First(N)$ and $Follow(N)$ are disjoint, i.e.,

$$N \overset{*}{\Rightarrow} \epsilon \implies First(N) \cup Follow(N) = \{\}$$

Because the first set fr an alternative includes ϵ if the alternative is nullable, the constraint that the first sets of all the alternatives are pairwise disjoint implies at most one alternative is nullable. [This is the reason for including ϵ in the First sets.]

Bottom up parsing

LR(x) parsing automaton

Don't forget to first add production: $S' \to E$.

LR(x) parsing action conflicts

There is no such thing as a *shift/shift* conflict

Derived LR(1) items

If a state has an LR(1) item of the form

$$[N \to a \cdot M\beta,\ T]$$

where the nonterminal M has productions

$$M \to a_1 \mid a_2 \mid \dots \mid a_m$$

and $T = \{a_1, a_2, \dots, a_m\}$, then the state also includes the derived items

$$[M \to \cdot a_1,\ T']$$
$$\dots$$
$$[M \to \cdot a_m,\ T']$$

where $T' = First(\beta a_1) \cup \dots \cup First(\beta a_n)$. If β is not nullable $T' = First(\beta)$ and if β is nullable, $T' = First(\beta) - \{\epsilon\} \cup T$.

LR(1) parsing algorithm

Put \$0 on the *Parsing stack*, and the input string, followed by \$, in the *Input* queue

1. Choose transition action based on lookahead. If it is
   (a) *shift*, dequeue start symbol of *Input* queue, and put dequeued symbol on the *Parsing stack*
   (b) *reduce*, pop start symbol of the **RHS** of the reduction, and all stack elements above the start symbol, off the stack. Transition to state indicated by number currently on top of stack. Put reduced symbol on the stack. *If LR(1), choose production s.t. $queue_0 \in T$, where T is look-ahead set.* Follow transition path of current state, based on the reduced symbol.
   (c) *accept*, do nothing
2. Put number indicating current state on the stack

   Repeat numbered process

Parameters

Kinds of parameter passing mechanisms

Call by const: the formal parameter acts as a read-only local variable that is initially assigned the value of the actual parameter expression.
Call by value: the formal parameter acts as a local variable that is initially assigned the value of the actual parameter expression.
Call by result: the formal parameter acts as a local variable whose final value is assigned to the actual parameter variable.
Call by value-result: a single parameter acts as both a value and a result parameter

Kinds of parameter passing

Call by reference: the formal parameter is really the address of the actual parameter variable; all references to the formal parameter are applied (via that address) to the actual parameter variable immediately. (In Pascal known as a **var** parameter).
Call by name: the actual parameter expression is evaluated every time the formal parameter is accessed.
Passing procedures (or functions) as parameters: need to pass the address of the procedure as well as the static link for the procedures environment.
Returning procedures (or functions): need to return the address of the procedure as well as the static link for the procedures environment - note that this requires the environment of the returned procedure to be maintained which means that the simple stack-based allocation of frames is not sufficient.

Dynamic Memory Allocation/Deallocation

Garbage collection schemes

Mark and sweep garbage collection consists of

- a phase that *marks* all the accessible objects
- a phase that *sweeps* up the objects left unmarked and adds them to the free list

Stop-and-copy (or two-space) garbage collection

- divides the available memory into two (large) spaces
- memory is allocated sequentially from one space until it runs out
- garbage collection consists of relocating all accessible objects from the first space to the second space
- because the objects are allocated sequentially in the second space, they are compacted
- when the copy is completed the roles of the spaces are swapped for the next garbage collection

Find all objects accessible from the runtime stack or global variables (either directly or indirectly) and copy them into the second space, allocating them sequentially (so that they are allocated more compactly), and placing a forwarding pointer with the old object to the new copy, so that other references to the (old) object can be updated to point to the new object.
Generational schemes use a scheme similar to the two-space scheme but make use of multiple spaces

- the spaces are organised based on the length of time its objects have survived
- the age of an object is the number of times it has been collected
- older objects are migrated to an old object space
- newer objects go in the new object space
- the objects in the old object space dont need to be copied when the new object space is garbage collected
- the old object space may have to be garbage collected at some stage

Regular expressions and finite state machines

Definitions

The empty closure of a state x in an NFA N, ϵ-closure(x, N), is the set of states in N that are reachable from x via any number of empty transitions.
The empty closure of a set of states X in an NFA N, ϵ-closure(X, N), is the set of states in N that are reachable from any of the states in X via any number of empty transitions.

Constructing the DFA from the NFA

- The label of the start state of the DFA consists of the set of states containing the start state s_0 of the NFA plus all the states in the NFA that are reachable from its start state by one or more empty transitions.

$$S_0 = \epsilon\text{-closure}(s_0, N)$$

- The process for forming a DFA works with a set of unmarked DFA states by selecting an unmarked DFA state and considering all transitions from it on symbols.
- The initial set of unmarked states contains just S_0.
  The followig process is repeated util there are no unmarked DFA states left.
- An unmarked DFA state S is selected (the first one is S_0).
- For each symbol a,
  - we consider the set of states that can be reached from any state in S by a transition on a; call this set of states X
  - if X is nonempty, we add a new state to the DFA labelled with $X' = \epsilon$-closure(X, N), unless a state with that label already exists
  - a transition from S to X' on a is added to the DFA.
- The state S is marked as having being processed.

Stack Machine

Definitions

Each activation contains:
a static link (at offset 0 from *fp*) also called an access link, it is used for accessing non-local variables;
a dynamic link (at offset 1) also called a control link, it contains the address of the activation record of the calling procedure (so the frame pointer can be restored on return from the procedure);
a return address (at offset 2) which is the address of the instruction to execute on return from the procedure (used for restoring the program counter on return);
local variables which are allocated sequentially on the stack with addresses relative to the frame pointer starting from offset 3; and
parameters which are allocated sequentially on the stack with negative addresses relative to the frame pointer.

Calling a procedure

1. any parameters to the procedure are pushed onto the stack;
2. the static link is pushed onto the stack;
3. the current frame pointer is pushed onto the stack to create the dynamic link;
4. the frame pointer is set so that it contains the address of the start of the new stack frame (i.e., the address of the location containing the static link);
5. the current value of the program counter (which is the address of the instruction after the call) is pushed onto the stack to form the return address;
6. the program counter is set to the address of the procedure and execution of the body of the procedure begins;
7. space is allocated on the stack for any local variables (using instruction **ALLOC_STACK**).

Returning from a procedure

1. the program counter is set to the return address in the current activation record;
2. the frame pointer is set to the dynamic link;
3. the stack pointer is set so that all the space used by the stack frame (but not parameters) is popped from the stack;
4. execution continues at the instruction addressed by the (restored) program counter; and
5. after return the calling procedure handles deallocating any parameters (using instruction **DEALLOC_STACK**)

Static semantics

$$\texttt{e} ::= \texttt{n} \mid \texttt{id} \mid \texttt{e.id} \mid \texttt{record(id, fe)}$$

$$\texttt{fe} ::= \texttt{id} \mapsto \texttt{e}$$

$$\mathit{syms} \vdash \texttt{n} : \mathit{int}$$

$$\frac{\texttt{id} \in \texttt{dom}(\mathit{syms}) \qquad \mathit{syms}(\texttt{id}) = \mathit{VarEntry}(T)}{\mathit{syms} \vdash \mathit{id} : T}$$

$$\frac{\mathit{syms} \vdash \texttt{e} : \mathit{RecordType}(\mathit{fields}) \qquad \texttt{id} \in \texttt{dom}(\mathit{fields})}{\mathit{syms} \vdash \texttt{e.id} : \mathit{fields}(\texttt{id})}$$

$$\frac{\begin{array}{c}\texttt{id} \in \texttt{dom}(\mathit{syms}) \\ \mathit{syms}(\texttt{id}) = \mathit{TypeEntry}(\mathit{RecordType}(\mathit{fields})) \\ \texttt{dom}(\mathit{fields}) = \texttt{dom(fe)} \\ \forall \texttt{id} \in \texttt{dom}(\mathit{fields}) \cdot \mathit{syms} \vdash \texttt{fe(id)} : \mathit{fields}(\texttt{id})\end{array}}{\mathit{syms} \vdash \texttt{record(id, fe)} : \mathit{RecordType}(\mathit{fields})}$$

$$\texttt{e} ::= \texttt{n} \mid \texttt{id} \mid \ \ldots \ \mid \lambda\ \texttt{id1} : \texttt{id2}\ \texttt{e} \mid \texttt{e1(e2)}$$

$$\frac{\begin{array}{c}\texttt{id} \in \texttt{dom}(\mathit{syms}) \\ \mathit{syms}(\texttt{id2}) = \mathit{TypeEntry}(T1) \\ \mathit{syms} \oplus \{\texttt{id1} \mapsto \mathit{VarEntry}(T1)\} \vdash \texttt{e} : T2\end{array}}{\mathit{syms} \vdash (\lambda\ \texttt{id1} : \texttt{id2}\ \texttt{e}) : (T1 \to T2)}$$

$$\frac{\mathit{syms} \vdash \texttt{e1} : T1 \to T2 \qquad \mathit{syms} \vdash \texttt{e2} : T1}{\mathit{syms} \vdash \texttt{e1(e2)} : T2}$$

$$\texttt{e} ::= \texttt{n} \mid \texttt{id} \mid \ \ldots \ \mid \texttt{let id = e in end}$$

$$\frac{\begin{array}{c}\mathit{syms} \vdash \texttt{e1} : T1 \qquad \mathit{syms} \vdash \texttt{e1} \xrightarrow{e} v1 \\ \mathit{syms} \oplus \{\texttt{id} \mapsto \mathit{ConstEntry}(T1, v1)\} \vdash \texttt{e2} : T2 \\ \mathit{syms} \oplus \{\texttt{id} \mapsto \mathit{ConstEntry}(T1, v1)\} \vdash \texttt{e2} \xrightarrow{e} v2\end{array}}{\begin{array}{c}\mathit{syms} \vdash \texttt{let id = e1 in e2 end} : T2 \\ \mathit{syms} \vdash \texttt{let id = e1 in e2 end} \xrightarrow{e} v2\end{array}}$$